Using Amazon DynamoDB global tables

AWS Prescriptive Guidance

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

AWS Prescriptive Guidance: Using Amazon DynamoDB global tables

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

Introduction **1**
Overview **2**
Key facts 2
Consistency modes 2
Key facts about MREC 3
Key facts about MRSC 4
Use cases 5
Write modes **7**
Write to any Region mode (no primary) 7
Write to one Region mode (single primary) 9
Write to your Region mode (mixed primary) 11
Routing strategies **14**
Client-driven request routing 14
Compute-layer request routing 16
Route 53 request routing 18
Global Accelerator request routing 19
Evacuation processes **21**
Evacuating a live Region 21
Evacuating an offline Region 21
Evacuating on offline MRSC table 22
Evacuating an offline MREC table 22
Throughput capacity planning **25**
Preparation checklist **27**
FAQ **29**
Conclusion and resources **33**
Document history **34**

Using Amazon DynamoDB global tables

Jason Hunter, Amazon Web Services

Global tables build on the global Amazon DynamoDB footprint to provide you with a fully managed, multi-Region, and multi-active database that delivers 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.

Overview

Key facts

- There are two versions of global tables: version 11.29 (legacy) (sometimes called *v1*) and version 2019.11.21 (current) (sometimes called *v2*). 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 do not specify a consistency mode when you create a global table, the global table defaults to MREC. A global table cannot contain replicas that are configured with different consistency modes. You cannot change a global table's consistency mode after its creation.

Key facts about MREC

- Global tables that use MREC 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. (For more information, see the AWS re:Invent 2024 presentation, [Multi-Region strong consistency with Amazon DynamoDB global tables](#).) This communication tends to add latency that increases based on the round-trip latency between the Region that is being accessed and the nearest Region that participates in the global table. 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 that use MRSC do not support the transaction APIs.
- An MRSC global table must be deployed in exactly three Regions. You can configure an 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 cannot perform read or write operations on a witness. A witness does not 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 cannot 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 it's configured in, from the output of the DescribeTable API. The witness is owned and managed by DynamoDB and does 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)
  - AP Region set: Asia Pacific (Tokyo), Asia Pacific (Seoul), and Asia Pacific (Osaka)
- MRSC global tables cannot span Region sets. For example, an MRSC global table cannot contain replicas from both US and EU Region sets.
- 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 reported only for the Region in which an operation occurred.
- The local Region accepts all read and write operations as long as a second Region that hosts a replica or witness is available to establish quorum. If a second Region isn't available, the local Region can service only 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 will return errors. However, eventually consistent read operations 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.

Use cases

MREC global tables provide these benefits:

- **Lower-latency read operations**. You 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**. An end user can write to a nearby Region to reduce network latency and the time to complete the write operation. The write traffic must be carefully routed to ensure that there are no conflicts. Techniques for routing are discussed in a later section.
- **Seamless Region migration**. You can add a new Region and then delete the old Region to migrate a deployment from one Region to another, without any 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 (that is, move away some or all requests that go 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 attain 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 that's equal to the replication delay between replicas—usually a few seconds depending on the replica Regions. For more information, see Consistency modes in the DynamoDB documentation.

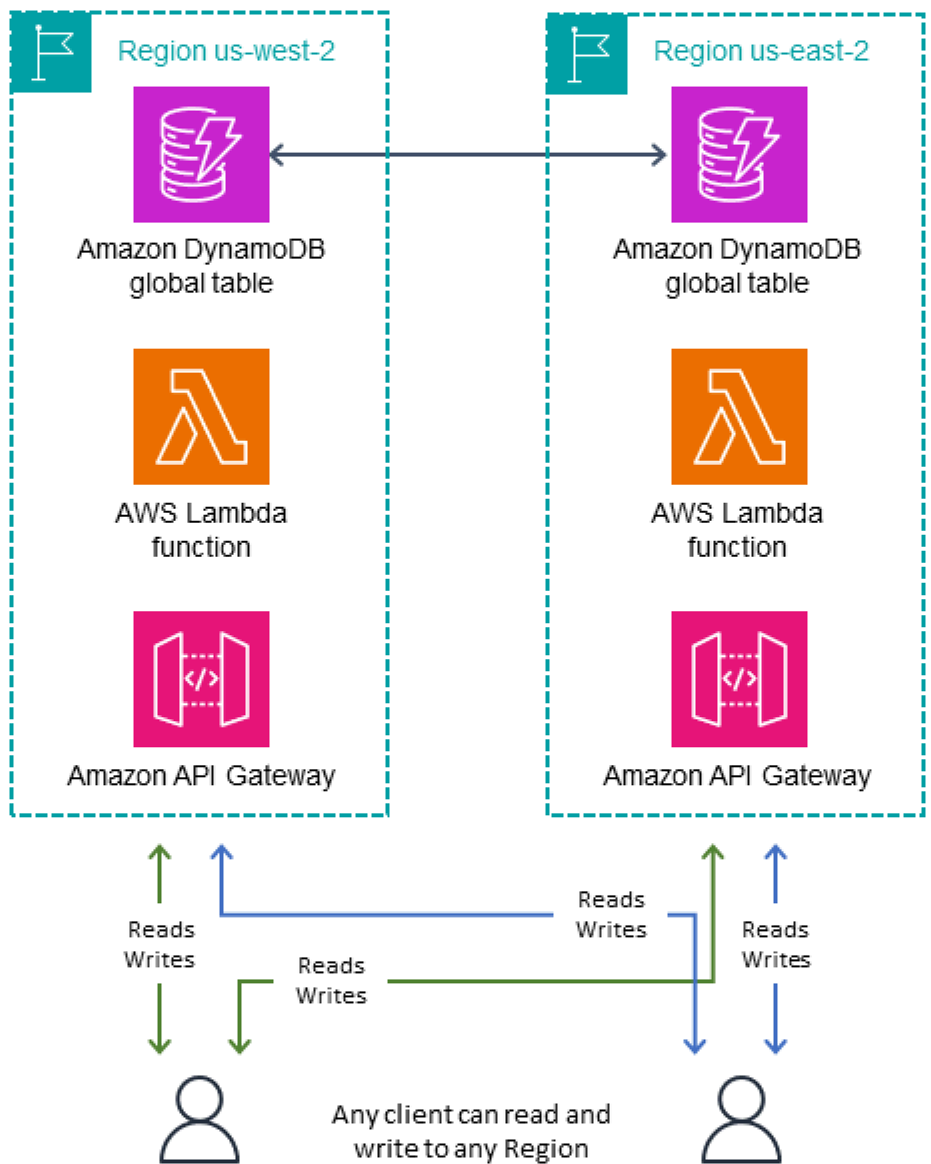
Write modes

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* write mode, illustrated in the following diagram, is fully active-active and doesn't impose restrictions on where a write operation may occur. Any Region can accept a write request at any time. This is the simplest mode; however, it can be used only with some types of applications. This mode is suitable for all MRSC tables. It's also suitable for MREC tables when all write operations are idempotent. *Idempotent* means that they are safely repeatable so that concurrent or repeated write operations across Regions are not in conflict—for example, when a user updates their contact data. It also works well for an append-only dataset where all write operations are unique inserts under a deterministic primary key, which is a special case of being idempotent. Lastly, this mode is suitable for MREC where the risk of conflicting write operations is 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 MREC tables any recent write operations 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 isn't 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.

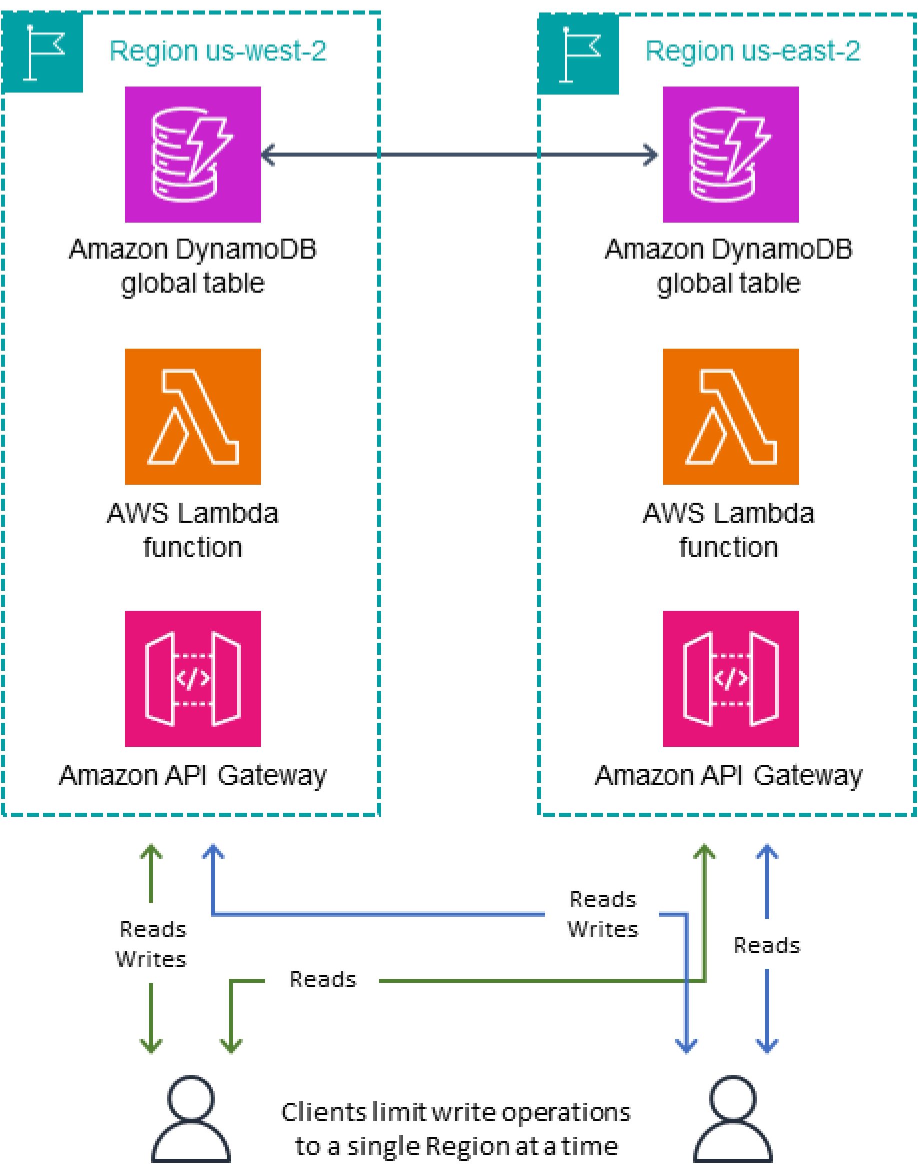
Write to one Region mode (single primary)

The *write to one Region* write mode, illustrated in the following diagram, is active-passive and routes all table write operations to a single active Region. (DynamoDB doesn't have a notion of a single active Region; the layer 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 read operations can go to any of the replica Regions to achieve lower latencies. Strongly consistent read operations must go to the single primary Region.



It's sometimes necessary to change the active Region in response to a Regional failure, as discussed later. 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) might keep a downscaled 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 more information, you can read the blog post [Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby].

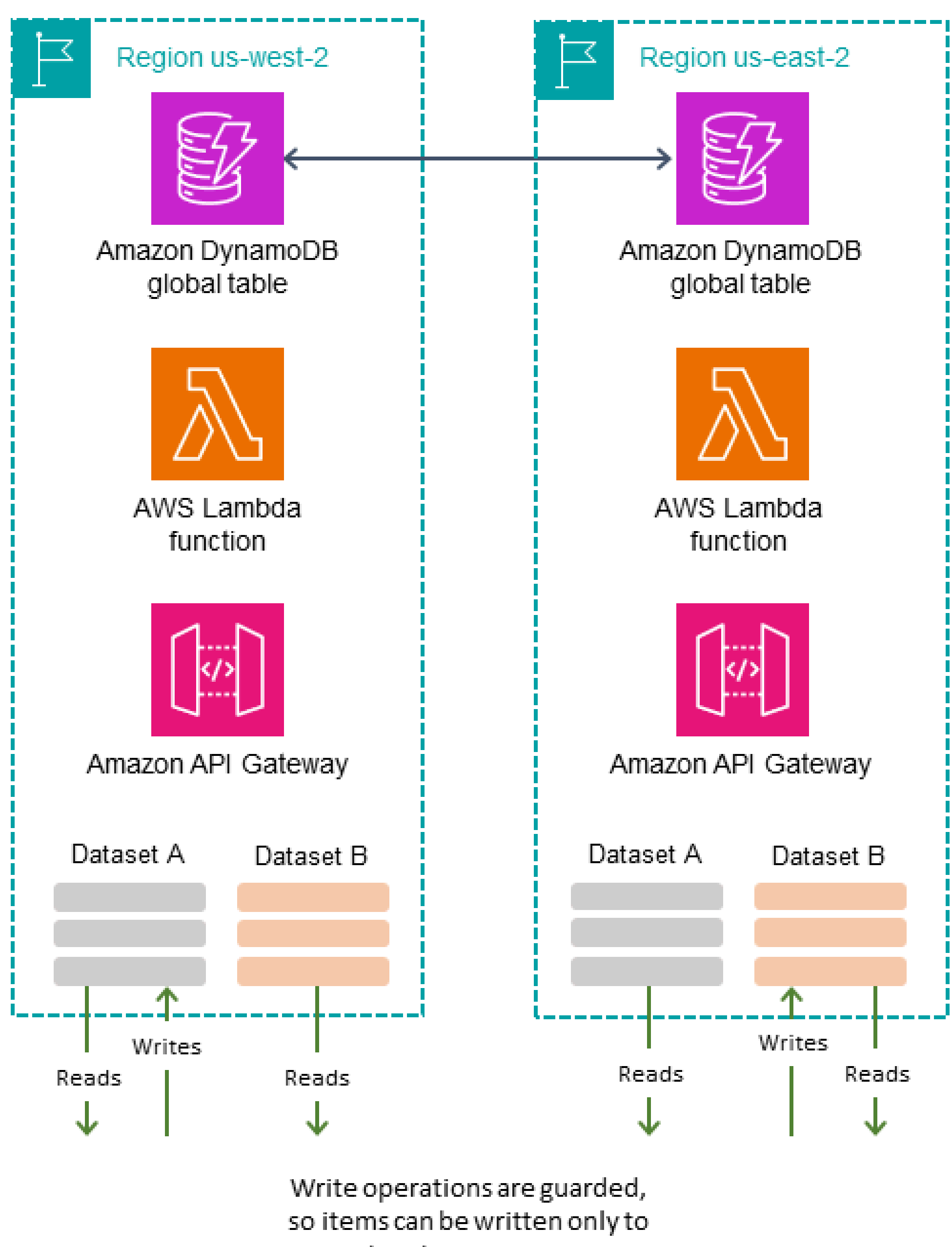
Using the *write to one Region* mode works well when you use global tables for low-latency, globally distributed read operations. An example is a large social media company that needs to have the same reference data available in every Region around the world. They don't update the data often, but when 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 mode (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 might 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

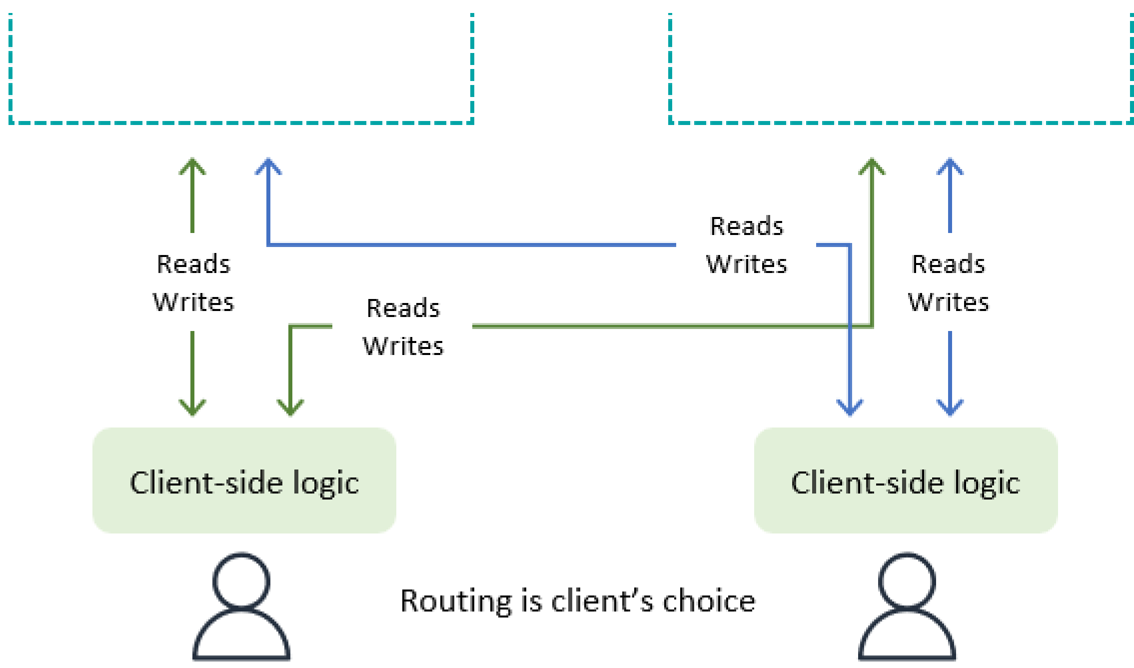
Perhaps the most complex piece of a global table deployment is managing request routing. Requests must first go from an end user to a Region that's chosen and routed in some manner. The request encounters some stack of services in that Region, including a compute layer that perhaps consists of a load balancer backed by an AWS Lambda function, container, or Amazon Elastic Compute Cloud (Amazon EC2) node, and possibly other services, including maybe 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 techniques for routing an end user request to a Region for processing. The right choice depends on your write mode and your failover considerations. This section discusses four options: client-driven, compute-layer, Amazon Route 53, and AWS 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.

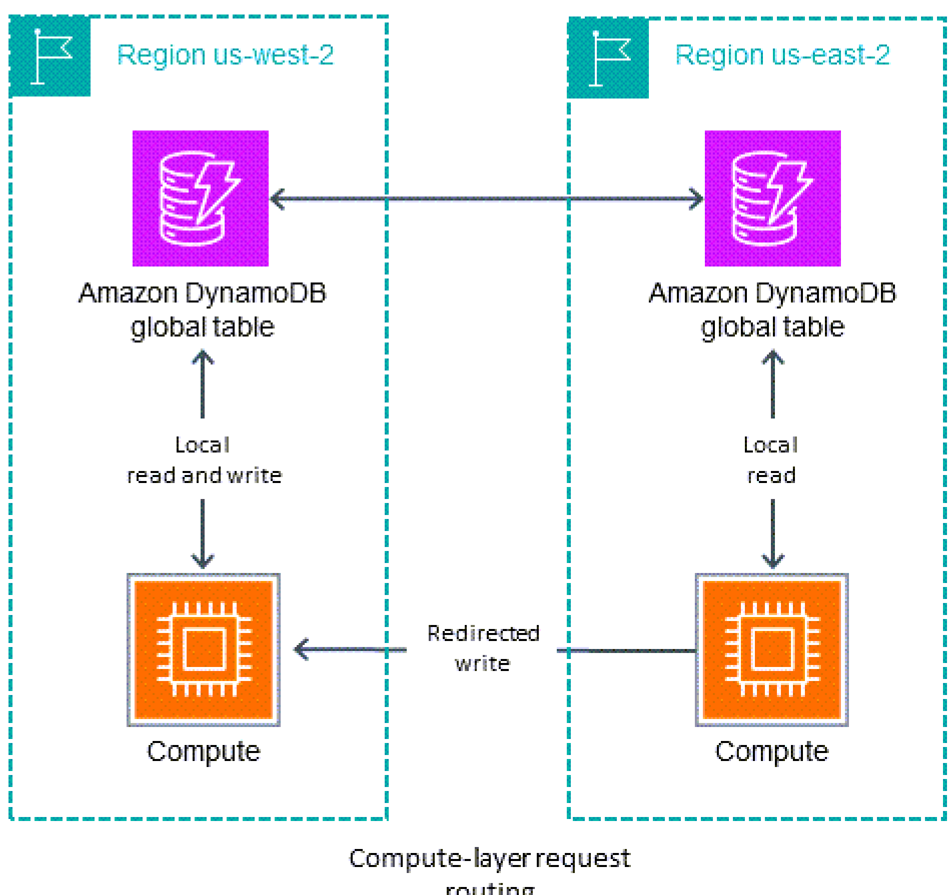
With *write to one Region* mode, the client needs a mechanism to route its write requests to the currently active Region. This could be a basic mechanism, such as empirically testing which Region is presently accepting write requests (noting any write rejections and falling back to an alternate). Or it can be a complex mechanism, such as using a global coordinator to query for the current application state (perhaps built on the Amazon Application Recovery Controller (ARC) routing control, which provides a five-Region, quorum-driven system to maintain global state for needs such as this). The client can decide if read requests can go to any Region for eventual consistency or must be routed to the active Region for strong consistency.

With the *write to your Region* mode, the client needs to determine the home Region for the dataset it's working with. 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 assignment to use with its credentials from a global login system.

For example, a financial services company that helps users manage their business finances through the web uses global tables with a *write to your Region* mode. Each user must log in to a central service. That service returns credentials as well as the endpoint for the Region where those credentials will work. The Region that's returned is based on where the user's dataset is currently homed. 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 microservice. 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.



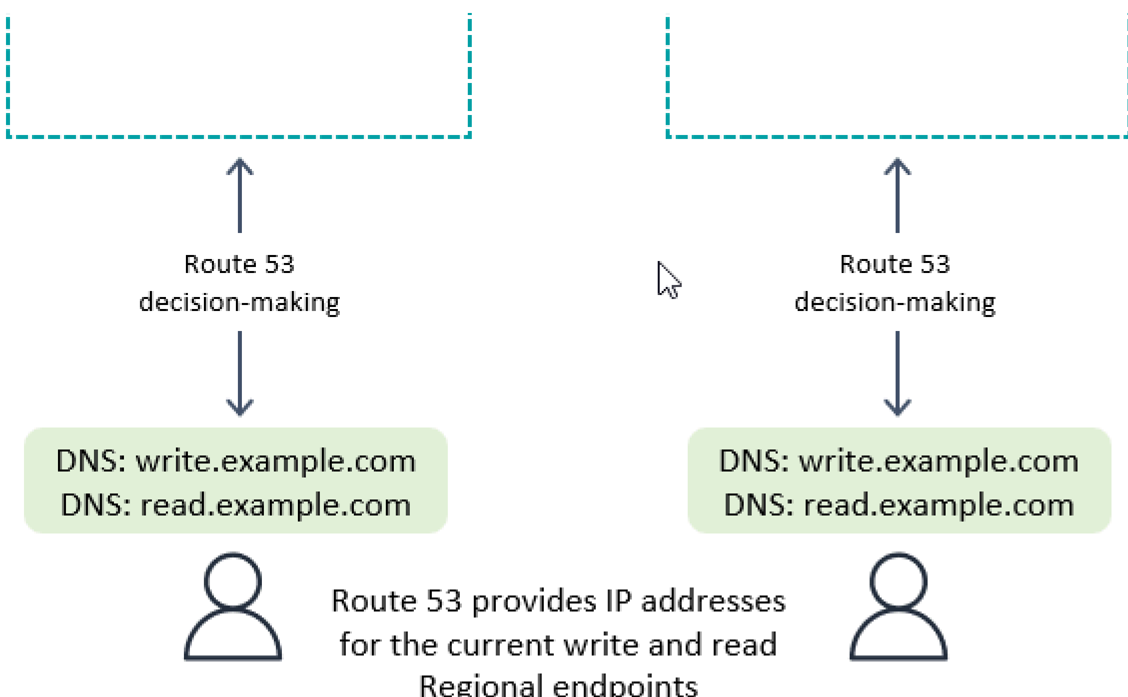
Compute-layer request routing

presented at re:Invent 2022

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 . 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 Route 53 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 that corresponds to the Regional endpoint(s) it determines 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 freedom to return the Region based on any complex internal rules, such as choosing the Region in the closest network or geographic proximity, or any other choice.

With *write to one Region* mode, you can configure Route 53 to return the currently active Region (by using 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

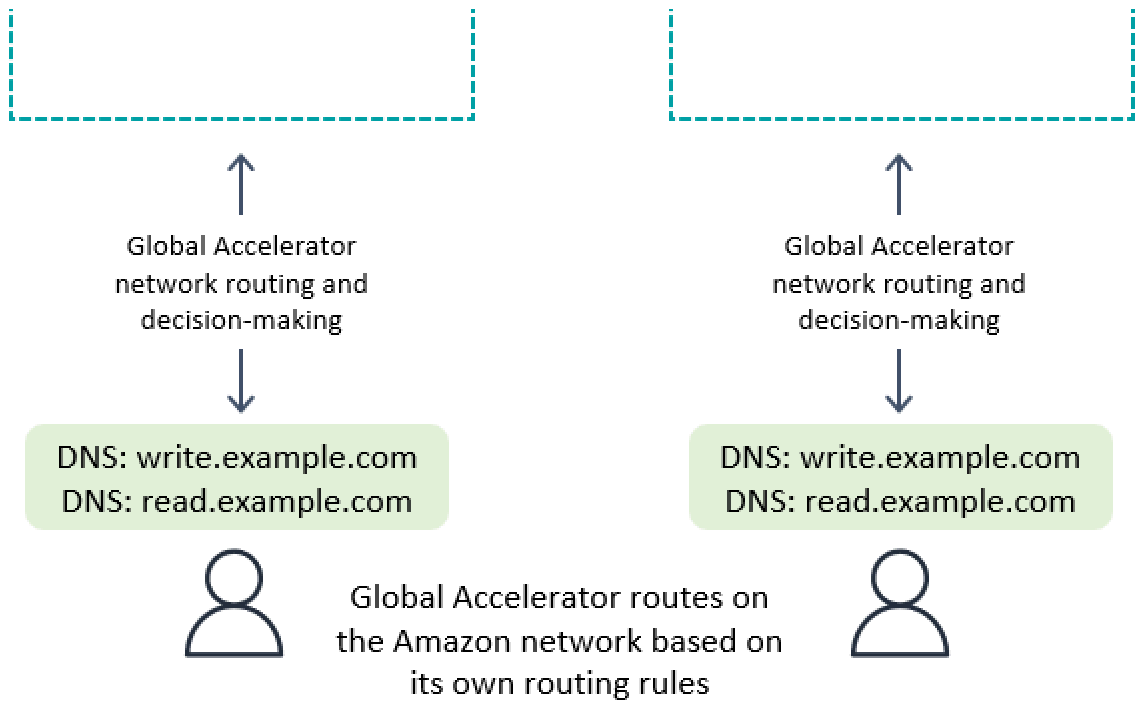
Clients cache the IP addresses in the response from Route 53 for a time indicated by the time to live (TTL) setting on the domain name. A longer TTL extends the recovery time

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 gets back an anycast static IP address that routes to the nearest AWS edge location. Starting from that edge location, all traffic gets routed on the private AWS network to some endpoints (Network Load Balancers, Application Load Balancers, EC2 instances, or elastic IP addresses) 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 backend, Global Accelerator works seamlessly. The client connects to the nearest edge location and doesn't have to be concerned about which Region receives the request.

With *write to one Region* mode, 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 write activity, possibly 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 on 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) 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

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 might float independently between Regions.
- One reason you might see different provisioned write capacity is due to the time to live (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. Remember, TTL is not 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 table to accept a higher level of read traffic.

ARC has readiness checks that can be useful in confirming that DynamoDB Regions have similar table settings and account quotas, whether or not you use Route 53 to route requests. These readiness checks also help you adjust account-level quotas to make them match.

Preparation checklist

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.)
- Plan your routing strategy, based on your write mode.
- Define your evacuation plan, 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. You should also use synthetic canaries (artificial requests designed to detect failures) as well as live observation of customer traffic. Not all issues 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 conditions where you would evacuate a Region. 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 might 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 Hubto evaluate the resilience of your entire application (including global tables). This service provides a comprehensive view of the resiliency status of your application portfolio 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.

FAQ

This section answers frequently asked questions about DynamoDB 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 WCUs and WRUs.
- rWCU and rWRU charges 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 might still 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 do global tables support?

- Global tables (current, version 2019) support 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 (Frankfurt)
  - AP Region set: Asia Pacific (Tokyo), Asia Pacific (Seoul), and Asia Pacific (Osaka).

How are GSIs handled with global tables?

- In global tables (current, version 2019), when you create a GSI in one Region, it's automatically created in other participating Regions and automatically backfilled.

How do I stop the 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 cannot 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 does 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 `Regionattribute` 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 Streams 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, 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 (DAX) cache?

- 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 back up 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 by using AWS 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, keep 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: 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 GitHub repository amazon-dynamodb-table-to-global-table-cdk.
- 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:

- Amazon DynamoDB documentation
- Amazon Application Recovery Controller (ARC)
- ARC readiness checks (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 Amazon DynamoDB global tables (AWS re:Invent 2024 presentation)

Document history

Change	Description	Date
Added information about MRSC support	Updated the guide to reflect consistency modes and multi-Region strong consistency (MRSC) support.	September 3, 2025
Updated AWS Global Accelerator information	Corrected the endpoints for Global Accelerator request routing.	March 14, 2024
Updated AWS Region support information	Updated the FAQ to indicate that global tables now support all AWS Regions.	November 15, 2023
Initial publication	—	May 19, 2023