用户指南

Amazon Aurora DSQL

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

Amazon Aurora DSQL: 用户指南

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

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

什么是 Amazon Aurora DSQL？ 1
何时使用 1
主要 功能 1
AWS 区域可用性 2
多区域集群 4
定价 5
接下来做什么？ 5
开始使用 6
先决条件 6
创建单区域集群 6
连接到集群 7
运行 SQL 命令 7
创建多区域集群 8
问题排查 11
身份验证和授权 12
管理集群 12
连接到集群 12
PostgreSQL 和 IAM 角色 13
将 IAM 策略操作与 Aurora DSQL 结合使用 14
使用 IAM 策略操作连接到集群 14
使用 IAM 策略操作管理集群 14
使用 IAM 和 PostgreSQL 撤销授权 15
生成身份验证令牌 16
控制台 16
AWS CloudShell 17
AWS CLI 18
Aurora DSQL SDK 19
数据库角色和 IAM 身份验证 27
IAM 角色 27
IAM 用户 28
Connect 28
查询 28
查看映射 29
撤销 29

Aurora DSQL 和 PostgreSQL 30
兼容性亮点 30
分布式架构优势 31
SQL 兼容性 31
支持的数据类型 31
支持的 SQL 功能 36
支持的 SQL 命令子集 40
迁移指南 58
并发控制 65
事务冲突 65
优化事务性能的准则 65
DDL 和分布式事务 65
主键 67
数据结构和存储 67
选择主键的准则 67
序列和标识列 68
序列操作函数 68
标识列 70
使用序列和标识列 71
异步索引 73
语法 73
参数 73
使用说明 74
创建索引 75
查询索引 75
唯一索引构建失败 77
唯一性违规 77
系统表和命令 79
系统表 79
有用的系统查询 89
ANALYZE 命令 90
EXPLAIN 计划 91
PostgreSQL EXPLAIN 计划 91
关键元素 92
过滤 93
阅读 EXPLAIN 计划 93

EXPLAIN ANALYZE 中的 DPU 97
管理 Aurora DSQL 集群 100
单区域集群 100
使用 AWS SDK 100
使用 AWS CLI 138
多区域集群 142
使用 AWS SDK 142
使用 AWS CLI 195
CloudFormation 201
初始配置 201
查找集群 202
更新配置 202
Aurora DSQL 集群生命周期 203
集群状态 203
查看集群状态 205
使用 Aurora DSQL 进行编程 207
连接器 207
JDBC 连接器 208
Python 连接器 212
Go 连接器 223
Node.js 连接器 229
Ruby 连接器 237
.NET 连接器 243
访问 Aurora DSQL 248
SQL 客户端 249
DBeaver 249
JetBrains DataGrip 252
psql 253
VSCode 254
问题排查 256
数据库连接工具 256
Aurora DSQL 适配器 256
数据库驱动程序示例 257
ORM 和框架示例 258
加载数据 260
选择加载方法 260

Aurora DSQL 加载器 261
迁移路径 265
使用 PostgreSQL \copy 267
其他资源 268
生成式人工智能 268
AWS Labs Aurora DSQL MCP 服务器 268
Aurora DSQL 操控：技能和能力 276
查询编辑器 281
先决条件 281
使用查询编辑器 282
查询编辑器：将 JupyterLab 与 Aurora DSQL 结合使用 283
开始使用 283
示例笔记本 286
延伸阅读 286
备份和还原 287
开始使用 AWS Backup 287
还原备份 287
配置多区域集群 288
还原多区域集群 288
监控和合规性 288
其他资源 288
监控和日志记录 290
使用 CloudWatch 进行监控 290
可观测性 290
使用量 291
使用 Cloudtrail 进行日志记录 293
管理事件 293
数据事件 295
安全性 297
AWS 托管策略 298
AmazonAuroraDSQLFullAccess 298
AmazonAuroraDSQLReadOnlyAccess 299
AmazonAuroraDSQLConsoleFullAccess 299
AuroraDSQLServiceRolePolicy 301
策略更新 301
数据保护 306

- 数据加密 ... 307
- 见证区域中的数据保护 ... 308
- SSL/TLS 证书 ... 308

数据加密 ... 307
- KMS 密钥类型 ... 315
- 静态加密 ... 315
- 使用 KMS 和数据密钥 ... 316
- 授权 KMS 密钥 ... 318
- 加密上下文 ... 320
- 监控 AWS KMS ... 320
- 创建加密集群 ... 323
- 移除或更新密钥 ... 325
- 注意事项 ... 327

Identity and access management ... 327
- 受众 ... 328
- 使用身份进行身份验证 ... 328
- 使用策略管理访问 ... 329
- Aurora DSQL 如何与 IAM 协同工作 ... 331
- 基于身份的策略示例 ... 335
- 问题排查 ... 340

基于资源的策略 ... 342
- 何时使用 ... 342
- 使用策略创建 ... 343
- 添加和编辑策略 ... 346
- 查看策略 ... 349
- 移除策略 ... 350
- 策略示例 ... 352
- 阻止公有访问 ... 356
- API 操作 ... 359

使用服务相关角色 ... 361
- Aurora DSQL 的服务相关角色权限 ... 361
- 创建服务相关角色 ... 362
- 编辑服务相关角色 ... 362
- 删除服务相关角色 ... 362
- Aurora DSQL 服务相关角色的受支持区域 ... 363

使用 IAM 条件键 ... 363

在特定区域中创建集群 363
在特定区域中创建多区域集群 363
创建具有特定见证区域的多区域集群 364
事件响应 365
合规性验证 366
恢复能力 366
备份与还原 366
复制 366
高可用性 367
故障注入测试 367
基础设施安全性 368
使用 AWS PrivateLink 管理集群 368
配置和漏洞分析 378
防止跨服务混淆座席 379
安全最佳实践 380
检测性安全最佳实践 380
预防性安全最佳实践 381
为资源添加标签 383
名称标签 383
标记要求 383
标记使用说明 383
注意事项 385
限额和限制 387
集群配额 387
数据库限制 388
API 参考 391
问题排查 251
连接错误 392
身份验证错误 392
授权错误 393
SQL 错误 394
OCC 错误 394
SSL/TLS 连接 395
提供反馈 396
反馈渠道 396
有效的功能请求 396

文档历史记录 397

什么是 Amazon Aurora DSQL？

Amazon Aurora DSQL 是一个针对事务性工作负载进行了优化的无服务器、分布式关系数据库服务。Aurora DSQL 提供了几乎无限的规模，并且不需要您管理基础设施。主动-主动高可用性架构可提供 99.99% 的单区域可用性和 99.999% 的多区域可用性。

何时使用 Aurora DSQL

Aurora DSQL 针对受益于 ACID 事务和关系数据模型的事务性工作负载进行了优化。由于 Aurora DSQL 是无服务器的，因此非常适合微服务、无服务器和事件驱动型架构的应用程序模式。Aurora DSQL 与 PostgreSQL 兼容，因此，您可以使用熟悉的驱动程序、对象关联映射（ORM）、框架和 SQL 功能。

Aurora DSQL 可自动管理系统基础设施，并根据工作负载扩展计算、I/O 和存储。由于您没有服务器可供预置或管理，因此，您不必担心与预置、修补或基础设施升级相关的维护停机时间。

Aurora DSQL 有助于您构建和维护在任何规模下始终可用的企业应用程序。主动-主动无服务器设计可自动执行故障恢复，因此您无需担心传统的数据库失效转移。您的应用程序受益于多可用区和多区域可用性，而且您不必担心最终一致性或与失效转移相关的数据丢失。

Aurora DSQL 中的主要功能

以下主要功能有助于您创建无服务器分布式数据库，以支持您的高可用性应用程序：

分布式架构

Aurora DSQL 由以下多租户组件组成：

- 中继和连接
- 计算和数据库
- 事务日志、并发控制和隔离
- 存储服务

控制面板协调上述的各个组件。每个组件均跨三个可用区（AZ）提供冗余，并可自动进行集群扩展和在组件出现故障时自我修复。要详细了解此架构如何支持高可用性，请参阅 Amazon Aurora DSQL 中的韧性。

单区域和多区域集群

Aurora DSQL 集群可提供以下优势：

- 同步数据复制
- 一致的读取操作
- 自动故障恢复
- 多个可用区或区域之间的数据一致性

如果基础设施组件出现故障，Aurora DSQL 会自动将请求路由到正常运行的基础设施，而无需手动干预。Aurora DSQL 通过强一致性、快照隔离、原子性以及跨可用区和跨区域持久性，提供了原子性、一致性、隔离性和持久性（ACID）事务。

多区域对等集群可提供与单区域集群相同的韧性和连接性。但是，它们通过提供两个区域端点（每个对等集群区域中各有一个端点）来提高可用性。对等集群的这两个端点都提供单个逻辑数据库。它们可用于并发读取和写入操作，并提供强数据一致性。您可以构建同时在多个区域中运行的应用程序来提高性能和韧性，并且知道读取器始终看到相同的数据。

与 PostgreSQL 的兼容性

Aurora DSQL 中的分布式数据库层（计算）基于 PostgreSQL 的当前主要版本。您可以使用熟悉的 PostgreSQL 驱动程序和工具（例如 `psql`）连接到 Aurora DSQL。Aurora DSQL 目前与 PostgreSQL 版本 16 兼容，并支持一系列广泛的 PostgreSQL 功能、表达式和数据类型。有关支持的 SQL 功能的更多信息，请参阅 Aurora DSQL 中的 SQL 功能兼容性。

Aurora DSQL 的区域可用性

借助 Amazon Aurora DSQL，您可以跨多个 AWS 区域部署数据库实例，以支持全球应用程序并满足数据驻留要求。区域可用性决定了您可以在何处创建和管理 Aurora DSQL 数据库集群。需要设计高度可用、全球分布式数据库系统的数据库管理员和应用程序架构师通常需要了解其工作负载的区域支持。常见用例包括设置跨区域灾难恢复、从地理位置较近的数据库实例为用户提供服务以减少延迟，以及在特定位置维护数据副本以实现合规性。

下表显示了 Aurora DSQL 当前可用的 AWS 区域以及每个 AWS 区域的端点。

区域名称	区域	端点	协议
美国东部（俄亥俄州）	us-east-2	dsql.us-east-2.api.aws	HTTPS
		dsql-fips.us-east-2.api.aws	HTTPS
美国东部（弗吉尼	us-east-1	dsql.us-east-1.api.aws	HTTPS

区域名称	区域	端点	协议
亚州北部）		dsql-fips.us-east-1.api.aws	HTTPS
美国西部（俄勒冈州）	us-west-2	dsql.us-west-2.api.aws dsql-fips.us-west-2.api.aws	HTTPS HTTPS
亚太地区（墨尔本）	ap-southeast-4	dsql.ap-southeast-4.api.aws	HTTPS
亚太地区（大阪）	ap-northeast-3	dsql.ap-northeast-3.api.aws	HTTPS
亚太地区（首尔）	ap-northeast-2	dsql.ap-northeast-2.api.aws	HTTPS
亚太地区（悉尼）	ap-southeast-2	dsql.ap-southeast-2.api.aws	HTTPS
亚太地区（东京）	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS
加拿大（中部）	ca-central-1	dsql.ca-central-1.api.aws dsql-fips.ca-central-1.api.aws	HTTPS HTTPS
加拿大西部（卡尔加里）	ca-west-1	dsql.ca-west-1.api.aws dsql-fips.ca-west-1.api.aws	HTTPS HTTPS
欧洲地区（法兰克福）	eu-central-1	dsql.eu-central-1.api.aws	HTTPS

区域名称	区域	端点	协议
欧洲地区（爱尔兰）	eu-west-1	dsql.eu-west-1.api.aws	HTTPS
欧洲地区（伦敦）	eu-west-2	dsql.eu-west-2.api.aws	HTTPS
欧洲地区（巴黎）	eu-west-3	dsql.eu-west-3.api.aws	HTTPS

Aurora DSQL 的多区域集群可用性

您可以在特定的 AWS 区域集中创建 Aurora DSQL 多区域集群。每个区域集对地理上相关的区域进行分组，而这些区域可以在多区域集群中协同工作。

美国区域

- 美国东部（弗吉尼亚州北部）
- 美国东部（俄亥俄州）
- 美国西部（俄勒冈州）

亚太区域

- 亚太地区（大阪）
- 亚太地区（首尔）
- 亚太地区（东京）

欧洲区域

- 欧洲地区（法兰克福）
- 欧洲地区（爱尔兰）
- 欧洲地区（伦敦）
- Europe (Paris)

重要限制

必须在单个区域集中创建多区域集群。例如，您无法创建一个同时包含美国东部（弗吉尼亚州北部）区域和欧洲地区（爱尔兰）区域的集群。

Important

Aurora DSQL 目前不支持跨洲多区域集群。

Aurora DSQL 的定价

有关费用信息，请参阅 Aurora DSQL pricing。

接下来做什么？

有关 Aurora DSQL 中核心组件的信息以及如何开始使用该服务，请参阅以下内容：

- Aurora DSQL 入门
- Aurora DSQL 中的 SQL 功能兼容性
- 使用兼容 PostgreSQL 的客户端访问 Aurora DSQL
- Aurora DSQL 和 PostgreSQL

Aurora DSQL 入门

Amazon Aurora DSQL 是一个针对事务性工作负载进行了优化的无服务器、完全托管式、分布式关系数据库。在以下各节中，您将了解如何创建单区域和多区域 Aurora DSQL 集群、连接到这些集群以及如何创建和加载示例架构。您将使用 AWS 管理控制台访问集群，并可选择使用其他 PostgreSQL 客户端与数据库进行交互。最后，您将设置好正常运行的 Aurora DSQL 集群，该集群可用于测试或生产工作负载。

主题

- 先决条件
- 步骤 1：创建 Aurora DSQL 单区域集群
- 步骤 2：连接到 Aurora DSQL 集群
- 步骤 3：在 Aurora DSQL 中运行示例 SQL 命令
- 步骤 4（可选）：创建多区域集群
- 问题排查

先决条件

在可以开始使用 Aurora DSQL 之前，确保满足以下先决条件：

- 您的 IAM 身份必须具有登录控制台的权限。
- 您的 IAM 身份必须满足以下条件：
  - 对 AWS 账户中的任何资源执行任何操作的访问权限
  - 已附加 `AmazonAuroraDSQLConsoleFullAccess` AWS 托管式策略。

步骤 1：创建 Aurora DSQL 单区域集群

Aurora DSQL 的基本单元是集群，您可以在其中存储数据。在本任务中，您将在单个 AWS 区域中创建集群。

在 Aurora DSQL 中创建单区域集群

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql。

2. 选择创建集群，然后选择单区域。
3. （可选）更改默认名称标签的值。
4. （可选）为此集群添加其他标签。
5. （可选）在集群设置中，选择以下任一选项：
   - 选择自定义加密设置（高级）以选择或创建 AWS KMS key。如果您使用客户自主管理型密钥，请确保密钥策略向 Aurora DSQL 授予所需的权限。有关更多信息，请参阅 客户托管密钥的密钥策略。
   - 选择启用删除保护以防止删除操作移除您的集群。默认情况下，删除保护功能处于选中状态。
   - 选择基于资源的策略（高级）可为此集群指定访问控制策略。
6. 选择创建集群。
7. 控制台将返回到集群页面。此时将显示一个通知横幅，指示正在创建集群。选择集群 ID 以打开集群详细信息视图。

步骤 2：连接到 Aurora DSQL 集群

Aurora DSQL 支持多种连接到集群的方式，包括 DSQL 查询编辑器、AWS CloudShell、本地 psql 客户端和其他兼容 PostgreSQL 的工具。在此步骤中，您将使用 Aurora DSQL 查询编辑器进行连接，可通过该编辑器快速开始与新集群进行交互。

使用查询编辑器进行连接

1. 在 Aurora DSQL 控制台（https://console.aws.amazon.com/dsql）中，打开集群页面，确认您的集群已创建完成且其状态为“活动”。
2. 从列表中选择您的集群，或选择集群 ID 以打开集群详细信息页面。
3. 选择使用查询编辑器进行连接。
4. 为刚刚创建的集群选择以管理员身份连接。
   - （可选）您可以使用自定义角色进行连接，请参阅使用数据库角色和 IAM 身份验证。

步骤 3：在 Aurora DSQL 中运行示例 SQL 命令

通过运行 SQL 语句测试 Aurora DSQL 集群。在查询编辑器中打开集群后，分步选择并运行每个示例查询。

在 Aurora DSQL 中运行示例 SQL 命令

1. 创建名为 `test` 的架构。

```
CREATE SCHEMA IF NOT EXISTS test;
```

2. 创建使用自动生成的 UUID 作为主键的 hello_world 表。

```
CREATE TABLE IF NOT EXISTS test.hello_world (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    message VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. 插入示例行。

```
INSERT INTO test.hello_world (message)
VALUES ('Hello, World!!');
```

4. 读取插入的值。

```
SELECT * FROM test.hello_world;
```

5. （可选）清除

```
DROP TABLE test.hello_world;
DROP SCHEMA test;
```

步骤 4（可选）：创建多区域集群

创建多区域集群时，您指定以下区域：

远程区域

这是您在其中创建第二个集群的区域。您在此区域中创建第二个集群，并使其与初始集群对等。Aurora DSQL 将初始集群上的所有写入内容复制到远程集群。您可以在任何集群上进行读取和写入。

见证区域

该区域接收写入多区域集群的所有数据。但是，见证区域不托管客户端端点，也不提供用户数据访问。在见证区域中维护着有限时段的加密事务日志。此日志有助于恢复，并在某个区域变为不可用时支持事务仲裁。

使用以下过程可创建初始集群，在不同的区域中创建第二个集群，然后使两个集群对等以创建多区域集群。它还演示了来自这两个区域端点的跨区域写入复制和一致读取。

创建多区域集群

1. 登录 Aurora DSQL 控制台。
2. 在导航窗格中，选择集群。
3. 选择创建集群，然后选择多区域。
4. （可选）更改默认名称标签的值。
5. （可选）为此集群添加其他标签。
6. 在多区域设置中，为初始集群选择以下选项：
   - 在见证区域中，选择一个区域。目前，多区域集群中的见证区域仅支持位于美国的区域。
   - （可选）在远程区域集群 ARN 中，输入另一个区域中现有集群的 ARN。如果不存在可用作多区域集群中第二个集群的集群，请在创建初始集群后完成设置。
7. （可选）在集群设置中，为初始集群选择以下任一选项：
   - 选择自定义加密设置（高级）以选择或创建 AWS KMS key。如果您使用客户自主管理型密钥，请确保密钥策略向 Aurora DSQL 授予所需的权限。有关更多信息，请参阅客户托管密钥的密钥策略。
   - 选择启用删除保护以防止删除操作移除您的集群。默认情况下，删除保护功能处于选中状态。
   - 选择基于资源的策略（高级）可为此集群指定访问控制策略。
8. 选择创建集群以创建初始集群。如果您在上一步中未输入 ARN，则控制台会显示集群设置待处理通知。
9. 在集群设置待处理通知中，选择完成多区域集群设置。此操作会启动在另一个区域中创建第二个集群。
10. 为第二个集群选择以下选项之一：
    - 添加远程区域集群 ARN：如果集群存在，并且您希望它成为多区域集群中的第二个集群，请选择此选项。

- 在其它区域中创建集群：选择此选项以创建第二个集群。在远程区域中，选择此第二个集群的区域。

11. 选择在 ***your-second-region*** 中创建集群，其中 *your-second-region* 是第二个集群的位置。控制台将在您的第二个区域中打开。
12. （可选）为第二个集群选择集群设置。例如，可以选择 AWS KMS key。如果您使用客户自主管理型密钥，请确保密钥策略向 Aurora DSQL 授予所需的权限。有关更多信息，请参阅 客户托管密钥的密钥策略。
13. 选择创建集群以创建第二个集群。
14. 选择在 ***initial-cluster-region*** 中对等，其中 *initial-cluster-region* 是托管您创建的第一个集群的区域。
15. 出现提示时，选择确认。此步骤将完成创建多区域集群。

连接到第二个集群

1. 打开 Aurora DSQL 控制台，然后选择第二个集群的区域。
2. 选择集群。
3. 选择多区域集群中的第二个集群所对应的行。
4. 选择使用查询编辑器进行连接。
5. 选择以管理员身份进行连接。
6. 按照步骤 3：在 Aurora DSQL 中运行示例 SQL 命令中的步骤操作来创建示例架构和表，并插入数据。

从托管初始集群的区域中的第二个集群查询数据

1. 在 Aurora DSQL 控制台中，选择初始集群的区域。
2. 选择集群。
3. 选择多区域集群中的第二个集群所对应的行。
4. 选择使用查询编辑器进行连接。
5. 选择以管理员身份进行连接。
6. 查询您插入到第二个集群的数据。

Example

```
SELECT * FROM test.hello_world;
```

问题排查

请参阅 Aurora DSQL 文档的故障排除部分。

Aurora DSQL 的身份验证和授权

Aurora DSQL 使用 IAM 角色和策略进行集群授权。可以将 IAM 角色与 PostgreSQL database roles 关联以进行数据库授权。这种方法将 IAM 中的优势与 PostgreSQL privileges 相结合。Aurora DSQL 使用这些功能为您的集群、数据库和数据提供全面的授权和访问策略。

使用 IAM 管理集群

要管理集群，请使用 IAM 进行身份验证和授权：

IAM 身份验证

要在管理 Aurora DSQL 集群时对 IAM 身份进行身份验证，必须使用 IAM。可以使用 AWS 管理控制台、AWS CLI 或 AWS SDK 提供身份验证。

IAM 授权

要管理 Aurora DSQL 集群，请使用 Aurora DSQL 的 IAM 操作授予授权。例如，要描述集群，请确保 IAM 身份拥有执行 IAM 操作 `dsql:GetCluster` 的权限，如以下示例策略操作所示。

```
{
  "Effect": "Allow",
  "Action": "dsql:GetCluster",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
}
```

有关更多信息，请参阅 使用 IAM 策略操作管理集群。

使用 IAM 连接到集群

要连接到集群，请使用 IAM 进行身份验证和授权：

IAM 身份验证

使用 IAM 身份（具有连接到集群的授权）生成临时身份验证令牌。要了解更多信息，请参阅在 Amazon Aurora DSQL 中生成身份验证令牌。

IAM 授权

向您用于与集群的端点建立连接的 IAM 身份授予执行以下 IAM 策略操作的权限：

- 如果您使用的是 admin 角色，请使用 dsql:DbConnectAdmin。Aurora DSQL 会为您创建和管理此角色。以下示例 IAM 策略操作支持 admin 连接到 *my-cluster*。

```
{
  "Effect": "Allow",
  "Action": "dsql:DbConnectAdmin",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
}
```

- 如果您使用的是自定义数据库角色，请使用 dsql:DbConnect。您可以通过在数据库中使用 SQL 命令来创建和管理此角色。以下示例 IAM 策略操作可让自定义数据库角色连接到 *my-cluster* 长达一小时。

```
{
  "Effect": "Allow",
  "Action": "dsql:DbConnect",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
}
```

建立连接后，您的角色可获得长达一小时的连接授权。

使用 PostgreSQL 数据库角色和 IAM 角色与数据库进行交互

PostgreSQL 使用角色的概念来管理数据库访问权限。根据设置角色的方式，可以将角色视为一个数据库用户或一组数据库用户。可以使用 SQL 命令创建 PostgreSQL 角色。要管理数据库级授权，请向 PostgreSQL 数据库角色授予 PostgreSQL 权限。

Aurora DSQL 支持两种类型的数据库角色：admin 角色和自定义角色。Aurora DSQL 会自动在 Aurora DSQL 集群中为您创建一个预定义的 admin 角色。您不能修改 admin 角色。当您以 admin 身份连接到数据库时，可以发出 SQL 来创建新的数据库级角色，以便与您的 IAM 角色关联。要让 IAM 角色连接到数据库，请将自定义数据库角色与 IAM 角色相关联。

身份验证

使用 admin 角色连接到集群。连接数据库后，使用命令 AWS IAM GRANT 将自定义数据库角色与获得授权可连接到集群的 IAM 身份相关联，如下例所示。

```
AWS IAM GRANT custom-db-role TO 'arn:aws:iam::account-id:role/iam-role-name';
```

要了解更多信息，请参阅授权数据库角色连接到集群。

Authorization*

使用 admin 角色连接到集群。运行 SQL 命令以设置自定义数据库角色并授予权限。要了解更多信息，请参阅 PostgreSQL 文档中的 PostgreSQL database roles 和 PostgreSQL privileges。

将 IAM 策略操作与 Aurora DSQL 结合使用

您使用的 IAM 策略操作取决于您用于连接到集群的角色：要么是 admin，要么是自定义数据库角色。该策略还取决于该角色所需的 IAM 操作。

使用 IAM 策略操作连接到集群

当您使用默认数据库角色 admin 连接到集群时，请使用具有授权的 IAM 身份来执行以下 IAM 策略操作。

```
"dsql:DbConnectAdmin"
```

当您使用自定义数据库角色连接到集群时，请先将 IAM 角色与数据库角色关联。您用于连接到集群的 IAM 身份必须具有执行以下 IAM 策略操作的授权。

```
"dsql:DbConnect"
```

要了解有关自定义数据库角色的更多信息，请参阅使用数据库角色和 IAM 身份验证。

使用 IAM 策略操作管理集群

当管理 Aurora DSQL 集群时，请仅为角色需要执行的操作指定策略操作。例如，如果您的角色只需要获取集群信息，则可以将角色权限限制为仅 GetCluster 和 ListClusters 权限，如以下示例策略所示

JSON

```
{
"Version":"2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
```

```
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
    }
  ]
}
```

以下示例策略显示了所有可用于管理集群的 IAM 策略操作。

JSON

```
{
"Version":"2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:UpdateCluster",
        "dsql:DeleteCluster",
        "dsql:ListClusters",
        "dsql:TagResource",
        "dsql:ListTagsForResource",
        "dsql:UntagResource"
      ],
      "Resource" : "*"
    }
  ]
}
```

使用 IAM 和 PostgreSQL 撤销授权

您可以撤销 IAM 角色用于访问数据库级角色的权限：

撤销管理员连接到集群的授权

要撤销使用 `admin` 角色连接到集群的授权，请撤销 IAM 身份对 `dsql:DbConnectAdmin` 的访问权限。编辑 IAM 策略或将策略与身份分离。

从 IAM 身份撤销连接授权后，Aurora DSQL 会拒绝来自该 IAM 身份的所有新的连接尝试。任何使用 IAM 身份的活跃连接都可能在连接的持续时间内保持授权状态。有关连接持续时间的更多信息，请参阅配额和限制。

撤销自定义角色连接到集群的授权

要撤销 `admin` 以外的数据库角色的访问权限，请撤销 IAM 身份对 `dsql:DbConnect` 的访问权限。编辑 IAM 策略或将策略与身份分离。

也可以在数据库中使用命令 `AWS IAM REVOKE` 来取消数据库角色和 IAM 之间的关联。要了解有关从数据库角色撤销访问权限的更多信息，请参阅从 IAM 角色撤销数据库授权。

您无法管理预定义 `admin` 数据库角色的权限。要了解如何管理自定义数据库角色的权限，请参阅 PostgreSQL privileges。对权限的修改将在 Aurora DSQL 成功提交修改事务后的下一个事务中生效。

在 Amazon Aurora DSQL 中生成身份验证令牌

要使用 SQL 客户端连接到 Amazon Aurora DSQL，请生成要用作密码的身份验证令牌。此令牌仅用于对连接进行身份验证。建立连接后，连接将保持有效，即使身份验证令牌过期也是如此。

如果您使用 AWS 管理控制台、AWS CLI 或 SDK 创建身份验证令牌，默认情况下，该令牌将在 15 分钟后自动过期。最大持续时间为 604800 秒，也就是一周。要再次从客户端连接到 Aurora DSQL，您可以使用相同的身份验证令牌（如果该令牌尚未过期），也可以生成一个新令牌。

要开始生成令牌，请创建 IAM 策略和 a cluster in Aurora DSQL。然后，使用 AWS 管理控制台、AWS CLI 或 AWS SDK 来生成令牌。

您必须至少拥有使用 IAM 连接到集群中列出的 IAM 权限，具体取决于您使用哪个数据库角色进行连接。

主题

- 使用 AWS 管理控制台在 Aurora DSQL 中生成身份验证令牌
- 使用 AWS CloudShell 在 Aurora DSQL 中生成身份验证令牌
- 使用 AWS CLI 在 Aurora DSQL 中生成身份验证令牌
- 使用 SDK 在 Aurora DSQL 中生成令牌

使用 AWS 管理控制台在 Aurora DSQL 中生成身份验证令牌

Aurora DSQL 使用令牌而不是密码来对用户进行身份验证。您可以从控制台生成令牌。

生成身份验证令牌

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql。
2. 选择您要为其生成身份验证令牌的集群的集群 ID。如果您尚未创建集群，请按照步骤 1：创建 Aurora DSQL 单区域集群或步骤 4（可选）：创建多区域集群中的步骤操作。
3. 选择连接，然后选择获取令牌。
4. 选择是要以 `admin` 身份还是要使用自定义数据库角色进行连接。
5. 复制生成的身份验证令牌并将其用于使用 SQL 客户端访问 Aurora DSQL。

要了解有关 Aurora DSQL 中的自定义数据库角色和 IAM 的更多信息，请参阅身份验证和授权。

使用 AWS CloudShell 在 Aurora DSQL 中生成身份验证令牌

在使用 AWS CloudShell 生成身份验证令牌之前，请确保您创建 Aurora DSQL 集群。

使用 AWS CloudShell 生成身份验证令牌

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql。
2. 在 AWS 管理控制台的左下角，选择 AWS CloudShell。
3. 运行以下命令以生成 `admin` 角色的身份验证令牌。将 *`us-east-1`* 替换为您的区域，并将 *`your_cluster_endpoint`* 替换为您自己的集群的端点。

   > **Note**
   >
   > 如果您没有以 `admin` 身份进行连接，请改用 `generate-db-connect-auth-token`。

   ```
   aws dsql generate-db-connect-admin-auth-token \
     --expires-in 3600 \
     --region us-east-1 \
     --hostname your_cluster_endpoint
   ```

   如果您遇到问题，请参阅排查 IAM 问题和如何使用 IAM 策略解决“访问被拒绝”或“未经授权的操作”错误？
4. 使用以下命令，通过 `psql` 开始与集群的连接。

```
PGSSLMODE=require \
psql --dbname postgres \
  --username admin \
  --host cluster_endpoint
```

5. 您应该会看到要求您提供密码的提示符。复制您生成的令牌，并确保不包含任何额外的空格或字符。将其从 psql 粘贴到以下提示符下。

```
Password for user admin:
```

6. 按 Enter 键。您应该看到 PostgreSQL 提示符。

```
postgres=>
```

如果您收到拒绝访问错误，请确保您的 IAM 身份具有 dsql:DbConnectAdmin 权限。如果您拥有此权限，但继续收到拒绝访问错误，请参阅排查 IAM 问题和如何使用 IAM 策略解决“访问被拒绝”或“未经授权的操作”错误？

要了解有关 Aurora DSQL 中的自定义数据库角色和 IAM 的更多信息，请参阅身份验证和授权。

使用 AWS CLI 在 Aurora DSQL 中生成身份验证令牌

当集群处于 ACTIVE 状态时，可以在 CLI 上使用 aws dsql 命令来生成身份验证令牌。使用以下任一方法：

> **Note**
>
> 令牌生成是一项本地操作，它使用您当前的 IAM 凭证对请求进行签名。它不会联系 AWS 以验证凭证。如果您的凭证已过期或无效，令牌生成仍会成功，但连接尝试会失败。在生成令牌之前，请确保您的 IAM 凭证有效。

- 如果您要使用 admin 角色进行连接，请使用 generate-db-connect-admin-auth-token 选项。
- 如果您要使用自定义数据库角色进行连接，请使用 generate-db-connect-auth-token 选项。

下面的示例使用以下属性为 admin 角色生成身份验证令牌。

- *your_cluster_endpoint*：集群的端点 它遵循格式 *your_cluster_identifier*.dsql.*region*.on.aws，如示例 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws 所示。
- *region*：AWS 区域，例如 us-east-2 或 us-east-1。

以下示例将令牌的到期时间设置为 3600 秒（1 小时）。

Linux and macOS

```
aws dsql generate-db-connect-admin-auth-token \
  --region region \
  --expires-in 3600 \
  --hostname your_cluster_endpoint
```

Windows

```
aws dsql generate-db-connect-admin-auth-token ^
  --region=region ^
  --expires-in=3600 ^
  --hostname=your_cluster_endpoint
```

使用 SDK 在 Aurora DSQL 中生成令牌

当集群处于 ACTIVE 状态时，您可以为其生成身份验证令牌。SDK 示例使用以下属性为 admin 角色生成身份验证令牌：

- *your_cluster_endpoint*（或 *yourClusterEndpoint*）：Aurora DSQL 集群的端点。命名格式为 *your_cluster_identifier*.dsql.*region*.on.aws，如示例 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws 中所示。
- *region*（*RegionEndpoint*）：您的集群所在的 AWS 区域，例如 us-east-2 或 us-east-1。

Python SDK

Tip

AWS 建议使用适用于 Python 的 Aurora DSQL 连接器，它会自动处理令牌生成。

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 generate_db_connect_admin_auth_token。
- 如果您要使用自定义数据库角色进行连接，请使用 generate_connect_auth_token。

```
import boto3

def generate_token(your_cluster_endpoint, region):
    client = boto3.client("dsql", region_name=region)
    # use `generate_db_connect_auth_token` instead if you are not connecting as
 admin.
    token = client.generate_db_connect_admin_auth_token(your_cluster_endpoint,
 region)
    print(token)
    return token
```

C++ SDK

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 GenerateDBConnectAdminAuthToken。
- 如果您要使用自定义数据库角色进行连接，请使用 GenerateDBConnectAuthToken。

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;

std::string generateToken(String yourClusterEndpoint, String region) {
    DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQLClient client{clientConfig};
    std::string token = "";

    // If you are not using the admin role to connect, use
 GenerateDBConnectAuthToken instead
    const auto presignedString =
 client.GenerateDBConnectAdminAuthToken(yourClusterEndpoint, region);
    if (presignedString.IsSuccess()) {
        token = presignedString.GetResult();
    } else {
        std::cerr << "Token generation failed." << std::endl;
    }

    std::cout << token << std::endl;
    return token;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    // Replace with your cluster endpoint and region
    std::string token = generateToken("your_cluster_endpoint.dsql.us-east-1.on.aws",
 "us-east-1");
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript SDK

Tip

AWS 建议使用适用于 Node.js 的 Aurora DSQL 连接器，它会自动处理令牌生成。

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 getDbConnectAdminAuthToken。
- 如果您要使用自定义数据库角色进行连接，请使用 getDbConnectAuthToken。

```
import { DsqlSigner } from "@aws-sdk/dsql-signer";

async function generateToken(yourClusterEndpoint, region) {
  const signer = new DsqlSigner({
    hostname: yourClusterEndpoint,
    region,
  });
  try {
    // Use `getDbConnectAuthToken` if you are _not_ logging in as the `admin` user
    const token = await signer.getDbConnectAdminAuthToken();
    console.log(token);
    return token;
  } catch (error) {
      console.error("Failed to generate token: ", error);
      throw error;
  }
}
```

Java SDK

> Tip
>
> AWS 建议使用使用 JDBC 连接器连接到 Aurora DSQL 集群，它会自动处理令牌生成。

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 generateDbConnectAdminAuthToken。
- 如果您要使用自定义数据库角色进行连接，请使用 generateDbConnectAuthToken。

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.services.dsql.DsqlUtilities;
import software.amazon.awssdk.regions.Region;

public class GenerateAuthToken {
    public static String generateToken(String yourClusterEndpoint, Region region) {
        DsqlUtilities utilities = DsqlUtilities.builder()
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.builder().build())
                .build();

        // Use `generateDbConnectAuthToken` if you are _not_ logging in as `admin`
 user
        String token = utilities.generateDbConnectAdminAuthToken(builder -> {
            builder.hostname(yourClusterEndpoint)
                    .region(region);
        });

        System.out.println(token);
        return token;
    }
}
```

Rust SDK

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 db_connect_admin_auth_token。
- 如果您要使用自定义数据库角色进行连接，请使用 db_connect_auth_token。

```
use aws_config::{BehaviorVersion, Region};
use aws_sdk_dsql::auth_token::{AuthTokenGenerator, Config};

async fn generate_token(your_cluster_endpoint: String, region: String) -> String {
    let sdk_config = aws_config::load_defaults(BehaviorVersion::latest()).await;
    let signer = AuthTokenGenerator::new(
        Config::builder()
            .hostname(&your_cluster_endpoint)
            .region(Region::new(region))
            .build()
            .unwrap(),
    );

    // Use `db_connect_auth_token` if you are _not_ logging in as `admin` user
    let token = signer.db_connect_admin_auth_token(&sdk_config).await.unwrap();
    println!("{}", token);
    token.to_string()
}
```

Ruby SDK

> **Tip**
>
> AWS 建议使用使用 Ruby 连接器连接到 Aurora DSQL 集群，它会自动处理令牌生成。

您可以通过以下方式生成令牌：

- 如果您要使用 admin 角色进行连接，请使用 generate_db_connect_admin_auth_token。
- 如果您要使用自定义数据库角色进行连接，请使用 generate_db_connect_auth_token。

```
require 'aws-sdk-dsql'

def generate_token(your_cluster_endpoint, region)
  credentials = Aws::CredentialProviderChain.new.resolve

  token_generator = Aws::DSQL::AuthTokenGenerator.new({
    :credentials => credentials
  })

  # if you're not using admin role, use generate_db_connect_auth_token instead
```

```
  token = token_generator.generate_db_connect_admin_auth_token({
    :endpoint => your_cluster_endpoint,
    :region => region
  })
end
```

PHP SDK

您可以通过以下方式生成令牌：

- 如果您要使用 `admin` 角色进行连接，请使用 `generateDbConnectAdminAuthToken`。
- 如果您要使用自定义数据库角色进行连接，请使用 `generateDbConnectAuthToken`。

```
<?php
require 'vendor/autoload.php';

use Aws\DSQL\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

function generateToken(string $yourClusterEndpoint, string $region): string {
    $provider = CredentialProvider::defaultProvider();
    $generator = new AuthTokenGenerator($provider);

    // Use generateDbConnectAuthToken if you are not connecting as admin
    $token = $generator->generateDbConnectAdminAuthToken($yourClusterEndpoint,
 $region);

    echo $token . PHP_EOL;
    return $token;
}
```

.NET

> **Tip**
>
> AWS 建议使用[使用 .NET 连接器连接到 Aurora DSQL 集群]，它会自动处理令牌生成。

官方适用于 .NET 的 SDK 不包括用于为 Aurora DSQL 生成身份验证令牌的内置 API 调用。而是必须使用 `DSQLAuthTokenGenerator`，这是一个实用程序类。以下代码示例展示了如何为 .NET 生成身份验证令牌。

您可以通过以下方式生成令牌：

- 如果您要使用 `admin` 角色进行连接，请使用 `DbConnectAdmin`。
- 如果您要使用自定义数据库角色进行连接，请使用 `DbConnect`。

以下示例使用 `DSQLAuthTokenGenerator` 实用程序类为具有 `admin` 角色的用户生成身份验证令牌。将 *`insert-dsql-cluster-endpoint`* 替换为您的集群端点。

```
using Amazon;
using Amazon.DSQL.Util;

var yourClusterEndpoint = "insert-dsql-cluster-endpoint";

// Use `DSQLAuthTokenGenerator.GenerateDbConnectAuthToken` if you are _not_ logging
 in as `admin` user
var token =
 DSQLAuthTokenGenerator.GenerateDbConnectAdminAuthToken(RegionEndpoint.USEast1,
 yourClusterEndpoint);

Console.WriteLine(token);
```

Go

AWS 建议使用[使用 Go 连接器连接到 Aurora DSQL 集群]，它会自动处理令牌生成。

适用于 Go 的 AWS SDK v2 提供了一种内置方法，用于在 `github.com/aws/aws-sdk-go-v2/feature/dsql/auth` 软件包中生成身份验证令牌。

- 如果您要使用 `admin` 角色进行连接，请使用 `auth.GenerateDBConnectAdminAuthToken`。

- 如果您要使用自定义数据库角色进行连接，请使用 `auth.GenerateDbConnectAuthToken`。

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dsql/auth"
)

func main() {
 ctx := context.Background()

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion("region"))
 if err != nil {
  panic(err)
 }

 // Use auth.GenerateDbConnectAuthToken for non-admin users
 token, err := auth.GenerateDBConnectAdminAuthToken(ctx, "yourClusterEndpoint",
 "region", cfg.Credentials)
 if err != nil {
  panic(err)
 }

 fmt.Println(token)
}
```

使用数据库角色和 IAM 身份验证

Aurora DSQL 支持使用 IAM 角色和 IAM 用户进行身份验证。您可以使用任一方法来对 Aurora DSQL 数据库进行身份验证和访问。

IAM 角色

IAM 角色是您的 AWS 账户中具有特定权限但不与特定人员关联的身份。使用 IAM 角色可提供临时安全凭证。您可以通过多种方式临时代入 IAM 角色：

- 通过在 AWS 管理控制台中切换角色
- 通过调用 AWS CLI 或 AWS API 操作
- 通过使用自定义 URL

代入角色后，可以使用该角色的临时凭证访问 Aurora DSQL。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的 IAM 身份。

IAM 用户

IAM 用户是您 AWS 账户内的一个身份，该身份对单个人员或应用程序具有特定权限或与单个人员或应用程序关联。IAM 用户拥有可用于访问 Aurora DSQL 的长期凭证，如密码和访问密钥。

> **Note**
>
> 要通过 IAM 身份验证来运行 SQL 命令，您可以在下面的示例中使用 IAM 角色 ARN 或 IAM 用户 ARN。

授权数据库角色连接到集群

创建 IAM 角色并使用 IAM 策略操作 `dsql:DbConnect` 授予连接授权。

IAM 策略还必须授予访问集群资源的权限。使用通配符 (`*`) 或按照将 IAM 条件键与 Amazon Aurora DSQL 结合使用中的说明操作。

授权数据库角色在数据库中使用 SQL

您必须使用具有授权的 IAM 角色才能连接到集群。

1. 使用 SQL 实用程序连接到 Aurora DSQL 集群。

   使用具有 IAM 身份（有权执行 IAM 操作 `dsql:DbConnectAdmin`）的 `admin` 数据库角色连接到集群。

2. 创建新的数据库角色，确保指定 `WITH LOGIN` 选项。

   ```
   CREATE ROLE example WITH LOGIN;
   ```

3. 将该数据库角色与 IAM 角色 ARN 关联。

```
AWS IAM GRANT example TO 'arn:aws:iam::012345678912:role/example';
```

4. 向数据库角色授予数据库级权限

   以下示例使用 GRANT 命令在数据库中提供授权。

```
GRANT USAGE ON SCHEMA myschema TO example;
GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA myschema TO example;
```

有关更多信息，请参阅 PostgreSQL 文档中的 PostgreSQL GRANT 和 PostgreSQL Privileges。

查看 IAM 到数据库的角色映射

要查看 IAM 角色与数据库角色之间的映射，请查询 `sys.iam_pg_role_mappings` 系统表。

```
SELECT * FROM sys.iam_pg_role_mappings;
```

输出示例：

```
 iam_oid |                   arn                    | pg_role_oid | pg_role_name |
 grantor_pg_role_oid | grantor_pg_role_name
---------+-----------------------------------------+-------------+-------------
+---------------------+---------------------
   26398 | arn:aws:iam::012345678912:role/example |       26396 | example      |
       15579 | admin
(1 row)
```

此表显示了 IAM 角色（由其 ARN 标识）和 PostgreSQL 数据库角色之间的所有映射。

从 IAM 角色撤销数据库授权

要撤销数据库授权，请使用 `AWS IAM REVOKE` 操作。

```
AWS IAM REVOKE example FROM 'arn:aws:iam::012345678912:role/example';
```

要了解有关撤销授权的更多信息，请参阅使用 IAM 和 PostgreSQL 撤销授权。

Aurora DSQL 和 PostgreSQL

Aurora DSQL 是与 PostgreSQL 兼容的分布式关系数据库，专为事务性工作负载而设计。Aurora DSQL 使用核心 PostgreSQL 组件，例如解析器、规划器、优化器和类型系统。

Aurora DSQL 的设计可确保所有受支持的 PostgreSQL 语法都提供兼容的行为并生成完全相同的查询结果。例如，Aurora DSQL 提供与 PostgreSQL 完全相同的类型转换、算术运算以及数值精度和小数位数。任何偏差都记录在案。

Aurora DSQL 还引入了高级功能，例如乐观并发控制和分布式架构管理。借助这些功能，您可以利用 PostgreSQL 的熟悉的工具，同时受益于现代、云原生、分布式应用程序的性能和可扩展性。

PostgreSQL 兼容性亮点

Aurora DSQL 目前基于 PostgreSQL 版本 16。关键亮点包括以下各项：

线路协议

Aurora DSQL 使用标准 PostgreSQL v3 线路协议。这样就可以与标准 PostgreSQL 客户端、驱动程序和工具集成。例如，Aurora DSQL 与 `psql`、`pgjdbc` 和 `psycopg` 兼容。

SQL 语法

Aurora DSQL 支持事务性工作负载中常用的各种标准 PostgreSQL 表达式和函数。支持的 SQL 表达式与 PostgreSQL 生成完全相同的结果，包括以下各项：

- 空值的处理
- 排序顺序行为
- 数值运算的小数位数和精度
- 字符串操作的等效性

有关更多信息，请参阅 Aurora DSQL 中的 SQL 功能兼容性。

事务管理

Aurora DSQL 保留了 PostgreSQL 的主要特征，例如 ACID 事务和等同于 PostgreSQL 可重复读取的隔离级别。有关更多信息，请参阅 Aurora DSQL 中的并发控制。

分布式架构优势

Aurora DSQL 的分布式、无共享设计提供了超越传统单节点数据库的性能和可扩展性优势。主要功能包括以下各项：

乐观并发控制（OCC）

Aurora DSQL 使用乐观并发控制模型。这种无锁方法可防止事务相互阻塞，消除死锁，并支持高吞吐量的并行执行。这些功能使得 Aurora DSQL 对于需要大规模一致性能的应用程序特别有价值。有关更多示例，请参阅 Aurora DSQL 中的并发控制。

异步 DDL 操作

Aurora DSQL 异步运行 DDL 操作，从而支持在架构更改期间不间断地读取和写入。其分布式架构可让 Aurora DSQL 执行以下操作：

- 将 DDL 操作作为后台任务运行，从而最大限度地减少中断。
- 将目录更改协调为强一致性分布式事务。这可以确保跨所有节点的原子可见性，即使在故障或并发操作期间也是如此。
- 跨多个可用区以完全分布式、无中心节点的方式运行，且计算层和存储层已分离。

有关在 PostgreSQL 中使用 EXPLAIN 命令的更多信息，请参阅 Aurora DSQL 中的 DDL 和分布式事务。

Aurora DSQL 中的 SQL 功能兼容性

在以下各节中，了解 Aurora DSQL 对 PostgreSQL 数据类型和 SQL 命令的支持。

主题

- Aurora DSQL 中支持的数据类型
- Aurora DSQL 支持的 SQL
- Aurora DSQL 中支持的 SQL 命令子集
- 从 PostgreSQL 迁移到 Aurora DSQL

Aurora DSQL 中支持的数据类型

Aurora DSQL 支持常用 PostgreSQL 类型的子集。

主题

- 数值数据类型
- 字符数据类型
- 日期和时间数据类型
- 其它数据类型
- 查询运行时数据类型

数值数据类型

Aurora DSQL 支持以下 PostgreSQL 数值数据类型。

名称	别名	范围和精度	存储大小	索引支持
smallint	int2	-32768 到 +32767	2 字节	是
integer	int, int4	-2147483648 到 +2147483647	4 字节	是
bigint	int8	-9223372036854775808 到 +9223372036854775807	8 字节	是
real	float4	6 位十进制精度	4 字节	是
double precision	float8	15 位十进制精度	8 字节	是
numeric [(*p*, *s*)]	decimal [(*p*, *s*)] dec[(*p*,*s*)]	可选择的精度的精确数字。最大精度为 38，最大小数位数为 37。[1] 默认值为 numeric (18,6)。	8 字节 + 每个精度位 2 字节。最大大小为 27 字节。	是

[1]：如果您在运行 CREATE TABLE 或 ALTER TABLE ADD COLUMN 时没有显式指定大小，Aurora DSQL 会强制使用默认值。Aurora DSQL 在您运行 INSERT 或 UPDATE 语句时会应用限制。

字符数据类型

Aurora DSQL 支持以下 PostgreSQL 字符数据类型。

名称	别名	说明	Aurora DSQL 限制	存储大小	索引支持
character [(*n*)]	char [(*n*)]	固定长度字符串	4096 字节[1]	可变，最大可达 4100 字节	是
character varying [(*n*)]	varchar [(*n*)]	长度可变的字符串	65535 字节[1]	可变，最大可达 65539 字节	是
bpchar [(*n*)]		如果长度固定，则这是 char 的别名。如果长度可变，则这是 varchar 的别名，其中尾随空格在语义上微不足道。	4096 字节[1]	可变，最大可达 4100 字节	是
text		长度可变的字符串	1 MiB[1]	可变，最大可达 1 MiB	是

1：如果您在运行 CREATE TABLE 或 ALTER TABLE ADD COLUMN 时没有显式指定大小，则 Aurora DSQL 会强制使用默认值。Aurora DSQL 在您运行 INSERT 或 UPDATE 语句时会应用限制。

日期和时间数据类型

Aurora DSQL 支持以下 PostgreSQL 日期和时间数据类型。

名称	别名	说明	Range	解析	存储大小	索引支持
date		日历日期（年、月、日）	4713 BC – 5874897 AD	1 天	4 字节	是

名称	别名	说明	Range	解析	存储大小	索引支持
time [(*p*)] [without time zone]	times	一天中的时间，不包括时区	0 – 1	1 微秒	8 字节	是
time [(*p*)] with time zone	timet	一天中的时间，包括时区	00:00:00+1559 – 24:00:00 –1559	1 微秒	12 字节	否
timestamp [(*p*)] [without time zone]		日期和时间，不包括时区	4713 BC – 294276 AD	1 微秒	8 字节	是
timestamp [(*p*)] with time zone	times tz	日期和时间，包括时区	4713 BC – 294276 AD	1 微秒	8 字节	是
interval [fields] [(*p*)]		时间跨度	-178000000 年 – 178000000 年	1 微秒	16 字节	否

其它数据类型

Aurora DSQL 支持以下其它 PostgreSQL 数据类型。

名称	别名	说明	Aurora DSQL 限制	存储大小	索引支持
boolean	bool	逻辑布尔值 (true/false)		1 字节	是
bytea		二进制数据（“字节数组”）	1 MiB[1]	可变，最大可达 1 MiB 限制	否

名称	别名	说明	Aurora DSQL 限制	存储大小	索引支持
UUID		通用唯一标识符		16 字节	是

1：如果您在运行 CREATE TABLE 或 ALTER TABLE ADD COLUMN 时没有显式指定大小，则 Aurora DSQL 会强制使用默认值。Aurora DSQL 在您运行 INSERT 或 UPDATE 语句时会应用限制。

查询运行时数据类型

查询运行时数据类型是在查询执行时使用的内部数据类型。这些类型不同于您在架构中定义的 PostgreSQL 兼容类型，如 varchar 和 integer。相反，这些类型是 Aurora DSQL 在处理查询时使用的运行时表示形式。

仅在查询运行时期间才支持以下数据类型：

数组类型

Aurora DSQL 支持所支持数据类型的数组。例如，您可能具有一个整数数组。函数 string_to_array 使用逗号分隔符 (,) 将字符串拆分为 PostgreSQL 样式的数组，如以下示例所示。在查询执行期间，可以在表达式、函数输出或临时计算中使用数组。

```
SELECT string_to_array('1,2', ',');
```

该函数返回类似于以下内容的响应：

```
 string_to_array
-----------------
 {1,2}
(1 row)
```

inet 类型

此数据类型表示 IPv4、IPv6 主机地址及其子网。此类型在解析日志、根据 IP 子网进行筛选或在查询中进行网络计算时很有用。有关更多信息，请参阅 PostgreSQL 文档中的 inet。

JSON 运行时函数

Aurora DSQL 支持将 JSON 和 JSONB 作为运行时数据类型来进行查询处理。将 JSON 数据存储为 `text` 列，然后在查询执行期间强制转换为 JSON，以便使用 PostgreSQL JSON 函数和运算符。

Aurora DSQL 支持 [section 9.1.6 JSON Functions and Operators](https://www.postgresql.org/docs/16/functions-json.html) 中的大多数 PostgreSQL 函数，并且具有相同的行为。

返回 JSON 或 JSONB 类型的函数可能需要进一步转换为 `text` 才能正确显示。

```
SELECT json_build_array(1, 2, 'foo', 4, 5)::text;
```

该函数返回类似于以下内容的响应：

```
    json_build_array
 ---------------------
   [1, 2, "foo", 4, 5]
 (1 row)
```

Aurora DSQL 支持的 SQL

Aurora DSQL 支持各种核心 PostgreSQL SQL 功能。在以下各节中，您可以了解有关 PostgreSQL 表达式的一般支持。此列表并不详尽。

SELECT 命令

Aurora DSQL 支持 `SELECT` 命令的以下子句。

主要子句	支持的子句
`FROM`	
`GROUP BY`	`ALL`, `DISTINCT`
`ORDER BY`	`ASC`, `DESC`, `NULLS`
`LIMIT`	
`DISTINCT`	

主要子句	支持的子句
HAVING	
USING	
WITH（公用表表达式）	
INNER JOIN	ON
OUTER JOIN	LEFT, RIGHT, FULL, ON
CROSS JOIN	ON
UNION	ALL
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR UPDATE	

数据定义语言 (DDL)

Aurora DSQL 支持以下 PostgreSQL DDL 命令。

命令	主要子句	支持的子句
CREATE	TABLE	有关 CREATE TABLE 命令支持的语法的信息，请参阅 CREATE TABLE。
ALTER	TABLE	有关 ALTER TABLE 命令支持的语法的信息，请参阅 ALTER TABLE。
DROP	TABLE	
CREATE	[UNIQUE] INDEX ASYNC	您可以将此命令与以下参数结合使用：ON、NULLS FIRST、NULLS LAST。

命令	主要子句	支持的子句
		有关 CREATE INDEX ASYNC 命令支持的语法的信息，请参阅 Aurora DSQL 中的异步索引。
DROP	INDEX	
CREATE	VIEW	有关 CREATE VIEW 命令支持的语法的更多信息，请参阅 CREATE VIEW。
ALTER	VIEW	有关 ALTER VIEW 命令支持的语法的信息，请参阅 ALTER VIEW。
DROP	VIEW	有关 DROP VIEW 命令支持的语法的信息，请参阅 DROP VIEW。
CREATE	SEQUENCE	有关 CREATE SEQUENCE 命令支持的语法的信息，请参阅 CREATE SEQUENCE。
ALTER	SEQUENCE	有关 ALTER SEQUENCE 命令支持的语法的信息，请参阅 ALTER SEQUENCE。
DROP	SEQUENCE	有关 DROP SEQUENCE 命令支持的语法的信息，请参阅 DROP SEQUENCE。
CREATE	ROLE, WITH	
CREATE	FUNCTION	LANGUAGE SQL
CREATE	DOMAIN	

数据操作语言 (DML)

Aurora DSQL 支持以下 PostgreSQL DML 命令。

命令	主要子句	支持的子句
INSERT	INTO	VALUES

命令	主要子句	支持的子句
		SELECT
UPDATE	SET	WHERE (SELECT) FROM, WITH
DELETE	FROM	USING, WHERE

数据控制语言（DCL）

Aurora DSQL 支持以下 PostgreSQL DCL 命令。

命令	支持的子句
GRANT	ON, TO
REVOKE	ON, FROM, CASCADE, RESTRICT

事务控制语言（TCL）

Aurora DSQL 支持以下 PostgreSQL TCL 命令。

命令	支持的子句	别名
COMMIT	[WORK | TRANSACTION] [AND NO CHAIN]	END
BEGIN	[WORK | TRANSACTION] [ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
START TRANSACTION	[ISOLATION LEVEL REPEATABLE READ]	

命令	支持的子句	别名
	[READ WRITE | READ ONLY]	
ROLLBACK	[WORK | TRANSACTION] [AND NO CHAIN]	ABORT

实用程序命令

Aurora DSQL 支持以下 PostgreSQL 实用程序命令：

- EXPLAIN
- ANALYZE（仅限关系名称）

Aurora DSQL 中支持的 SQL 命令子集

本节提供有关支持的 SQL 命令的详细信息，重点介绍具有大量参数集和子命令的命令。例如，PostgreSQL 中的 CREATE TABLE 提供了许多子句和参数，Aurora DSQL 支持其中的一个子集。本节介绍使用 Aurora DSQL 支持的熟悉的 PostgreSQL 语法元素支持的常见 SQL 命令的子集。

主题

- CREATE TABLE
- ALTER TABLE
- CREATE SEQUENCE
- ALTER SEQUENCE
- DROP SEQUENCE
- CREATE VIEW
- ALTER VIEW
- DROP VIEW

CREATE TABLE

CREATE TABLE 定义一个新表。

```
CREATE TABLE [ IF NOT EXISTS ] table_name ( [
```

```
  { column_name data_type [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression )|
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY ( sequence_options ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] index_parameters |
  PRIMARY KEY index_parameters |

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |

and like_option is:

{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | GENERATED | IDENTITY |
 INDEXES | STATISTICS | ALL }

index_parameters in UNIQUE, and PRIMARY KEY constraints are:
[ INCLUDE ( column_name [, ... ] ) ]
```

标识列

> **Note**
>
> 使用标识列时，应谨慎考虑缓存值。有关更多信息，请参阅 CREATE SEQUENCE 页面上的“重要提示”标注。
> 有关如何根据工作负载模式以最佳方式使用标识列的指导，请参阅使用序列和标识列。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY (*sequence_options*) 子句将列创建为标识列。它将附加一个隐式序列，在新插入的行中，该列将自动具有序列中分配给它的值。这样的列隐式为 NOT NULL。

子句 ALWAYS 和 BY DEFAULT 决定了在 INSERT 和 UPDATE 命令中如何显式处理用户指定的值。

在 INSERT 命令中，如果选择了 ALWAYS，则只有在 INSERT 语句指定 OVERRIDING SYSTEM VALUE 时才接受用户指定的值。如果选择了 BY DEFAULT，则优先使用用户指定的值。

在 UPDATE 命令中，如果选择 ALWAYS，则将列更新为除 DEFAULT 以外的任何值都将遭到拒绝。如果选择 BY DEFAULT，则可以正常更新该列。（UPDATE 命令没有 OVERRIDING 子句。）

sequence_options 子句可用于覆盖序列的参数。可用的选项包括为 CREATE SEQUENCE 显示的选项，另加 SEQUENCE NAME *name*。如果没有 SEQUENCE NAME，则系统会为序列选择一个未使用的名称。

ALTER TABLE

ALTER TABLE 更改表的定义。

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema

where action is one of:

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET
 sequence_option | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

标识列操作

SET GENERATED { ALWAYS | BY DEFAULT } / SET *sequence_option* / RESTART

这些格式更改列是否为标识列，或更改现有标识列的生成属性。有关详细信息，请参阅 CREATE TABLE。比如 SET DEFAULT，这些格式只影响后续 INSERT 和 UPDATE 命令的行为；它们不会导致表中已有的行发生变化。

sequence_option 是 ALTER SEQUENCE（如 INCREMENT BY）支持的选项。这些格式更改现有标识列所基于的序列。

DROP IDENTITY [IF EXISTS]

此格式从列中移除标识属性。如果指定了 DROP IDENTITY IF EXISTS 并且该列不是标识列，则不会引发任何错误。在这种情况下，将改为发出通知。

CREATE SEQUENCE

CREATE SEQUENCE：定义新的序列生成器。

> **Important**
>
> 在 PostgreSQL 中，指定 CACHE 是可选的，默认为 1。在 Amazon Aurora DSQL 等分布式系统中，序列操作涉及协调，在高并发情况下，缓存大小为 1 可能会增加协调开销。虽然较大的缓存值支持从本地预分配的范围中提供序列号，从而提高吞吐量，但未使用的保留值可能会丢失，从而使间隙和排序效果更加明显。由于应用程序对分配顺序与吞吐量的敏感度不同，因此 Amazon Aurora DSQL 要求显式指定 CACHE 并且目前支持 CACHE = 1 或 CACHE >= 65536，并明确区分和严格顺序生成更密切的分配行为与针对高度并发工作负载进行优化的分配。
>
> 当使用 CACHE >= 65536 时，序列值仍能保证唯一性，但在不同会话间可能不会按严格的递增顺序生成，并且可能会出现间隙，尤其是在缓存的值未被充分使用的情况下。这些特征与在并发使用情况下缓存序列的 PostgreSQL 语义一致，其中两个系统都保证不同的值，但不保证在会话间采用严格顺序排序。
>
> 在单个客户端会话中，序列值可能并非始终严格增加，尤其是在显式事务之外。此行为类似于使用连接池的 PostgreSQL 部署。通过使用 CACHE = 1 或在显式事务内获取序列值，可以实现更接近单会话 PostgreSQL 环境的分配行为。
>
> 使用 CACHE = 1，序列分配遵循 PostgreSQL 的非缓存序列行为。
>
> 有关如何根据工作负载模式以最佳方式使用序列的指导，请参阅使用序列和标识列。

支持的语法

```
CREATE SEQUENCE [ IF NOT EXISTS ] name CACHE cache
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ [ NO ] CYCLE ]
    [ START [ WITH ] start ]
    [ OWNED BY { table_name.column_name | NONE } ]

where data_type is BIGINT
      and cache = 1 or cache >= 65536
```

说明

`CREATE SEQUENCE` 创建新的序列号生成器。这涉及创建并初始化一个新的名为 *name* 的特殊单行表。生成器将归发出命令的用户所有。

如果提供了架构名称，则在指定的架构中创建序列。否则，将在当前架构中创建序列。序列名称必须与同一架构中的任何其它关系（表、序列、索引、视图、实体化视图或外部表）的名称不同。

创建序列后，您可以使用函数 `nextval`、`currval` 和 `setval` 对序列进行操作。序列操作函数中介绍了这些函数。

尽管您无法直接更新序列，但您可以使用如下查询：

```
SELECT * FROM name;
```

来检查序列的某些参数和当前状态。尤其是，序列的 `last_value` 字段显示任何会话分配的最后一个值。（当然，如果其它会话正在积极地进行 `nextval` 调用，则此值在打印时可能已经过时了。）可以在 `pg_sequences` 视图中观察其它参数，例如 *increment* 和 *maxvalue*。

参数

IF NOT EXISTS

如果同名的关系已存在，不引发错误。在这种情况下，将发出通知。请注意，不能保证现有关系与应已创建的序列有任何相似之处 — 它甚至可能不是序列。

name

要创建的序列的名称（可选择架构限定）。

data_type

可选子句 AS *data_type* 指定序列的数据类型。有效值为 bigint。bigint 为默认值。数据类型决定序列的默认最小值和最大值。

INCREMENT

可选子句 INCREMENT BY *increment* 指定将哪个值添加到当前序列值以创建新值。正值将生成升序序列，负值将生成降序序列。默认值是 1。

***minvalue*/NO MINVALUE**

可选子句 MINVALUE *minvalue* 确定序列可以生成的最小值。如果未提供此子句或指定了 NO MINVALUE，将使用默认值。升序序列的默认值为 1。降序序列的默认值是数据类型的最小值。

***maxvalue*/NO MAXVALUE**

可选子句 MAXVALUE *maxvalue* 确定序列的最大值。如果未提供此子句或指定了 NO MAXVALUE，将使用默认值。升序序列的默认值是数据类型的最大值。降序序列的默认值为 -1。

CYCLE / NO CYCLE

当升序序列或降序序列分别达到 *maxvalue* 或 *minvalue* 时，CYCLE 选项支持序列进行循环。如果达到限制，则生成的下一个数字将分别是 *minvalue* 或 *maxvalue*。

如果指定了 NO CYCLE，则在序列达到其最大值之后对 nextval 的任何调用都将返回错误。如果 CYCLE 或 NO CYCLE 都未指定，则 NO CYCLE 为默认值。

##

可选子句 START WITH *start* 支持序列从任何地方开始。升序序列的默认起始值为 *minvalue*，降序序列的默认起始值为 *maxvalue*。

cache

子句 CACHE *cache* 指定要预分配多少个序列号并将其存储在内存中，以便更快地进行访问。Aurora DSQL 中 CACHE 的可接受值为 1 或任何 >= 65536 的数字。最小值为 1（一次只能生成一个值，这意味着没有缓存）。

OWNED BY *table_name.column_name* / OWNED BY NONE

OWNED BY 选项使序列与特定的表列相关联，这样，如果删除该列（或其整个表），也将自动删除该序列。指定的表必须与序列具有相同的所有者且位于相同架构中。默认值 OWNED BY NONE 指定不存在此类关联。

备注

使用 `DROP SEQUENCE` 移除序列。

序列基于 `bigint` 算法，因此范围不能超过八字节整数的范围（-9223372036854775808 到 9223372036854775807）。

由于 `nextval` 和 `setval` 调用从不会回滚，因此，如果需要“无间隙”分配序列号，则无法使用序列对象。

在一次访问序列对象期间，每个会话都会分配和缓存连续的序列值，并相应地增加序列对象的 `last_value`。然后，在该会话中下一个 *cache*-1 使用 `nextval` 时，只返回预分配的值，而不涉及序列对象。因此，当会话结束时，任何已分配但未在该会话中使用的数字都将丢失，从而导致序列中出现“空洞”。

此外，尽管可以保证多个会话分配不同的序列值，但在考虑所有会话时，这些值可能会以非顺序方式生成。例如，如果 *cache* 设置为 10，则会话 A 可能会保留值 1..10 并返回 `nextval=1`，然后会话 B 可能会在会话 A 生成 `nextval=2` 之前保留值 11..20 并返回 `nextval=11`。因此，如果 *cache* 设置为 1，则可以安全地假设 `nextval` 值是按顺序生成的；如果 *cache* 设置大于 1，则只应假设 `nextval` 值都是不同的，而不是纯粹按顺序生成的。此外，`last_value` 将反映任何会话保留的最新值，无论 `nextval` 是否已返回该值。

另一个注意事项是，在其它会话用完它们已缓存的任何预分配值之前，这些会话将不会注意到对此类序列执行的 `setval`。

示例

创建一个名为 `serial` 的升序序列，从 101 开始：

```
CREATE SEQUENCE serial CACHE 65536 START 101;
```

从此序列中选择下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     101
```

从此序列中选择下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     102
```

在 INSERT 命令中使用此序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

使用 setval 将序列重置为特定值：

```
SELECT setval('serial', 200);
SELECT nextval('serial');

 nextval
---------
     201
```

兼容性

除以下例外情况之外，CREATE SEQUENCE 符合 SQL 标准：

- 获取下一个值是使用 nextval() 函数而不是标准的 NEXT VALUE FOR 表达式完成的。
- OWNED BY 子句是 PostgreSQL 扩展。

ALTER SEQUENCE

ALTER SEQUENCE：更改序列生成器的定义。

Important

使用序列时，应谨慎考虑缓存值。有关更多信息，请参阅 CREATE SEQUENCE 页面上的“重要提示”标注。

有关如何根据工作负载模式以最佳方式使用序列的指导，请参阅使用序列和标识列。

支持的语法

```
ALTER SEQUENCE [ IF EXISTS ] name
```

```
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ [ NO ] CYCLE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema

where cache is 1 or cache >= 65536
```

说明

`ALTER SEQUENCE` 更改现有序列生成器的参数。`ALTER SEQUENCE` 命令中未特别设置的任何参数都将保留其先前的设置。

您必须拥有该序列的所有权才能使用 `ALTER SEQUENCE`。要更改序列的架构，您还必须对新架构具有 `CREATE` 权限。要更改所有者，您必须能够 `SET ROLE` 以使用新的拥有角色，并且该角色必须对序列的架构拥有 `CREATE` 权限。（这些限制强制要求：更改所有者不会执行任何您无法通过删除并重新创建序列来完成的事情。但是，无论如何，超级用户都可以更改任何序列的所有权。）

参数

name

要更改的序列的名称（可选择架构限定）。

IF EXISTS

如果序列不存在，不引发错误。在这种情况下，将发出通知。

INCREMENT

子句 `INCREMENT BY` *`increment`* 是可选的。正值将生成升序序列，负值将生成降序序列。如果未指定，将保留旧的增量值。

***minvalue*/NO MINVALUE**

可选子句 `MINVALUE` *`minvalue`* 确定序列可以生成的最小值。如果指定了 `NO MINVALUE`，则将分别对升序序列和降序序列使用默认值 1 和数据类型的最小值。如果两个选项都未指定，则将保持当前的最小值。

maxvalue/NO MAXVALUE

可选子句 MAXVALUE *maxvalue* 确定序列的最大值。如果指定了 NO MAXVALUE，则将分别使用数据类型的最大值（对于升序序列）和 -1（对于降序序列）作为默认值。如果两个选项都未指定，则将保持当前的最大值。

CYCLE

当升序降序或降序序列分别达到 *maxvalue* 或 *minvalue* 时，可以使用可选的 CYCLE 关键字来使序列循环。如果达到限制，则生成的下一个数字将分别是 *minvalue* 或 *maxvalue*。

NO CYCLE

如果指定了可选的 NO CYCLE 关键字，则在序列达到其最大值之后对 nextval 的任何调用都将返回错误。如果 CYCLE 或 NO CYCLE 都未指定，则将保持旧的循环行为。

##

可选子句 START WITH *start* 更改序列的已记录的起始值。这对当前序列值没有影响；它只是设置将来的 ALTER SEQUENCE RESTART 命令将使用的值。

####

可选子句 RESTART [WITH *restart*] 更改序列的当前值。这类似于通过 is_called = false 调用 setval 函数：下次调用 nextval 将返回指定的值。在没有 *restart* 值的情况下编写 RESTART 等同于提供由 CREATE SEQUENCE 记录或由 ALTER SEQUENCE START WITH 上次设置的起始值。

与 setval 调用相比，针对序列的 RESTART 操作是事务性的，它会阻止并发事务从同一序列中获取数字。如果这不是所需的操作模式，则应使用 setval。

cache

子句 CACHE *cache* 支持预分配序列号并将其存储在内存中，以便更快地进行访问。该值必须为 1 或某个 >= 65536 的值。如果未指定，将保留旧的缓存值。有关缓存行为的更多信息，请参阅 CREATE SEQUENCE 下的指南。

OWNED BY *table_name*.*column_name* / OWNED BY NONE

OWNED BY 选项使序列与特定的表列相关联，这样，如果删除该列（或其整个表），也将自动删除该序列。如果指定，则此关联将替换先前为序列指定的任何关联。指定的表必须与序列具有相同的所有者，并且与序列处于相同的架构中。指定 OWNED BY NONE 会移除任何现有的关联，使序列变为“独立的”。

new_owner

序列的新所有者的用户名。

new_name

序列的新名称。

new_schema

序列的新架构。

备注

ALTER SEQUENCE 不会立即影响具有预分配（缓存）序列值的后端（当前后端除外）中的 nextval 结果。在注意到更改的序列生成参数之前，它们将用完所有缓存的值。当前后端将立即受到影响。

ALTER SEQUENCE 不会影响序列的 currval 状态。

ALTER SEQUENCE 可能会导致 OCC 发生其它事务。

出于历史原因，ALTER TABLE 也可以与序列一起使用；但仅支持与上述格式等效的 ALTER TABLE 变体用于序列。

示例

在 105 处重新启动一个名为 serial 的序列：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

ALTER SEQUENCE 符合 SQL 标准，但 AS、START WITH、OWNED BY、OWNER TO、RENAME TO 和 SET SCHEMA 子句除外，它们是 PostgreSQL 扩展。

DROP SEQUENCE

DROP SEQUENCE：移除序列。

支持的语法

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

说明

DROP SEQUENCE 移除序列号生成器。序列只能由其所有者或超级用户删除。

参数

IF EXISTS

如果序列不存在，不引发错误。在这种情况下，将发出通知。

name

序列的名称（可选择架构限定）。

CASCADE

自动删除依赖于序列的对象，进而删除依赖于这些对象的所有对象。

RESTRICT

如果任何对象依赖于该序列，则拒绝删除该序列。这是默认值。

示例

要移除序列 seq：

```
DROP SEQUENCE seq;
```

兼容性

DROP SEQUENCE 符合 SQL 标准，但该标准仅允许每个命令删除一个序列，并且 IF EXISTS 选项是一个 PostgreSQL 扩展，不在此标准之列。

CREATE VIEW

CREATE VIEW 定义新的持久视图。Aurora DSQL 不支持临时视图；仅支持持久视图。

支持的语法

```
CREATE [ OR REPLACE ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
```

```
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

说明

CREATE VIEW 定义查询的视图。此视图并未实际实现。相反，每次在查询中引用视图时，都会运行查询。

CREATE or REPLACE VIEW 与之类似，但如果同名的视图已经存在，则将替换该视图。新查询生成的列必须与现有视图查询生成的列相同（即顺序相同且数据类型相同的列名），但它可能会在列表末尾添加其它列。产生输出列的计算方法可能有所不同。

如果提供架构名称（例如 CREATE VIEW myschema.myview ...），则在指定的架构中创建视图。否则，将在当前架构中创建该视图。

视图的名称必须与同一架构中任何其它关系（表、索引、视图）的名称不同。

参数

CREATE VIEW 支持各种参数来控制可自动更新的视图的行为。

RECURSIVE

创建递归视图。语法：CREATE RECURSIVE VIEW [schema .] view_name (column_names) AS SELECT ...; 等同于 CREATE VIEW [schema .] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...) SELECT column_names FROM view_name;。

必须为递归视图指定视图列名列表。

name

要创建的视图的名称，可以选择使用架构进行限定。必须为递归视图指定列名列表。

column_name

要用于视图中各列的名称的可选列表。如果未提供，则从查询中推断出列名。

WITH (view_option_name [= view_option_value] [, ...])

此子句为视图指定可选参数；支持以下参数。

- check_option (enum)：此参数可以为 local 或 cascaded，等同于指定 WITH [CASCADED | LOCAL] CHECK OPTION。

- `security_barrier (boolean)`：如果视图旨在提供行级安全性，则应使用此参数。Aurora DSQL 目前不支持行级安全性，但此选项仍会强制首先评估视图的 `WHERE` 条件（以及任何使用标记为 `LEAKPROOF` 的运算符的条件）。
- `security_invoker (boolean)`：此选项会导致根据视图用户而不是视图所有者的权限来检查底层基本关系。有关完整详细信息，请参阅下面的备注。

可以使用 `ALTER VIEW` 在现有视图上更改上述所有选项。

`query`

`SELECT` 或 `VALUES` 命令，它们将提供视图的列和行。

`WITH [ CASCADED | LOCAL ] CHECK OPTION`

此选项控制可自动更新的视图的行为。指定此选项后，将检查视图上的 `INSERT` 和 `UPDATE` 命令，以确保新行满足视图定义条件（也就是说，检查新行以确保它们在视图中可见）。否则，将拒绝更新。如果未指定 `CHECK OPTION`，则支持视图上的 `INSERT` 和 `UPDATE` 命令创建在视图中不可见的行。

`LOCAL`：仅根据在视图本身中直接定义的条件来检查新行。不检查在底层基本视图上定义的任何条件（除非它们也指定了 `CHECK OPTION`）。

`CASCADED`：根据视图和所有底层基本视图的条件检查新行。如果指定了 `CHECK OPTION`，但既未指定 `LOCAL` 也未指定 `CASCADED`，则假定为 `CASCADED`。

> **Note**
>
> `CHECK OPTION` 不得与 `RECURSIVE` 视图一起使用。仅在可自动更新的视图上才支持 `CHECK OPTION`。

备注

使用 `DROP VIEW` 语句可删除视图。

应仔细考虑视图列的名称和数据类型。例如，不建议使用 CREATE VIEW vista AS SELECT 'Hello World';，因为列名称默认为 `?column?;`。此外，列的数据类型默认为 `text`，这可能不是您想要的。

更好的方法是显式指定列名和数据类型，例如：`CREATE VIEW vista AS SELECT text 'Hello World' AS hello;`。

默认情况下，对视图中引用的底层基本关系的访问权限由视图所有者的权限决定。在某些情况下，这可用于提供对底层表的安全但受限的访问。但是，并非所有视图都能防止篡改。

- 如果视图的 security_invoker 属性设置为 true，则对底层基本关系的访问权限取决于执行查询的用户的权限，而不是视图所有者的权限。因此，安全调用程序视图的用户必须对该视图及其底层基本关系拥有相关权限。
- 如果任何底层基本关系是安全调用程序视图，则会被视为直接从原始查询访问了该视图。因此，安全调用程序视图将始终使用当前用户的权限来检查其底层基本关系，即使从没有 security_invoker 属性的视图访问该视图也是如此。
- 视图中调用的函数的处理方式与使用视图直接从查询中调用函数的处理方式相同。因此，视图的用户必须具有权限来调用该视图使用的所有函数。视图中的函数是以执行查询的用户或函数所有者的权限来执行的，具体取决于函数是定义为 SECURITY INVOKER 还是 SECURITY DEFINER。
- 创建或替换视图的用户必须对视图查询中引用的任何架构具有 USAGE 权限，才能在这些架构中查找引用的对象。
- 在现有视图上使用 CREATE OR REPLACE VIEW 时，仅更改视图的定义 SELECT 规则以及任何 WITH (...)参数及其 CHECK OPTION。其它视图属性（包括所有权、权限和非 SELECT 规则）保持不变。您必须拥有视图才能替换它（这包括成为拥有角色的成员）。

可更新视图

简单视图可自动更新：系统将支持在视图上使用 INSERT、UPDATE 和 DELETE 语句，其方式与在常规表上相同。如果视图满足以下所有条件，则该视图可自动更新：

- 视图在其 FROM 列表中必须确切只有一个条目，该条目必须是一个表或另一个可更新的视图。
- 视图定义不得在顶层包含 WITH、DISTINCT、GROUP BY、HAVING、LIMIT 或 OFFSET 子句。
- 视图定义不得在顶层包含集合操作（UNION、INTERSECT 或 EXCEPT）。
- 视图的选择列表不得包含任何聚合、窗口函数或返回集合的函数。

可自动更新的视图可能包含可更新和不可更新的列的组合。如果列是对底层基本关系的可更新列的简单引用，则该列是可更新的。否则，该列是只读的，如果 INSERT 或 UPDATE 语句尝试为其赋值，则会发生错误。

默认情况下，未满足所有这些条件的更复杂的视图是只读的：系统不支持在视图上执行插入、更新或删除操作。

Note

在视图上执行插入、更新或删除操作的用户必须对该视图具有相应的插入、更新或删除权限。默认情况下，视图的所有者必须对底层基本关系拥有相关的权限，而执行更新的用户不需要对底层基本关系拥有任何权限。但是，如果视图将 security_invoker 设置为 true，则执行更新的用户（而不是视图所有者）必须对底层基本关系拥有相关权限。

示例

创建由所有喜剧电影组成的视图。

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

使用 `LOCAL CHECK OPTION` 创建视图。

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

创建递归视图。

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

兼容性

`CREATE OR REPLACE VIEW` 是 PostgreSQL 语言扩展。`WITH ( ... )` 子句也是扩展，安全屏障视图和安全调用程序视图也是如此。Aurora DSQL 支持这些语言扩展。

ALTER VIEW

`ALTER VIEW` 语句支持更改现有视图的各种属性，并且 Aurora DSQL 支持此命令的所有 PostgreSQL 语法。

支持的语法

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_column_name
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

说明

ALTER VIEW 更改视图的各种辅助属性。（如果要修改视图的定义查询，请使用 CREATE OR REPLACE VIEW。）您必须拥有该视图的所有权才能使用 ALTER VIEW。要更改视图的架构，您还必须对新架构具有 CREATE 权限。要更改所有者，您必须能够 SET ROLE 以使用新的拥有角色，并且该角色必须对视图的架构拥有 CREATE 权限。

参数

name

现有视图的名称（可选择架构限定）。

column_name

现有列的名称，或现有列的新名称。

IF EXISTS

如果视图不存在，不引发错误。在这种情况下，将发出通知。

SET/DROP DEFAULT

这些表单为列设置或移除默认值。视图列的默认值会替换为任何以视图为目标的 INSERT 或 UPDATE 命令。

new_owner

视图的新所有者的用户名。

new_name

视图的新名称。

new_schema

视图的新架构。

SET (view_option_name [= view_option_value] [, ...])

设置视图选项。以下是支持的选项：

- `check_option (enum)`：更改视图的检查选项。值必须为 `local` 或 `cascaded`。
- `security_barrier (boolean)`：更改视图的 security-barrier 属性。
- `security_invoker (boolean)`：更改视图的 security-invoker 属性。

RESET (view_option_name [, ...])

将视图选项重置为其默认值。

示例

将视图 `foo` 重命名为 `bar`：

```
ALTER VIEW foo RENAME TO bar;
```

将默认列值附加到可更新的视图：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

兼容性

`ALTER VIEW` 是 Aurora DSQL 支持的 SQL 标准的 PostgreSQL 扩展。

DROP VIEW

`DROP VIEW` 语句移除现有视图。Aurora DSQL 支持此命令的完整 PostgreSQL 语法。

支持的语法

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

说明

`DROP VIEW` 删除现有视图。要执行此命令，您必须是视图的所有者。

参数

`IF EXISTS`

如果视图不存在，不引发错误。在这种情况下，将发出通知。

`name`

要移除的视图的名称（可选择架构限定）。

`CASCADE`

自动删除依赖于视图的对象（如其它视图），进而删除依赖于这些对象的所有对象。

`RESTRICT`

如果任何对象依赖于该视图，则拒绝删除该视图。这是默认值。

示例

```
DROP VIEW kinds;
```

兼容性

此命令符合 SQL 标准，除了该标准只支持每个命令删除一个视图，并且 `IF EXISTS` 选项除外（这是 Aurora DSQL 支持的 PostgreSQL 扩展）。

从 PostgreSQL 迁移到 Aurora DSQL

Aurora DSQL 设计为与 PostgreSQL 兼容，支持诸如 ACID 事务、辅助索引、联接和标准 DML 操作等核心关系功能。大多数现有 PostgreSQL 应用程序只需进行极少的更改，即可迁移到 Aurora DSQL。

本节提供将应用程序迁移到 Aurora DSQL 的实用指南，包括框架兼容性、迁移模式和架构注意事项。

框架和 ORM 兼容性

Aurora DSQL 使用标准 PostgreSQL 线路协议，同时确保与 PostgreSQL 驱动程序和框架的兼容性。大多数流行的 ORM 与 Aurora DSQL 一起工作，而只需进行极少的更改或不需要更改。有关参考实现和可用的 ORM 集成，请参阅 the section called “Aurora DSQL 适配器”。

常见迁移模式

从 PostgreSQL 迁移到 Aurora DSQL 时，某些功能的工作原理会有所不同或具有其它语法。本节提供有关常见迁移场景的指导。

DDL 操作替代方案

Aurora DSQL 为传统的 PostgreSQL DDL 操作提供了现代替代方案：

索引创建

使用 `CREATE INDEX ASYNC` 而非 `CREATE INDEX` 来创建非阻止索引。

优势：在大型表上创建零停机时间索引。

数据移除

使用 `DELETE FROM table_name`，而不使用 `TRUNCATE`。

替代方案：要完全重新创建表，请使用 `DROP TABLE`，后跟 `CREATE TABLE`。

系统配置

Aurora DSQL 是完全托管式的，因此，配置是根据工作负载模式自动处理的。使用 AWS 管理控制台或 API 来管理集群设置。

优点：无需进行数据库调整或参数管理。

架构设计模式

调整以下常见的 PostgreSQL 模式以实现 Aurora DSQL 兼容性：

引用完整性模式

Aurora DSQL 支持表关系和 `JOIN` 操作。为实现引用完整性，请在应用程序层中实现验证。这种设计与现代分布式数据库模式保持一致，在这种模式中，应用程序层验证可提供更大的灵活性，并可避免级联操作带来的性能瓶颈。

模式：使用一致的命名约定、验证逻辑和事务边界，在应用程序层中实施引用完整性检查。许多大规模应用程序更喜欢这种方法，以便更好地控制错误处理和性能。

临时数据处理

使用带有清理逻辑的 CTE、子查询或常规表，而不是临时表。

替代方案：使用会话特定的名称创建表，然后在应用程序中对其进行清理。

了解架构差异

Aurora DSQL 的分布式无服务器架构在若干方面有意与传统 PostgreSQL 有所不同。这些差异使 Aurora DSQL 具有简单性和可扩展性等主要优势。

简化的数据库模型

每个集群单个数据库

Aurora DSQL 为每个集群提供一个名为 `postgres` 的内置数据库。

迁移提示：如果应用程序使用多个数据库，请创建单独的 Aurora DSQL 集群来进行逻辑分离，或者在单个集群中使用架构。

无临时表

对于临时数据处理，应使用公用表表达式（CTE）和子查询，它们可为复杂查询提供灵活的替代方案。

替代方案：将带有 `WITH` 子句的 CTE 用于临时结果集，或者使用带有唯一命名的常规表来存储会话特定的数据。

自动存储管理

Aurora DSQL 消除了表空间和手动存储管理。存储空间会根据数据模式自动扩展和优化。

优势：无需监控磁盘空间、规划存储分配或管理表空间配置。

现代应用程序模式

Aurora DSQL 鼓励采用可提高可维护性和性能的现代应用程序开发模式：

应用程序级逻辑，而不是数据库触发器

要获得类似触发器的功能，请在应用程序层中实现事件驱动的逻辑。

迁移策略：将触发逻辑移至应用程序代码，将事件驱动架构与 EventBridge 等 AWS 服务结合使用，或者使用应用程序日志记录来实现审计跟踪记录。

用于处理数据的 SQL 函数

Aurora DSQL 支持基于 SQL 的函数，但不支持像 PL/pgSQL 这样的程序性语言。

替代方案：使用 SQL 函数进行数据转换，或者将复杂逻辑移至应用程序层或 AWS Lambda 函数。

乐观并发控制，而不是悲观锁定

Aurora DSQL 使用乐观并发控制（OCC），这是一种不同于传统数据库锁定机制的无锁方法。Aurora DSQL 不是获取用于阻止其它事务的锁，而是支持事务在不阻止的情况下继续进行，并在提交时检测冲突。这样可以消除死锁，并防止慢速事务阻止其它操作。

主要区别：发生冲突时，Aurora DSQL 返回序列化错误，而不是让事务等待锁定。这要求应用程序实现重试逻辑，类似于处理传统数据库中的锁定超时，但冲突可以立即得到解决，而不是导致阻止等待。

设计模式：使用重试机制实现幂等事务逻辑。设计架构，通过使用随机主键并在键范围内分布更新，最大限度地减少争用。有关更多信息，请参阅 Aurora DSQL 中的并发控制。

关系和引用完整性

Aurora DSQL 支持表之间的外键关系，包括 `JOIN` 操作。为实现引用完整性，请在应用程序层中实现验证。尽管强制执行引用完整性可能很有价值，但级联操作（如级联删除）可能会造成意想不到的性能问题，例如，删除包含 1000 个订单项的订单会变成一个 1001 行的事务。出于这个原因，许多客户避免使用外键限制。

设计模式：在应用程序层中实施引用完整性检查，使用最终一致性模式，或利用 AWS 服务进行数据验证。

操作简化

Aurora DSQL 消除了许多传统的数据库维护任务，从而减少了操作开销：

无需手动维护

Aurora DSQL 自动管理存储优化、统计数据收集和性能调整。诸如 `VACUUM` 等传统维护命令由系统处理。

优点：不需要数据库维护时段、vacuum 调度和系统参数调整。

自动分区和扩展

Aurora DSQL 会根据访问模式自动对您的数据进行分区和分配。使用 UUID 或应用程序生成的 ID，以实现最佳分配。

迁移提示：移除手动分区逻辑，让 Aurora DSQL 处理数据分配。使用 UUID 或应用程序生成的 ID，以实现最佳分配。如果您的应用程序需要序列标识符，请参阅序列和标识列。

使用人工智能工具进行代理式迁移

人工智能编码代理可以通过分析架构、转换代码以及使用内置安全检查执行 DDL 迁移，来加快向 Aurora DSQL 迁移。

使用 Kiro 进行迁移

像 Kiro 这样的编码代理有助于您分析 PostgreSQL 代码并将其迁移到 Aurora DSQL：

- 架构分析：上传您现有的架构文件，并要求 Kiro 确定潜在的兼容性问题和提出替代方案
- 代码转换：提供您的应用程序代码，并要求 Kiro 协助重构触发逻辑、将序列替换为 UUID 或修改事务模式
- 迁移规划：要求 Kiro 根据您的特定应用程序架构制定分步迁移计划
- DDL 迁移：使用带有内置安全检查和用户验证功能的表重新创建模式执行架构修改

提示示例：

```
"Analyze this PostgreSQL schema for DSQL compatibility and suggest alternatives for any
 unsupported features"

"Help me refactor this trigger function into application-level logic for DSQL
 migration"

"Create a migration checklist for moving my Django application from PostgreSQL to DSQL"

"Drop the legacy_status column from the orders table"

"Change the price column from VARCHAR to DECIMAL in the products table"
```

使用表重新创建进行 DDL 迁移

将人工智能代理与 Aurora DSQL MCP 服务器结合使用时，某些 ALTER TABLE 操作使用可安全迁移数据的表重新创建模式。代理处理复杂性，同时让您随时了解每个步骤的情况。

以下操作使用表重新创建模式：

操作	方法
DROP COLUMN	从新表中排除列

操作	方法
ALTER COLUMN TYPE	在迁移期间强制转换数据类型
ALTER COLUMN SET/DROP NOT NULL	在新表定义中更改约束
ALTER COLUMN SET/DROP DEFAULT	在新表定义中定义默认值
ADD/DROP CONSTRAINT	在新表中包含或移除约束
MODIFY PRIMARY KEY	使用唯一性验证定义新 PK
拆分/合并列	使用 SPLIT_PART、SUBSTRING 或 CONCAT

无需重新创建表，即可直接支持以下 ALTER TABLE 操作：

- `ALTER TABLE ... RENAME COLUMN`：重命名列
- `ALTER TABLE ... RENAME TO`：重命名表
- `ALTER TABLE ... ADD COLUMN`：添加新列

安全功能：在执行 DDL 迁移时，人工智能代理会在任何破坏性操作（例如 DROP TABLE）之前，提供迁移计划、验证数据兼容性、确认行计数并请求明确批准。

批量迁移：对于超过 3000 行的表，代理会自动以 500-1000 行的增量进行批量迁移，以保持在事务限制范围内。

Aurora DSQL MCP 服务器

借助 Aurora DSQL 模型上下文协议（MCP）服务器，人工智能助手能够直接连接到 Aurora DSQL 集群并搜索 Aurora DSQL 文档。这样，人工智能就能够：

- 分析您的现有架构并提出迁移更改建议
- 使用表重新创建模式执行 DDL 迁移
- 在迁移期间测试查询并验证兼容性
- 根据最新的 Aurora DSQL 文档提供准确、最新的指导

要将 Aurora DSQL MCP 服务器与其它人工智能助手一起使用，请参阅 Aurora DSQL MCP 服务器的设置说明。

Aurora DSQL 有关 PostgreSQL 兼容性的注意事项

Aurora DSQL 的功能支持与自行管理的 PostgreSQL 有所不同，后者支持其分布式架构、无服务器操作和自动扩展。大多数应用程序无需修改，即可在这些差异范围内运行。

有关一般注意事项，请参阅使用 Amazon Aurora DSQL 的注意事项。有关配额和限制，请参阅 Amazon Aurora DSQL 中的集群配额和数据库限制。

- Aurora DSQL 对于每个集群使用单个名为 `postgres` 的内置数据库。为了进行逻辑分离，请创建单独的 Aurora DSQL 集群或在单个集群中使用架构。
- `postgres` 数据库使用 UTF-8 字符编码，该编码提供广泛的国际字符支持。
- 数据库仅使用 `C` 排序规则。
- Aurora DSQL 使用 `UTC` 作为系统时区。Postgres 在内部以 UTC 格式存储所有可识别时区的日期和时间。您可以设置 `TimeZone` 配置参数来转换它向客户端显示的方式，并将其作为客户端输入的默认值，服务器将使用该默认值在内部转换为 UTC。
- PostgreSQL `Repeatable Read` 的事务隔离级别是固定的。
- 事务具有以下约束：
  - DDL 和 DML 操作需要单独的事务
  - 一个事务只能包含 1 条 DDL 语句
  - 一个事务最多可以修改 3000 行，而无论二级索引的数量如何
  - 3000 行的限制适用于所有 DML 语句（`INSERT`、`UPDATE`、`DELETE`）
- 数据库连接在 1 小时后超时。
- Aurora DSQL 通过架构级别的授权来管理权限。管理员用户使用 `CREATE SCHEMA` 创建架构，并使用 `GRANT USAGE ON SCHEMA` 授予访问权限。管理员用户管理公有架构中的对象，而非管理员用户则在用户创建的架构中创建对象以明确所有权边界。有关更多信息，请参阅 授权数据库角色在数据库中使用 SQL。

需要迁移方面的帮助吗？

如果您遇到对迁移至关重要但 Aurora DSQL 目前不支持的功能，请参阅提供有关 Amazon Aurora DSQL 的反馈，以了解有关如何与 AWS 分享反馈的信息。

Aurora DSQL 中的并发控制

并发可让多个会话同时访问和修改数据，而不会损害数据完整性和一致性。Aurora DSQL 在实施现代、无锁并发控制机制的同时提供 PostgreSQL 兼容性。它通过快照隔离来保持完全的 ACID 合规性，同时确保数据一致性和可靠性。

Aurora DSQL 的一个关键优势是其无锁架构，这消除了常见的数据库性能瓶颈。Aurora DSQL 可防止慢速事务阻塞其它操作，并消除死锁风险。这种方法使 Aurora DSQL 对于性能和可扩展性至关重要的高吞吐量应用程序特别有价值。

事务冲突

Aurora DSQL 使用乐观并发控制（OCC），其工作原理与传统的基于锁的系统不同。OCC 不使用锁，而是在提交时评估冲突。如果多个事务在更新同一行时发生冲突，Aurora DSQL 会按如下方式管理事务：

- 提交时间最早的事务由 Aurora DSQL 处理。
- 冲突的事务会收到 PostgreSQL 序列化错误，指示需要重试。

设计应用程序以实施重试逻辑来处理冲突。理想的设计模式是幂等的，尽可能将事务重试作为第一选择。建议采用的逻辑类似于标准 PostgreSQL 锁定超时或死锁情况下的中止和重试逻辑。然而，OCC 要求您的应用程序更频繁地实施此逻辑。

优化事务性能的准则

要优化性能，请尽量减少对单个键或小键范围的高度争用。要实现此目标，请按照以下准则设计架构，使其在集群键范围内分散更新：

- 为表选择一个随机主键。
- 避免使用会增加单个键争用的模式。即使在事务量增长的情况下，这种方法也能确保最佳性能。

Aurora DSQL 中的 DDL 和分布式事务

数据定义语言（DDL）在 Aurora DSQL 中的行为与在 PostgreSQL 中不同。Aurora DSQL 具有一个多可用区分布式和无共享数据库层，该数据库层在多租户计算和存储实例集的基础之上构建。由于不存在单个主数据库节点或中心节点，因此数据库目录是分布式的。这样，Aurora DSQL 将 DDL 架构更改作为分布式事务进行管理。

具体而言，DDL 在 Aurora DSQL 中的行为有所不同，如下所示：

并发控制错误

如果您运行一个事务，而另一个事务更新资源，则 Aurora DSQL 会返回并发控制违规错误。例如，请考虑以下操作序列：

1. 在会话 1 中，用户向表 `mytable` 中添加一列。
2. 在会话 2 中，用户尝试向 `mytable` 中插入一行。

   Aurora DSQL 返回错误 `SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (OC001).`

DDL 和 DML 在同一个事务中

Aurora DSQL 中的事务只能包含一个 DDL 语句，而不能同时拥有 DDL 和 DML 语句。此限制意味着您无法在同一个事务中创建表并将数据插入到同一个表中。例如，Aurora DSQL 支持以下顺序事务。

```
BEGIN;
  CREATE TABLE mytable (ID_col integer);
COMMIT;

BEGIN;
  INSERT into FOO VALUES (1);
COMMIT;
```

Aurora DSQL 不支持以下事务，其中同时包括 `CREATE` 和 `INSERT` 语句。

```
BEGIN;
  CREATE TABLE FOO (ID_col integer);
  INSERT into FOO VALUES (1);
COMMIT;
```

异步 DDL

在标准 PostgreSQL 中，诸如 `CREATE INDEX` 之类的 DDL 操作会锁定受影响的表，使其不可用于从其它会话中读取和写入。在 Aurora DSQL 中，这些 DDL 语句使用后台管理器异步运行。对受影响表的访问不受阻止。因此，大型表上的 DDL 可以在不停机或不影响性能的情况下运行。有关 Aurora DSQL 中异步作业管理器的更多信息，请参阅 Aurora DSQL 中的异步索引。

Aurora DSQL 中的主键

在 Aurora DSQL 中，主键是一种在物理上组织表数据的功能。它类似于 PostgreSQL 中的 CLUSTER 操作或其它数据库中的聚集索引。在定义主键时，Aurora DSQL 会创建一个包含表中所有列的索引。Aurora DSQL 中的主键结构可确保高效的数据访问和管理。

数据结构和存储

在定义主键时，Aurora DSQL 会按主键顺序存储表数据。这种按索引组织的结构支持主键查找来直接检索所有列值，而不是像传统 B 树索引那样跟随指向数据的指针。与 PostgreSQL 中仅对数据进行一次重组的 CLUSTER 操作不同，Aurora DSQL 会自动并持续保持这种顺序。这种方法可提高依赖于主键访问的查询的性能。

Aurora DSQL 还使用主键来为表和索引中的每一行生成集群范围的唯一键。此唯一键也构成了分布式数据管理的基础。它支持跨多个节点对数据进行自动分区，并支持可扩展存储和高并发性。因此，主键结构有助于 Aurora DSQL 自动扩展并高效地管理并发工作负载。

选择主键的准则

在 Aurora DSQL 中选择和使用主键时，请考虑以下准则：

- 创建表时定义主键。以后您无法更改此键或添加新的主键。主键成为用于数据分区和自动扩展写入吞吐量的集群范围键的一部分。如果未指定主键，Aurora DSQL 将分配一个合成的隐藏 ID。
- 对于写入量较高的表，请避免使用单调递增的整数作为主键。这可能会由于将所有新的插入内容定向到单个分区而导致性能问题。相反，应将主键与随机分布结合使用，以确保写入操作在各存储分区之间均匀分布。
- 对于不经常更改或只读的表，可以使用升序键。升序键的示例包括时间戳或序列号。密集键有许多紧密间隔或重复的值。您可以使用升序键，即使它是密集键，因为写入性能并不那么重要。
- 如果全表扫描不能满足您的性能要求，请选择更高效的访问方法。在大多数情况下，这意味着使用与查询中最常用的联接和查找键相匹配的主键。
- 主键中各列的最大组合大小为 1 KiB。有关更多信息，请参阅 Aurora DSQL 中的数据库限制和 Aurora DSQL 中支持的数据类型。
- 主键或二级索引中最多可以包含 8 列。有关更多信息，请参阅 Aurora DSQL 中的数据库限制和 Aurora DSQL 中支持的数据类型。

序列和标识列

序列和标识列生成整数值，在需要紧凑或用户可读的标识符时非常有用。这些值涉及在 CREATE SEQUENCE 文档中描述的分配和缓存行为。

主题

- 序列操作函数
- 标识列
- 使用序列和标识列

序列操作函数

本节介绍用于操作序列对象的函数，也称为序列生成器或就称为序列。序列对象是使用 CREATE SEQUENCE 创建的特殊单行表。序列对象常用于为表的行生成唯一标识符。序列函数提供了简单、多用户安全的方法，用于从序列对象中获取连续的序列值。

Important

使用序列时，应谨慎考虑缓存值。有关更多信息，请参阅 CREATE SEQUENCE 页面上的“重要提示”标注。

有关如何根据工作负载模式以最佳方式使用序列的指导，请参阅使用序列和标识列。

函数	说明
`nextval ( regclass ) # bigint`	使序列对象前进到其下一个值并返回该值。这是以原子方式完成的：即使多个会话同时运行 nextval，每个会话也会安全地接收一个不同的序列值。如果序列对象是使用默认参数创建的，则后续的 nextval 调用将返回从 1 开始递增的值。其它行为可以通过在 CREATE SEQUENCE 命令中使用适当的参数来获得。此函数需要针对序列的 USAGE 或 UPDATE 权限。
`setval ( regclass, bigint [, boolean ] ) # bigint`	设置序列对象的当前值，并可选设置其 is_called 标志。双参数格式将序列的 last_value 字段设置为指定的值，并将其 is_called 字段设置为 true，

函数	说明

	这意味着下一个 nextval 将在返回值之前使序列前进。currval 将报告的值也会设置为指定的值。在三参数格式中，is_called 可以设置为 true 或 false。true 与双参数格式效果相同。如果将其设置为 false，则下一个 nextval 将确切返回指定的值，并且序列前进从后一个 nextval 开始。此外，此处并未更改 currval 报告的值。例如：
	`SELECT setval('myseq', 42);           -- Next nextval will return 43`
	`SELECT setval('myseq', 42, true);     -- Same as above`
	`SELECT setval('myseq', 42, false);    -- Next nextval will return 42`
	setval 返回的结果只是其第二个参数的值。此函数需要针对序列的 UPDATE 权限。

currval (regclass) # bigint	返回当前会话中此序列的 nextval 最近获得的值。（如果在此会话中从未为此序列调用过 nextval，则会报告错误。）由于这会返回会话本地值，因此，无论其它会话是否因为当前会话运行 nextval 而运行了它，都会给出一个可预测的答案。此函数需要针对序列的 USAGE 或 SELECT 权限。
lastval () # bigint	返回 nextval 在当前事务中最近返回的值。此函数与 currval 相同（除了它不是将序列名称作为参数），它引用在当前事务中最近应用了 nextval 的任何序列。如果在当前事务中尚未调用 nextval，则调用 lastval 是错误的。此函数需要针对上次使用的序列的 USAGE 或 SELECT 权限。

> **Warning**
>
> 如果发出调用的事务稍后中止，则不会回收 nextval 获得的值以供重用。这意味着，事务中止或数据库崩溃可能会导致已分配值的序列中出现间隙。在不中止事务的情况下，也可能出现

这种情况。例如，带有 ON CONFLICT 子句的 INSERT 将计算要插入的元组（包括执行任何必需的 nextval 调用），然后再检测到任何可能导致其遵循 ON CONFLICT 规则的冲突。因此，Aurora DSQL 的序列对象不能用于获取“无间隙”序列。
同样，由 setval 所做的序列状态更改会立即对其它事务可见，并且在发出调用的事务回滚时不会撤消。

序列函数要执行操作的序列由 regclass 参数指定，该参数只是序列在 pg_class 系统目录中的 OID。但是，您不必手动查找 OID，因为 regclass 数据类型的输入转换器将为您完成此项工作。有关详细信息，请参阅有关 Object Identifier Types 的 PostgreSQL 文档。

标识列

Important

使用标识列时，应谨慎考虑缓存值。有关更多信息，请参阅 CREATE SEQUENCE 页面上的“重要提示”标注。
有关如何根据工作负载模式以最佳方式使用标识列的指导，请参阅使用序列和标识列。

标识列是根据隐式序列自动生成的特殊列。它可以用来生成键值。要创建标识列，请在 CREATE TABLE 中使用 GENERATED ... AS IDENTITY 子句，例如：

```
CREATE TABLE people (
    id bigint GENERATED ALWAYS AS IDENTITY (CACHE 70000),
    ...
);
```

或者：

```
CREATE TABLE people (
    id bigint GENERATED BY DEFAULT AS IDENTITY (CACHE 70000),
    ...
);
```

有关更多信息，请参阅CREATE TABLE。

如果对带有标识列的表执行 INSERT 命令，但没有为标识列显式指定任何值，则会插入由隐式序列生成的值。例如，使用前面的定义并假设其它适当的列，编写：

```
INSERT INTO people (name, address) VALUES ('A', 'foo');
INSERT INTO people (name, address) VALUES ('B', 'bar');
```

将为从 1 开始的 id 列生成值并生成下面的表数据：

```
 id | name | address
----+------+---------
  1 | A    | foo
  2 | B    | bar
```

或者，可以指定关键字 DEFAULT 来代替值，以显式请求序列生成的值：

```
INSERT INTO people (id, name, address) VALUES (DEFAULT, 'C', 'baz');
```

同样，可以在 UPDATE 命令中使用关键字 DEFAULT。

这样，在许多方面，标识列的行为类似于具有默认值的列。

列定义中的子句 ALWAYS 和 BY DEFAULT 决定了在 INSERT 和 UPDATE 命令中如何显式处理用户指定的值。在 INSERT 命令中，如果选择了 ALWAYS，则只有在 INSERT 语句指定 OVERRIDING SYSTEM VALUE 时才接受用户指定的值。如果选择了 BY DEFAULT，则优先使用用户指定的值。因此，使用 BY DEFAULT 会产生更类似于使用默认值时的行为，其中，默认值可以由显式值覆盖，而 ALWAYS 可提供更多保护，以防止意外插入显式值。

标识列的数据类型必须为序列支持的数据类型之一。（请参见 CREATE SEQUENCE。）关联序列的属性可以在创建标识列时指定（请参阅 CREATE TABLE），也可以在之后更改（请参阅 ALTER TABLE）。

标识列会自动标记为 NOT NULL。但是，身份列并不能保证唯一性。（序列通常返回唯一值，但可以重置序列，也可以手动将值插入标识列中，如前面所述。）需要使用 PRIMARY KEY 或 UNIQUE 约束来强制实施唯一性。

使用序列和标识列

本节有助于您了解如何根据工作负载模式以最佳方式使用序列和标识列。

> **Important**
>
> 有关分配和缓存行为的更多详细信息，请参阅 CREATE SEQUENCE 页面上的“重要提示”标注。

选择标识符类型

Amazon Aurora DSQL 既支持基于 UUID 的标识符，也支持使用序列或身份列生成的整数值。这些选项的不同之处在于分配值的方式以及值在负载下的扩展方式。

UUID 值无需协调即可生成，非常适合频繁或跨许多会话创建标识符的工作负载。由于 Amazon Aurora DSQL 专为分布式操作而设计，因此避免协调通常是有益的。因此，建议将 UUID 作为默认标识符类型，尤其是对于可扩展性很重要且不需要对标识符进行严格排序的工作负载中的主键。

序列和标识列生成紧凑的整数值，方便用于用户可读的标识符、报告和外部接口。当出于可用性或集成原因而首选数字标识符时，可以考虑将序列或标识列与基于 UUID 的标识符结合使用。当需要整数序列或标识值时，选择合适的缓存大小将成为工作负载设计的重要部分。有关选择缓存大小的指导，请参阅以下章节。

选择缓存大小

选择适当的缓存值是有效地使用序列和标识列的重要部分。缓存设置决定了标识符分配在负载下的行为方式，从而影响系统吞吐量以及值反映分配顺序的紧密程度。

在以下情况下，较大的缓存大小 `CACHE >= 65536` 非常适合：

- 标识符以高频率生成
- 许多会话并发插入
- 工作负载可以容忍间隙和明显的排序效果

例如，大容量事件摄取工作负载（例如 IoT 或遥测）以及诸如作业运行 ID、支持案例参考或内部订单号等操作标识符通常会受益于较大的缓存大小，其中标识符频繁生成且不要求严格的排序。

在以下情况下，缓存大小为 1 将更好地保持一致：

- 分配率相对较低
- 随着时间推移，预计标识符将更紧密地遵循分配顺序
- 使间隙最小化比最大吞吐量更重要

诸如分配账户或参考编号之类的工作负载会与缓存大小为 1 更好地保持一致，在这些工作负载中，标识符生成频率较低，并且需要更紧密的排序。

Aurora DSQL 中的异步索引

`CREATE INDEX ASYNC` 命令在指定表的一列或多列上创建索引。此命令是一种异步 DDL 操作，不会阻止其它事务。当您运行 `CREATE INDEX ASYNC` 时，Aurora DSQL 立即返回 `job_id`。

您可以使用 `sys.jobs` 系统视图来监控此异步作业的状态。当索引创建作业正在进行时，您可以使用以下过程和命令：

`sys.wait_for_job(job_id)`*`'your_index_creation_job_id'`*

阻止当前会话，直到指定的作业完成或失败。返回一个布尔值，指示成功或失败。

`DROP INDEX`

取消正在进行的索引构建作业。

异步索引创建过程完成后，Aurora DSQL 更新系统目录以将索引标记为活动状态。

> **Note**
>
> 请注意，在此更新期间访问同一命名空间中的对象的并发事务可能会遇到并发错误。

当 Aurora DSQL 完成异步索引任务时，它会更新系统目录以显示该索引处于活动状态。如果此时其它事务引用同一命名空间中的对象，则您可能会看到并发错误。

语法

`CREATE INDEX ASYNC` 使用下面的语法。

```
CREATE [ UNIQUE ] INDEX ASYNC [ IF NOT EXISTS ] name ON table_name
    ( { column_name } [ NULLS { FIRST | LAST } ] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
```

参数

`UNIQUE`

指示 Aurora DSQL 在它创建索引时和您每次添加数据时，检查表中是否存在重复值。如果指定此参数，则会导致重复条目的插入和更新操作会生成错误。

IF NOT EXISTS

指示如果已存在同名的索引，则 Aurora DSQL 不应引发异常。在这种情况下，Aurora DSQL 不会创建新索引。请注意，您尝试创建的索引的结构可能与现有的索引截然不同。如果您指定此参数，则需要索引名称。

name

索引的名称。您不能在此参数中包含架构的名称。

Aurora DSQL 在与其父表相同的架构中创建索引。索引的名称必须与架构中任何其它对象（例如表或索引）的名称不同。

如果您未指定名称，Aurora DSQL 将根据父表和索引列的名称自动生成名称。例如，如果您运行 `CREATE INDEX ASYNC on table1 (col1, col2)`，Aurora DSQL 会自动将索引命名为 `table1_col1_col2_idx`。

NULLS FIRST | LAST

空列和非空列的排序顺序。`FIRST` 表示 Aurora DSQL 应先对空列进行排序，然后再对非空列进行排序。`LAST` 表示 Aurora DSQL 应先对非空列进行排序，之后再对空列进行排序。

INCLUDE

要作为非键列包含在索引中的列的列表。您不能在索引扫描搜索限定条件中使用非键列。就索引的唯一性而言，Aurora DSQL 会忽略该列。

NULLS DISTINCT | NULLS NOT DISTINCT

指定 Aurora DSQL 是否应将空值视为唯一索引中的不同值。默认值为 `DISTINCT`，这意味着唯一索引可以在一列中包含多个空值。`NOT DISTINCT` 表示索引不能在一列中包含多个空值。

使用说明

请考虑以下准则：

- `CREATE INDEX ASYNC` 命令不引入锁。它也不会影响 Aurora DSQL 用来创建索引的基表。
- 在架构迁移操作期间，`sys.wait_for_job(job_id)`*`'your_index_creation_job_id'`* 过程很有用。它可确保后续的 DDL 和 DML 操作以新创建的索引为目标。
- 每当 Aurora DSQL 运行新的异步任务时，它都会检查 `sys.jobs` 视图，并删除状态为 `completed` 或 `failed` 超过 30 分钟的任务。这样，`sys.jobs` 主要显示正在进行的任务，而不包含有关旧任务的信息。

- 如果 Aurora DSQL 无法构建异步索引，则索引将保持 INVALID。对于唯一索引，DML 操作受唯一性约束所制约，直至您删除索引。我们建议您删除无效的索引并重新创建它们。

创建索引：示例

以下示例演示如何创建架构、表和索引。

1. 创建名为 `test.departments` 的文件。

```
CREATE SCHEMA test;

CREATE TABLE test.departments (name varchar(255) primary key NOT null,
     manager varchar(255),
     size varchar(4));
```

2. 在表中插入一行。

```
INSERT INTO test.departments VALUES ('Human Resources', 'John Doe', '10')
```

3. 创建异步索引。

```
CREATE INDEX ASYNC test_index on test.departments(name, manager, size);
```

`CREATE INDEX` 命令返回作业 ID，如下所示。

```
job_id
--------------------------
jh2gbtx4mzhgfkbimtgwn5j45y
```

`job_id` 表示 Aurora DSQL 已经提交了新作业来创建索引。可以使用过程 `sys.wait_for_job(job_id)`*'your_index_creation_job_id'* 来阻止会话中的其它工作，直到作业完成或超时。

查询索引创建的状态：示例

查询 `sys.jobs` 系统视图以查看索引的创建状态，如以下示例所示。

```
SELECT * FROM sys.jobs where job_id = 'wqhu6ewifze5xitg3umt24h5ua';
```

Aurora DSQL 返回与下面类似的响应。

```
          job_id            |  status    | details |  job_type    | class_id | object_id
 |    object_name     |        start_time         |       update_time
----------------------------+-----------+---------+-------------+---------+-----------
+-------------------+------------------------+------------------------
 wqhu6ewifze5xitg3umt24h5ua | completed |         | INDEX_BUILD |     1259 |     26433
 | public.nt2_c1_idx | 2025-09-25 22:07:31+00 | 2025-09-25 22:07:46+00
```

状态列可以是以下值之一。

Status	说明
submitted	任务已提交，但是 Aurora DSQL 尚未开始处理该任务。
processing	Aurora DSQL 正在处理该任务。
failed	任务失败。有关更多信息，请参阅详细信息列。如果 Aurora DSQL 未能构建索引，Aurora DSQL 不会自动移除索引定义。您必须使用 DROP INDEX 命令手动移除索引。
completed	Aurora DSQL 已成功完成任务。

也可以通过目录表 pg_index 和 pg_class 查询索引的状态。具体而言，属性 indisvalid 和 indisimmediate 可以告诉您索引处于什么状态。当 Aurora DSQL 创建索引时，索引的初始状态为 INVALID。索引的 indisvalid 标志返回 FALSE 或 f，表示该索引无效。如果标志返回 TRUE 或 t，则索引已就绪。

```
SELECT relname AS index_name, indisvalid as is_valid, pg_get_indexdef(indexrelid) AS
 index_definition
from pg_index, pg_class
WHERE pg_class.oid = indexrelid AND indrelid = 'test.departments'::regclass;
```

```
    index_name    | is_valid |
 index_definition
------------------+----------
+---------------------------------------------------------------------------------------------------
 department_pkey  |     t    | CREATE UNIQUE INDEX department_pkey ON test.departments
 USING btree_index (title) INCLUDE (name, manager, size)
 test_index1      |     t    | CREATE INDEX test_index1 ON test.departments USING
 btree_index (name, manager, size)
```

唯一索引构建失败

如果异步唯一索引构建作业显示失败状态以及详细信息 `Found duplicate key while validating index for UCVs`，这表示由于违反唯一性约束而无法构建唯一索引。

解决唯一索引构建失败

1. 移除主表中与在唯一二级索引中指定的键有重复条目的所有行。
2. 删除失败的索引。
3. 发出新的创建索引命令。

检测主表中的唯一性违规

以下 SQL 查询有助于您识别表的指定列中的重复值。当您需要对当前未设置为主键或没有唯一约束的列（例如用户表中的电子邮件地址）强制实施唯一性时，这特别有用。

以下示例演示如何创建示例用户表，在其中填充包含已知重复项的测试数据，然后运行检测查询。

定义表架构

```
-- Drop the table if it exists
DROP TABLE IF EXISTS users;

-- Create the users table with a simple integer primary key
CREATE TABLE users (
    user_id INTEGER PRIMARY KEY,
    email VARCHAR(255),
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
);
```

插入包含一组重复电子邮件地址的示例数据

```
-- Insert sample data with explicit IDs
INSERT INTO users (user_id, email, first_name, last_name) VALUES
    (1, 'john.doe@example.com', 'John', 'Doe'),
    (2, 'jane.smith@example.com', 'Jane', 'Smith'),
    (3, 'john.doe@example.com', 'Johnny', 'Doe'),
    (4, 'alice.wong@example.com', 'Alice', 'Wong'),
    (5, 'bob.jones@example.com', 'Bob', 'Jones'),
    (6, 'alice.wong@example.com', 'Alicia', 'Wong'),
    (7, 'bob.jones@example.com', 'Robert', 'Jones');
```

运行重复项检测查询

```
-- Query to find duplicates
WITH duplicates AS (
    SELECT email, COUNT(*) as duplicate_count
    FROM users
    GROUP BY email
    HAVING COUNT(*) > 1
)
SELECT u.*, d.duplicate_count
FROM users u
INNER JOIN duplicates d ON u.email = d.email
ORDER BY u.email, u.user_id;
```

查看所有包含重复电子邮件地址的记录

```
 user_id |         email          | first_name | last_name |         created_at
 | duplicate_count
---------+------------------------+------------+-----------
+----------------------------+-----------------
       4 | akua.mansa@example.com | Akua       | Mansa     | 2025-05-21 20:55:53.714432
 |               2
       6 | akua.mansa@example.com | Akua       | Mansa     | 2025-05-21 20:55:53.714432
 |               2
       1 | john.doe@example.com   | John       | Doe       | 2025-05-21 20:55:53.714432
 |               2
       3 | john.doe@example.com   | Johnny     | Doe       | 2025-05-21 20:55:53.714432
 |               2
```

```
(4 rows)
```

如果我们现在尝试使用索引创建语句，它就会失败：

```
postgres=> CREATE UNIQUE INDEX ASYNC idx_users_email ON users(email);
      job_id
----------------------------
 ve32upmjz5dgdknpbleeca5tri
(1 row)

postgres=> select * from sys.jobs;
           job_id           |  status   |                        details
       |  job_type   | class_id | object_id |      object_name       |       start_time
       |      update_time
----------------------------+-----------
+------------------------------------------------------+-------------+----------
+-----------+-----------------------+-----------------------+-----------------------
 qpn6aqlkijgmzilyidcpwrpova | completed |
          | DROP       |     1259 |     26384 |                        | 2025-05-20
 00:47:10+00 | 2025-05-20 00:47:32+00
 ve32upmjz5dgdknpbleeca5tri | failed    | Found duplicate key while validating index
 for UCVs | INDEX_BUILD |     1259 |     26396 | public.idx_users_email | 2025-05-20
 00:49:49+00 | 2025-05-20 00:49:56+00
(2 rows)
```

Aurora DSQL 中的系统表和命令

请参阅以下各节来了解 Aurora DSQL 中支持的系统表和目录，以及用于获取有关系统的信息（如版本）的有用查询。

系统表

Aurora DSQL 与 PostgreSQL 兼容，因此 Aurora DSQL 中还存在许多来自 PostgreSQL 的 system catalog tables 和 views。

重要的 PostgreSQL 目录表和视图

下表介绍了您可能在 Aurora DSQL 中使用的最常见的表和视图。

名称	描述
pg_namespace	有关所有架构的信息
pg_tables	有关所有表的信息
pg_attribute	有关所有属性的信息
pg_views	有关（预）定义视图的信息
pg_class	描述所有表、列、索引和类似对象
pg_stats	有关计划程序统计数据的视图
pg_user	有关用户的信息
pg_roles	有关用户和组的信息
pg_indexes	列出所有索引
pg_constraint	列出对表的约束

支持和不支持的目录表

下表指示在 Aurora DSQL 中支持哪些表和不支持哪些表。

名称	适用于 Aurora DSQL
pg_aggregate	否
pg_am	是
pg_amop	否
pg_amproc	否
pg_attrdef	是
pg_attribute	是

名称	适用于 Aurora DSQL
pg_authid	否（使用 pg_roles）
pg_auth_members	支持
pg_cast	是
pg_class	是
pg_collation	是
pg_constraint	是
pg_conversion	否
pg_database	否
pg_db_role_setting	是
pg_default_acl	是
pg_depend	是
pg_description	是
pg_enum	否
pg_event_trigger	否
pg_extension	否
pg_foreign_data_wrapper	否
pg_foreign_server	否
pg_foreign_table	否
pg_index	是
pg_inherits	是

名称	适用于 Aurora DSQL
pg_init_privs	否
pg_language	否
pg_largeobject	否
pg_largeobject_metadata	是
pg_namespace	是
pg_opclass	否
pg_operator	是
pg_opfamily	否
pg_parameter_acl	是
pg_partitioned_table	否
pg_policy	否
pg_proc	否
pg_publication	否
pg_publication_namespace	否
pg_publication_rel	否
pg_range	是
pg_replication_origin	否
pg_rewrite	否
pg_seclabel	否
pg_sequence	否

名称	适用于 Aurora DSQL
pg_shdepend	是
pg_shdescription	是
pg_shseclabel	否
pg_statistic	是
pg_statistic_ext	否
pg_statistic_ext_data	否
pg_subscription	否
pg_subscription_rel	否
pg_tablespace	否
pg_transform	否
pg_trigger	否
pg_ts_config	是
pg_ts_config_map	是
pg_ts_dict	是
pg_ts_parser	是
pg_ts_template	是
pg_type	是
pg_user_mapping	否

支持和不支持的系统视图

下表指示在 Aurora DSQL 中支持哪些视图和不支持哪些视图。

名称	适用于 Aurora DSQL
pg_available_extensions	否
pg_available_extension_versions	否
pg_backend_memory_contexts	是
pg_config	否
pg_cursors	否
pg_file_settings	否
pg_group	是
pg_hba_file_rules	否
pg_ident_file_mappings	否
pg_indexes	是
pg_locks	否
pg_matviews	否
pg_policies	否
pg_prepared_statements	否
pg_prepared_xacts	否
pg_publication_tables	否
pg_replication_origin_status	否
pg_replication_slots	否
pg_roles	是
pg_rules	否

名称	适用于 Aurora DSQL
pg_seclabels	否
pg_sequences	否
pg_settings	是
pg_shadow	是
pg_shmem_allocations	是
pg_stats	是
pg_stats_ext	否
pg_stats_ext_exprs	否
pg_tables	是
pg_timezone_abbrevs	是
pg_timezone_names	是
pg_user	是
pg_user_mappings	否
pg_views	是
pg_stat_activity	否
pg_stat_replication	否
pg_stat_replication_slots	否
pg_stat_wal_receiver	否
pg_stat_recovery_prefetch	否
pg_stat_subscription	否

名称	适用于 Aurora DSQL
pg_stat_subscription_stats	否
pg_stat_ssl	是
pg_stat_gssapi	否
pg_stat_archiver	否
pg_stat_io	否
pg_stat_bgwriter	否
pg_stat_wal	否
pg_stat_database	否
pg_stat_database_conflicts	否
pg_stat_all_tables	否
pg_stat_all_indexes	否
pg_statio_all_tables	否
pg_statio_all_indexes	否
pg_statio_all_sequences	否
pg_stat_slru	否
pg_statio_user_tables	否
pg_statio_user_sequences	否
pg_stat_user_functions	否
pg_stat_user_indexes	否
pg_stat_progress_analyze	否

名称	适用于 Aurora DSQL
pg_stat_progress_basebackup	否
pg_stat_progress_cluster	否
pg_stat_progress_create_index	否
pg_stat_progress_vacuum	否
pg_stat_sys_indexes	否
pg_stat_sys_tables	否
pg_stat_xact_all_tables	否
pg_stat_xact_sys_tables	否
pg_stat_xact_user_functions	否
pg_stat_xact_user_tables	否
pg_statio_sys_indexes	否
pg_statio_sys_sequences	否
pg_statio_sys_tables	否
pg_statio_user_indexes	否

sys.jobs 视图

sys.jobs 提供有关异步作业的状态信息。例如，在您创建异步索引后，Aurora DSQL 将返回 job_uuid。您可以将此 job_uuid 与 sys.jobs 结合使用来查找作业的状态。

```
SELECT * FROM sys.jobs;
```

Aurora DSQL 返回与下面类似的响应。

```
            job_id              |  status    | details |  job_type    | class_id | object_id
 |     object_name      |          start_time          |        update_time
```

```
----------------------------+-----------+---------+-------------+----------+-----------
+-------------------+------------------------+------------------------
 wqhu6ewifze5xitg3umt24h5ua | completed |         | INDEX_BUILD |     1259 |     26433
 | public.nt2_c1_idx | 2025-09-25 22:07:31+00 | 2025-09-25 22:07:46+00
 kkngzf33dndl3daacxehpx5eba | completed |         | ANALYZE     |     1259 |     26419
 | public.nt         | 2025-09-25 21:57:05+00 | 2025-09-25 21:57:27+00
 fyopxjb6ovdn7po6lrkj63cyea | completed |         | DROP        |     1259 |     26422
 |                   | 2025-09-25 22:05:57+00 | 2025-09-25 22:06:03+00
```

下表介绍 sys.jobs 视图中的各列。

sys.jobs 视图列

列	类型	说明
job_id	text	一个 32 位的 UUID，表示作业。
status	text	作业的当前状态。可能的值为 submitted 、processing 、completed 和 failed。有关更多信息，请参阅 sys.jobs 状态值。
details	text	有关作业的任何相关详细信息。如果任务失败，则会提供详细原因。
job_type	text	异步作业的类型。可能的值为：INDEX_BUILD – 异步索引构建。ANALYZE – 系统提交的自动分析作业。DROP – 在 DROP TABLE 或 DROP INDEX 操作后移除物理数据。
class_id	oid	包含该对象的目录表的 OID。
object_id	oid	对象的 OID。
object_name	text	对象的完全限定名称。DROP 作业无法引用已经删除的对象。如果已经删除引用的对象，则 object_name 可能为 NULL。
start_time	timestamp with time zone	提交作业的时间戳。

列	类型	说明
update_time	timestamp with time zone	上次更新作业行的时间戳。

sys.jobs 状态值

Status	说明
submitted	任务已提交，但是 Aurora DSQL 尚未开始处理该任务。
processing	Aurora DSQL 正在处理该任务。
failed	任务失败。有关更多信息，请参阅 details 列。
completed	Aurora DSQL 已成功完成任务。

sys.iam_pg_role_mappings 视图

视图 sys.iam_pg_role_mappings 提供有关授予 IAM 用户的权限的信息。例如，如果 DQSLDBConnect 是一个 IAM 角色，该角色为非管理员提供 Aurora DSQL 访问权限，并且向名为 testuser 的用户授予了 DQSLDBConnect 角色和相应的权限，则您可以查询 sys.iam_pg_role_mappings 视图来查看向哪些用户授予了哪些权限。

```
SELECT * FROM sys.iam_pg_role_mappings;
```

有用的系统元数据查询

使用这些查询可以获取表统计数据和系统元数据，而无需执行诸如全表扫描之类的高代价操作。

获取表的估计行计数

要在不执行全表扫描的情况下获取表中的大致行计数，请使用以下查询：

```
SELECT reltuples FROM pg_class WHERE relname = 'table_name';
```

该命令返回的输出类似于下方内容：

```
  reltuples
--------------
 9.993836e+08
```

对于 Aurora DSQL 中的大型表，这种方法比 SELECT COUNT(*) 更高效。

获取当前 Aurora DSQL 主要版本

要获取 Aurora DSQL 集群的当前主要版本，请使用以下查询：

```
SELECT * FROM sys.dsql_major_version();
```

该命令返回的输出类似于下方内容：

```
 dsql_major_version
--------------------
                  1
```

这将返回 Aurora DSQL 中 SQL 连接处于开启状态的主要版本。

获取当前 PostgreSQL 版本

要获取 Aurora DSQL 集群的当前 PostgreSQL 版本，请使用以下查询：

```
SHOW server_version;
```

该命令返回的输出类似于下方内容：

```
 server_version
----------------
 16.13
```

这将返回 Aurora DSQL 中 SQL 连接处于开启状态的 PostgreSQL 版本。

ANALYZE 命令

ANALYZE 命令收集有关数据库中表内容的统计数据，并将结果存储在 pg_stats 系统视图中。随后，查询计划程序使用这些统计数据来帮助确定最有效的查询执行计划。

在 Aurora DSQL 中，您无法在显式事务中运行 ANALYZE 命令。ANALYZE 不受数据库事务超时限制的约束。

为了减少手动干预的需要并将统计数据始终保持为最新，Aurora DSQL 会自动将 ANALYZE 作为后台进程运行。此后台作业会根据在表中观察到的变化率自动触发。它与自上次分析以来已插入、更新或删除的行（元组）数相关联。

ANALYZE 在后台异步运行，可以通过以下查询在系统视图 sys.jobs 中监控其活动：

```
SELECT * FROM sys.jobs WHERE job_type = 'ANALYZE';
```

重要注意事项

Note

ANALYZE 作业的计费方式与 Aurora DSQL 中的其它异步作业相同。修改表时，这可能会间接触发自动后台统计数据收集作业，这可能会因关联的系统级活动而产生计量费用。

自动触发的后台 ANALYZE 作业收集的统计数据类型与手动 ANALYZE 相同，默认情况下会将其应用于用户表。系统表和目录表不包括在此自动化流程中。

使用 Aurora DSQL EXPLAIN 解释计划

Aurora DSQL 使用与 PostgreSQL 类似的 EXPLAIN 计划结构，但增加了一些反映其分布式架构和执行模型的关键内容。

在本文档中，我们将概述 Aurora DSQL EXPLAIN 计划，重点介绍与 PostgreSQL 相比的相似之处和不同之处。我们将介绍 Aurora DSQL 中可用的各种类型的扫描操作，并协助您了解运行查询的成本。

PostgreSQL 与 Aurora DSQL EXPLAIN 计划

Aurora DSQL 建立在 PostgreSQL 数据库基础之上，与 PostgreSQL 共享大多数计划结构，但存在影响查询执行和优化的关键架构差异：

功能	PostgreSQL	Aurora DSQL
数据存储	堆存储	没有堆，所有行都通过唯一标识符编制索引

功能	PostgreSQL	Aurora DSQL
主键	主键索引与表数据是分开的	主键索引是所有额外列均为 INCLUDE 列的表
二级索引	标准二级索引	工作原理与 PostgreSQL 相同，能够包含非键列
筛选功能	索引条件、堆筛选条件	索引条件、存储筛选条件、查询处理器筛选条件
扫描类型	顺序扫描、索引扫描、仅限索引扫描	全面扫描、仅限索引扫描、索引扫描
查询执行	对数据库为本地的	分布式（计算和存储是分开的）

Aurora DSQL 直接按主键顺序存储表数据，而不是存储在单独的堆中。每行都由一个唯一键（通常是主键）标识，这使数据库能够更高效地优化查找。架构差异解释了为什么 Aurora DSQL 经常在 PostgreSQL 可能选择顺序扫描的情况下使用仅限索引扫描。

另一个关键区别是，Aurora DSQL 将计算与存储分开，从而可以在执行路径中更早地应用筛选条件，以减少数据移动并提高性能。

有关在 PostgreSQL 中使用 EXPLAIN 计划的更多信息，请参阅 PostgreSQL EXPLAIN 文档。

Aurora DSQL EXPLAIN 中的关键元素

Aurora DSQL EXPLAIN 计划提供有关如何执行查询的详细信息，包括筛选发生的位置以及从存储中检索哪些列。了解此输出有助于优化查询性能。

索引条件

用于导航索引的条件。效率最高的筛选，可减少扫描的数据。在 Aurora DSQL 中，可以在执行计划的多个层面上应用索引条件。

投影

从存储中检索的列。预测越少意味着性能越好。

存储筛选条件

在存储级别上应用的条件。比查询处理器筛选条件更高效。

查询处理器筛选条件

> 在查询处理器级别应用的条件。需要在筛选之前传输所有数据，这会导致更高的数据移动和处理开销。

Aurora DSQL 中的筛选条件

Aurora DSQL 将计算与存储分开，这意味着在查询执行期间应用筛选条件的位置会对性能产生重大影响。在传输大量数据之前应用筛选的条件可减少延迟并提高效率。越早应用筛选条件，需要处理、移动和扫描的数据就越少，从而加快查询速度。

Aurora DSQL 可以在查询路径的多个阶段应用筛选条件。了解这些阶段是解释查询计划和优化性能的关键。

级别	筛选条件类型	描述
1	索引条件	在扫描索引时应用。限制从存储中读取的数据量，并减少发送到计算层的数据。
2	存储筛选条件	在从存储中读取数据之后但在将其发送到计算之前应用。这里的一个例子是针对索引的包含列的筛选条件。减少数据传输，但不会减少读取量。
3	查询处理器筛选条件	在数据到达计算层后应用。必须先传输所有数据，这会增加延迟和成本。目前，Aurora DSQL 无法对存储执行所有筛选和投影操作，因此某些查询可能会被迫回退到这种类型的筛选。

阅读 Aurora DSQL EXPLAIN 计划

了解如何阅读 EXPLAIN 计划是优化查询性能的关键。在本节中，我们将介绍 Aurora DSQL 查询计划的真实示例，展示不同扫描类型的行为，解释应用筛选条件的位置，并重点介绍优化的机会。

这些示例中使用的示例表

下面的示例引用两个表：`transaction` 和 `account`。

`transaction` 表没有主键，这将导致 Aurora DSQL 在查询表时执行全表扫描。

account 表对 customer_id 具有索引。该索引包括 balance 和 status 作为覆盖列，支持直接从索引中满足某些查询，而无需从基表中读取。但是，索引不包括 created_at，因此，引用该列的查询需要额外的表访问权限。

```
CREATE TABLE transaction (
    account_id uuid,
    transaction_date timestamp,
    description text
);

CREATE TABLE account (
    customer_id uuid,
    balance numeric,
    status varchar,
    created_at timestamp
);

CREATE INDEX ASYNC idx1 ON account (customer_id) INCLUDE (balance, status);
```

全面扫描示例

Aurora DSQL 既有顺序扫描（功能上与 PostgreSQL 相同），也有全面扫描。这两者之间的唯一区别是，全面扫描可以对存储进行额外的筛选。因此，与顺序扫描相比，几乎始终优先选择全面扫描。由于相似性，我们将只介绍更有趣的全面扫描的示例。

全面扫描将主要用于没有主键的表。由于 Aurora DSQL 主键默认情况下为完全覆盖索引，因此在 PostgreSQL 使用顺序扫描的许多情况下，Aurora DSQL 很可能会对主键使用仅限索引扫描。与大多数其它数据库一样，没有索引的表的扩展性会很差。

```
EXPLAIN SELECT account_id FROM transaction WHERE transaction_date > '2025-01-01' AND
 description LIKE '%external%';
```

```
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------
 Full Scan (btree-table) on transaction  (cost=125100.05..177933.38 rows=33333
 width=16)
   Filter: (description ~~ '%external%'::text)
   -> Storage Scan on transaction (cost=12510.05..17793.38 rows=66666 width=16)
        Projections: account_id, description
        Filters: (transaction_date > '2025-01-01 00:00:00'::timestamp without time
 zone)
```

```
        -> B-Tree Scan on transaction (cost=12510.05..17793.38 rows=100000 width=30)
```

此计划显示了在不同阶段应用的两个筛选条件。`transaction_date > '2025-01-01'` 条件应用于存储层，从而减少返回的数据量。稍后，在数据传输之后，在查询处理器中应用 `description LIKE '%external%'` 条件，这会降低效率。将更具选择性的筛选条件推送到存储层或索引层通常可以提高性能。

仅限索引扫描示例

仅限索引扫描是 Aurora DSQL 中最优的扫描类型，因为它们可以最大限度地减少到存储层的往返次数，并且可以进行最多的筛选。但仅因为您看到了仅限索引扫描，并不意味着您拥有最好的计划。由于可能发生的筛选级别各不相同，因此仍要注意可能发生筛选的不同位置，这一点至关重要。

```
EXPLAIN SELECT balance FROM account
WHERE customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'
AND balance > 100
AND status = 'pending';
```

```
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using idx1 on account  (cost=725.05..1025.08 rows=8 width=18)
   Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
   Filter: (balance > '100'::numeric)
   -> Storage Scan on idx1 (cost=12510.05..17793.38 rows=9 width=16)
        Projections: balance
        Filters: ((status)::text = 'pending'::text)
        -> B-Tree Scan on idx1 (cost=12510.05..17793.38 rows=10 width=30)
            Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
```

在此计划中，首先在索引扫描期间评估索引条件 (`customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'`)，这是效率最高的阶段，因为它会限制从存储中读取的数据量。存储筛选条件 `status = 'pending'` 是在读取数据之后但在将数据发送到计算层之前应用的，从而减少了传输的数据量。最后，查询处理器筛选条件 `balance > 100` 在数据移动后最后运行，因此效率最低。其中，索引条件的性能最佳，因为它直接控制扫描的数据量。

索引扫描示例

索引扫描与仅限索引扫描类似，不同之处在于前者需要另一个步骤，即调用基表。由于 Aurora DSQL 可以指定存储筛选条件，因此它能够对索引调用和查找调用指定存储筛选条件。

为了明确这一点，Aurora DSQL 将计划呈现为两个节点。这样，您可以清楚地看到添加包含列对从存储返回的行有多大帮助。

```
EXPLAIN SELECT balance FROM account
WHERE customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'
AND balance > 100
AND status = 'pending'
AND created_at > '2025-01-01';
```

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx1 on account  (cost=728.18..1132.20 rows=3 width=18)
   Filter: (balance > '100'::numeric)
   Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
   -> Storage Scan on idx1 (cost=12510.05..17793.38 rows=8 width=16)
        Projections: balance
        Filters: ((status)::text = 'pending'::text)
        -> B-Tree Scan on account (cost=12510.05..17793.38 rows=10 width=30)
            Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
   -> Storage Lookup on account (cost=12510.05..17793.38 rows=4 width=16)
        Filters: (created_at > '2025-01-01 00:00:00'::timestamp without time zone)
        -> B-Tree Lookup on transaction (cost=12510.05..17793.38 rows=8 width=30)
```

此计划显示了筛选是如何跨多个阶段发生的：

- 有关 `customer_id` 的索引条件会尽早筛选数据。
- 有关 `status` 的存储筛选条件会进一步缩小结果范围，然后再将结果发送到计算中。
- 有关 `balance` 的查询处理器筛选条件将在稍后传输后应用。
- 从基表中提取其它列时，会评估有关 `created_at` 的查找筛选条件。

将常用列添加为 `INCLUDE` 字段通常可以消除这种查找并提高性能。

最佳实践

- 将筛选条件与索引列对齐，以便更早地推送筛选。
- 使用 INCLUDE 列以支持仅限索引扫描并避免查找。
- 在调查性能问题时验证行估计值。Aurora DSQL 根据数据更改率在后台运行 `ANALYZE`，从而自动管理统计数据。如果估计值看起来不准确，则可以手动运行 `ANALYZE` 以立即刷新统计数据。
- 避免对大型表进行未编入索引的查询，以防止代价高昂的全面扫描。

了解 EXPLAIN ANALYZE 中的 DPU

Aurora DSQL 在 `EXPLAIN ANALYZE VERBOSE` 计划输出中提供语句级分布式处理单元（DPU）信息，以便您能够更深入地了解开发过程中的查询成本。此部分将阐释 DPU 的定义以及如何在 `EXPLAIN ANALYZE VERBOSE` 输出中解读它们。

什么是 DPU？

分布式处理单元（DPU）是 Aurora DSQL 中用于量化工作完成量的标准化度量单位，由以下部分组成：

- ComputeDPU – 执行 SQL 查询所花费的时间
- ReadDPU – 从存储中读取数据所使用的资源
- WriteDPU – 向存储中写入数据所使用的资源
- MultiRegionWriteDPU – 用于在多区域配置中将写入内容复制到对等集群的资源。

EXPLAIN ANALYZE VERBOSE 中的 DPU 使用量

Aurora DSQL 对 `EXPLAIN ANALYZE VERBOSE` 进行了扩展，以便在输出末尾包含语句级 DPU 使用量估算值。这可让您即时了解查询成本，并帮助您识别工作负载成本驱动因素、优化查询性能，并更准确地预测资源使用量。

以下示例展示如何解释 EXPLAIN ANALYZE VERBOSE 输出中包含的语句级 DPU 估算值。

示例 1：SELECT 查询

```
EXPLAIN ANALYZE VERBOSE SELECT * FROM test_table;
```

```
QUERY PLAN
----------------------------------------------------
Index Only Scan using test_table_pkey on public.test_table  (cost=125100.05..171100.05
 rows=1000000 width=36) (actual time=2.973..4.482 rows=120 loops=1)
  Output: id, context
  -> Storage Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000 width=36)
 (actual rows=120 loops=1)
      Projections: id, context
      -> B-Tree Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000
 width=36) (actual rows=120 loops=1)
Query Identifier: qymgw1m77maoe
```

```
Planning Time: 11.415 ms
Execution Time: 4.528 ms
Statement DPU Estimate:
  Compute: 0.01607 DPU
  Read: 0.04312 DPU
  Write: 0.00000 DPU
  Total: 0.05919 DPU
```

在此示例中，SELECT 语句执行了一次仅索引扫描，因此大部分成本来自读取 DPU（0.04312）和计算 DPU（0.01607），前者表示从存储中检索数据所使用的资源，后者反映处理并返回结果所使用的计算资源。由于该查询未修改数据，因此不存在写入 DPU。总 DPU（0.05919）为计算 DPU、读取 DPU 与写入 DPU 的总和。

示例 2：INSERT 查询

```
EXPLAIN ANALYZE VERBOSE INSERT INTO test_table VALUES (1, 'name1'), (2, 'name2'), (3,
 'name3');
```

```
QUERY PLAN
----------------------------------------------------
Insert on public.test_table  (cost=0.00..0.04 rows=0 width=0) (actual time=0.055..0.056
 rows=0 loops=1)
  ->  Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=122) (actual
 time=0.003..0.008 rows=3 loops=1)
        Output: "*VALUES*".column1, "*VALUES*".column2
Query Identifier: jtkjkexhjotbo
Planning Time: 0.068 ms
Execution Time: 0.543 ms
Statement DPU Estimate:
  Compute: 0.01550 DPU
  Read: 0.00307 DPU (Transaction minimum: 0.00375)
  Write: 0.01875 DPU (Transaction minimum: 0.05000)
  Total: 0.03732 DPU
```

此语句主要执行写入操作，因此大部分成本与写入 DPU 相关。计算 DPU（0.01550）表示处理并插入值所使用的计算资源。读取 DPU（0.00307）反映系统轻量级读取操作（如目录查询或索引检查）所使用的资源。

请注意读取 DPU 和写入 DPU 旁边显示的事务最低计费标准。这些值代表单次事务的基准成本，仅在操作涉及数据读取或写入时适用。这并不意味着每个事务都会自动产生 0.00375 读取 DPU 或 0.05 写入 DPU 的费用。相反，这些最低计费标准仅在事务级别进行成本聚合时适用，且仅当该事务内实际发

生读取或写入操作时生效。由于范围差异，EXPLAIN ANALYZE VERBOSE 中的语句级估算值可能与 CloudWatch 或计费数据中报告的事务级指标不完全一致。

利用 DPU 信息进行优化

单语句 DPU 估算为您提供了一种强大的查询优化方式，其价值不仅限于缩短执行时间。常见使用案例包括：

- 成本感知：了解某一查询相对于其他查询的成本高低。
- 架构优化：比较索引或架构变更对性能与资源效率产生的影响。
- 预算规划：基于观测到的 DPU 使用量来估算工作负载成本。
- 查询比较：根据相对 DPU 使用量来评估替代查询方法。

解释 DPU 信息

使用 EXPLAIN ANALYZE VERBOSE 中的 DPU 数据时，请记住以下最佳实践：

- 定向地使用 DPU 数据：将报告的 DPU 值视为了解查询相对成本的依据，而非与 CloudWatch 指标或计费数据完全一致的精确值。预计会出现差异，因为 EXPLAIN ANALYZE VERBOSE 报告的是语句级成本，而 CloudWatch 聚合的是事务级活动。此外，CloudWatch 还包括 EXPLAIN ANALYZE VERBOSE 有意排除的后台操作（例如 ANALYZE 或压缩）和事务开销（BEGIN/COMMIT）。
- 在分布式系统中，DPU 在不同的运行间存在波动是正常现象，并不表示出现错误。缓存、执行计划更改、并发性或数据分布偏移等因素都可能导致同一查询在不同的运行时消耗的资源量存在差异。
- 批量处理小型操作：如果您的工作负载发出许多小型语句，建议将其批量合并为大型操作（不超过 10 MB）。这可减少四舍五入开销，并生成更具参考价值的成本估算值。
- 用于调优，而非计费：EXPLAIN ANALYZE VERBOSE 中的 DPU 数据专为成本感知、查询调优和优化设计，并非计费级指标。要获取权威的成本和使用量数据，请始终以 CloudWatch 指标或月度账单报告为准。

管理 Aurora DSQL 集群

Aurora DSQL 提供了多种配置选项，有助于您建立适合自己需求的数据库基础设施。要设置 Aurora DSQL 集群基础设施，请查看以下各节。

主题

- 配置单区域集群
- 配置多区域集群
- 使用 AWS CloudFormation 配置 Aurora DSQL 集群
- Aurora DSQL 集群生命周期

本指南中讨论的特性和功能可确保 Aurora DSQL 环境将具有更高的韧性和响应能力，并且能够在应用程序增长和演变时为其提供支持。

配置单区域集群

使用 AWS CLI 或您偏好的编程语言（包括 Python、C++、JavaScript、Java、Rust、Ruby、.NET 和 Golang），针对某个 AWS 区域配置和管理集群。AWS CLI 通过 shell 命令提供快速访问，而 AWS 软件开发工具包（SDK）通过原生语言支持实现编程控制。

主题

- 使用 AWS SDK
- 使用 AWS CLI

使用 AWS SDK

这些 AWS SDK 让您能够使用偏好的编程语言，实现对 Aurora DSQL 的程序化访问。以下各节演示如何使用不同编程语言执行常见的集群操作。

创建集群

以下示例演示如何使用不同编程语言来创建单区域集群。

Python

要在单个 AWS 区域中创建集群，请使用以下示例。

```
import boto3


def create_cluster(region):
    try:
        client = boto3.client("dsql", region_name=region)
        tags = {"Name": "Python single region cluster"}
        cluster = client.create_cluster(tags=tags, deletionProtectionEnabled=True)
        print(f"Initiated creation of cluster: {cluster["identifier"]}")

        print(f"Waiting for {cluster["arn"]} to become ACTIVE")
        client.get_waiter("cluster_active").wait(
            identifier=cluster["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        return cluster
    except:
        print("Unable to create cluster")
        raise


def main():
    region = "us-east-1"
    response = create_cluster(region)
    print(f"Created cluster: {response["arn"]}")


if __name__ == "__main__":
    main()
```

C++

以下示例可让您在单个 AWS 区域中创建集群。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/CreateClusterRequest.h>
```

```
#include <aws/dsql/model/GetClusterRequest.h>
#include <iostream>
#include <thread>
#include <chrono>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
 * Creates a single-region cluster in Amazon Aurora DSQL
 */
CreateClusterResult CreateCluster(const Aws::String& region) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Create the cluster
    CreateClusterRequest createClusterRequest;
    createClusterRequest.SetDeletionProtectionEnabled(true);
    createClusterRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    // Add tags
    Aws::Map<Aws::String, Aws::String> tags;
    tags["Name"] = "cpp single region cluster";
    createClusterRequest.SetTags(tags);

    auto createOutcome = client.CreateCluster(createClusterRequest);
    if (!createOutcome.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region << ": "
                  << createOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to create cluster in " + region);
    }

    auto cluster = createOutcome.GetResult();
    std::cout << "Created " << cluster.GetArn() << std::endl;

    return cluster;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
```

```
    {
        try {
            // Define region for the single-region setup
            Aws::String region = "us-east-1";

            auto cluster = CreateCluster(region);

            std::cout << "Created single region cluster:" << std::endl;
            std::cout << "Cluster ARN: " << cluster.GetArn() << std::endl;
            std::cout << "Cluster Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(cluster.GetStatus()) << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要在单个 AWS 区域中创建集群，请使用以下示例。

```
import { DSQLClient, CreateClusterCommand, waitUntilClusterActive } from "@aws-sdk/
client-dsql";

async function createCluster(region) {

    const client = new DSQLClient({ region });

    try {
        const createClusterCommand = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript single region cluster"
            },
        });
        const response = await client.send(createClusterCommand);

        console.log(`Waiting for cluster ${response.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client,
```

```
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response.identifier
            }
        );
        console.log(`Cluster Id ${response.identifier} is now active`);
        return;
    } catch (error) {
        console.error(`Unable to create cluster in ${region}: `, error.message);
        throw error;
    }
}

async function main() {
    const region = "us-east-1";

    await createCluster(region);
}

main();
```

Java

使用以下示例在单个 AWS 区域中创建集群。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;

import java.time.Duration;
import java.util.Map;

public class CreateCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
```

```
        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            CreateClusterRequest request = CreateClusterRequest.builder()
                    .deletionProtectionEnabled(true)
                    .tags(Map.of("Name", "java single region cluster"))
                    .build();
            CreateClusterResponse cluster = client.createCluster(request);
            System.out.println("Created " + cluster.arn());

            // The DSQL SDK offers a built-in waiter to poll for a cluster's
            // transition to ACTIVE.
            System.out.println("Waiting for cluster to become ACTIVE");
            WaiterResponse<GetClusterResponse> waiterResponse =
 client.waiter().waitUntilClusterActive(
                    getCluster -> getCluster.identifier(cluster.identifier()),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );
            waiterResponse.matched().response().ifPresent(System.out::println);
        }
    }
}
```

Rust

要在单个 AWS 区域中创建集群，请使用以下示例。

```
use aws_config::{BehaviorVersion, Region, load_defaults};
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;
use aws_sdk_dsql::{Client, Config};
use std::collections::HashMap;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
```

```
    let region_provider = Region::new(region);

    let config = load_defaults(BehaviorVersion::latest())
        .region(region_provider)
        .load()
        .await;

    let config = Config::new(&config);

    Client::from_conf(config)
}

/// Create a cluster without delete protection and a name
pub async fn create_cluster(region: &'static str) -> GetClusterOutput {
    let client = dsql_client(region).await;

    let tags = HashMap::from([
        (String::from("Name"), String::from("rust single region cluster")),
    ]);

    println!("Creating cluster in {region}");
    let cluster = client
        .create_cluster()
        .set_tags(Some(tags))
        .deletion_protection_enabled(true)
        .send()
        .await
        .unwrap();

    println!("Created {}", cluster.arn);

    println!("Waiting for {} to become ACTIVE", cluster.arn);
    let cluster_output = client
        .wait_until_cluster_active()
        .identifier(&cluster.identifier)
        .send()
        .await
        .unwrap();

    cluster_output
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
```

```
    let region = "us-east-1";

    let cluster = create_cluster(region).await;

    println!("Created single region cluster:");
    println!("{:#?}", cluster);

    Ok(())
}
```

Ruby

要在单个 AWS 区域中创建集群，请使用以下示例。

```
require "aws-sdk-dsql"
require "pp"

def create_cluster(region)
  client = Aws::DSQL::Client.new(region: region)

  puts "Creating cluster in #{region}"
  cluster = client.create_cluster(
    deletion_protection_enabled: true,
    tags: {
      Name: "ruby single region cluster"
    }
  )
  puts "Created #{cluster.arn}"

  puts "Waiting for #{cluster.arn} to become ACTIVE"
  cluster = client.wait_until(:cluster_active, identifier: cluster.identifier) do |
w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end

  cluster
rescue Aws::Errors::ServiceError => e
  abort "Failed to create cluster: #{e.message}"
end

def main
  region = "us-east-1"
```

```
    cluster = create_cluster(region)

    puts "Created single region cluster:"
    pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

.NET

要在单个 AWS 区域中创建集群，请使用以下示例。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;
using Amazon.Runtime.Credentials;
using Amazon.Runtime.Endpoints;

namespace DSQLExamples.examples
{
    public class CreateCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = new DefaultAWSCredentialsChain().GetCredentials();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Create a cluster with deletion protection enabled and a name tag.
```

```
        /// </summary>
        public static async Task<CreateClusterResponse> Create(RegionEndpoint
 region)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp single region cluster" }
                };

                var createClusterRequest = new CreateClusterRequest
                {
                    DeletionProtectionEnabled = true,
                    Tags = tags
                };

                var cluster = await client.CreateClusterAsync(createClusterRequest);
                Console.WriteLine($"Created {cluster.Arn}");

                return cluster;
            }
        }

        public static async Task Main()
        {
            var region = RegionEndpoint.USEast1;

            var cluster = await Create(region);

            Console.WriteLine("Created single region cluster:");
            Console.WriteLine($"Cluster ARN: {cluster.Arn}");
        }
    }
}
```

Golang

要在单个 AWS 区域中创建集群，请使用以下示例。

```
package main

import (
 "context"
```

```
 "fmt"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func CreateCluster(ctx context.Context, region string) error {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 client := dsql.NewFromConfig(cfg)

 deleteProtect := true

 input := &dsql.CreateClusterInput{
  DeletionProtectionEnabled: &deleteProtect,
  Tags: map[string]string{
   "Name": "go single-region cluster",
  },
 }

 clusterProperties, err := client.CreateCluster(context.Background(), input)

 if err != nil {
  return fmt.Errorf("failed to create cluster. %v", err)
 }

 // Create the waiter with our custom options
 waiter := dsql.NewClusterActiveWaiter(client, func(o
 *dsql.ClusterActiveWaiterOptions) {
  o.MaxDelay = 30 * time.Second
  o.MinDelay = 10 * time.Second
  o.LogWaitAttempts = true
 })

 // Create the input for the clusterProperties to monitor
 clusterInput := &dsql.GetClusterInput{
  Identifier: clusterProperties.Identifier,
```

```
    }

    fmt.Printf("Waiting for cluster %s to become ACTIVE\n", *clusterProperties.Arn)
    err = waiter.Wait(ctx, clusterInput, 5*time.Minute)
    if err != nil {
     return fmt.Errorf("error waiting for cluster to become active: %w", err)
    }

    fmt.Printf("Created single region cluster: %s\n", *clusterProperties.Arn)
    return nil
}

func main() {
 // Set up context with timeout
 ctx, cancel := context.WithTimeout(context.Background(), 10*time.Minute)
 defer cancel()

 err := CreateCluster(ctx, "us-east-1")
 if err != nil {
  fmt.Printf("failed to create cluster: %v", err)
  panic(err)
 }

}
```

获取集群

以下示例演示如何使用不同编程语言来获取有关单区域集群的信息。

Python

要获取有关单区域集群的信息，请使用以下示例。

```
import boto3
from datetime import datetime
import json


def get_cluster(region, identifier):
    try:
        client = boto3.client("dsql", region_name=region)
        return client.get_cluster(identifier=identifier)
    except:
```

```
        print(f"Unable to get cluster {identifier} in region {region}")
        raise


def main():
    region = "us-east-1"
    cluster_id = "<your cluster id>"
    response = get_cluster(region, cluster_id)

    print(json.dumps(response, indent=2, default=lambda obj: obj.isoformat() if
 isinstance(obj, datetime) else None))


if __name__ == "__main__":
    main()
```

C++

使用以下示例可获取有关单区域集群的信息。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/GetClusterRequest.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
 * Retrieves information about a cluster in Amazon Aurora DSQL
 */
GetClusterResult GetCluster(const Aws::String& region, const Aws::String&
 identifier) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Get the cluster
    GetClusterRequest getClusterRequest;
    getClusterRequest.SetIdentifier(identifier);
```

```
    auto getOutcome = client.GetCluster(getClusterRequest);
    if (!getOutcome.IsSuccess()) {
        std::cerr << "Failed to retrieve cluster " << identifier << " in " << region
 << ": "
                  << getOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to retrieve cluster " + identifier + " in
 region " + region);
    }

    return getOutcome.GetResult();
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define region and cluster ID
            Aws::String region = "us-east-1";
            Aws::String clusterId = "<your cluster id>";

            auto cluster = GetCluster(region, clusterId);

            // Print cluster details
            std::cout << "Cluster Details:" << std::endl;
            std::cout << "ARN: " << cluster.GetArn() << std::endl;
            std::cout << "Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(cluster.GetStatus()) << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要获取有关单区域集群的信息，请使用以下示例。

```
import { DSQLClient, GetClusterCommand } from "@aws-sdk/client-dsql";
```

```
async function getCluster(region, clusterId) {

  const client = new DSQLClient({ region });

  const getClusterCommand = new GetClusterCommand({
    identifier: clusterId,
  });

  try {
    return await client.send(getClusterCommand);
  } catch (error) {
    if (error.name === "ResourceNotFoundException") {
      console.log("Cluster ID not found or deleted");
    }
    throw error;
  }
}

async function main() {
  const region = "us-east-1";
  const clusterId = "<CLUSTER_ID>";

  const response = await getCluster(region, clusterId);
  console.log("Cluster: ", response);
}

main();
```

Java

以下示例可用于获取有关单区域集群的信息。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

public class GetCluster {
```

```
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            GetClusterResponse cluster = client.getCluster(r ->
 r.identifier(clusterId));
            System.out.println(cluster);
        } catch (ResourceNotFoundException e) {
            System.out.printf("Cluster %s not found in %s%n", clusterId, region);
        }
    }
}
```

Rust

以下示例可用于获取有关单区域集群的信息。

```
use aws_config::load_defaults;
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
```

```
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Get a ClusterResource from DSQL cluster identifier
pub async fn get_cluster(region: &'static str, identifier: &'static str) ->
 GetClusterOutput {
    let client = dsql_client(region).await;
    client
        .get_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap()
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";

    let cluster = get_cluster(region, "<your cluster id>").await;
    println!("{:#?}", cluster);

    Ok(())
}
```

Ruby

以下示例可用于获取有关单区域集群的信息。

```
require "aws-sdk-dsql"
require "pp"

def get_cluster(region, identifier)
  client = Aws::DSQL::Client.new(region: region)
  client.get_cluster(identifier: identifier)
rescue Aws::Errors::ServiceError => e
  abort "Unable to retrieve cluster #{identifier} in region #{region}: #{e.message}"
end

def main
```

```
  region = "us-east-1"
  cluster_id = "<your cluster id>"
  cluster = get_cluster(region, cluster_id)
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

.NET

以下示例可用于获取有关单区域集群的信息。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;

namespace DSQLExamples.examples
{
    public class GetCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Get information about a DSQL cluster.
        /// </summary>
```

```
        public static async Task<GetClusterResponse> Get(RegionEndpoint region,
 string identifier)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var getClusterRequest = new GetClusterRequest
                {
                    Identifier = identifier
                };

                return await client.GetClusterAsync(getClusterRequest);
            }
        }

        private static async Task Main()
        {
            var region = RegionEndpoint.USEast1;
            var clusterId = "<your cluster id>";

            var response = await Get(region, clusterId);
            Console.WriteLine($"Cluster ARN: {response.Arn}");
        }
    }
}
```

Golang

以下示例可用于获取有关单区域集群的信息。

```
package main

import (
 "context"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func GetCluster(ctx context.Context, region, identifier string) (clusterStatus
 *dsql.GetClusterOutput, err error) {
```

```
 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Initialize the DSQL client
 client := dsql.NewFromConfig(cfg)

 input := &dsql.GetClusterInput{
  Identifier: aws.String(identifier),
 }
 clusterStatus, err = client.GetCluster(context.Background(), input)

 if err != nil {
  log.Fatalf("Failed to get cluster: %v", err)
 }

 log.Printf("Cluster ARN: %s", *clusterStatus.Arn)

 return clusterStatus, nil
}

func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
 defer cancel()

 // Example cluster identifier
 identifier := "<CLUSTER_ID>"
 region := "us-east-1"

 _, err := GetCluster(ctx, region, identifier)
 if err != nil {
  log.Fatalf("Failed to get cluster: %v", err)
 }
}
```

更新集群

以下示例演示如何使用不同编程语言来更新单区域集群。

Python

要更新单区域集群，请使用以下示例。

```
import boto3


def update_cluster(region, cluster_id, deletion_protection_enabled):
    try:
        client = boto3.client("dsql", region_name=region)
        return client.update_cluster(identifier=cluster_id,
 deletionProtectionEnabled=deletion_protection_enabled)
    except:
        print("Unable to update cluster")
        raise


def main():
    region = "us-east-1"
    cluster_id = "<your cluster id>"
    deletion_protection_enabled = False
    response = update_cluster(region, cluster_id, deletion_protection_enabled)
    print(f"Updated {response["arn"]} with deletion_protection_enabled:
 {deletion_protection_enabled}")


if __name__ == "__main__":
    main()
```

C++

使用以下示例可更新单区域集群。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/UpdateClusterRequest.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;
```

```
/**
 * Updates a cluster in Amazon Aurora DSQL
 */
UpdateClusterResult UpdateCluster(const Aws::String& region, const
 Aws::Map<Aws::String, Aws::String>& updateParams) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Create update request
    UpdateClusterRequest updateRequest;
    updateRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    // Set identifier (required)
    if (updateParams.find("identifier") != updateParams.end()) {
        updateRequest.SetIdentifier(updateParams.at("identifier"));
    } else {
        throw std::runtime_error("Cluster identifier is required for update
 operation");
    }

    // Set deletion protection if specified
    if (updateParams.find("deletion_protection_enabled") != updateParams.end()) {
        bool deletionProtection = (updateParams.at("deletion_protection_enabled") ==
 "true");
        updateRequest.SetDeletionProtectionEnabled(deletionProtection);
    }

    // Execute the update
    auto updateOutcome = client.UpdateCluster(updateRequest);
    if (!updateOutcome.IsSuccess()) {
        std::cerr << "Failed to update cluster: " <<
 updateOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to update cluster");
    }

    return updateOutcome.GetResult();
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
```

```
        try {
            // Define region and update parameters
            Aws::String region = "us-east-1";
            Aws::String clusterId = "<your cluster id>";

            // Create parameter map
            Aws::Map<Aws::String, Aws::String> updateParams;
            updateParams["identifier"] = clusterId;
            updateParams["deletion_protection_enabled"] = "false";

            auto updatedCluster = UpdateCluster(region, updateParams);

            std::cout << "Updated " << updatedCluster.GetArn() << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要更新单区域集群，请使用以下示例。

```
import { DSQLClient, UpdateClusterCommand } from "@aws-sdk/client-dsql";

export async function updateCluster(region, clusterId, deletionProtectionEnabled) {

  const client = new DSQLClient({ region });

  const updateClusterCommand = new UpdateClusterCommand({
    identifier: clusterId,
    deletionProtectionEnabled: deletionProtectionEnabled
  });

  try {
    return await client.send(updateClusterCommand);
  } catch (error) {
    console.error("Unable to update cluster", error.message);
    throw error;
  }
```

```
}

async function main() {
  const region = "us-east-1";
  const clusterId = "<CLUSTER_ID>";
  const deletionProtectionEnabled = false;

  const response = await updateCluster(region, clusterId,
 deletionProtectionEnabled);
  console.log(`Updated ${response.arn}`);
}

main();
```

Java

使用以下示例可更新单区域集群。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsql.model.UpdateClusterResponse;

public class UpdateCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            UpdateClusterRequest request = UpdateClusterRequest.builder()
                    .identifier(clusterId)
                    .deletionProtectionEnabled(false)
                    .build();
            UpdateClusterResponse cluster = client.updateCluster(request);
            System.out.println("Updated " + cluster.arn());
```

```
        }
    }
}
```

Rust

使用以下示例可更新单区域集群。

```
use aws_config::load_defaults;
use aws_sdk_dsql::operation::update_cluster::UpdateClusterOutput;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Update a DSQL cluster and set delete protection to false. Also add new tags.
pub async fn update_cluster(region: &'static str, identifier: &'static str) ->
 UpdateClusterOutput {
    let client = dsql_client(region).await;
    // Update delete protection
    let update_response = client
        .update_cluster()
        .identifier(identifier)
        .deletion_protection_enabled(false)
        .send()
```

```
        .await
        .unwrap();

    update_response
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";

    let cluster = update_cluster(region, "<your cluster id>").await;
    println!("{:#?}", cluster);

    Ok(())
}
```

Ruby

使用以下示例可更新单区域集群。

```
require "aws-sdk-dsql"

def update_cluster(region, update_params)
  client = Aws::DSQL::Client.new(region: region)
  client.update_cluster(update_params)
rescue Aws::Errors::ServiceError => e
  abort "Unable to update cluster: #{e.message}"
end

def main
  region = "us-east-1"
  cluster_id = "<your cluster id>"
  updated_cluster = update_cluster(region, {
    identifier: cluster_id,
    deletion_protection_enabled: false
  })
  puts "Updated #{updated_cluster.arn}"
end

main if $PROGRAM_NAME == __FILE__
```

.NET

使用以下示例可更新单区域集群。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;

namespace DSQLExamples.examples
{
    public class UpdateCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Update a DSQL cluster and set delete protection to false.
        /// </summary>
        public static async Task<UpdateClusterResponse> Update(RegionEndpoint
 region, string identifier)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var updateClusterRequest = new UpdateClusterRequest
                {
                    Identifier = identifier,
                    DeletionProtectionEnabled = false
                };
```

```
                UpdateClusterResponse response = await
 client.UpdateClusterAsync(updateClusterRequest);
                Console.WriteLine($"Updated {response.Arn}");

                return response;
            }
        }

        private static async Task Main()
        {
            var region = RegionEndpoint.USEast1;
            var clusterId = "<your cluster id>";

            await Update(region, clusterId);
        }
    }
}
```

Golang

使用以下示例可更新单区域集群。

```
package main

import (
 "context"
 "github.com/aws/aws-sdk-go-v2/config"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func UpdateCluster(ctx context.Context, region, id string, deleteProtection bool)
 (clusterStatus *dsql.UpdateClusterOutput, err error) {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Initialize the DSQL client
 client := dsql.NewFromConfig(cfg)
```

```
input := dsql.UpdateClusterInput{
  Identifier:                &id,
  DeletionProtectionEnabled: &deleteProtection,
 }

 clusterStatus, err = client.UpdateCluster(context.Background(), &input)

 if err != nil {
  log.Fatalf("Failed to update cluster: %v", err)
 }

 log.Printf("Cluster updated successfully: %v", clusterStatus.Status)
 return clusterStatus, nil
}

func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
 defer cancel()

 // Example cluster identifier
 identifier := "<CLUSTER_ID>"
 region := "us-east-1"
 deleteProtection := false

 _, err := UpdateCluster(ctx, region, identifier, deleteProtection)
 if err != nil {
  log.Fatalf("Failed to update cluster: %v", err)
 }
}
```

删除集群

以下示例演示如何使用不同编程语言来删除单区域集群。

Python

要在单个 AWS 区域中删除集群，请使用以下示例。

```
import boto3


def delete_cluster(region, identifier):
```

```
    try:
        client = boto3.client("dsql", region_name=region)
        cluster = client.delete_cluster(identifier=identifier)
        print(f"Initiated delete of {cluster["arn"]}")

        print("Waiting for cluster to finish deletion")
        client.get_waiter("cluster_not_exists").wait(
            identifier=cluster["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )
    except:
        print("Unable to delete cluster " + identifier)
        raise


def main():
    region = "us-east-1"
    cluster_id = "<cluster id>"  # Use a placeholder in docs
    delete_cluster(region, cluster_id)
    print(f"Deleted {cluster_id}")


if __name__ == "__main__":
    main()
```

C++

要在单个 AWS 区域中删除集群，请使用以下示例。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/DeleteClusterRequest.h>
#include <aws/dsql/model/GetClusterRequest.h>
#include <iostream>
#include <thread>
#include <chrono>

using namespace Aws;
using namespace Aws::DSQL;
```

```
using namespace Aws::DSQL::Model;

/**
 * Deletes a single-region cluster in Amazon Aurora DSQL
 */
void DeleteCluster(const Aws::String& region, const Aws::String& identifier) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Delete the cluster
    DeleteClusterRequest deleteRequest;
    deleteRequest.SetIdentifier(identifier);
    deleteRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto deleteOutcome = client.DeleteCluster(deleteRequest);
    if (!deleteOutcome.IsSuccess()) {
        std::cerr << "Failed to delete cluster " << identifier << " in " << region
 << ": "
                  << deleteOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to delete cluster " + identifier + " in " +
 region);
    }

    auto cluster = deleteOutcome.GetResult();
    std::cout << "Initiated delete of " << cluster.GetArn() << std::endl;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define region and cluster ID
            Aws::String region = "us-east-1";
            Aws::String clusterId = "<your cluster id>";

            DeleteCluster(region, clusterId);

            std::cout << "Deleted " << clusterId << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
```

```
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要在单个 AWS 区域中删除集群，请使用以下示例。

```
import { DSQLClient, DeleteClusterCommand, waitUntilClusterNotExists } from "@aws-sdk/client-dsql";

async function deleteCluster(region, clusterId) {

  const client = new DSQLClient({ region });

  try {
    const deleteClusterCommand = new DeleteClusterCommand({
      identifier: clusterId,
    });
    const response = await client.send(deleteClusterCommand);

    console.log(`Waiting for cluster ${response.identifier} to finish deletion`);

    await waitUntilClusterNotExists(
      {
        client: client,
        maxWaitTime: 300 // Wait for 5 minutes
      },
      {
        identifier: response.identifier
      }
    );
    console.log(`Cluster Id ${response.identifier} is now deleted`);
    return;
  } catch (error) {
    if (error.name === "ResourceNotFoundException") {
      console.log("Cluster ID not found or already deleted");
    } else {
      console.error("Unable to delete cluster: ", error.message);
    }
    throw error;
  }
```

```
}

async function main() {
  const region = "us-east-1";
  const clusterId = "<CLUSTER_ID>";

  await deleteCluster(region, clusterId);
}

main();
```

Java

要在单个 AWS 区域中删除集群，请使用以下示例。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

import java.time.Duration;

public class DeleteCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";


        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            DeleteClusterResponse cluster = client.deleteCluster(r ->
 r.identifier(clusterId));
            System.out.println("Initiated delete of " + cluster.arn());

            // The DSQL SDK offers a built-in waiter to poll for deletion.
```

```
            System.out.println("Waiting for cluster to finish deletion");
            client.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );
            System.out.println("Deleted " + cluster.arn());
        } catch (ResourceNotFoundException e) {
            System.out.printf("Cluster %s not found in %s%n", clusterId, region);
        }
    }
}
```

Rust

要在单个 AWS 区域中删除集群，请使用以下示例。

```
use aws_config::load_defaults;
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}
```

```
/// Delete a DSQL cluster
pub async fn delete_cluster(region: &'static str, identifier: &'static str) {
    let client = dsql_client(region).await;
    let delete_response = client
        .delete_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap();
    println!("Initiated delete of {}", delete_response.arn);

    println!("Waiting for cluster to finish deletion");
    client
        .wait_until_cluster_not_exists()
        .identifier(identifier)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap();
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";
    let cluster_id = "<cluster to be deleted>";

    delete_cluster(region, cluster_id).await;
    println!("Deleted {cluster_id}");

    Ok(())
}
```

Ruby

要在单个 AWS 区域中删除集群，请使用以下示例。

```
require "aws-sdk-dsql"

def delete_cluster(region, identifier)
  client = Aws::DSQL::Client.new(region: region)
  cluster = client.delete_cluster(identifier: identifier)
  puts "Initiated delete of #{cluster.arn}"

  # The DSQL SDK offers built-in waiters to poll for deletion.
```

```
  puts "Waiting for cluster to finish deletion"
  client.wait_until(:cluster_not_exists, identifier: cluster.identifier) do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end
rescue Aws::Errors::ServiceError => e
  abort "Unable to delete cluster #{identifier} in #{region}: #{e.message}"
end

def main
  region = "us-east-1"
  cluster_id = "<your cluster id>"
  delete_cluster(region, cluster_id)
  puts "Deleted #{cluster_id}"
end

main if $PROGRAM_NAME == __FILE__
```

.NET

要在单个 AWS 区域中删除集群，请使用以下示例。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;

namespace DSQLExamples.examples
{
    public class DeleteSingleRegionCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
```

```
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Delete a DSQL cluster.
        /// </summary>
        public static async Task Delete(RegionEndpoint region, string identifier)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var deleteRequest = new DeleteClusterRequest
                {
                    Identifier = identifier
                };

                var deleteResponse = await client.DeleteClusterAsync(deleteRequest);
                Console.WriteLine($"Initiated deletion of {deleteResponse.Arn}");
            }
        }

        private static async Task Main()
        {
            var region = RegionEndpoint.USEast1;
            var clusterId = "<cluster to be deleted>";

            await Delete(region, clusterId);
        }
    }
}
```

Golang

要在单个 AWS 区域中删除集群，请使用以下示例。

```
package main

import (
 "context"
 "fmt"
 "log"
 "time"
```

```
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func DeleteSingleRegion(ctx context.Context, identifier, region string) error {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Initialize the DSQL client
 client := dsql.NewFromConfig(cfg)

 // Create delete cluster input
 deleteInput := &dsql.DeleteClusterInput{
  Identifier: &identifier,
 }

 // Delete the cluster
 result, err := client.DeleteCluster(ctx, deleteInput)
 if err != nil {
  return fmt.Errorf("failed to delete cluster: %w", err)
 }

 fmt.Printf("Initiated deletion of cluster: %s\n", *result.Arn)

 // Create waiter to check cluster deletion
 waiter := dsql.NewClusterNotExistsWaiter(client, func(options
 *dsql.ClusterNotExistsWaiterOptions) {
  options.MinDelay = 10 * time.Second
  options.MaxDelay = 30 * time.Second
  options.LogWaitAttempts = true
 })

 // Create the input for checking cluster status
 getInput := &dsql.GetClusterInput{
  Identifier: &identifier,
 }

 // Wait for the cluster to be deleted
 fmt.Printf("Waiting for cluster %s to be deleted...\n", identifier)
 err = waiter.Wait(ctx, getInput, 5*time.Minute)
```

```
	if err != nil {
		return fmt.Errorf("error waiting for cluster to be deleted: %w", err)
	}

	fmt.Printf("Cluster %s has been successfully deleted\n", identifier)
	return nil
}

func DeleteCluster(ctx context.Context) {
}

// Example usage in main function
func main() {
	// Your existing setup code for client configuration...

	ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
	defer cancel()

	// Example cluster identifier
	// Need to make sure that cluster does not have delete protection enabled
	identifier := "<CLUSTER_ID>"
	region := "us-east-1"

	err := DeleteSingleRegion(ctx, identifier, region)
	if err != nil {
		log.Fatalf("Failed to delete cluster: %v", err)
	}

}
```

有关更多代码示例及相关示例，请访问 Aurora DSQL 示例 GitHub 存储库。

使用 AWS CLI

AWS CLI 提供了用于管理 Aurora DSQL 集群的命令行界面。以下示例演示常见的集群管理操作：

创建集群

使用 create-cluster 命令创建集群。

Note

集群创建是一个异步操作。调用 GetCluster API，直到状态变为 ACTIVE。集群变为活动状态后，即可连接到该集群。

Example命令

```
aws dsql create-cluster --region us-east-1
```

Note

要在创建过程中禁用删除保护，请包括 --no-deletion-protection-enabled 标志。

Example响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "CREATING",
    "creationTime": "2024-05-25T16:56:49.784000-07:00",
    "deletionProtectionEnabled": true,
    "tag": {},
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    }
}
```

描述集群

使用 get-cluster 命令获取有关集群的信息。

Example命令

```
aws dsql get-cluster \
  --region us-east-1 \
  --identifier your_cluster_id
```

Example响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2024-11-27T00:32:14.434000-08:00",
    "deletionProtectionEnabled": false,
    "encryptionDetails": {
        "encryptionType": "CUSTOMER_MANAGED_KMS_KEY",
        "kmsKeyArn": "arn:aws:kms:us-east-1:111122223333:key/123a456b-c789-01de-2f34-
g5hi6j7k8lm9",
        "encryptionStatus": "ENABLED"
    }
}
```

更新集群

使用 update-cluster 命令更新现有集群。

> **Note**
>
> 更新是异步操作。调用 GetCluster API，直到状态变为 ACTIVE，以查看您的更改。

Example命令

```
aws dsql update-cluster \
  --region us-east-1 \
  --no-deletion-protection-enabled \
  --identifier your_cluster_id
```

Example响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "UPDATING",
    "creationTime": "2024-05-24T09:15:32.708000-07:00"
}
```

删除集群

使用 delete-cluster 命令删除现有集群。

Note

您只能删除禁用了删除保护的集群。默认情况下，创建新集群时会启用删除保护。

Example命令

```
aws dsql delete-cluster \
  --region us-east-1 \
  --identifier your_cluster_id
```

Example响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "DELETING",
    "creationTime": "2024-05-24T09:16:43.778000-07:00"
}
```

列出集群

使用 list-clusters 命令列出您的集群。

Example命令

```
aws dsql list-clusters --region us-east-1
```

Example响应

```
{
    "clusters": [
        {
            "identifier": "abc0def1baz2quux3quux4quuux",
            "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
abc0def1baz2quux3quux4quuux"
        },
        {
```

```
            "identifier": "abc0def1baz2quux3quux5quuuux",
            "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
abc0def1baz2quux3quux5quuuux"
        }
    ]
}
```

配置多区域集群

使用 AWS CLI 或您偏好的编程语言（包括 Python、C++、JavaScript、Java、Rust、Ruby、.NET 和 Golang），跨多个 AWS 区域配置和管理集群。AWS CLI 通过 shell 命令提供快速访问，而 AWS SDK 通过原生语言支持实现编程控制。

主题

- 使用 AWS SDK
- 使用 AWS CLI

使用 AWS SDK

这些 AWS SDK 让您能够使用偏好的编程语言，实现对 Aurora DSQL 的程序化访问。以下各节演示如何使用不同编程语言执行常见的集群操作。

创建集群

以下示例演示如何使用不同编程语言来创建多区域集群。

Python

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
import boto3


def create_multi_region_clusters(region_1, region_2, witness_region):
    try:
        client_1 = boto3.client("dsql", region_name=region_1)
        client_2 = boto3.client("dsql", region_name=region_2)

        # We can only set the witness region for the first cluster
        cluster_1 = client_1.create_cluster(
```

```
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region},
            tags={"Name": "Python multi region cluster"}
        )
        print(f"Created {cluster_1["arn"]}")

        # For the second cluster we can set witness region and designate cluster_1
 as a peer
        cluster_2 = client_2.create_cluster(
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_1["arn"]]},
            tags={"Name": "Python multi region cluster"}
        )

        print(f"Created {cluster_2["arn"]}")
        # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
        client_1.update_cluster(
            identifier=cluster_1["identifier"],
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_2["arn"]]}
        )
        print(f"Added {cluster_2["arn"]} as a peer of {cluster_1["arn"]}")

        # Now that multiRegionProperties is fully defined for both clusters
        # they'll begin the transition to ACTIVE
        print(f"Waiting for {cluster_1["arn"]} to become ACTIVE")
        client_1.get_waiter("cluster_active").wait(
            identifier=cluster_1["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        print(f"Waiting for {cluster_2["arn"]} to become ACTIVE")
        client_2.get_waiter("cluster_active").wait(
            identifier=cluster_2["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )
```

```
        return (cluster_1, cluster_2)

    except:
        print("Unable to create cluster")
        raise


def main():
    region_1 = "us-east-1"
    region_2 = "us-east-2"
    witness_region = "us-west-2"
    (cluster_1, cluster_2) = create_multi_region_clusters(region_1, region_2,
 witness_region)
    print("Created multi region clusters:")
    print("Cluster id: " + cluster_1['arn'])
    print("Cluster id: " + cluster_2['arn'])


if __name__ == "__main__":
    main()
```

C++

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/CreateClusterRequest.h>
#include <aws/dsql/model/UpdateClusterRequest.h>
#include <aws/dsql/model/MultiRegionProperties.h>
#include <aws/dsql/model/GetClusterRequest.h>
#include <iostream>
#include <thread>
#include <chrono>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
 * Creates multi-region clusters in Amazon Aurora DSQL
 */
```

```
std::pair<CreateClusterResult, CreateClusterResult> CreateMultiRegionClusters(
    const Aws::String& region1,
    const Aws::String& region2,
    const Aws::String& witnessRegion) {

    // Create clients for each region
    DSQL::DSQLClientConfiguration clientConfig1;
    clientConfig1.region = region1;
    DSQL::DSQLClient client1(clientConfig1);

    DSQL::DSQLClientConfiguration clientConfig2;
    clientConfig2.region = region2;
    DSQL::DSQLClient client2(clientConfig2);

    std::cout << "Creating cluster in " << region1 << std::endl;

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region
    MultiRegionProperties multiRegionProps1;
    multiRegionProps1.SetWitnessRegion(witnessRegion);
    createClusterRequest1.SetMultiRegionProperties(multiRegionProps1);

    // Add tags
    Aws::Map<Aws::String, Aws::String> tags;
    tags["Name"] = "cpp multi region cluster 1";
    createClusterRequest1.SetTags(tags);
    createClusterRequest1.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome1 = client1.CreateCluster(createClusterRequest1);
    if (!createOutcome1.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region1 << ": "
                  << createOutcome1.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }

    auto cluster1 = createOutcome1.GetResult();
    std::cout << "Created " << cluster1.GetArn() << std::endl;

    // Create second cluster
    std::cout << "Creating cluster in " << region2 << std::endl;

    CreateClusterRequest createClusterRequest2;
```

```
    createClusterRequest2.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region and cluster1 as peer
    MultiRegionProperties multiRegionProps2;
    multiRegionProps2.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> clusters;
    clusters.push_back(cluster1.GetArn());
    multiRegionProps2.SetClusters(clusters);

    tags["Name"] = "cpp multi region cluster 2";
    createClusterRequest2.SetMultiRegionProperties(multiRegionProps2);
    createClusterRequest2.SetTags(tags);
    createClusterRequest2.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome2 = client2.CreateCluster(createClusterRequest2);
    if (!createOutcome2.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region2 << ": "
                  << createOutcome2.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }

    auto cluster2 = createOutcome2.GetResult();
    std::cout << "Created " << cluster2.GetArn() << std::endl;

    // Now that we know the cluster2 arn we can set it as a peer of cluster1
    UpdateClusterRequest updateClusterRequest;
    updateClusterRequest.SetIdentifier(cluster1.GetIdentifier());

    MultiRegionProperties updatedProps;
    updatedProps.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> updatedClusters;
    updatedClusters.push_back(cluster2.GetArn());
    updatedProps.SetClusters(updatedClusters);

    updateClusterRequest.SetMultiRegionProperties(updatedProps);
    updateClusterRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto updateOutcome = client1.UpdateCluster(updateClusterRequest);
    if (!updateOutcome.IsSuccess()) {
        std::cerr << "Failed to update cluster in " << region1 << ": "
                  << updateOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to update multi-region clusters");
```

```
    }

    std::cout << "Added " << cluster2.GetArn() << " as a peer of " <<
 cluster1.GetArn() << std::endl;

    return std::make_pair(cluster1, cluster2);
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define regions for the multi-region setup
            Aws::String region1 = "us-east-1";
            Aws::String region2 = "us-east-2";
            Aws::String witnessRegion = "us-west-2";

            auto [cluster1, cluster2] = CreateMultiRegionClusters(region1, region2,
 witnessRegion);

            std::cout << "Created multi region clusters:" << std::endl;
            std::cout << "Cluster 1 ARN: " << cluster1.GetArn() << std::endl;
            std::cout << "Cluster 2 ARN: " << cluster2.GetArn() << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
import { DSQLClient, CreateClusterCommand, UpdateClusterCommand,
 waitUntilClusterActive } from "@aws-sdk/client-dsql";

async function createMultiRegionCluster(region1, region2, witnessRegion) {

    const client1 = new DSQLClient({ region: region1 });
    const client2 = new DSQLClient({ region: region2 });
```

```
    try {
        // We can only set the witness region for the first cluster
        console.log(`Creating cluster in ${region1}`);
        const createClusterCommand1 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 1"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion
            }
        });
        const response1 = await client1.send(createClusterCommand1);
        console.log(`Created ${response1.arn}`);

        // For the second cluster we can set witness region and designate the first
 cluster as a peer
        console.log(`Creating cluster in ${region2}`);
        const createClusterCommand2 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 2"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response1.arn]
            }
        });
        const response2 = await client2.send(createClusterCommand2);
        console.log(`Created ${response2.arn}`);

        // Now that we know the second cluster arn we can set it as a peer of the
 first cluster
        const updateClusterCommand = new UpdateClusterCommand({
            identifier: response1.identifier,
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response2.arn]
            }
        });
        await client1.send(updateClusterCommand);
        console.log(`Added ${response2.arn} as a peer of ${response1.arn}`);
```

```
        // Now that multiRegionProperties is fully defined for both clusters they'll
 begin the transition to ACTIVE
        console.log(`Waiting for cluster ${response1.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client1,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response1.identifier
            }
        );
        console.log(`Cluster 1 is now active`);

        console.log(`Waiting for cluster ${response2.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client2,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response2.identifier
            }
        );
        console.log(`Cluster 2 is now active`);
        console.log("The multi region clusters are now active");
        return;
    } catch (error) {
        console.error("Failed to create cluster: ", error.message);
        throw error;
    }
}

async function main() {
    const region1 = "us-east-1";
    const region2 = "us-east-2";
    const witnessRegion = "us-west-2";

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();
```

Java

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.DsqlClientBuilder;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;

import java.time.Duration;
import java.util.Map;

public class CreateMultiRegionCluster {

    public static void main(String[] args) {
        Region region1 = Region.US_EAST_1;
        Region region2 = Region.US_EAST_2;
        Region witnessRegion = Region.US_WEST_2;

        DsqlClientBuilder clientBuilder = DsqlClient.builder()
                .credentialsProvider(DefaultCredentialsProvider.create());

        try (
            DsqlClient client1 = clientBuilder.region(region1).build();
            DsqlClient client2 = clientBuilder.region(region2).build()
        ) {
            // We can only set the witness region for the first cluster
            System.out.println("Creating cluster in " + region1);
            CreateClusterRequest request1 = CreateClusterRequest.builder()
                    .deletionProtectionEnabled(true)
                    .multiRegionProperties(mrp ->
 mrp.witnessRegion(witnessRegion.toString()))
                    .tags(Map.of("Name", "java multi region cluster"))
                    .build();
            CreateClusterResponse cluster1 = client1.createCluster(request1);
            System.out.println("Created " + cluster1.arn());
```

```
            // For the second cluster we can set the witness region and designate
            // cluster1 as a peer.
            System.out.println("Creating cluster in " + region2);
            CreateClusterRequest request2 = CreateClusterRequest.builder()
                    .deletionProtectionEnabled(true)
                    .multiRegionProperties(mrp ->

 mrp.witnessRegion(witnessRegion.toString()).clusters(cluster1.arn())
                    )
                    .tags(Map.of("Name", "java multi region cluster"))
                    .build();
            CreateClusterResponse cluster2 = client2.createCluster(request2);
            System.out.println("Created " + cluster2.arn());

            // Now that we know the cluster2 ARN we can set it as a peer of cluster1
            UpdateClusterRequest updateReq = UpdateClusterRequest.builder()
                    .identifier(cluster1.identifier())
                    .multiRegionProperties(mrp ->

 mrp.witnessRegion(witnessRegion.toString()).clusters(cluster2.arn())
                    )
                    .build();
            client1.updateCluster(updateReq);
            System.out.printf("Added %s as a peer of %s%n", cluster2.arn(),
 cluster1.arn());

            // Now that MultiRegionProperties is fully defined for both clusters
 they'll begin
            // the transition to ACTIVE.
            System.out.printf("Waiting for cluster %s to become ACTIVE%n",
 cluster1.arn());
            GetClusterResponse activeCluster1 =
 client1.waiter().waitUntilClusterActive(
                    getCluster -> getCluster.identifier(cluster1.identifier()),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            ).matched().response().orElseThrow();

            System.out.printf("Waiting for cluster %s to become ACTIVE%n",
 cluster2.arn());
            GetClusterResponse activeCluster2 =
 client2.waiter().waitUntilClusterActive(
```

```
                    getCluster -> getCluster.identifier(cluster2.identifier()),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            ).matched().response().orElseThrow();

            System.out.println("Created multi region clusters:");
            System.out.println(activeCluster1);
            System.out.println(activeCluster2);
        }
    }
}
```

Rust

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
use aws_config::{BehaviorVersion, Region, load_defaults};
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;
use aws_sdk_dsql::types::MultiRegionProperties;
use aws_sdk_dsql::{Client, Config};
use std::collections::HashMap;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}
```

```
/// Create a cluster without delete protection and a name
pub async fn create_multi_region_clusters(
    region_1: &'static str,
    region_2: &'static str,
    witness_region: &'static str,
) -> (GetClusterOutput, GetClusterOutput) {
    let client_1 = dsql_client(region_1).await;
    let client_2 = dsql_client(region_2).await;

    let tags = HashMap::from([(
        String::from("Name"),
        String::from("rust multi region cluster"),
    )]);

    // We can only set the witness region for the first cluster
    println!("Creating cluster in {region_1}");
    let cluster_1 = client_1
        .create_cluster()
        .set_tags(Some(tags.clone()))
        .deletion_protection_enabled(true)
        .multi_region_properties(
            MultiRegionProperties::builder()
                .witness_region(witness_region)
                .build(),
        )
        .send()
        .await
        .unwrap();
    let cluster_1_arn = &cluster_1.arn;
    println!("Created {cluster_1_arn}");

    // For the second cluster we can set witness region and designate cluster_1 as a
 peer
    println!("Creating cluster in {region_2}");
    let cluster_2 = client_2
        .create_cluster()
        .set_tags(Some(tags))
        .deletion_protection_enabled(true)
        .multi_region_properties(
            MultiRegionProperties::builder()
                .witness_region(witness_region)
                .clusters(&cluster_1.arn)
                .build(),
```

```
        )
        .send()
        .await
        .unwrap();
    let cluster_2_arn = &cluster_2.arn;
    println!("Created {cluster_2_arn}");

    // Now that we know the cluster_2 arn we can set it as a peer of cluster_1
    client_1
        .update_cluster()
        .identifier(&cluster_1.identifier)
        .multi_region_properties(
            MultiRegionProperties::builder()
                .witness_region(witness_region)
                .clusters(&cluster_2.arn)
                .build(),
        )
        .send()
        .await
        .unwrap();
    println!("Added {cluster_2_arn} as a peer of {cluster_1_arn}");

    // Now that the multi-region properties are fully defined for both clusters
    // they'll begin the transition to ACTIVE
    println!("Waiting for {cluster_1_arn} to become ACTIVE");
    let cluster_1_output = client_1
        .wait_until_cluster_active()
        .identifier(&cluster_1.identifier)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap()
        .into_result()
        .unwrap();

    println!("Waiting for {cluster_2_arn} to become ACTIVE");
    let cluster_2_output = client_2
        .wait_until_cluster_active()
        .identifier(&cluster_2.identifier)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap()
        .into_result()
        .unwrap();
```

```
    (cluster_1_output, cluster_2_output)
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region_1 = "us-east-1";
    let region_2 = "us-east-2";
    let witness_region = "us-west-2";

    let (cluster_1, cluster_2) =
        create_multi_region_clusters(region_1, region_2, witness_region).await;

    println!("Created multi region clusters:");
    println!("{:#?}", cluster_1);
    println!("{:#?}", cluster_2);

    Ok(())
}
```

Ruby

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
require "aws-sdk-dsql"
require "pp"

def create_multi_region_clusters(region_1, region_2, witness_region)
  client_1 = Aws::DSQL::Client.new(region: region_1)
  client_2 = Aws::DSQL::Client.new(region: region_2)

  # We can only set the witness region for the first cluster
  puts "Creating cluster in #{region_1}"
  cluster_1 = client_1.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region
    },
    tags: {
      Name: "ruby multi region cluster"
    }
  )
  puts "Created #{cluster_1.arn}"
```

```
  # For the second cluster we can set witness region and designate cluster_1 as a
 peer
  puts "Creating cluster in #{region_2}"
  cluster_2 = client_2.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_1.arn ]
    },
    tags: {
      Name: "ruby multi region cluster"
    }
  )
  puts "Created #{cluster_2.arn}"

  # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
  client_1.update_cluster(
    identifier: cluster_1.identifier,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_2.arn ]
    }
  )
  puts "Added #{cluster_2.arn} as a peer of #{cluster_1.arn}"

  # Now that multi_region_properties is fully defined for both clusters
  # they'll begin the transition to ACTIVE
  puts "Waiting for #{cluster_1.arn} to become ACTIVE"
  cluster_1 = client_1.wait_until(:cluster_active, identifier: cluster_1.identifier)
 do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end

  puts "Waiting for #{cluster_2.arn} to become ACTIVE"
  cluster_2 = client_2.wait_until(:cluster_active, identifier: cluster_2.identifier)
 do |w|
    w.max_attempts = 30
    w.delay = 10
  end

  [ cluster_1, cluster_2 ]
rescue Aws::Errors::ServiceError => e
```

```
  abort "Failed to create multi-region clusters: #{e.message}"
end

def main
  region_1 = "us-east-1"
  region_2 = "us-east-2"
  witness_region = "us-west-2"

  cluster_1, cluster_2 = create_multi_region_clusters(region_1, region_2,
 witness_region)

  puts "Created multi region clusters:"
  pp cluster_1
  pp cluster_2
end

main if $PROGRAM_NAME == __FILE__
```

Golang

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
package main

import (
 "context"
 "fmt"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/dsql"
 dtypes "github.com/aws/aws-sdk-go-v2/service/dsql/types"
)

func CreateMultiRegionClusters(ctx context.Context, witness, region1, region2
 string) error {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region1))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }
```

```
// Create a DSQL region 1 client
client := dsql.NewFromConfig(cfg)

cfg2, err := config.LoadDefaultConfig(ctx, config.WithRegion(region2))
if err != nil {
 log.Fatalf("Failed to load AWS configuration: %v", err)
}

// Create a DSQL region 2 client
client2 := dsql.NewFromConfig(cfg2, func(o *dsql.Options) {
 o.Region = region2
})

// Create cluster
deleteProtect := true

// We can only set the witness region for the first cluster
input := &dsql.CreateClusterInput{
 DeletionProtectionEnabled: &deleteProtect,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String(witness),
 },
 Tags: map[string]string{
  "Name": "go multi-region cluster",
 },
}

clusterProperties, err := client.CreateCluster(context.Background(), input)

if err != nil {
 return fmt.Errorf("failed to create first cluster: %v", err)
}

// create second cluster
cluster2Arns := []string{*clusterProperties.Arn}

// For the second cluster we can set witness region and designate the first cluster
as a peer
input2 := &dsql.CreateClusterInput{
 DeletionProtectionEnabled: &deleteProtect,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String("us-west-2"),
  Clusters:      cluster2Arns,
```

```
 },
 Tags: map[string]string{
  "Name": "go multi-region cluster",
 },
}

clusterProperties2, err := client2.CreateCluster(context.Background(), input2)

if err != nil {
 return fmt.Errorf("failed to create second cluster: %v", err)
}

// link initial cluster to second cluster
cluster1Arns := []string{*clusterProperties2.Arn}

// Now that we know the second cluster arn we can set it as a peer of the first
cluster
input3 := dsql.UpdateClusterInput{
 Identifier: clusterProperties.Identifier,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String("us-west-2"),
  Clusters:      cluster1Arns,
 }}

_, err = client.UpdateCluster(context.Background(), &input3)

if err != nil {
 return fmt.Errorf("failed to update cluster to associate with first cluster. %v",
err)
}

// Create the waiter with our custom options for first cluster
waiter := dsql.NewClusterActiveWaiter(client, func(o
*dsql.ClusterActiveWaiterOptions) {
 o.MaxDelay = 30 * time.Second // Creating a multi-region cluster can take a few
minutes
 o.MinDelay = 10 * time.Second
 o.LogWaitAttempts = true
})

// Now that multiRegionProperties is fully defined for both clusters
// they'll begin the transition to ACTIVE

// Create the input for the clusterProperties to monitor for first cluster
```

```
 getInput := &dsql.GetClusterInput{
  Identifier: clusterProperties.Identifier,
 }

 // Wait for the first cluster to become active
 fmt.Printf("Waiting for first cluster %s to become active...\n",
 *clusterProperties.Identifier)
 err = waiter.Wait(ctx, getInput, 5*time.Minute)
 if err != nil {
  return fmt.Errorf("error waiting for first cluster to become active: %w", err)
 }

 // Create the waiter with our custom options
 waiter2 := dsql.NewClusterActiveWaiter(client2, func(o
 *dsql.ClusterActiveWaiterOptions) {
  o.MaxDelay = 30 * time.Second // Creating a multi-region cluster can take a few
 minutes
  o.MinDelay = 10 * time.Second
  o.LogWaitAttempts = true
 })

 // Create the input for the clusterProperties to monitor for second
 getInput2 := &dsql.GetClusterInput{
  Identifier: clusterProperties2.Identifier,
 }

 // Wait for the second cluster to become active
 fmt.Printf("Waiting for second cluster %s to become active...\n",
 *clusterProperties2.Identifier)
 err = waiter2.Wait(ctx, getInput2, 5*time.Minute)
 if err != nil {
  return fmt.Errorf("error waiting for second cluster to become active: %w", err)
 }

 fmt.Printf("Cluster %s is now active\n", *clusterProperties.Identifier)
 fmt.Printf("Cluster %s is now active\n", *clusterProperties2.Identifier)
 return nil
}

// Example usage in main function
func main() {
 // Set up context with timeout
 ctx, cancel := context.WithTimeout(context.Background(), 10*time.Minute)
 defer cancel()
```

```
    err := CreateMultiRegionClusters(ctx, "us-west-2", "us-east-1", "us-east-2")
    if err != nil {
     fmt.Printf("failed to create multi-region clusters: %v", err)
     panic(err)
    }

}
```

.NET

要创建多区域集群，请使用以下示例。创建多区域集群可能需要一些时间。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;
using Amazon.Runtime.Endpoints;

namespace DSQLExamples.examples
{
    public class CreateMultiRegionClusters
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region,
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
```

```
        /// Create multi-region clusters with a witness region.
        /// </summary>
        public static async Task<(CreateClusterResponse, CreateClusterResponse)>
 Create(
            RegionEndpoint region1,
            RegionEndpoint region2,
            RegionEndpoint witnessRegion)
        {
            using (var client1 = await CreateDSQLClient(region1))
            using (var client2 = await CreateDSQLClient(region2))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp multi region cluster" }
                };

                // We can only set the witness region for the first cluster
                var createClusterRequest1 = new CreateClusterRequest
                {
                    DeletionProtectionEnabled = true,
                    Tags = tags,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName
                    }
                };

                var cluster1 = await
 client1.CreateClusterAsync(createClusterRequest1);
                var cluster1Arn = cluster1.Arn;
                Console.WriteLine($"Initiated creation of {cluster1Arn}");

                // For the second cluster we can set witness region and designate
 cluster1 as a peer
                var createClusterRequest2 = new CreateClusterRequest
                {
                    DeletionProtectionEnabled = true,
                    Tags = tags,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName,
                        Clusters = new List<string> { cluster1.Arn }
                    }
                };
```

```
                var cluster2 = await
 client2.CreateClusterAsync(createClusterRequest2);
                var cluster2Arn = cluster2.Arn;
                Console.WriteLine($"Initiated creation of {cluster2Arn}");

                // Now that we know the cluster2 arn we can set it as a peer of
 cluster1
                var updateClusterRequest = new UpdateClusterRequest
                {
                    Identifier = cluster1.Identifier,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName,
                        Clusters = new List<string> { cluster2.Arn }
                    }
                };

                await client1.UpdateClusterAsync(updateClusterRequest);
                Console.WriteLine($"Added {cluster2Arn} as a peer of
 {cluster1Arn}");

                return (cluster1, cluster2);
            }
        }

        private static async Task Main()
        {
            var region1 = RegionEndpoint.USEast1;
            var region2 = RegionEndpoint.USEast2;
            var witnessRegion = RegionEndpoint.USWest2;

            var (cluster1, cluster2) = await Create(region1, region2,
 witnessRegion);

            Console.WriteLine("Created multi region clusters:");
            Console.WriteLine($"Cluster 1: {cluster1.Arn}");
            Console.WriteLine($"Cluster 2: {cluster2.Arn}");
        }
    }
}
```

获取集群

以下示例演示如何使用不同编程语言来获取有关多区域集群的信息。

Python

要获取有关多区域集群的信息，请使用以下示例。

```
import boto3
from datetime import datetime
import json


def get_cluster(region, identifier):
    try:
        client = boto3.client("dsql", region_name=region)
        return client.get_cluster(identifier=identifier)
    except:
        print(f"Unable to get cluster {identifier} in region {region}")
        raise


def main():
    region = "us-east-1"
    cluster_id = "<your cluster id>"
    response = get_cluster(region, cluster_id)

    print(json.dumps(response, indent=2, default=lambda obj: obj.isoformat() if
 isinstance(obj, datetime) else None))


if __name__ == "__main__":
    main()
```

C++

使用以下示例可获取有关多区域集群的信息。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/GetClusterRequest.h>
```

```
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
 * Retrieves information about a cluster in Amazon Aurora DSQL
 */
GetClusterResult GetCluster(const Aws::String& region, const Aws::String&
 identifier) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Get the cluster
    GetClusterRequest getClusterRequest;
    getClusterRequest.SetIdentifier(identifier);

    auto getOutcome = client.GetCluster(getClusterRequest);
    if (!getOutcome.IsSuccess()) {
        std::cerr << "Failed to retrieve cluster " << identifier << " in " << region
 << ": "
                  << getOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to retrieve cluster " + identifier + " in
 region " + region);
    }

    return getOutcome.GetResult();
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define region and cluster ID
            Aws::String region = "us-east-1";
            Aws::String clusterId = "<your cluster id>";

            auto cluster = GetCluster(region, clusterId);

            // Print cluster details
```

```
            std::cout << "Cluster Details:" << std::endl;
            std::cout << "ARN: " << cluster.GetArn() << std::endl;
            std::cout << "Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(cluster.GetStatus()) << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要获取有关多区域集群的信息，请使用以下示例。

```
import { DSQLClient, GetClusterCommand } from "@aws-sdk/client-dsql";

async function getCluster(region, clusterId) {

  const client = new DSQLClient({ region });

  const getClusterCommand = new GetClusterCommand({
    identifier: clusterId,
  });

  try {
    return await client.send(getClusterCommand);
  } catch (error) {
    if (error.name === "ResourceNotFoundException") {
      console.log("Cluster ID not found or deleted");
    }
    throw error;
  }
}

async function main() {
  const region = "us-east-1";
  const clusterId = "<CLUSTER_ID>";

  const response = await getCluster(region, clusterId);
  console.log("Cluster: ", response);
```

```
}

main();
```

Java

以下示例可用于获取有关多区域集群的信息。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

public class GetCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            GetClusterResponse cluster = client.getCluster(r ->
 r.identifier(clusterId));
            System.out.println(cluster);
        } catch (ResourceNotFoundException e) {
            System.out.printf("Cluster %s not found in %s%n", clusterId, region);
        }
    }
}
```

Rust

以下示例可用于获取有关多区域集群的信息。

```
use aws_config::load_defaults;
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Get a ClusterResource from DSQL cluster identifier
pub async fn get_cluster(region: &'static str, identifier: &'static str) ->
 GetClusterOutput {
    let client = dsql_client(region).await;
    client
        .get_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap()
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";

    let cluster = get_cluster(region, "<your cluster id>").await;
```

```
    println!("{:#?}", cluster);

    Ok(())
}
```

Ruby

以下示例可用于获取有关多区域集群的信息。

```
require "aws-sdk-dsql"
require "pp"

def get_cluster(region, identifier)
  client = Aws::DSQL::Client.new(region: region)
  client.get_cluster(identifier: identifier)
rescue Aws::Errors::ServiceError => e
  abort "Unable to retrieve cluster #{identifier} in region #{region}: #{e.message}"
end

def main
  region = "us-east-1"
  cluster_id = "<your cluster id>"
  cluster = get_cluster(region, cluster_id)
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

.NET

以下示例可用于获取有关多区域集群的信息。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;

namespace DSQLExamples.examples
{
```

```
    public class GetCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Get information about a DSQL cluster.
        /// </summary>
        public static async Task<GetClusterResponse> Get(RegionEndpoint region,
 string identifier)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var getClusterRequest = new GetClusterRequest
                {
                    Identifier = identifier
                };

                return await client.GetClusterAsync(getClusterRequest);
            }
        }

        private static async Task Main()
        {
            var region = RegionEndpoint.USEast1;
            var clusterId = "<your cluster id>";

            var response = await Get(region, clusterId);
            Console.WriteLine($"Cluster ARN: {response.Arn}");
        }
    }
```

```
}
```

Golang

以下示例可用于获取有关多区域集群的信息。

```
package main

import (
 "context"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func GetCluster(ctx context.Context, region, identifier string) (clusterStatus
 *dsql.GetClusterOutput, err error) {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Initialize the DSQL client
 client := dsql.NewFromConfig(cfg)

 input := &dsql.GetClusterInput{
  Identifier: aws.String(identifier),
 }
 clusterStatus, err = client.GetCluster(context.Background(), input)

 if err != nil {
  log.Fatalf("Failed to get cluster: %v", err)
 }

 log.Printf("Cluster ARN: %s", *clusterStatus.Arn)

 return clusterStatus, nil
}
```

```
func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
 defer cancel()

 // Example cluster identifier
 identifier := "<CLUSTER_ID>"
 region := "us-east-1"

 _, err := GetCluster(ctx, region, identifier)
 if err != nil {
  log.Fatalf("Failed to get cluster: %v", err)
 }
}
```

更新集群

以下示例演示如何使用不同编程语言来更新多区域集群。

Python

要更新多区域集群，请使用以下示例。

```
import boto3


def update_cluster(region, cluster_id, deletion_protection_enabled):
    try:
        client = boto3.client("dsql", region_name=region)
        return client.update_cluster(identifier=cluster_id,
 deletionProtectionEnabled=deletion_protection_enabled)
    except:
        print("Unable to update cluster")
        raise


def main():
    region = "us-east-1"
    cluster_id = "<your cluster id>"
    deletion_protection_enabled = False
    response = update_cluster(region, cluster_id, deletion_protection_enabled)
    print(f"Updated {response["arn"]} with deletion_protection_enabled:
 {deletion_protection_enabled}")
```

```
if __name__ == "__main__":
    main()
```

C++

使用以下示例可更新多区域集群。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/UpdateClusterRequest.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
 * Updates a cluster in Amazon Aurora DSQL
 */
UpdateClusterResult UpdateCluster(const Aws::String& region, const
 Aws::Map<Aws::String, Aws::String>& updateParams) {
    // Create client for the specified region
    DSQL::DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQL::DSQLClient client(clientConfig);

    // Create update request
    UpdateClusterRequest updateRequest;
    updateRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    // Set identifier (required)
    if (updateParams.find("identifier") != updateParams.end()) {
        updateRequest.SetIdentifier(updateParams.at("identifier"));
    } else {
        throw std::runtime_error("Cluster identifier is required for update
 operation");
    }

    // Set deletion protection if specified
    if (updateParams.find("deletion_protection_enabled") != updateParams.end()) {
```

```
        bool deletionProtection = (updateParams.at("deletion_protection_enabled") ==
 "true");
        updateRequest.SetDeletionProtectionEnabled(deletionProtection);
    }

    // Execute the update
    auto updateOutcome = client.UpdateCluster(updateRequest);
    if (!updateOutcome.IsSuccess()) {
        std::cerr << "Failed to update cluster: " <<
 updateOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Unable to update cluster");
    }

    return updateOutcome.GetResult();
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define region and update parameters
            Aws::String region = "us-east-1";
            Aws::String clusterId = "<your cluster id>";

            // Create parameter map
            Aws::Map<Aws::String, Aws::String> updateParams;
            updateParams["identifier"] = clusterId;
            updateParams["deletion_protection_enabled"] = "false";

            auto updatedCluster = UpdateCluster(region, updateParams);

            std::cout << "Updated " << updatedCluster.GetArn() << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要更新多区域集群，请使用以下示例。

```
import { DSQLClient, UpdateClusterCommand } from "@aws-sdk/client-dsql";

export async function updateCluster(region, clusterId, deletionProtectionEnabled) {

  const client = new DSQLClient({ region });

  const updateClusterCommand = new UpdateClusterCommand({
    identifier: clusterId,
    deletionProtectionEnabled: deletionProtectionEnabled
  });

  try {
    return await client.send(updateClusterCommand);
  } catch (error) {
    console.error("Unable to update cluster", error.message);
    throw error;
  }
}

async function main() {
  const region = "us-east-1";
  const clusterId = "<CLUSTER_ID>";
  const deletionProtectionEnabled = false;

  const response = await updateCluster(region, clusterId,
 deletionProtectionEnabled);
  console.log(`Updated ${response.arn}`);
}

main();
```

Java

使用以下示例可更新多区域集群。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
```

```
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsql.model.UpdateClusterResponse;

public class UpdateCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            UpdateClusterRequest request = UpdateClusterRequest.builder()
                    .identifier(clusterId)
                    .deletionProtectionEnabled(false)
                    .build();
            UpdateClusterResponse cluster = client.updateCluster(request);
            System.out.println("Updated " + cluster.arn());
        }
    }
}
```

Rust

使用以下示例可更新多区域集群。

```
use aws_config::load_defaults;
use aws_sdk_dsql::operation::update_cluster::UpdateClusterOutput;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
```

```
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Update a DSQL cluster and set delete protection to false. Also add new tags.
pub async fn update_cluster(region: &'static str, identifier: &'static str) ->
 UpdateClusterOutput {
    let client = dsql_client(region).await;
    // Update delete protection
    let update_response = client
        .update_cluster()
        .identifier(identifier)
        .deletion_protection_enabled(false)
        .send()
        .await
        .unwrap();

    update_response
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";

    let cluster = update_cluster(region, "<your cluster id>").await;
    println!("{:#?}", cluster);

    Ok(())
}
```

Ruby

使用以下示例可更新多区域集群。

```
require "aws-sdk-dsql"

def update_cluster(region, update_params)
```

```
  client = Aws::DSQL::Client.new(region: region)
  client.update_cluster(update_params)
rescue Aws::Errors::ServiceError => e
  abort "Unable to update cluster: #{e.message}"
end

def main
  region = "us-east-1"
  cluster_id = "<your cluster id>"
  updated_cluster = update_cluster(region, {
    identifier: cluster_id,
    deletion_protection_enabled: false
  })
  puts "Updated #{updated_cluster.arn}"
end

main if $PROGRAM_NAME == __FILE__
```

.NET

使用以下示例可更新多区域集群。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;

namespace DSQLExamples.examples
{
    public class UpdateCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
```

```
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Update a DSQL cluster and set delete protection to false.
        /// </summary>
        public static async Task<UpdateClusterResponse> Update(RegionEndpoint
 region, string identifier)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var updateClusterRequest = new UpdateClusterRequest
                {
                    Identifier = identifier,
                    DeletionProtectionEnabled = false
                };

                UpdateClusterResponse response = await
 client.UpdateClusterAsync(updateClusterRequest);
                Console.WriteLine($"Updated {response.Arn}");

                return response;
            }
        }

        private static async Task Main()
        {
            var region = RegionEndpoint.USEast1;
            var clusterId = "<your cluster id>";

            await Update(region, clusterId);
        }
    }
}
```

Golang

使用以下示例可更新多区域集群。

```
package main
```

```
import (
 "context"
 "github.com/aws/aws-sdk-go-v2/config"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func UpdateCluster(ctx context.Context, region, id string, deleteProtection bool)
 (clusterStatus *dsql.UpdateClusterOutput, err error) {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Initialize the DSQL client
 client := dsql.NewFromConfig(cfg)

 input := dsql.UpdateClusterInput{
  Identifier:                &id,
  DeletionProtectionEnabled: &deleteProtection,
 }

 clusterStatus, err = client.UpdateCluster(context.Background(), &input)

 if err != nil {
  log.Fatalf("Failed to update cluster: %v", err)
 }

 log.Printf("Cluster updated successfully: %v", clusterStatus.Status)
 return clusterStatus, nil
}

func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
 defer cancel()

 // Example cluster identifier
 identifier := "<CLUSTER_ID>"
 region := "us-east-1"
 deleteProtection := false
```

```
    _, err := UpdateCluster(ctx, region, identifier, deleteProtection)
    if err != nil {
     log.Fatalf("Failed to update cluster: %v", err)
    }
   }
```

删除集群

以下示例演示如何使用不同编程语言来删除多区域集群。

Python

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
import boto3


def delete_multi_region_clusters(region_1, cluster_id_1, region_2, cluster_id_2):
    try:

        client_1 = boto3.client("dsql", region_name=region_1)
        client_2 = boto3.client("dsql", region_name=region_2)

        client_1.delete_cluster(identifier=cluster_id_1)
        print(f"Deleting cluster {cluster_id_1} in {region_1}")

        # cluster_1 will stay in PENDING_DELETE state until cluster_2 is deleted

        client_2.delete_cluster(identifier=cluster_id_2)
        print(f"Deleting cluster {cluster_id_2} in {region_2}")

        # Now that both clusters have been marked for deletion they will transition
        # to DELETING state and finalize deletion
        print(f"Waiting for {cluster_id_1} to finish deletion")
        client_1.get_waiter("cluster_not_exists").wait(
            identifier=cluster_id_1,
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        print(f"Waiting for {cluster_id_2} to finish deletion")
```

```
        client_2.get_waiter("cluster_not_exists").wait(
            identifier=cluster_id_2,
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

    except:
        print("Unable to delete cluster")
        raise


def main():
    region_1 = "us-east-1"
    cluster_id_1 = "<cluster 1 id>"
    region_2 = "us-east-2"
    cluster_id_2 = "<cluster 2 id>"

    delete_multi_region_clusters(region_1, cluster_id_1, region_2, cluster_id_2)
    print(f"Deleted {cluster_id_1} in {region_1} and {cluster_id_2} in {region_2}")


if __name__ == "__main__":
    main()
```

C++

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
#include <aws/core/Aws.h>
#include <aws/core/utils/Outcome.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/DeleteClusterRequest.h>
#include <aws/dsql/model/GetClusterRequest.h>
#include <iostream>
#include <thread>
#include <chrono>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

/**
```

```
 * Deletes multi-region clusters in Amazon Aurora DSQL
 */
void DeleteMultiRegionClusters(
    const Aws::String& region1,
    const Aws::String& clusterId1,
    const Aws::String& region2,
    const Aws::String& clusterId2) {

    // Create clients for each region
    DSQL::DSQLClientConfiguration clientConfig1;
    clientConfig1.region = region1;
    DSQL::DSQLClient client1(clientConfig1);

    DSQL::DSQLClientConfiguration clientConfig2;
    clientConfig2.region = region2;
    DSQL::DSQLClient client2(clientConfig2);

    // Delete the first cluster
    std::cout << "Deleting cluster " << clusterId1 << " in " << region1 <<
 std::endl;

    DeleteClusterRequest deleteRequest1;
    deleteRequest1.SetIdentifier(clusterId1);
    deleteRequest1.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto deleteOutcome1 = client1.DeleteCluster(deleteRequest1);
    if (!deleteOutcome1.IsSuccess()) {
        std::cerr << "Failed to delete cluster " << clusterId1 << " in " << region1
 << ": "
                  << deleteOutcome1.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to delete multi-region clusters");
    }

    // cluster1 will stay in PENDING_DELETE state until cluster2 is deleted
    std::cout << "Deleting cluster " << clusterId2 << " in " << region2 <<
 std::endl;

    DeleteClusterRequest deleteRequest2;
    deleteRequest2.SetIdentifier(clusterId2);
    deleteRequest2.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto deleteOutcome2 = client2.DeleteCluster(deleteRequest2);
    if (!deleteOutcome2.IsSuccess()) {
```

```
        std::cerr << "Failed to delete cluster " << clusterId2 << " in " << region2
 << ": "
                  << deleteOutcome2.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to delete multi-region clusters");
    }
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            Aws::String region1 = "us-east-1";
            Aws::String clusterId1 = "<your cluster id 1>";
            Aws::String region2 = "us-east-2";
            Aws::String clusterId2 = "<your cluster id 2>";

            DeleteMultiRegionClusters(region1, clusterId1, region2, clusterId2);

            std::cout << "Deleted " << clusterId1 << " in " << region1
                      << " and " << clusterId2 << " in " << region2 << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
import { DSQLClient, DeleteClusterCommand, waitUntilClusterNotExists } from "@aws-
sdk/client-dsql";

async function deleteMultiRegionClusters(region1, cluster1_id, region2, cluster2_id)
 {

    const client1 = new DSQLClient({ region: region1 });
    const client2 = new DSQLClient({ region: region2 });

    try {
```

```
        const deleteClusterCommand1 = new DeleteClusterCommand({
            identifier: cluster1_id,
        });
        const response1 = await client1.send(deleteClusterCommand1);

        const deleteClusterCommand2 = new DeleteClusterCommand({
            identifier: cluster2_id,
        });
        const response2 = await client2.send(deleteClusterCommand2);

        console.log(`Waiting for cluster1 ${response1.identifier} to finish 
deletion`);
        await waitUntilClusterNotExists(
            {
                client: client1,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response1.identifier
            }
        );
        console.log(`Cluster1 Id ${response1.identifier} is now deleted`);

        console.log(`Waiting for cluster2 ${response2.identifier} to finish 
deletion`);
        await waitUntilClusterNotExists(
            {
                client: client2,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response2.identifier
            }
        );
        console.log(`Cluster2 Id ${response2.identifier} is now deleted`);
        return;
    } catch (error) {
        if (error.name === "ResourceNotFoundException") {
            console.log("Some or all Cluster ARNs not found or already deleted");
        } else {
            console.error("Unable to delete multi-region clusters: ", 
error.message);
        }
        throw error;
```

```
    }
}

async function main() {
    const region1 = "us-east-1";
    const cluster1_id = "<CLUSTER_ID_1>";
    const region2 = "us-east-2";
    const cluster2_id = "<CLUSTER_ID_2>";

    const response = await deleteMultiRegionClusters(region1, cluster1_id, region2,
 cluster2_id);
    console.log(`Deleted ${cluster1_id} in ${region1} and ${cluster2_id} in
 ${region2}`);
}

main();
```

Java

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.DsqlClientBuilder;
import software.amazon.awssdk.services.dsql.model.DeleteClusterRequest;

import java.time.Duration;

public class DeleteMultiRegionClusters {

    public static void main(String[] args) {
        Region region1 = Region.US_EAST_1;
        String clusterId1 = "<your cluster id 1>";
        Region region2 = Region.US_EAST_2;
        String clusterId2 = "<your cluster id 2>";

        DsqlClientBuilder clientBuilder = DsqlClient.builder()
                .credentialsProvider(DefaultCredentialsProvider.create());

        try (
```

```
            DsqlClient client1 = clientBuilder.region(region1).build();
            DsqlClient client2 = clientBuilder.region(region2).build()
        ) {
            System.out.printf("Deleting cluster %s in %s%n", clusterId1, region1);
            DeleteClusterRequest request1 = DeleteClusterRequest.builder()
                    .identifier(clusterId1)
                    .build();
            client1.deleteCluster(request1);

            // cluster1 will stay in PENDING_DELETE until cluster2 is deleted
            System.out.printf("Deleting cluster %s in %s%n", clusterId2, region2);
            DeleteClusterRequest request2 = DeleteClusterRequest.builder()
                    .identifier(clusterId2)
                    .build();
            client2.deleteCluster(request2);

            // Now that both clusters have been marked for deletion they will
 transition
            // to DELETING state and finalize deletion.
            System.out.printf("Waiting for cluster %s to finish deletion%n",
 clusterId1);
            client1.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId1),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );

            System.out.printf("Waiting for cluster %s to finish deletion%n",
 clusterId2);
            client2.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId2),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );

            System.out.printf("Deleted %s in %s and %s in %s%n", clusterId1,
 region1, clusterId2, region2);
        }
    }
```

```
}
```

Rust

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
use aws_config::{BehaviorVersion, Region, load_defaults};
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::{Client, Config};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults.credentials_provider().unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Create a cluster without delete protection and a name
pub async fn delete_multi_region_clusters(
    region_1: &'static str,
    cluster_id_1: &'static str,
    region_2: &'static str,
    cluster_id_2: &'static str,
) {
    let client_1 = dsql_client(region_1).await;
    let client_2 = dsql_client(region_2).await;

    println!("Deleting cluster {cluster_id_1} in {region_1}");
    client_1
        .delete_cluster()
        .identifier(cluster_id_1)
        .send()
```

```
        .await
        .unwrap();

    // cluster_1 will stay in PENDING_DELETE state until cluster_2 is deleted
    println!("Deleting cluster {cluster_id_2} in {region_2}");
    client_2
        .delete_cluster()
        .identifier(cluster_id_2)
        .send()
        .await
        .unwrap();

    // Now that both clusters have been marked for deletion they will transition
    // to DELETING state and finalize deletion
    println!("Waiting for {cluster_id_1} to finish deletion");
    client_1
        .wait_until_cluster_not_exists()
        .identifier(cluster_id_1)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap();

    println!("Waiting for {cluster_id_2} to finish deletion");
    client_2
        .wait_until_cluster_not_exists()
        .identifier(cluster_id_2)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap();
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region_1 = "us-east-1";
    let cluster_id_1 = "<cluster 1 to be deleted>";
    let region_2 = "us-east-2";
    let cluster_id_2 = "<cluster 2 to be deleted>";

    delete_multi_region_clusters(region_1, cluster_id_1, region_2,
 cluster_id_2).await;
    println!("Deleted {cluster_id_1} in {region_1} and {cluster_id_2} in
 {region_2}");

    Ok(())
```

```
}
```

Ruby

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
require "aws-sdk-dsql"

def delete_multi_region_clusters(region_1, cluster_id_1, region_2, cluster_id_2)
  client_1 = Aws::DSQL::Client.new(region: region_1)
  client_2 = Aws::DSQL::Client.new(region: region_2)

  puts "Deleting cluster #{cluster_id_1} in #{region_1}"
  client_1.delete_cluster(identifier: cluster_id_1)

  # cluster_1 will stay in PENDING_DELETE state until cluster_2 is deleted
  puts "Deleting #{cluster_id_2} in #{region_2}"
  client_2.delete_cluster(identifier: cluster_id_2)

  # Now that both clusters have been marked for deletion they will transition
  # to DELETING state and finalize deletion
  puts "Waiting for #{cluster_id_1} to finish deletion"
  client_1.wait_until(:cluster_not_exists, identifier: cluster_id_1) do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end

  puts "Waiting for #{cluster_id_2} to finish deletion"
  client_2.wait_until(:cluster_not_exists, identifier: cluster_id_2) do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end
rescue Aws::Errors::ServiceError => e
  abort "Failed to delete multi-region clusters: #{e.message}"
end

def main
  region_1 = "us-east-1"
  cluster_id_1 = "<your cluster id 1>"
  region_2 = "us-east-2"
  cluster_id_2 = "<your cluster id 2>"
```

```
  delete_multi_region_clusters(region_1, cluster_id_1, region_2, cluster_id_2)
  puts "Deleted #{cluster_id_1} in #{region_1} and #{cluster_id_2} in #{region_2}"
end

main if $PROGRAM_NAME == __FILE__
```

.NET

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime.Credentials;
using Amazon.Runtime.Endpoints;

namespace DSQLExamples.examples
{
    public class DeleteMultiRegionClusters
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
 region)
        {
            var awsCredentials = await
 DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region,
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Delete multi-region clusters.
        /// </summary>
        public static async Task Delete(
            RegionEndpoint region1,
```

```
            string clusterId1,
            RegionEndpoint region2,
            string clusterId2)
        {
            using (var client1 = await CreateDSQLClient(region1))
            using (var client2 = await CreateDSQLClient(region2))
            {
                var deleteRequest1 = new DeleteClusterRequest
                {
                    Identifier = clusterId1
                };

                var deleteResponse1 = await
 client1.DeleteClusterAsync(deleteRequest1);
                Console.WriteLine($"Initiated deletion of {deleteResponse1.Arn}");

                // cluster 1 will stay in PENDING_DELETE state until cluster 2 is
 deleted
                var deleteRequest2 = new DeleteClusterRequest
                {
                    Identifier = clusterId2
                };

                var deleteResponse2 = await
 client2.DeleteClusterAsync(deleteRequest2);
                Console.WriteLine($"Initiated deletion of {deleteResponse2.Arn}");
            }
        }

        private static async Task Main()
        {
            var region1 = RegionEndpoint.USEast1;
            var cluster1 = "<cluster 1 to be deleted>";
            var region2 = RegionEndpoint.USEast2;
            var cluster2 = "<cluster 2 to be deleted>";

            await Delete(region1, cluster1, region2, cluster2);
        }
    }
}
```

Golang

要删除多区域集群，请使用以下示例。删除多区域集群可能需要一些时间。

```
package main

import (
 "context"
 "fmt"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/dsql"
)

func DeleteMultiRegionClusters(ctx context.Context, region1, clusterId1, region2,
 clusterId2 string) error {
 // Load the AWS configuration for region 1
 cfg1, err := config.LoadDefaultConfig(ctx, config.WithRegion(region1))
 if err != nil {
  return fmt.Errorf("unable to load SDK config for region %s: %w", region1, err)
 }

 // Load the AWS configuration for region 2
 cfg2, err := config.LoadDefaultConfig(ctx, config.WithRegion(region2))
 if err != nil {
  return fmt.Errorf("unable to load SDK config for region %s: %w", region2, err)
 }

 // Create DSQL clients for both regions
 client1 := dsql.NewFromConfig(cfg1)
 client2 := dsql.NewFromConfig(cfg2)

 // Delete cluster in region 1
 fmt.Printf("Deleting cluster %s in %s\n", clusterId1, region1)
 _, err = client1.DeleteCluster(ctx, &dsql.DeleteClusterInput{
  Identifier: aws.String(clusterId1),
 })
 if err != nil {
  return fmt.Errorf("failed to delete cluster in region %s: %w", region1, err)
 }

 // Delete cluster in region 2
 fmt.Printf("Deleting cluster %s in %s\n", clusterId2, region2)
 _, err = client2.DeleteCluster(ctx, &dsql.DeleteClusterInput{
```

```
 Identifier: aws.String(clusterId2),
})
if err != nil {
 return fmt.Errorf("failed to delete cluster in region %s: %w", region2, err)
}

// Create waiters for both regions
waiter1 := dsql.NewClusterNotExistsWaiter(client1, func(options
*dsql.ClusterNotExistsWaiterOptions) {
 options.MinDelay = 10 * time.Second
 options.MaxDelay = 30 * time.Second
 options.LogWaitAttempts = true
})

waiter2 := dsql.NewClusterNotExistsWaiter(client2, func(options
*dsql.ClusterNotExistsWaiterOptions) {
 options.MinDelay = 10 * time.Second
 options.MaxDelay = 30 * time.Second
 options.LogWaitAttempts = true
})

// Wait for cluster in region 1 to be deleted
fmt.Printf("Waiting for cluster %s to finish deletion\n", clusterId1)
err = waiter1.Wait(ctx, &dsql.GetClusterInput{
 Identifier: aws.String(clusterId1),
}, 5*time.Minute)
if err != nil {
 return fmt.Errorf("error waiting for cluster deletion in region %s: %w", region1,
err)
}

// Wait for cluster in region 2 to be deleted
fmt.Printf("Waiting for cluster %s to finish deletion\n", clusterId2)
err = waiter2.Wait(ctx, &dsql.GetClusterInput{
 Identifier: aws.String(clusterId2),
}, 5*time.Minute)
if err != nil {
 return fmt.Errorf("error waiting for cluster deletion in region %s: %w", region2,
err)
}

fmt.Printf("Successfully deleted clusters %s in %s and %s in %s\n",
 clusterId1, region1, clusterId2, region2)
return nil
```

```
}

// Example usage in main function
func main() {
 ctx, cancel := context.WithTimeout(context.Background(), 10*time.Minute)
 defer cancel()

 err := DeleteMultiRegionClusters(
  ctx,
  "us-east-1",      // region1
  "<CLUSTER_ID_1>", // clusterId1
  "us-east-2",      // region2
  "<CLUSTER_ID_2>", // clusterId2
 )
 if err != nil {
  log.Fatalf("Failed to delete multi-region clusters: %v", err)
 }
}
```

有关更多代码示例及相关示例，请访问 Aurora DSQL 示例 GitHub 存储库。

使用 AWS CLI

AWS CLI 提供了用于管理多区域 Aurora DSQL 集群的命令行界面。以下示例演示如何创建、配置和删除多区域集群。

连接到多区域集群

多区域对等集群提供两个区域端点，每个对等集群 AWS 区域中各一个。这两个端点都提供了一个逻辑数据库，该数据库支持并发读写操作，并具有强数据一致性。除了对等集群之外，多区域集群还有一个见证区域，它存储有限时段内的加密事务日志，用于提高多区域持久性和可用性。多区域见证区域没有端点。

创建多区域集群

要创建多区域集群，首先创建一个带有见证区域的集群。然后，您使此集群与第二个集群对等，第二个集群与第一个集群共享相同的见证区域。以下示例显示了如何在美国东部（弗吉尼亚州北部）和美国东部（俄亥俄州）创建以美国西部（俄勒冈州）作为见证区域的集群。

步骤 1：在美国东部（弗吉尼亚州北部）创建集群 1

要在美国东部（弗吉尼亚州北部）AWS 区域创建具有多区域属性的集群，请使用以下命令。

```
aws dsql create-cluster \
--region us-east-1 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example响应：

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "UPDATING",
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
   }
   "creationTime": "2024-05-24T09:15:32.708000-07:00"
}
```

> **Note**
>
> API 操作成功后，集群进入 PENDING_SETUP 状态。集群创建将保持 PENDING_SETUP 状态，直至您使用其对等集群的 ARN 更新该集群。

步骤 2：在美国东部（俄亥俄州）创建集群 2

要在美国东部（俄亥俄州）AWS 区域创建具有多区域属性的集群，请使用以下命令。

```
aws dsql create-cluster \
--region us-east-2 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example响应：

```
{
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5",
    "status": "PENDING_SETUP",
    "creationTime": "2025-05-06T06:51:16.145000-07:00",
    "deletionProtectionEnabled": true,
    "multiRegionProperties": {
        "witnessRegion": "us-west-2",
```

```
        "clusters": [
            "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"
        ]
    }
}
```

API 操作成功后，集群将转为 PENDING_SETUP 状态。集群创建将保持 PENDING_SETUP 状态，直到您通过用于实现对等的另一个集群的 ARN 对其进行更新。

步骤 3：使美国东部（弗吉尼亚州北部）与美国东部（俄亥俄州）的集群对等

要使美国东部（弗吉尼亚州北部）集群与美国东部（俄亥俄州）集群对等，请使用 update-cluster 命令。指定美国东部（弗吉尼亚州北部）集群名称以及带有美国东部（俄亥俄州）集群的 ARN 的 JSON 字符串。

```
aws dsql update-cluster \
--region us-east-1 \
--identifier 'foo0bar1baz2quux3quuxquux4' \
--multi-region-properties '{"witnessRegion": "us-west-2","clusters": ["arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"]}'
```

Example响应

```
{
    "identifier": "foo0bar1baz2quux3quuxquux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

步骤 4：使美国东部（俄亥俄州）与美国东部（弗吉尼亚州北部）的集群对等

要使美国东部（俄亥俄州）集群与美国东部（弗吉尼亚州北部）集群对等，请使用 update-cluster 命令。指定美国东部（俄亥俄州）集群名称以及带有美国东部（弗吉尼亚州北部）集群的 ARN 的 JSON 字符串。

Example

```
aws dsql update-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5' \
```

```
--multi-region-properties '{"witnessRegion": "us-west-2", "clusters":
 ["arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4"]}'
```

Example响应

```
{
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:51:16.145000-07:00"
}
```

Note

成功实现对等后，这两个集群都会从“PENDING_SETUP”转换为“CREATING”，最后在准备就绪可供使用时变为“ACTIVE”状态。

查看多区域集群属性

描述集群时，您可以查看不同 AWS 区域中集群的多区域属性。

Example

```
aws dsql get-cluster \
--region us-east-1 \
--identifier 'foo0bar1baz2quux3quuxquux4'
```

Example响应

```
{
    "identifier": "foo0bar1baz2quux3quuxquux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4",
    "status": "PENDING_SETUP",
    "encryptionDetails": {
    "encryptionType": "AWS_OWNED_KMS_KEY",
    "encryptionStatus": "ENABLED"
},
    "creationTime": "2024-11-27T00:32:14.434000-08:00",
    "deletionProtectionEnabled": false,
    "multiRegionProperties": {
```

```
        "witnessRegion": "us-west-2",
        "clusters": [
           "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4",
           "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"
        ]
    }
}
```

在创建过程中使集群对等

可以通过在集群创建过程中包含对等信息来减少步骤数。在美国东部（弗吉尼亚州北部）创建第一个集群（步骤 1）后，您可以在美国东部（俄亥俄州）创建第二个集群，同时通过包含第一个集群的 ARN 来启动对等过程。

Example

```
aws dsql create-cluster \
--region us-east-2 \
--multi-region-properties '{"witnessRegion":"us-west-2","clusters": ["arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4"]}'
```

这合并执行了步骤 2 和步骤 4，但您仍需要完成步骤 3（使用第二个集群的 ARN 更新第一个集群），才能建立对等关系。完成所有步骤后，这两个集群将经历与标准过程相同的状态转换：从 PENDING_SETUP 转换为 CREATING，最后在准备好可供使用时转换为 ACTIVE。

删除多区域集群

要删除多区域集群，您需要完成两个步骤。

1. 关闭每个集群的删除保护。
2. 在各自的 AWS 区域中分别删除每个对等集群

更新和删除美国东部（弗吉尼亚州北部）的集群

1. 使用 `update-cluster` 命令关闭删除保护。

   ```
   aws dsql update-cluster \
     --region us-east-1 \
     --identifier 'foo0bar1baz2quux3quuxquux4' \
     --no-deletion-protection-enabled
   ```

2. 使用 `delete-cluster` 命令删除集群。

```
aws dsql delete-cluster \
  --region us-east-1 \
  --identifier 'foo0bar1baz2quux3quuxquux4'
```

此命令将返回以下响应。

```
{
    "identifier": "foo0bar1baz2quux3quuxquux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
foo0bar1baz2quux3quuxquux4",
    "status": "PENDING_DELETE",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

> **Note**
>
> 集群将转换为 `PENDING_DELETE` 状态。直至删除美国东部（俄亥俄州）中的对等集群后，删除才完成。

更新和删除美国东部（俄亥俄州）的集群

1. 使用 `update-cluster` 命令关闭删除保护。

```
aws dsql update-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quux4quuux' \
--no-deletion-protection-enabled
```

2. 使用 `delete-cluster` 命令删除集群。

```
aws dsql delete-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5'
```

该命令返回以下响应：

```
{
```

```
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/
foo0bar1baz2quux3quuxquux5",
    "status": "PENDING_DELETE",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

> **Note**
>
> 集群将转换为 PENDING_DELETE 状态。几秒钟后，系统会在验证后自动将两个对等集群转换为 DELETING 状态。

使用 AWS CloudFormation 配置 Aurora DSQL 集群

您可以使用相同的 CloudFormation 资源 `AWS::DSQL::Cluster` 来部署和管理单区域和多区域 Aurora DSQL 集群。

有关如何使用 `AWS::DSQL::Cluster` 资源来创建、修改和管理集群的更多信息，请参阅 Amazon Aurora DSQL resource type reference。

创建初始集群配置

首先，创建一个 AWS CloudFormation 模板来定义多区域集群：

```
---
Resources:
  MRCluster:
    Type: AWS::DSQL::Cluster
    Properties:
      DeletionProtectionEnabled: true
      MultiRegionProperties:
        WitnessRegion: us-west-2
```

使用以下 AWS CLI 命令在两个区域中创建堆栈：

```
aws cloudformation create-stack --region us-east-2 \
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

```
aws cloudformation create-stack --region us-east-1 \
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

查找集群标识符

检索集群的物理资源 ID：

```
aws cloudformation describe-stack-resources -region us-east-2 \
    --stack-name MRCluster \
    --query 'StackResources[].PhysicalResourceId'
[
  "auabudrks5jwh4mjt6o5xxhr4y"
]
```

```
aws cloudformation describe-stack-resources -region us-east-1 \
    --stack-name MRCluster \
    --query 'StackResources[].PhysicalResourceId'
[
  "imabudrfon4p2z3nv2jo4rlajm"
]
```

更新集群配置

更新 AWS CloudFormation 模板以包含两个集群 ARN：

```
---
Resources:
  MRCluster:
    Type: AWS::DSQL::Cluster
    Properties:
      DeletionProtectionEnabled: true
      MultiRegionProperties:
        WitnessRegion: us-west-2
        Clusters:
        - arn:aws:dsql:us-east-2:123456789012:cluster/auabudrks5jwh4mjt6o5xxhr4y
        - arn:aws:dsql:us-east-1:123456789012:cluster/imabudrfon4p2z3nv2jo4rlajm
```

将更新后的配置应用于这两个区域：

```
aws cloudformation update-stack --region us-east-2 \
```

```
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

```
aws cloudformation update-stack --region us-east-1 \
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

Aurora DSQL 集群生命周期

了解 Aurora DSQL 集群生命周期有助于高效管理集群。本节介绍集群状态定义和可优化成本的缩容至零功能。

定义 Aurora DSQL 集群状态

Aurora DSQL 集群状态提供有关集群运行状况和连接的重要信息。可以使用 AWS 管理控制台、AWS CLI 或 Aurora DSQL API 查看集群和集群实例的状态。

下表描述了 Aurora DSQL 集群的每种可能状态以及每种状态的含义。

Status	说明
Creating	Aurora DSQL 正在尝试为集群创建或配置资源。当集群处于这种状态时，任何连接尝试都将失败。
活动	集群正在运行，可供使用。
Idle	当集群闲置时间足够长，促使 Aurora DSQL 缩减运行中的资源以降低容量和成本时，集群将变为空闲状态。当您连接到空闲的集群时，Aurora DSQL 会将集群转换回活跃状态。
非活动	当空闲集群上长时间没有活动时，该集群将变为非活动状态。在此暂停状态下，运行中的资源会缩容至零，但您的数据将保留。当您尝试连接到非活跃的集群时，Aurora DSQL 会自动将该集群转换回活跃状态。恢复时间取决于集群大小。
Updating	当您更改集群配置时，集群会变为正在更新状态。
删除	当您提交删除集群请求时，集群会变为正在删除状态。
已删除	已成功删除了集群。

Status	说明
失败	Aurora DSQL 无法创建集群，因为它遇到了错误。
待设置	仅适用于多区域集群。当您在带有见证区域的第一个区域中创建多区域集群时，多区域集群将进入待设置状态。集群创建将暂停，直到您在辅助区域中创建另一个集群并使这两个集群对等。
待删除	仅适用于多区域集群。当您从多区域集群中删除某个集群时，多区域集群会进入待删除状态。在您删除最后一个对等集群后，多区域集群会变为正在删除状态。

使用空闲集群和非活动集群

当 Aurora DSQL 检测到集群在一段时间内无连接活动时，它会将集群切换至空闲状态，并减少运行中的资源以最大限度地降低容量与成本。如果集群长时间无连接活动，空闲集群将自动切换至非活动状态，此时运行中的资源将缩容至零，但您的数据将保留。

要恢复正常操作，只需像往常一样连接到集群即可。当您成功连接到集群时，Aurora DSQL 会自动将该集群切换至活动状态。

> **Note**
> 首次尝试连接到空闲或非活动集群时，连接速度会比平时慢。

要求集群处于活动状态才能执行的操作

某些操作要求您的集群处于活动状态。要在空闲或非活动集群上执行这些操作，您需要通过连接到集群来将集群转换回活动状态。

备份操作

要求集群处于活动状态才能执行备份操作。如果集群处于空闲或非活动状态，则备份操作失败，并显示以下错误：

```
"Error": {
    "Code": "FailedPrecondition",
    "Message": "Cluster 'cluster-id' is in state 'IDLE' and can't be backed up.
```

```
    In order to take a backup of your cluster, it must be in Active state. Please
    connect to your cluster to transition it to Active to perform the backup."
}
```

要继续进行备份，请执行以下操作：

1. 使用首选数据库客户端或 Aurora DSQL 控制台连接到集群以将其唤醒。
2. 等待自动切换至活动状态。
3. 在集群完全运行正常后，启动备份操作。

> **Note**
>
> 在集群为空闲状态或非活动状态之前创建的现有备份保持有效且不受影响。在连接到集群以实施自动唤醒前，尝试在集群上创建新备份的操作将失败。

查看 Aurora DSQL 集群状态

要查看集群的状态，请使用 AWS 管理控制台、AWS CLI 或 Aurora DSQL API。

控制台

按照以下步骤在 AWS 管理控制台中查看集群状态：

在控制台中查看集群状态

1. 打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql。
2. 在导航窗格中选择 Clusters (集群)。
3. 在控制面板中查看每个集群的状态。

AWS CLI

运行以下 AWS CLI 命令来检查单个集群的状态。

```
aws dsql get-cluster --identifier cluster-id --query status --output text
```

运行以下命令以列出所有集群的状态。

```
for id in $(aws dsql list-clusters --query 'clusters[*].identifier' --output text); do

  cluster_status=$(aws dsql get-cluster --identifier "$id" --query 'status' --output
 text)
  echo "$id    $cluster_status"
done
```

此示例输出显示了两个活跃的集群和一个正在删除的集群。

```
aaabbb2bkx555xa7p42qd5cdef    ACTIVE
abcde123efghi77t35abcdefgh    ACTIVE
12abc6lqasc5bbbbbbbbbbbbbb    DELETING
```

使用 Aurora DSQL 进行编程

Aurora DSQL 为您提供以下工具，以便以编程方式管理 Aurora DSQL 资源。

AWS Command Line Interface (AWS CLI)

可以在命令行 Shell 中使用 AWS CLI 来创建和管理资源。对于 Aurora DSQL 等 AWS 服务，AWS CLI 提供对 API 的直接访问。有关 Aurora DSQL 命令的语法和示例，请参阅《AWS CLI Command Reference》中的 dsql。

AWS 软件开发工具包（SDK）

AWS 为许多流行的技术和编程语言提供 SDK。这些 SDK 使您能够更轻松地从应用程序中使用该语言或技术调用 AWS 服务。有关这些 SDK 的更多信息，请参阅用于在 AWS 上开发和管理应用程序的工具。

Aurora DSQL API

此 API 是 Aurora DSQL 的另一个编程接口。使用此 API 时，必须正确格式化每个 HTTPS 请求，并向每个请求添加有效的数字签名。有关更多信息，请参阅 API 参考。

CloudFormation

AWS::DSQL::Cluster 是一种 CloudFormation 资源，使您能够创建和管理 Aurora DSQL 集群，作为基础设施即代码的一部分。CloudFormation 有助于您通过代码定义整个 AWS 环境，从而更轻松地以一致且可靠的方式预置、更新和复制基础设施。

当您在 CloudFormation 模板中使用 AWS::DSQL::Cluster 资源时，您能够以声明方式将 Aurora DSQL 集群与其它云资源一起预置。这有助于确保数据基础设施与应用程序堆栈的其余部分一起部署和管理。

适用于 Aurora DSQL 的连接器

Aurora DSQL 提供了专用连接器，用于扩展现有的数据库驱动程序，从而实现与 AWS 服务的无缝 IAM 身份验证和集成。这些连接器设计为可用于流行的编程语言和框架，同时保持了与现有 PostgreSQL 工作流的兼容性。

我们已计划在未来版本中推出更多连接器。有关连接器可用性的最新信息，请参阅 Aurora DSQL 示例存储库。

使用 JDBC 连接器连接到 Aurora DSQL 集群

Aurora DSQL Connector for JDBC 设计为身份验证插件，该插件对 PostgreSQL JDBC 驱动程序的功能进行扩展，使应用程序能够使用 IAM 凭证向 Aurora DSQL 进行身份验证。该连接器不直接连接到数据库，而是在底层 PostgreSQL JDBC 驱动程序之上，提供无缝的 IAM 身份验证。

适用于 JDBC 的 Aurora DSQL 连接器设计为与 PostgreSQL JDBC 驱动程序配合使用，提供与 Aurora DSQL 的 IAM 身份验证要求的无缝集成。

适用于 JDBC 的 Aurora DSQL 连接器与 PostgreSQL JDBC 驱动程序结合使用，为 Aurora DSQL 实现了基于 IAM 的身份验证。该连接器引入了与 AWS Identity and Access Management（IAM）等 AWS 身份验证服务的深度集成。

关于连接器

Aurora DSQL 是一种分布式 SQL 数据库服务，面向兼容 PostgreSQL 的应用程序提供高可用性和可扩展性。Aurora DSQL 要求使用基于 IAM 的身份验证以及限时令牌，而现有 JDBC 驱动程序本身不支持这种方法。

设计适用于 JDBC 的 Aurora DSQL 连接器的主要理念是，在 PostgreSQL JDBC 驱动程序之上添加一个身份验证层来处理 IAM 令牌生成，使得用户无需更改现有 JDBC 工作流即可连接到 Aurora DSQL。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：使用 AWS 凭证生成身份验证令牌，其生命周期可配置

适用于 JDBC 的 Aurora DSQL 连接器针对这些要求而设计，在建立连接时自动生成 IAM 身份验证令牌。

适用于 JDBC 的 Aurora DSQL 连接器的益处

尽管 Aurora DSQL 提供了与 PostgreSQL 兼容的接口，但现有 PostgreSQL 驱动程序目前不支持 Aurora DSQL 的 IAM 身份验证要求。通过适用于 JDBC 的 Aurora DSQL 连接器，客户可以继续使用现有的 PostgreSQL 工作流，同时通过以下方式启用 IAM 身份验证：

- 自动令牌生成：使用 AWS 凭证自动生成 IAM 令牌

- 无缝集成：适用于现有 JDBC 连接模式
- AWS 凭证支持：支持各种 AWS 凭证提供方（默认、基于配置文件等）

将适用于 JDBC 的 Aurora DSQL 连接器与连接池结合使用

适用于 JDBC 的 Aurora DSQL 连接器可与 HikariCP 等连接池库结合使用。连接器在建立连接期间处理 IAM 令牌生成，使得连接池可以正常操作。

主要 功能

自动令牌生成

IAM 令牌使用 AWS 凭证自动生成。

无缝集成

使用现有 JDBC 连接模式而无需更改工作流。

AWS 凭证支持

支持各种 AWS 凭证提供方（默认、基于配置文件等）。

连接池兼容性

与 HikariCP 等连接池库无缝结合使用。

先决条件

在开始之前，请确保您已完成以下先决条件：

- 在 Aurora DSQL 中创建集群。
- 安装 Java 开发工具包（JDK）。请确保您使用的是版本 17 或更高版本。
- 设置适当的 IAM 权限，以允许应用程序连接到 Aurora DSQL。
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）。

使用适用于 JDBC 的 Aurora DSQL 连接器

要在 Java 应用程序中使用适用于 JDBC 的 Aurora DSQL 连接器，请按照以下步骤操作：

1. 将以下依赖项添加到您的 Maven 项目中。

```
<dependencies>
    <!-- Aurora DSQL Connector for JDBC -->
    <dependency>
        <groupId>software.amazon.dsql</groupId>
        <artifactId>aurora-dsql-jdbc-connector</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

对于 Gradle 项目，请添加此依赖项：

```
implementation("software.amazon.dsql:aurora-dsql-jdbc-connector:1.0.0")
```

2. 使用 AWS DSQL PostgreSQL 连接器格式，创建与 Aurora DSQL 集群的基本连接：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DsqlJdbcConnectorExample {
    public static void main(String[] args) {
        // Using AWS DSQL PostgreSQL Connector prefix
        String jdbcUrl = "jdbc:aws-dsql:postgresql://your-cluster.dsql.us-
east-1.on.aws/postgres?user=admin";

        try (Connection connection = DriverManager.getConnection(jdbcUrl)) {
            // Use the connection
            try (Statement statement = connection.createStatement()) {
                // Create a table
                statement.execute("CREATE TABLE IF NOT EXISTS test_table (id UUID
 PRIMARY KEY DEFAULT gen_random_uuid(), name VARCHAR(100))");

                // Insert data
                statement.execute("INSERT INTO test_table (name) VALUES ('Test
 Name')");

                // Query data
                try (ResultSet resultSet = statement.executeQuery("SELECT * FROM
 test_table")) {
                    while (resultSet.next()) {
```

```
                        System.out.println("ID: " + resultSet.getInt("id") + ",
 Name: " + resultSet.getString("name"));
                    }
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

配置属性

适用于 JDBC 的 Aurora DSQL 连接器支持以下连接属性：

用户

确定连接的用户以及使用的令牌生成方法。示例：`admin`

token-duration-secs

令牌的有效期长度，以秒为单位。有关令牌限制的更多信息，请参阅在 Amazon Aurora DSQL 中生成身份验证令牌。

配置文件

用于实例化 ProfileCredentialsProvider，来通过提供的配置文件名称生成令牌。

region

用于 Aurora DSQL 连接的 AWS 区域。这是可选的。提供此项时，将覆盖从 URL 中提取的区域。

database

要连接的数据库的名称。默认值为 `postgres`。

日志记录

启用日志记录功能，用于对使用 Aurora DSQL JDBC 连接器时可能遇到的任何问题进行故障排除。

该连接器使用 Java 的内置日志记录系统（java.util.logging）。您可以通过创建 `logging.properties` 文件来配置日志记录级别：

```
# Set root logger level to INFO for clean output
.level = INFO

# Show Aurora DSQL Connector for JDBC FINE logs for detailed debugging
software.amazon.dsql.level = FINE

# Console handler configuration
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Detailed formatter pattern with timestamp and logger name
java.util.logging.SimpleFormatter.format = %1$tH:%1$tM:%1$tS.%1$tL [%4$s] %3$s - %5$s%n
```

示例

有关更全面的示例和使用案例，请参阅适用于 JDBC 的 Aurora DSQL 连接器存储库

适用于 Python 的 Aurora DSQL 连接器

Aurora DSQL Connector for Python 集成了 IAM 身份验证，用于将 Python 应用程序连接到 Amazon Aurora DSQL 集群。其内部采用 psycopg、psycopg2 和 asyncpg 客户端库。

经设计，适用于 Python 的 Aurora DSQL 连接器可作为身份验证插件，对 psycopg、psycopg2 和 asyncpg 客户端库的功能进行扩展，使应用程序能够使用 IAM 凭证与 Amazon Aurora DSQL 进行身份验证。该连接器不直接连接到数据库，而是在底层客户端库之上，提供无缝的 IAM 身份验证。

关于连接器

Amazon Aurora DSQL 是一种分布式 SQL 数据库服务，面向兼容 PostgreSQL 的应用程序提供高可用性和可扩展性。Aurora DSQL 要求使用基于 IAM 的身份验证以及限时令牌，而现有 Python 库本身不支持这种方法。

设计适用于 Python 的 Aurora DSQL 连接器的理念是，在 psycopg、psycopg2 和 asyncpg 客户端库之上添加一个身份验证层来处理 IAM 令牌生成，使得用户无需更改现有工作流即可连接到 Aurora DSQL。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌

- 令牌生成：使用 AWS 凭证生成身份验证令牌，其生命周期可配置

适用于 Python 的 Aurora DSQL 连接器针对这些要求而设计，在建立连接时自动生成 IAM 身份验证令牌。

功能

- 自动 IAM 身份验证：通过 AWS 凭证自动生成 IAM 令牌
- 基于 psycopg、psycopg2 和 asyncpg 构建：借助 psycopg、psycopg2 和 asyncpg 客户端库
- 无缝集成：适用于现有的 psycopg、psycopg2 和 asyncpg 连接模式，而无需更改工作流
- 区域自动发现：从 DSQL 集群主机名中提取 AWS 区域
- AWS 凭证支持：支持各种 AWS 凭证提供者（默认、基于配置文件、自定义）
- 连接池兼容性：适用于 psycopg、psycopg2 和 asyncpg 内置连接池

快速入门指南

要求

- Python 3.10 或更高版本
- 有权访问 Aurora DSQL 集群
- 设置适当的 IAM 权限，以允许应用程序连接到 Aurora DSQL。
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）。

安装

```
pip install aurora-dsql-python-connector
```

单独安装 psycopg、psycopg2 或 asyncpg

适用于 Python 的 Aurora DSQL 连接器安装程序不会安装底层库。这些库需要单独安装，例如：

```
# Install psycopg and psycopg pool
pip install "psycopg[binary,pool]"
```

```
# Install psycopg2
pip install psycopg2-binary
```

```
# Install asyncpg
```

```
pip install asyncpg
```

注意。

只需安装所需的库。因此，如果客户端将使用 psycopg，则只需安装 psycopg。如果客户端将使用 psycopg2，则只需安装 psycopg2。如果客户端将使用 asyncpg，则只需安装 asyncpg。

如果客户端需要使用多个库，则需安装全部所需的库。

基本用法

psycopg

```
import aurora_dsql_psycopg as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = await dsql.connect(**config)
result = await conn.fetchrow("SELECT 1")
await conn.close()
print(result)
```

仅使用主机

psycopg

```
import aurora_dsql_psycopg as dsql

conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

conn = await dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

仅使用集群 ID

psycopg

```
import aurora_dsql_psycopg as dsql
```

```
conn = dsql.connect("your-cluster")
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

conn = dsql.connect("your-cluster")
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

conn = await dsql.connect("your-cluster")
```

注意。

在“仅使用集群 ID”场景中，将使用之前在机器上设置的区域，例如：

```
aws configure set region us-east-1
```

如果尚未设置区域，或给定的集群 ID 位于其他区域，则连接将失败。如需正常连接，请将区域作为参数提供，如以下示例中所示：

psycopg

```
import aurora_dsql_psycopg as dsql

config = {
        "region": "us-east-1",
}

conn = dsql.connect("your-cluster", **config)
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql
```

```
    config = {
            "region": "us-east-1",
    }

    conn = dsql.connect("your-cluster", **config)
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    config = {
            "region": "us-east-1",
    }

    conn = await dsql.connect("your-cluster", **config)
```

连接字符串

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/
postgres?user=admin&token_duration_secs=15")
```

高级配置

psycopg

```
import aurora_dsql_psycopg as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
    "profile": "default",
    "token_duration_secs": "15",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
    "profile": "default",
    "token_duration_secs": "15",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql
```

```
    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
        "profile": "default",
        "token_duration_secs": "15",
    }

    conn = await dsql.connect(**config)
    result = await conn.fetchrow("SELECT 1")
    await conn.close()
    print(result)
```

配置选项

Option	类型	必需	描述
host	string	是	DSQL 集群主机名或集群 ID
user	string	否	DSQL 用户名。默认值：admin
dbname	string	否	数据库名称。默认值：postgres
region	string	否	AWS 区域（如果未提供，则自动从主机名中检测）
port	int	否	默认值为 5432
custom_credentials_provider	CredentialProvider	否	自定义 AWS 凭证提供者
profile	string	否	IAM 配置文件名称。默认值：default。
token_duration_secs	int	否	令牌过期时间（以秒为单位）

此外，底层 psycopg、psycopg2 及 asyncpg 库的所有标准连接选项均受支持，但 DSQL 不支持的 asyncpg 参数 krbsrvname 和 gsslib 例外。

将适用于 Python 的 Aurora DSQL 连接器与连接池结合使用

适用于 Python 的 Aurora DSQL 连接器可与 psycopg、psycopg2 和 asyncpg 内置连接池结合使用。连接器在建立连接期间处理 IAM 令牌生成，使得连接池可以正常操作。

psycopg

对于 psycopg，该连接器实现了一个名为 DSQLConnection 的连接类，该类可以直接传递给 psycopg_pool.ConnectionPool 构造函数。对于异步操作，还提供了该类的异步版本 DSQLAsyncConnection。

```
from psycopg_pool import ConnectionPool as PsycopgPool

...
pool = PsycopgPool(
    "",
    connection_class=dsql.DSQLConnection,
    kwargs=conn_params,
    min_size=2,
    max_size=8,
    max_lifetime=3300
)
```

注意：连接 max_Lifetime 配置

max_lifetime 参数应设置为小于 3600 秒（1 小时），因为这是 Aurora DSQL 数据库允许的最大连接持续时间。通过将 max_lifetime 设置为更低的值，可让连接池主动管理连接回收，这比处理数据库引发的连接超时错误更高效。

psycopg2

对于 psycopg2，该连接器提供了一个名为 AuroraDSQLThreadedConnectionPool 的类，此类继承自 psycopg2.pool.ThreadedConnectionPool。AuroraDSQLThreadedConnectionPool 类只能覆盖内部 _connect 方法。实施的其余部分由 psycopg2.pool.ThreadedConnectionPool 提供，保持不变。

```
import aurora_dsql_psycopg2 as dsql

pool = dsql.AuroraDSQLThreadedConnectionPool(
        minconn=2,
        maxconn=8,
        **conn_params,
```

```
    )
```

asyncpg

对于 asyncpg，该连接器提供了一个 create_pool 函数，该函数可返回 asyncpg.Pool 的实例。

```
import asyncio
import os

import aurora_dsql_asyncpg as dsql

pool_params = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'user': "admin",
    "min_size": 2,
    "max_size": 5,
}

pool = await dsql.create_pool(**pool_params)
```

身份验证

该连接器通过使用 DSQL 客户端令牌生成器生成令牌来自动处理 DSQL 身份验证。如果未提供 AWS 区域，系统会自动从提供的主机名中解析该区域。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅用户指南。

管理员用户与普通用户

- 名为 "admin" 的用户会自动使用管理员身份验证令牌
- 所有其他用户都使用非管理员身份验证令牌
- 令牌将为每个连接动态生成

示例

有关完整示例代码，请参阅以下各部分中所示的示例。有关如何运行示例的说明，请参阅示例 READMDE 文件。

psycopg

示例 README

说明	示例
使用适用于 Python 的 Aurora DSQL 连接器进行基本连接	基本连接示例
使用适用于 Python 的 Aurora DSQL 连接器进行基本异步连接	基本异步连接示例
将适用于 Python 的 Aurora DSQL 连接器与连接池结合使用	带连接池的基本连接示例
	带连接池的并发连接示例
将适用于 Python 的 Aurora DSQL 连接器与异步连接池结合使用	带异步连接池的基本连接示例

psycopg2

示例 README

说明	示例
使用适用于 Python 的 Aurora DSQL 连接器进行基本连接	基本连接示例
将适用于 Python 的 Aurora DSQL 连接器与连接池结合使用	带连接池的基本连接示例
	带连接池的并发连接示例

asyncpg

示例 README

说明	示例
使用适用于 Python 的 Aurora DSQL 连接器进行基本连接	基本连接示例
将适用于 Python 的 Aurora DSQL 连接器与连接池结合使用	带连接池的基本连接示例
	带连接池的并发连接示例

使用 Go 连接器连接到 Aurora DSQL 集群

Aurora DSQL Connector for Go 使用自动 IAM 身份验证封装 pgx。此连接器处理令牌生成、SSL 配置和连接管理，因此您可以专注于应用程序逻辑。

关于连接器

Aurora DSQL 要求使用基于 IAM 的身份验证以及限时令牌，而现有 Go PostgreSQL 驱动程序并不原生支持这种方法。适用于 Go 的 Aurora DSQL 连接器在 pgx 驱动程序之上添加一个身份验证层来处理 IAM 令牌生成，使您无需更改现有 pgx 工作流程即可连接到 Aurora DSQL。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：连接器使用 AWS 凭证生成身份验证令牌，并且这些令牌具有可配置的生命周期

适用于 Go 的 Aurora DSQL 连接器旨在满足这些要求，并在建立连接时自动生成 IAM 身份验证令牌。

适用于 Go 的 Aurora DSQL 连接器的优势

通过适用于 Go 的 Aurora DSQL 连接器，您可以继续使用现有的 pgx 工作流程，同时通过以下方式启用 IAM 身份验证：

- 自动生成令牌：连接器会自动为每个连接生成 IAM 令牌
- 连接池：内置对 `pgxpool` 的支持，对于每个连接自动生成令牌
- 灵活的配置：通过区域自动检测功能支持完整端点或集群 ID
- AWS 凭证支持：支持 AWS 配置文件和自定义凭证提供商

主要 功能

自动令牌管理

连接器使用预先解析的凭证为每个新连接自动生成 IAM 令牌。

连接池

通过 `pgxpool` 实现连接池，对于每个连接自动生成令牌。

灵活的主机配置

通过自动区域检测功能同时支持完整集群端点或集群 ID。

SSL 安全性

SSL 始终处于启用状态，具有完整验证模式和直接 TLS 协商。

先决条件

- Go 1.24 或更高版本
- 配置了 AWS 凭证
- 一个 Aurora DSQL 集群

此连接器使用适用于 Go 的 AWS SDK v2 默认凭证链，该凭证链按以下顺序解析凭证：

1. 环境变量 (AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY)
2. 共享的凭证文件（~/.aws/credentials）
3. 共享的配置文件（~/.aws/config）
4. 适用于 Amazon EC2/ECS/Lambda 的 IAM 角色

安装

使用 Go 模块安装连接器：

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql
```

快速入门

以下示例演示如何创建连接池和执行查询：

```
package main

import (
    "context"
    "log"

    "github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql"
)

func main() {
    ctx := context.Background()

    // Create a connection pool
    pool, err := dsql.NewPool(ctx, dsql.Config{
        Host: "your-cluster.dsql.us-east-1.on.aws",
    })
    if err != nil {
        log.Fatal(err)
    }
    defer pool.Close()

    // Execute a query
    var greeting string
    err = pool.QueryRow(ctx, "SELECT 'Hello, DSQL!'").Scan(&greeting)
    if err != nil {
        log.Fatal(err)
    }
    log.Println(greeting)
}
```

配置选项

此连接器支持以下配置选项：

字段	类型	默认值	说明
主机	字符串	（必需）	集群端点或集群 ID
区域	字符串	（自动检测）	AWS 区域；如果主机是集群 ID，则为必需
用户	字符串	“admin”	数据库用户
数据库	字符串	“postgres”	数据库名称
端口	int	5432	数据库端口
配置文件	字符串	""	凭证的 AWS 配置文件名称
TokenDurationSecs	int	900（15 分钟）	令牌有效期以秒为单位（支持的最长时间：1 周，默认值：15 分钟）
MaxConns	int32	0	最大池连接数（0 = pgxpool 默认值）
MinConns	int32	0	最小池连接数（0 = pgxpool 默认值）
MaxConnLifetime	time.Duration	55 分钟	最大连接生命周期

连接字符串格式

该连接器支持 PostgreSQL 和 DSQL 连接字符串格式：

```
postgres://[user@]host[:port]/[database][?param=value&...]
dsql://[user@]host[:port]/[database][?param=value&...]
```

支持的查询参数：

- `region`：AWS 区域
- `profile`：AWS 配置文件名称
- `tokenDurationSecs`：令牌有效期，以秒为单位

示例：

```
// Full endpoint (region auto-detected)
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres")

// Using dsql:// scheme (also supported)
pool, _ := dsql.NewPool(ctx, "dsql://admin@cluster.dsql.us-east-1.on.aws/postgres")

// With explicit region
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/mydb?
region=us-east-1")

// With AWS profile
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?
profile=dev")
```

高级用法

主机配置

该连接器支持两种主机格式：

完整端点（自动检测区域）：

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
```

集群 ID（需要区域）：

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
    Host:   "your-cluster-id",
    Region: "us-east-1",
})
```

池配置调整

为您的工作负载配置连接池：

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:              "your-cluster.dsql.us-east-1.on.aws",
    MaxConns:          20,
    MinConns:          5,
    MaxConnLifetime:   time.Hour,
```

```
    MaxConnIdleTime:   30 * time.Minute,
    HealthCheckPeriod: time.Minute,
})
```

单一连接使用

对于简单的脚本或不需要连接池时：

```
conn, err := dsql.Connect(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
if err != nil {
    log.Fatal(err)
}
defer conn.Close(ctx)

// Use the connection
rows, err := conn.Query(ctx, "SELECT * FROM users")
```

使用 AWS 配置文件

为凭证指定 AWS 配置文件：

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:    "your-cluster.dsql.us-east-1.on.aws",
    Profile: "production",
})
```

OCC 重试

Aurora DSQL 使用乐观并发控制（OCC）。当两个事务修改相同的数据时，提交的第一个事务获胜，第二个事务收到 OCC 错误。

`occretry` 程序包为带指数回退和抖动的自动重试提供了帮助程序。使用以下命令进行安装：

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/occretry
```

使用 `WithRetry` 进行事务写入：

```
err := occretry.WithRetry(ctx, pool, occretry.DefaultConfig(), func(tx pgx.Tx) error {
    _, err := tx.Exec(ctx, "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
 100, fromID)
```

```
    if err != nil {
        return err
    }
    _, err = tx.Exec(ctx, "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
 100, toID)
    return err
})
```

对于 DDL 或单个语句，使用 `ExecWithRetry`：

```
err := occretry.ExecWithRetry(ctx, pool, occretry.DefaultConfig(),
    "CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY, name TEXT)")
```

> **Important**
>
> `WithRetry` 在内部管理 `BEGIN/COMMIT/ROLLBACK`。您的回调会收到一个事务，应仅包含数据库操作并可以安全地重试。

示例

有关更全面的示例和使用案例，请参阅适用于 Go 的 Aurora DSQL 连接器示例。

示例	说明
example_preferred	建议：带有并发查询的连接池
交易	使用 BEGIN/COMMIT/ROLLBACK 处理事务
occ_retry	使用指数回退处理 OCC 冲突
connection_string	使用连接字符串进行配置

适用于 Node.js 的 Aurora DSQL 连接器

适用于 node-postgres 的 Aurora DSQL 连接器和适用于 Postgres.js 的 Aurora DSQL 连接器均为身份验证插件，扩展了 node-postgres 和 Postgres.js 客户端的功能，使应用程序能够使用 IAM 凭证与 Aurora DSQL 进行身份验证。

适用于 node-postgres 的 Aurora DSQL 连接器

Aurora DSQL Connector for node-postgres 是一款基于 node-postgres 构建的 Node.js 连接器，它集成了 IAM 身份验证功能，可用于将 JavaScript/TypeScript 应用程序连接到 Amazon Aurora DSQL 集群。

经设计，Aurora DSQL 连接器可作为身份验证插件，对 node-postgres 客户端和池的功能进行扩展，使应用程序能够使用 IAM 凭证与 Amazon Aurora DSQL 进行身份验证。

关于连接器

Amazon Aurora DSQL 是一款兼容 PostgreSQL 的云原生分布式数据库。该数据库要求使用 IAM 身份验证和有时限的令牌，而传统的 Node.js 数据库驱动程序缺少此内置支持。

适用于 node-postgres 的 Aurora DSQL 连接器通过实施可与 node-postgres 无缝协作的身份验证中间件，弥补了这一不足。此方法可让开发人员保留现有的 node-postgres 代码，并通过自动化令牌管理，获得对 Aurora DSQL 集群的基于 IAM 的安全访问权限。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：使用 AWS 凭证生成身份验证令牌，其生命周期可配置

适用于 node-postgres 的 Aurora DSQL 连接器针对这些要求而设计，在建立连接时自动生成 IAM 身份验证令牌。

功能

- 自动 IAM 身份验证：处理 DSQL 令牌的生成与刷新
- 基于 node-postgres 构建：借助适用于 Node.js 的常用 PostgreSQL 客户端
- 无缝集成：适用于现有 node-postgres 连接模式
- 区域自动发现：从 DSQL 集群主机名中提取 AWS 区域
- 完整 TypeScript 支持：提供完全类型安全性
- AWS 凭证支持：支持各种 AWS 凭证提供者（默认、基于配置文件、自定义）
- 连接池兼容性：与内置连接池无缝协作

示例应用程序

example 中包含一个示例应用程序，展示了如何使用适用于 node-postgres 的 Aurora DSQL 连接器。要运行包含的示例，请参阅示例 README。

快速入门指南

要求

- Node.js 20+
- 有权访问 Aurora DSQL 集群
- 设置适当的 IAM 权限，以允许应用程序连接到 Aurora DSQL。
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）。

安装

```
npm install @aws/aurora-dsql-node-postgres-connector
```

对等依赖项

```
npm install @aws-sdk/credential-providers @aws-sdk/dsql-signer pg tsx
npm install --save-dev @types/pg
```

用法

客户端连接

```
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
});
await client.connect();
const result = await client.query("SELECT NOW()");
await client.end();
```

池连接

```
import { AuroraDSQLPool } from "@aws/aurora-dsql-node-postgres-connector";
```

```
const pool = new AuroraDSQLPool({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
  max: 3,
  idleTimeoutMillis: 60000,
});

const result = await pool.query("SELECT NOW()");
```

高级用法

```
import { fromNodeProviderChain } from "@aws-sdk/credential-providers";
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "example.dsql.us-east-1.on.aws",
  user: "admin",
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
});

await client.connect();
const result = await client.query("SELECT NOW()");
await client.end();
```

配置选项

Option	类型	必需	描述
`host`	`string`	是	DSQL 集群主机名
`username`	`string`	是	DSQL 用户名
`database`	`string`	否	数据库名称
`region`	`string`	否	AWS 区域（如果未提供，则自动从主机名中检测）
`port`	`number`	否	默认值为 5432

Option	类型	必需	描述
customCredentialsProvider	AwsCredentialIdentity / AwsCredentialIdentityProvider	否	自定义 AWS 凭证提供者
profile	string	否	IAM 配置文件名称。默认值为“default”
tokenDurationSecs	number	否	令牌过期时间（以秒为单位）

支持来自 Client/Pool 的所有其他参数。

身份验证

该连接器通过使用 DSQL 客户端令牌生成器生成令牌来自动处理 DSQL 身份验证。如果未提供 AWS 区域，系统会自动从提供的主机名中解析该区域。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅用户指南。

管理员用户与普通用户

- 名为“admin”的用户会自动使用管理员身份验证令牌
- 所有其他用户都使用普通身份验证令牌
- 令牌将为每个连接动态生成

适用于 Postgres.js 的 Aurora DSQL 连接器

Aurora DSQL Connector for Postgres.js 是一款基于 Postgres.js 构建的 Node.js 连接器，它集成了 IAM 身份验证功能，可用于将 JavaScript 应用程序连接到 Amazon Aurora DSQL 集群。

经设计，适用于 Postgres.js 的 Aurora DSQL 连接器可作为身份验证插件，对 Postgres.js 客户端的功能进行扩展，使应用程序能够使用 IAM 凭证与 Amazon Aurora DSQL 进行身份验证。该连接器不直接连接到数据库，而是在底层 Postgres.js 驱动程序之上，提供无缝的 IAM 身份验证。

关于连接器

Amazon Aurora DSQL 是一种分布式 SQL 数据库服务，面向兼容 PostgreSQL 的应用程序提供高可用性和可扩展性。Aurora DSQL 要求使用基于 IAM 的身份验证以及限时令牌，而现有 Node.js 驱动程序本身不支持这种方法。

设计适用于 Postgres.js 的 Aurora DSQL 连接器的理念是，在 Postgres.js 客户端之上添加一个身份验证层来处理 IAM 令牌生成，使得用户无需更改现有 Postgres.js 工作流即可连接到 Aurora DSQL。

适用于 Postgres.js 的 Aurora DSQL 连接器可以用于大多数版本的 Postgres.js。用户可通过直接安装 Postgres.js 来提供自己的版本。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：使用 AWS 凭证生成身份验证令牌，其生命周期可配置

适用于 Postgres.js 的 Aurora DSQL 连接器针对这些要求而设计，在建立连接时自动生成 IAM 身份验证令牌。

功能

- 自动 IAM 身份验证：处理 DSQL 令牌的生成与刷新
- 基于 Postgres.js 构建：借助适用于 Node.js 的高性能 PostgreSQL 客户端
- 无缝集成：适用于现有 Postgres.js 连接模式
- 区域自动发现：从 DSQL 集群主机名中提取 AWS 区域
- 完整 TypeScript 支持：提供完全类型安全性
- AWS 凭证支持：支持各种 AWS 凭证提供者（默认、基于配置文件、自定义）
- 连接池兼容性：与 Postgres.js 内置连接池无缝协作

快速入门指南

要求

- Node.js 20+
- 有权访问 Aurora DSQL 集群

- 设置适当的 IAM 权限，以允许应用程序连接到 Aurora DSQL。
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）。

安装

```
npm install @aws/aurora-dsql-postgresjs-connector
# Postgres.js is a peer-dependency, so users must install it themselves
npm install postgres
```

基本用法

```
import { auroraDSQLPostgres } from '@aws/aurora-dsql-postgresjs-connector';

const sql = auroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  username: 'admin',

});

// Execute queries
const result = await sql`SELECT current_timestamp`;
console.log(result);

// Clean up
await sql.end();
```

使用集群 ID 而非主机

```
const sql = auroraDSQLPostgres({
  host: 'your-cluster-id',
  region: 'us-east-1',
  username: 'admin',

});
```

连接字符串

```
const sql = AuroraDSQLPostgres(
  'postgres://admin@your-cluster.dsql.us-east-1.on.aws'
);
```

```
const result = await sql`SELECT current_timestamp`;
```

高级配置

```
import { fromNodeProviderChain } from '@aws-sdk/credential-providers';

const sql = AuroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  database: 'postgres',
  username: 'admin',
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
  tokenDurationSecs: 3600,                            // Token expiration (seconds)

  // Standard Postgres.js options
  max: 20,                                  // Connection pool size
  ssl: { rejectUnauthorized: false }        // SSL configuration
});
```

配置选项

Option	类型	必需	描述
host	string	是	DSQL 集群主机名或集群 ID
database	string?	否	数据库名称
username	string?	否	数据库用户名（如果未提供，则使用 admin）
region	string?	否	AWS 区域（如果未提供，则自动从主机名中检测）
customCredentialsProvider	AwsCredentialIdentityProvider?	否	自定义 AWS 凭证提供者
tokenDurationSecs	number?	否	令牌过期时间（以秒为单位）

还支持所有标准 Postgres.js 选项。

身份验证

该连接器通过使用 DSQL 客户端令牌生成器生成令牌来自动处理 DSQL 身份验证。如果未提供 AWS 区域，系统会自动从提供的主机名中解析该区域。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅用户指南。

管理员用户与普通用户

- 名为“admin”的用户会自动使用管理员身份验证令牌
- 所有其他用户都使用普通身份验证令牌
- 令牌将为每个连接动态生成

示例用法

GitHub 上提供了使用适用于 Postgres.js 的 Aurora DSQL 连接器的 JavaScript 示例。有关如何运行这些示例的说明，请参阅示例目录。

说明	示例
使用并发查询实现连接池，包括跨多个工作线程创建表、插入和读取	连接池示例（首选）
没有连接池的 CRUD 操作（创建表、插入、选择、删除）	没有连接池的示例

使用 Ruby 连接器连接到 Aurora DSQL 集群

Aurora DSQL Connector for Ruby 是一款基于 pg 构建的 Ruby 连接器，它集成了 IAM 身份验证功能，用于将 Ruby 应用程序连接到 Amazon Aurora DSQL 集群。

此连接器处理令牌生成、SSL 配置和连接池，因此您可以专注于应用程序逻辑。

关于连接器

Amazon Aurora DSQL 要求使用 IAM 身份验证以及限时令牌，而现有 Ruby PostgreSQL 驱动程序并不原生支持这种方法。适用于 Ruby 的 Aurora DSQL 连接器在 pg gem 之上添加一个身份验证层来处理 IAM 令牌生成，使您无需更改现有 pg 工作流程即可连接到 Aurora DSQL。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：连接器使用 AWS 凭证生成身份验证令牌，并且这些令牌具有可配置的生命周期

适用于 Ruby 的 Aurora DSQL 连接器了解这些要求，并在建立连接时自动生成 IAM 身份验证令牌。

功能

- 自动 IAM 身份验证：处理 Aurora DSQL 令牌生成与刷新
- 在 pg 之上构建：封装了适用于 Ruby 的常用 PostgreSQL gem
- 无缝集成：可与现有的 pg gem 工作流程结合使用
- 连接池：通过 `connection_pool` gem 提供内置支持，具有 max_lifetime 强制执行
- 区域自动检测：从 Aurora DSQL 集群主机名中提取 AWS 区域
- AWS 凭证支持：支持 AWS 配置文件和自定义凭证提供商
- OCC 重试：选择加入乐观并发控制重试，具有指数回退

示例应用程序

有关完整的示例，请参阅 GitHub 上的示例应用程序。

快速入门指南

要求

- Ruby 3.1 或更高版本
- 有权访问 Aurora DSQL 集群
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）

安装

添加到您的 Gemfile：

```
gem "aurora-dsql-ruby-pg"
```

或者直接安装：

```
gem install aurora-dsql-ruby-pg
```

用法

池连接

```
require "aurora_dsql_pg"

# Create a connection pool with OCC retry enabled
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

# Read
pool.with do |conn|
  result = conn.exec("SELECT 'Hello, DSQL!'")
  puts result[0]["?column?"]
end

# Write — you must wrap writes in a transaction
pool.with do |conn|
  conn.transaction do
    conn.exec_params("INSERT INTO users (id, name) VALUES (gen_random_uuid(), $1)",
 ["Alice"])
  end
end

pool.shutdown
```

单一 连接

对于简单的脚本或不需要连接池时：

```
conn = AuroraDsql::Pg.connect(host: "your-cluster.dsql.us-east-1.on.aws")
conn.exec("SELECT 1")
conn.close
```

高级用法

主机配置

该连接器支持完整的集群端点（自动检测区域）和集群 ID（需要区域）：

```
# Full endpoint (region auto-detected)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws"
)

# Cluster ID (region required)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster-id",
  region: "us-east-1"
)
```

AWS 配置文件

为凭证指定 AWS 配置文件：

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  profile: "production"
)
```

连接字符串格式

该连接器支持 PostgreSQL 连接字符串格式：

```
postgres://[user@]host[:port]/[database][?param=value&...]
postgresql://[user@]host[:port]/[database][?param=value&...]
```

支持的查询参数：`region`、`profile`、`tokenDurationSecs`。

```
# Full endpoint with profile
```

```
pool = AuroraDsql::Pg.create_pool(
  "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?profile=dev"
)
```

OCC 重试

Aurora DSQL 使用乐观并发控制（OCC）。当两个事务修改相同的数据时，提交的第一个事务获胜，第二个事务收到 OCC 错误。

OCC 重试为选择加入的。当创建池时设置 `occ_max_retries`，以在 `pool.with` 上启用带指数回退和抖动的自动重试功能：

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

pool.with do |conn|
  conn.transaction do
    conn.exec_params("UPDATE accounts SET balance = balance - $1 WHERE id = $2", [100,
 from_id])
    conn.exec_params("UPDATE accounts SET balance = balance + $1 WHERE id = $2", [100,
 to_id])
  end
end
```

> **Warning**
>
> `pool.with` 不会在事务中自动封装块。您必须自行为写入操作调用 `conn.transaction`。发生 OCC 冲突时，连接器会重新执行整个块，因此它应仅包含数据库操作并且可以安全地重试。

要对单个调用跳过重试，请传递 `retry_occ: false`：

```
pool.with(retry_occ: false) do |conn|
  conn.exec("SELECT 1")
end
```

配置选项

字段	类型	默认值	说明
host	字符串	（必需）	集群端点或集群 ID
region	字符串	（自动检测）	AWS 区域；如果主机是集群 ID，则为必需
用户	字符串	“admin”	数据库用户
database	字符串	“postgres”	数据库名称
端口	整数	5432	数据库端口
配置文件	字符串	nil	凭证的 AWS 配置文件名称
token_duration	整数	900（15 分钟）	令牌有效期以秒为单位（支持的最长时间：1 周，默认值：15 分钟）
credentials_provider	Aws::Credentials	nil	自定义凭证提供商
max_lifetime	整数	3300（55 分钟）	最大连接生命周期，以秒为单位
application_name	字符串	nil	application_name 的 ORM 前缀
日志记录程序	记录器	nil	OCC 重试警告的记录器
occ_max_retries	整数	nil（已禁用）	`pool.with` 上的 OCC 最大重试次数；设置后启用重试

`create_pool` 还接受带有选项哈希值的 `pool:` 关键字，您可以直接将其传递给 `ConnectionPool.new`。如果忽略 `pool:`，则连接器默认为 `{size: 5, timeout: 5}`。您提供的键仅会覆盖那些特定的默认值。

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  pool: { size: 10, timeout: 10 }
```

```
)
```

身份验证

该连接器通过使用 AWS 凭证生成令牌来自动处理 Aurora DSQL 身份验证。如果您未提供 AWS 区域，则连接器会从主机名中解析该区域。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅 Aurora DSQL 的身份验证和授权。

管理员用户与普通用户

- 名为“admin”的用户会自动使用管理员身份验证令牌
- 所有其他用户都使用普通身份验证令牌
- 该连接器为每个连接动态生成令牌

使用 .NET 连接器连接到 Aurora DSQL 集群

Amazon Aurora DSQL Connector for .NET 是一款基于 Npgsql 构建的 .NET 连接器，它集成了 IAM 身份验证功能，用于将 .NET 应用程序连接到 Amazon Aurora DSQL 集群。

此连接器处理令牌生成、SSL 配置和连接池，因此您可以专注于应用程序逻辑。

关于连接器

Amazon Aurora DSQL 要求使用 IAM 身份验证以及限时令牌，而现有 .NET PostgreSQL 驱动程序并不原生支持这种方法。适用于 .NET 的 Aurora DSQL 连接器在 Npgsql 之上添加一个身份验证层来处理 IAM 令牌生成，使您无需更改现有 Npgsql 工作流程即可连接到 Aurora DSQL。

什么是 Aurora DSQL 身份验证？

在 Aurora DSQL 中，身份验证包括：

- IAM 身份验证：所有连接都使用基于 IAM 的身份验证和限时令牌
- 令牌生成：连接器使用 AWS 凭证生成身份验证令牌，并且这些令牌具有可配置的生命周期

适用于 .NET 的 Aurora DSQL 连接器了解这些要求，并在建立连接时自动生成 IAM 身份验证令牌。

功能

- 自动 IAM 身份验证：处理 Aurora DSQL 令牌生成与刷新

- 基于 Npgsql 构建：封装了适用于 .NET 的常用 PostgreSQL 驱动程序
- 无缝集成：可与现有的 Npgsql 工作流程结合使用
- 连接池：通过 `NpgsqlDataSource` 提供内置支持，强制执行最长使用寿命
- 区域自动检测：从 Aurora DSQL 集群主机名中提取 AWS 区域
- AWS 凭证支持：支持 AWS 配置文件和自定义凭证提供商
- OCC 重试：选择加入乐观并发控制重试，具有指数回退
- SSL 强制执行：始终在 `verify-full` 模式和直接 TLS 协商中使用 SSL

示例应用程序

有关完整的示例，请参阅 GitHub 上的示例应用程序。

快速入门指南

要求

- .NET 8.0 或更高版本
- 有权访问 Aurora DSQL 集群
- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）

安装

将程序包添加到您的项目：

```
dotnet add package Amazon.AuroraDsql.Npgsql
```

用法

池连接

```
using Amazon.AuroraDsql.Npgsql;

// Create a connection pool
await using var ds = await AuroraDsql.CreateDataSourceAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws",
```

```
    OccMaxRetries = 3
});

// Read
await using (var conn = await ds.OpenConnectionAsync())
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT 'Hello, DSQL!'";
    var greeting = await cmd.ExecuteScalarAsync();
    Console.WriteLine(greeting);
}

// Transactional write with OCC retry
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "INSERT INTO users (id, name) VALUES (gen_random_uuid(), @name)";
    cmd.Parameters.AddWithValue("name", "Alice");
    await cmd.ExecuteNonQueryAsync();
});
```

单一 连接

对于简单的脚本或当您不需要连接池时：

```
await using var conn = await AuroraDsql.ConnectAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws"
});

await using var cmd = conn.CreateCommand("SELECT 1");
await cmd.ExecuteScalarAsync();
```

OCC 重试

Aurora DSQL 使用乐观并发控制（OCC）。当两个事务修改相同的数据时，提交的第一个事务获胜，第二个事务收到 OCC 错误。

OCC 重试为选择加入的。在配置中设置 `OccMaxRetries`，以启用带指数回退和抖动的自动重试功能。使用 `WithTransactionRetryAsync` 进行事务写入：

```
await ds.WithTransactionRetryAsync(async conn =>
```

```
{
    await using var cmd = conn.CreateCommand();

    cmd.CommandText = "UPDATE accounts SET balance = balance - 100 WHERE id = @from";
    cmd.Parameters.AddWithValue("from", fromId);
    await cmd.ExecuteNonQueryAsync();

    cmd.CommandText = "UPDATE accounts SET balance = balance + 100 WHERE id = @to";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("to", toId);
    await cmd.ExecuteNonQueryAsync();
});
```

对于 DDL 或单个语句，使用 `ExecWithRetryAsync`：

```
await ds.ExecWithRetryAsync("CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY,
 name TEXT)");
```

Important

`WithTransactionRetryAsync` 在内部管理 BEGIN/COMMIT/ROLLBACK，每次尝试都会打开一个全新的连接。您的回调应仅包含数据库操作，并且可以安全地重试。

配置选项

连接器还通过 `region` 和 `profile` 查询参数接受 `postgres://` 和 `postgresql://` 连接字符串。

字段	类型	默认值	说明
Host	string	（必需）	集群端点或 26 个字符集群 ID
Region	string?	（自动检测）	AWS 区域；如果 Host 是集群 ID，则为必需
User	string	"admin"	数据库用户
Database	string	"postgres"	数据库名称
Port	int	5432	数据库端口

字段	类型	默认值	说明
Profile	string?	null	凭证的 AWS 配置文件名称
CustomCredentialsProvider	AWSCredentials?	null	自定义 AWS 凭证提供商
TokenDurationSecs	int?	null（SDK 默认值，900 秒）	令牌有效期，以秒为单位
OccMaxRetries	int?	null（已禁用）	数据来源上重试方法的默认最大 OCC 重试次数
OrmPrefix	string?	null	附加在 application_name 之前的 ORM 前缀
LoggerFactory	ILoggerFactory?	null	用于重试警告和诊断的记录器工厂
ConfigureConnectionString	Action<NpgsqlConnectionStringBuilder>?	null	回调以覆盖池设置，或设置其它 Npgsql 连接字符串属性。SSL 和 Enlist 是安全不变量，不能被覆盖。

身份验证

该连接器通过使用 AWS 凭证生成令牌来自动处理 Aurora DSQL 身份验证。如果您未提供 AWS 区域，则连接器会从主机名中解析该区域。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅 Aurora DSQL 的身份验证和授权。

管理员用户与普通用户

- 名为“admin”的用户会自动使用管理员身份验证令牌
- 所有其他用户都使用普通身份验证令牌
- 该连接器为每个连接动态生成令牌

使用兼容 PostgreSQL 的客户端访问 Aurora DSQL

Aurora DSQL 使用 PostgreSQL 有线协议。您可以通过各种工具和客户端（例如，AWS CloudShell、psql、DBeaver 和 DataGrip）连接到 PostgreSQL。下表汇总了 Aurora DSQL 与常见 PostgreSQL 连接参数的对应关系：

PostgreSQL	Aurora DSQL	备注
角色（也称为用户或组）	数据库角色	Aurora DSQL 为您创建一个名为 `admin` 的角色。当您创建自定义数据库角色时，您必须使用 `admin` 角色将其与 IAM 角色关联，以便在连接到集群时进行身份验证。有关更多信息，请参阅 使用数据库角色和 IAM 身份验证。
主机（也称为主机名或主机规格）	集群端点	Aurora DSQL 单区域集群提供单个托管式端点，当区域内出现不可用性问题时会自动重定向流量。
端口	不适用：使用默认值 `5432`	这是 PostgreSQL 默认值。
数据库（dbname）	使用 `postgres`	Aurora DSQL 会在您创建集群时为您创建此数据库。
SSL 模式	始终在服务器端启用 SSL	在 Aurora DSQL 中，Aurora DSQL 支持 `require` SSL 模式。Aurora DSQL 会拒绝没有 SSL 的连接。
密码	身份验证令牌	Aurora DSQL 需要临时身份验证令牌，而不是长期密码。要了解更多信息，请参阅在 Amazon Aurora DSQL 中生成身份验证令牌。

连接时，Aurora DSQL 要求使用签名的 IAM 身份验证令牌替代传统密码。这些临时令牌通过 AWS 签名版本 4 生成，并且仅在建立连接期间使用。建立连接后，会话将保持活动状态，直至会话结束或客户端断开连接。

如果您尝试使用过期的令牌打开新会话，连接请求将失败，并且必须生成新令牌。有关更多信息，请参阅 在 Amazon Aurora DSQL 中生成身份验证令牌。

使用 SQL 客户端访问 Aurora DSQL

Aurora DSQL 支持通过多种兼容 PostgreSQL 的客户端来连接到您的集群。以下各部分介绍如何使用 AWS CloudShell 或本地命令行通过 PostgreSQL 进行连接，以及如何使用基于 GUI 的工具（例如 DBeaver 和 JetBrains DataGrip）进行连接。每个客户端均需要一个有效的身份验证令牌，如上一部分所述。

主题

- 使用 DBeaver 访问 Aurora DSQL
- 使用 JetBrains DataGrip 访问 Aurora DSQL
- 使用 PostgreSQL 交互式终端（psql）访问 Aurora DSQL
- 使用适用于 SQLTools 的 Aurora DSQL 驱动程序
- 问题排查

使用 DBeaver 访问 Aurora DSQL

DBeaver 是一款通用 SQL 客户端，可用于管理任何具有 JDBC 驱动程序的数据库。由于其强大的数据查看、编辑和管理功能，该工具被开发人员和数据库管理员广泛使用。使用 DBeaver 的云连接选项，可将 DBeaver 以原生方式连接到 Aurora DSQL。

DBeaver Pro

从版本 25.3 开始，DBeaver PRO 产品提供与 Aurora DSQL 的原生集成。按照 DBeaver Documentation 中的说明连接到 Aurora DSQL 集群。

DBeaver 社区版

DBeaver 社区版是免费的开源版本。请访问下载页面以查看安装说明。要从 DBeaver 社区版连接到 DSQL，您需要安装 Aurora DSQL Plugin for DBeaver。

Aurora DSQL Plugin for DBeaver 在 Aurora DSQL Connector for JDBC 之上构建，支持对 Aurora DSQL 集群进行 IAM 身份验证。它可通过 DBeaver UI 方便地安装，无需编写令牌生成代码或手动提供有效的 IAM 令牌，从而简化了身份验证，同时消除了与传统用户生成的密码关联的安全风险。

功能

- IAM 身份验证支持：使用 AWS IAM 凭证连接到 Aurora DSQL 集群，以实现安全、免密码的身份验证

- 自动驱动程序管理：无缝地安装和配置适用于 JDBC 的 Aurora DSQL 连接器
- 灵活的连接选项：在基于主机的连接配置或基于 JDBC URL 的连接配置之间选择

适用于 DBeaver 的 Aurora DSQL 插件安装

1. 打开 DBeaver 后，转至下拉菜单帮助 → 安装新软件
2. 单击添加以添加新的存储库
3. 输入：
   - 名称：`Aurora DSQL Plugin`
   - 位置：`https://awslabs.github.io/aurora-dsql-dbeaver-plugin/update-site/`
4. 选择适用于 JDBC 的 Aurora DSQL 连接器
5. 单击下一步，接受许可证，然后完成安装
6. 当系统提示时，重新启动 DBeaver

创建 Aurora DSQL 连接

1. 单击新建数据库连接
2. 选择 Aurora DSQL
3. 在服务器下，为连接方式设置选择以下选项之一
   - 主机
     - 为以下字段启用用户界面文本输入：
       - 端点：DSQL 集群端点
       - 用户名：DSQL 用户名（例如 admin）
       - AWS 配置文件：例如，默认配置文件，即未指定特定配置文件时使用的标准配置文件
       - AWS 区域（可选）：必须与您的 DSQL 集群所在的区域匹配，否则身份验证将失败
   - URL
     - 采用以下格式的 JDBC URL：

       ```
       jdbc:aws-dsql:postgresql://{cluster_endpoint}/{database}?
       user=admin&profile=default&region=us-east-1
       ```
     - 注意：在此模式下，仅启用 URL 输入。为了向 JDBC 连接字符串添加参数，请使用以 ? 开头的 URL 查询参数格式作为第一个参数，并为后续参数附加一个 &。
4. 单击测试连接以验证 Aurora DSQL 连接是否有效
5. 单击完成

问题排查

Windows Trust Store 问题

Windows 用户在从 Maven Central 下载适用于 JDBC 的 Aurora DSQL 连接器驱动程序时可能会遇到问题。

原因：Windows Trust Store 可能不包含访问 Maven Central 存储库所需的证书。

解决方案：

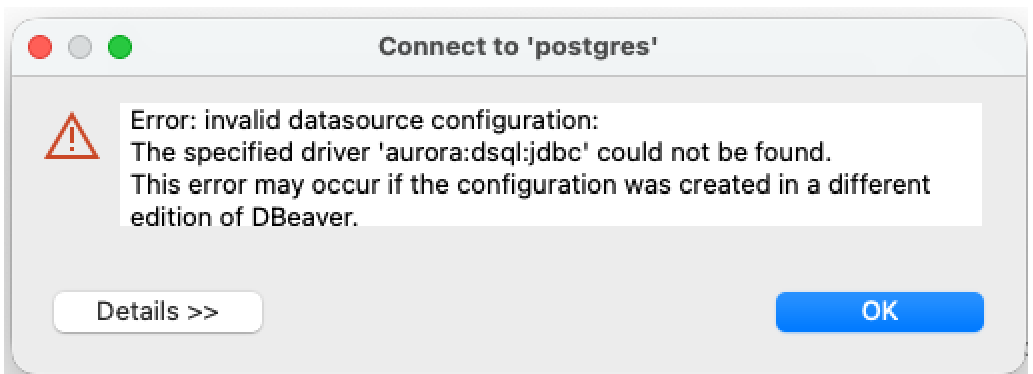
1. 以“管理员”身份运行 DBeaver
2. 取消选中此设置：Windows > 偏好设置 > 连接 >“使用 Windows Trust Store”

缺失驱动程序错误

如果您看到缺失驱动程序图标或连接错误，则说明您当前的 DBeaver 版本中可能未安装 Aurora DSQL（社区插件）。以下是一些错误示例及其修复方法：

- 创建与缺失驱动程序的新连接：

- 尝试在没有驱动程序的情况下进行连接：



原因：安装多个 DBeaver 版本时，连接设置是共享的，但驱动程序是按应用程序单独安装的。

解决方案：按照上述安装步骤重新安装 Aurora DSQL（社区插件）。

Important

DBeaver 为 PostgreSQL 数据库提供的管理功能（如会话管理器和锁定管理器）由于其独特的架构而不适用于 Aurora DSQL 数据库。虽然这些屏幕可供访问，但它们不提供有关数据库运行状况或状态的可靠信息。

使用 JetBrains DataGrip 访问 Aurora DSQL

JetBrains DataGrip 是一款跨平台 IDE，用于处理 SQL 和数据库，包括 PostgreSQL。DataGrip 包含一个强大的图形用户界面和一个智能 SQL 编辑器。要下载 DataGrip，请转至 JetBrains 网站上的下载页面。

在 JetBrains DataGrip 中设置新的 Aurora DSQL 连接

1. 选择新建数据来源，然后选择 PostgreSQL。
2. 在数据来源/常规选项卡中，输入以下信息：

    - 主机：使用您的集群端点。

      端口：Aurora DSQL 使用 PostgreSQL 默认值：`5432`

      数据库：Aurora DSQL 使用 PostgreSQL 默认值 `postgres`

      身份验证：选择 `User & Password` 。

      用户名：输入 `admin`。

      密码：生成令牌并将其粘贴到此字段中。

      URL：请勿修改此字段。它将根据其它字段自动填充。
3. 密码：通过生成身份验证令牌来提供密码。复制令牌生成器的结果输出，并将其粘贴到密码字段中。

    Note

    您必须在客户端连接中设置 SSL 模式。Aurora DSQL 支持 `PGSSLMODE=require and PGSSLMODE=verify-full`。Aurora DSQL 在服务器端强制执行 SSL 通信，并拒绝非

> SSL 连接。对于 `verify-full` 选项，您需要本地安装 SSL 证书。有关更多信息，请参阅 SSL/TLS 证书。

4. 您应该已连接到集群，并可以开始运行 SQL 语句：

> **Important**
>
> DataGrip 为 PostgreSQL 数据库提供的某些视图（例如会话）由于其独特的架构而不适用于 Aurora DSQL 数据库。虽然这些屏幕可供访问，但它们不提供有关连接到数据库的实际会话的可靠信息。

使用 PostgreSQL 交互式终端（psql）访问 Aurora DSQL

使用 AWS CloudShell 通过 PostgreSQL 交互式终端（psql）访问 Aurora DSQL

按照以下过程操作，使用 AWS CloudShell 通过 PostgreSQL 交互式终端访问 Aurora DSQL。有关更多信息，请参阅什么是 AWS CloudShell。

使用 AWS CloudShell 进行连接

1. 登录 Aurora DSQL 控制台。
2. 选择要在 CloudShell 中打开的集群。如果您尚未创建集群，请按照步骤 1：创建 Aurora DSQL 单区域集群或创建多区域集群中的步骤操作。
3. 选择使用查询编辑器进行连接，然后选择使用 CloudShell 进行连接。
4. 选择是要以管理员身份还是要使用自定义数据库角色进行连接。
5. 选择在 CloudShell 中启动，然后在以下 CloudShell 对话框中选择运行。

使用本地 CLI 通过 PostgreSQL 交互式终端（psql）访问 Aurora DSQL

使用 `psql`（一款基于终端的 PostgreSQL 前端实用程序）可通过交互方式输入查询，将查询发送到 PostgreSQL，并查看查询结果。

Note

要缩短查询响应时间，请使用 PostgreSQL 版本 17 客户端。如果您在不同的环境中使用 CLI，请务必手动设置 Python 版本 3.8+ 和 psql 版本 14+。

从 PostgreSQL Downloads 页面下载操作系统的安装程序。有关 psql 的更多信息，请参阅 PostgreSQL 网站上的 PostgreSQL 客户端应用程序。

如果您已经安装了 AWS CLI，请使用以下示例连接到集群。

```
# Aurora DSQL requires a valid IAM token as the password when connecting.
# Aurora DSQL provides tools for this and here we're using Python.
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token \
  --region us-east-1 \
  --expires-in 3600 \
  --hostname your_cluster_endpoint)

# Aurora DSQL requires SSL and will reject your connection without it.
export PGSSLMODE=require

# Connect with psql, which automatically uses the values set in PGPASSWORD and
 PGSSLMODE.
# Quiet mode suppresses unnecessary warnings and chatty responses but still outputs
 errors.
psql --quiet \
  --username admin \
  --dbname postgres \
  --host your_cluster_endpoint
```

使用适用于 SQLTools 的 Aurora DSQL 驱动程序

适用于 SQLTools 的 Aurora DSQL 驱动程序是 Amazon Aurora DSQL 的 Visual Studio 代码扩展，与 SQLTools 相集成。它使开发人员能够直接从 VS Code 连接和查询 Aurora DSQL 数据库。该驱动程序可从 Visual Studio Marketplace 和 Open VSX Registry 安装。Kiro、Cursor 和其它基于 VSCode 的 IDE 可以使用 Open VSX Registry，以便按照本页中描述的标准安装过程安装驱动程序。

功能

- 自动 IAM 身份验证
- 标准数据库操作，例如浏览架构、表和执行 SQL 查询。

安装

1. 打开“扩展”视图。
2. 搜索“适用于 SQLTools 的 Aurora DSQL 驱动程序”。
3. 单击“安装”。

注意。

如果 SQLTools 扩展尚不存在，则会自动安装该扩展。

身份验证

在 Aurora DSQL 中，所有连接都使用基于 IAM 的身份验证以及限时令牌。该驱动程序使用 Aurora DSQL Connector for node-postgres 来自动处理 Aurora DSQL 身份验证。

有关 Aurora DSQL 中的身份验证的更多信息，请参阅用户指南。

创建 Aurora DSQL 连接

先决条件

- 已配置 AWS 凭证（通过 AWS CLI、环境变量或 IAM 角色）。

Steps

1. 在左侧边栏中单击 SQLTools 图标。
2. 在 SQLTools 窗格中，将鼠标悬停在“连接”上方，然后单击“添加新连接”图标。
3. 在 SQLTools 的“设置”选项卡中，从列表中选择“Aurora DSQL 驱动程序”。
4. 填入连接参数。
    - AWS 区域
        - 可选：将从 Aurora DSQL 集群端点解析区域。
        - 在 DSQL 集群字段中仅指定集群 ID 时为必填项。
    - AWS 配置文件
        - 用于生成令牌。
        - 如果未指定，则使用默认配置文件。
5. 单击“测试连接按钮”以测试连接。
6. 单击“保存连接”。

问题排查

SQL 客户端的身份验证凭证过期

已建立的会话会在最多 1 小时内保持身份验证状态，或者直到出现显式断开连接或客户端超时。如果需要建立新连接，则必须在连接的密码字段中生成并提供新的身份验证令牌。尝试打开新会话（例如，列出新表或打开新的 SQL 控制台）会强制尝试新的身份验证。如果在连接设置中配置的身份验证令牌不再有效，则该新会话将失败，并且所有先前打开的会话将变为无效。在通过 `expires-in` 选项选择 IAM 身份验证令牌的持续时间时，请记住一点：该持续时间默认设置为 15 分钟，最大值可设置为 7 天。

此外，请参阅 Aurora DSQL 文档的故障排除部分。

Amazon Aurora DSQL 集群连接工具

Aurora DSQL 与许多第三方数据库驱动程序和 ORM 库兼容。AWS 提供了两种类型的工具来简化使用 Aurora DSQL 的过程：

- 连接器：扩展数据库驱动程序以自动处理 IAM 令牌生成的身份验证插件。当直接使用数据库驱动程序时，请使用连接器。
- 适配器和方言：对特定 ORM 框架的扩展，可提供 IAM 身份验证和改进的 Aurora DSQL 兼容性。当使用支持的 ORM 框架时，请使用适配器。

Aurora DSQL 适配器和方言

下表列出了 Aurora DSQL 可用的适配器和方言。

编程语言	ORM/框架	存储库链接
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/java/hibernate
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django

编程语言	ORM/框架	存储库链接
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm

数据库驱动程序示例

下表显示了使用第三方数据库驱动程序连接到 Aurora DSQL 的示例代码。

编程语言	驱动程序	存储库链接示例
C++	libpq	https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/libpq
C# (.NET)	Npgsql	https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/npgsql
Go	pgx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/pgx
Java	HikariCP + pgJDBC	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc
JavaScript	node-postgres (AWS Lambda)	https://github.com/aws-samples/aurora-dsql-samples/tree/main/lambda
JavaScript	node-postgres	https://github.com/aws-samples/aurora-dsql-samples/

编程语言	驱动程序	存储库链接示例
		tree/main/javascript/node-postgres
JavaScript	Postgres.js	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js
Python	asyncpg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/asyncpg
Python	Psycopg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg
Python	Psycopg2	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg2
Ruby	pg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/ruby-pg
Rust	SQLx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/sqlx

ORM 和框架示例

下表显示了将第三方 ORM 库和框架与 Aurora DSQL 结合使用的示例代码。

编程语言	ORM/框架	存储库链接示例
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/

编程语言	ORM/框架	存储库链接示例
		java/hibernate/examples/pet-clinic-app
Java	Liquibase	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/liquibase
Java	Spring Boot	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/spring_boot
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django/examples/pet-clinic-app
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy/examples/pet-clinic-app
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm/example
Ruby	Rails	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails
TypeScript	Prisma	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/prisma
TypeScript	Sequelize	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize

编程语言	ORM/框架	存储库链接示例
TypeScript	TypeORM	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm

将数据加载到 Aurora DSQL 中

无论您是从现有数据库进行迁移、从 Amazon Simple Storage Service 导入文件，还是从本地系统加载数据，Aurora DSQL 都提供了多种方法来获取数据。本节介绍适用于各种大小（从千兆字节到数百 TB）的数据加载的推荐工具和技术。

选择加载方法

Aurora DSQL 支持标准 PostgreSQL 数据加载命令，但是大规模高效加载数据需要处理并行化、连接管理和错误恢复。下表汇总了您的选项：

方法	适用于	注意事项
Aurora DSQL 加载器：开源实用程序，可在使用 Aurora DSQL 时轻松并行处理插入	大多数数据加载场景，尤其是迁移和批量导入	自动处理并行化、连接池、冲突解决和 IAM 身份验证。以源代码或二进制形式提供。
PostgreSQL **`\copy`**：客户端 `psql` 元命令	当您已经通过 `psql` 进行连接时，加载过程很简单	读取客户端上的文件并通过连接流式传输数据；您自行管理并行化
`INSERT` 事务：标准 `SQL DML`	小型数据集或应用程序驱动的插入	处理批量数据的方法最简单，但速度最慢

对于大多数数据加载任务，请使用 Aurora DSQL 加载器。它可以处理将数据加载到分布式数据库中的操作复杂性，包括跨多个连接并行执行和自动重试失败的操作。

Aurora DSQL 加载器

Aurora DSQL Loader 是一个开源命令行实用程序，旨在高效地将数据加载到 Aurora DSQL 集群中。它管理连接池，并行处理多个工作线程之间的数据传输，并自动处理冲突和重试。

主要 功能

Aurora DSQL 加载器提供以下功能：

并行加载

　可配置的工作线程支持跨多个连接并行加载数据，从而提高性能。

连接池

　管理与 Aurora DSQL 集群的连接池，自动处理 IAM 身份验证和连接生命周期。

多种文件格式支持

　支持 CSV（逗号分隔值）、TSV（制表符分隔值）和 Apache Parquet 列式格式。加载器会根据源 URI 扩展名自动检测文件格式。

自动架构推理

　与 `--if-not-exists` 标志一起使用时，加载器可以根据数据自动创建具有适当列类型的表。

冲突处理

　当目标表具有唯一约束时，使用以下 `--on-conflict` 选项配置加载器处理冲突的方式：跳过重复项、更新插入记录或返回错误。

容错能力

　自动重试和任务恢复功能可确保中断的加载可以从其停止点继续，而不是完全重新开始。

本地和 S3 源

　从本地文件系统路径或使用 S3 URI 直接从 Amazon S3 存储桶加载数据。

先决条件

在使用 Aurora DSQL 加载器之前，请确保您拥有以下各项：

- 具有有效端点的活动 Aurora DSQL 集群。
- 通过 AWS CLI (aws configure)、AWS 单点登录 (aws sso login) 或 IAM 角色配置的 AWS 凭证。
- IAM 权限：针对 Aurora DSQL 集群的 `dsql:DbConnectAdmin` 或 `dsql:DbConnect`。

- 对于 S3 源，从源存储桶进行读取的适当权限。

安装

从 GitHub 版本页面下载最新版本。预构建的二进制文件可用于常见平台。有关从源代码构建的说明，请参阅 Aurora DSQL Loader 存储库。

用法示例

以下示例演示了 Aurora DSQL 加载器的常见使用案例。

Example加载本地 CSV 文件

此示例将 CSV 文件从本地文件系统加载到现有表中：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table
```

Example从 Simple Storage Service（Amazon S3）加载数据

此示例从 Amazon S3 存储桶加载 Parquet 文件：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/data.parquet \
  --table my_table
```

Example自动表创建

此示例根据数据架构自动创建一个新表：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --if-not-exists
```

Example加载前验证

此示例无需实际加载数据即可验证您的配置：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --dry-run
```

Example恢复中断的加载

如果加载操作中断，则可以使用来自上次运行的作业 ID 恢复该操作：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --resume-job-id job-id \
  --manifest-dir ./loader-state
```

> **Note**
>
> 恢复时，加载器会跳过大多数已经完成的工作，但可能会重试某些记录。如果您的目标表具有唯一约束，请使用 `--on-conflict` 选项来处理重复项：例如，`DO NOTHING` 以跳过它们，或 `DO UPDATE` 以执行更新插入。

命令行选项

Aurora DSQL 加载器支持以下命令行选项：

`--endpoint`

（必需）Aurora DSQL 集群端点。示例：`cluster-id.dsql.region.on.aws`

`--source-uri`

（必需）数据文件的路径。可以是本地文件路径或 S3 URI（例如 `s3://bucket-name/file.parquet`）。

`--table`

（必需）Aurora DSQL 数据库中目标表的名称。

`--if-not-exists`

（可选）如果目标表不存在，则自动创建该表。加载器从数据中推理架构。

`--dry-run`

（可选）验证配置和数据，而不将其实际加载到数据库中。

`--resume-job-id`

（可选）使用指定的作业 ID 恢复先前中断的加载操作。

`--manifest-dir`

（可选）用于存储作业状态和清单的目录，用于恢复作业。

`--on-conflict`

（可选）指定在插入违反目标表上唯一约束的行时如何处理冲突。有效值为 `error`（返回错误）、`do-nothing`（跳过重复行）或 `do-update`（使用新值更新现有行）。

有关选项和其它配置参数的完整列表，请运行：

```
aurora-dsql-loader load --help
```

最佳实践

- 使用试运行进行验证：在将数据加载到生产表之前，请务必使用 `--dry-run` 测试您的加载配置。
- 为恢复定义唯一约束：如果您需要恢复中断的加载，请在目标表上定义唯一约束，然后使用 `--on-conflict` 选项来处理已加载的记录。
- 使用 Parquet 处理大型数据集：与 CSV 或 TSV 相比，Parquet 的列式格式通常可以为大型数据集提供更好的压缩和更快的加载速度。
- 保留清单目录：保留加载作业的清单目录，直到您确认加载成功完成，并在需要时启用恢复。
- 尽可能预先创建表：在加载数据之前，使用显式列数据类型和主键定义目标表。借助预先创建的架构，您可以控制类型精度和索引编制，与自动推理的架构相比，这通常可以提高查询性能。

问题排查

身份验证错误

验证您的 AWS 凭证配置正确，以及您的 IAM 身份对目标集群具有所需的 `dsql:DbConnect` 或 `dsql:DbConnectAdmin` 权限。

S3 访问权限错误

确保您的 IAM 身份对源存储桶和对象具有相应的 S3 读取权限。

架构推理错误

使用 `--if-not-exists` 时，请确保您的数据文件具有一致的列类型。列中的混合类型可能会导致架构推理失败。

恢复时出现重复键错误

如果您在恢复加载时遇到重复键错误，请向目标表添加唯一约束，以便加载器可以使用 `ON CONFLICT DO NOTHING` 来跳过已加载的记录。

有关其它故障排除信息，请参阅 Aurora DSQL Loader GitHub 存储库。

迁移路径

以下各节介绍如何将数据从常见源系统迁移到 Aurora DSQL。

从 PostgreSQL 迁移

要将数据从现有 PostgreSQL 数据库迁移到 Aurora DSQL：

1. 将 PostgreSQL 中的数据导出为 CSV 或 Parquet 格式。可以使用 PostgreSQL `COPY` 命令来导出每个表：

```
COPY my_table TO '/path/to/my_table.csv' WITH (FORMAT csv, HEADER true);
```

2. 在 Aurora DSQL 中创建目标表。您可以手动创建架构，也可以使用加载器的 `--if-not-exists` 标志从数据中推理模式。

3. 使用 Aurora DSQL 加载器加载导出的数据：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

Tip

对于大型迁移，考虑导出为 Parquet 格式，以获得更好的压缩和更快的加载。像 DuckDB 这样的工具可以高效地将 CSV 文件转换为 Parquet。

从 MySQL 迁移

要将数据从 MySQL 迁移到 Aurora DSQL：

1. 使用 SELECT INTO OUTFILE 或类似于 mysqldump 的工具（带 --tab 选项）将数据从 MySQL 导出为 CSV 格式：

```
SELECT * FROM my_table
INTO OUTFILE '/path/to/my_table.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

2. 使用与 PostgreSQL 兼容的相应数据类型在 Aurora DSQL 中创建目标表。
3. 使用 Aurora DSQL 加载器加载导出的数据：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

Note

MySQL 和 PostgreSQL 有不同的数据类型系统。在 Aurora DSQL 中创建表时，请查看您的架构并根据需要调整数据类型。

从 Amazon S3 加载

如果您的数据已经在 Amazon S3 中，您可以直接加载它，而无需下载到本地系统。Aurora DSQL 加载器原生支持 S3 URI：

```
aurora-dsql-loader load \
```

```
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/path/to/data.parquet \
  --table my_table
```

确保您的 IAM 身份对源对象拥有 `s3:GetObject` 权限。

使用 PostgreSQL \copy

如果您已经通过处理 IAM 身份验证的 `psql` 会话连接到 Aurora DSQL，则可以使用客户端 `\copy` 元命令从本地文件系统加载数据。与服务器端 `COPY` 语句不同，`\copy` 读取客户端计算机上的文件并通过现有连接流式传输数据，因此不需要服务器端文件访问权限。这种方法适用于简单的单线程加载。

Example使用 \copy 加载 CSV 文件

```
\copy my_table FROM '/path/to/data.csv' WITH (FORMAT csv, HEADER true);
```

直接使用 `\copy` 时，您负责：

- 加载多个文件或大型数据集时管理并行化
- 处理连接管理和身份验证令牌刷新
- 为失败的操作实施重试逻辑

INSERT 事务的最佳实践

使用 `INSERT` 语句将数据加载到 Aurora DSQL 时，请遵循以下做法以提高吞吐量和可靠性：

- 将行批处理成多行 INSERT：将多行分组为单个 `INSERT` 语句以减少往返次数。例如，`INSERT INTO my_table VALUES (1, 'a'), (2, 'b'), (3, 'c')` 比三个单独的语句效率更高。
- 使用参数化查询：使用带参数绑定的预准备语句而不是字符串连接。这样可以避免 SQL 注入风险，并支持数据库重用查询计划。
- 保持较小的事务量：Aurora DSQL 使用乐观并发控制，因此涉及许多行的大型事务更有可能遇到冲突。目标是数百行而不是数千行的事务。
- 实施重试逻辑：分布式系统中预计会出现诸如乐观并发控制（OCC）冲突之类的瞬态错误。通过重试失败的事务实施指数回退。
- 跨连接并行化：打开多个连接并在它们之间分配插入。每个连接可以同时处理不同的数据子集。

对于大多数使用案例，Aurora DSQL 加载器提供了一种更简单、更稳健的数据加载方法。

其他资源

- GitHub 上的 Aurora DSQL Loader：源代码、文档和问题跟踪
- 在 Amazon Aurora DSQL 中生成身份验证令牌：了解 Aurora DSQL 的 IAM 身份验证令牌
- 使用兼容 PostgreSQL 的客户端访问 Aurora DSQL：使用各种客户端和工具连接到 Aurora DSQL

适用于 Aurora DSQL 的生成式人工智能

此部分详细说明了如何在 Aurora DSQL 中使用生成式人工智能工具

AWS Labs Aurora DSQL MCP 服务器

适用于 Aurora DSQL 的 AWS Labs 模型上下文协议（MCP）服务器

功能

- 将人类可读的问题和命令转换为结构化的、与 Postgres 兼容的 SQL 查询，并对已配置的 Aurora DSQL 数据库执行这些查询。
- 默认情况下为只读模式，可通过 `--allow-writes` 启用事务
- 在请求间重用连接以提升性能
- 内置 Aurora DSQL 文档、搜索及最佳实践建议访问功能

可用工具

数据库操作

- readonly_query：向您的 DSQL 集群执行只读 SQL 查询
- transact：在事务中执行写入操作（需使用 `--allow-writes`）
- get_schema：检索表架构信息

文档与建议

- dsql_search_documentation：搜索 Aurora DSQL 文档
  - 参数：`search_phrase`（必需）、`limit`（可选）
- dsql_read_documentation：读取特定的 DSQL 文档页面
  - 参数：`url`（必需）、`start_index`（可选）、`max_length`（可选）

- dsql_recommend：获取 DSQL 最佳实践的建议
  - 参数：`url`（必需）

先决条件

1. 具有 Aurora DSQL 集群的 AWS 账户
2. 此 MCP 服务器只能在您的 LLM 客户端所在的主机上本地运行。
3. 设置具有 AWS 服务的访问权限的 AWS 凭证
   - 您需要一个 AWS 账户，其角色需包含以下权限：
     - `dsql:DbConnectAdmin`：以管理员用户身份连接到 DSQL 集群
     - `dsql:DbConnect`：使用自定义数据库角色连接到 DSQL 集群（仅在使用非管理员用户时需要此权限）
   - 通过 `aws configure` 或环境变量配置 AWS 凭证

安装

对于大多数工具，按照默认安装说明更新配置就应该足够了。

对 Claude Code 和 Codex 分别概述了说明。

默认安装：更新相关的 MCP 配置文件

使用 **uv**

1. 从 Astral 或 GitHub README 安装 uv
2. 使用 `uv python install 3.10` 安装 Python

在 MCP 客户端配置中配置 MCP 服务器（查找 MCP 配置文件）

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "command": "uvx",
      "args": [
        "awslabs.aurora-dsql-mcp-server@latest",
        "--cluster_endpoint",
        "[your dsql cluster endpoint, e.g. abcdefghijklmnopqrst234567.dsql.us-
east-1.on.aws]",
```

```
        "--region",
        "[your dsql cluster region, e.g. us-east-1]",
        "--database_user",
        "[your dsql username, e.g. admin]",
        "--profile",
        "[your aws profile, e.g. default]"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR"
      },
      "disabled": false,
      "autoApprove": []
    }
  }
}
```

Windows 安装

对于 Windows 用户，MCP 服务器配置格式略有不同：

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "disabled": false,
      "timeout": 60,
      "type": "stdio",
      "command": "uv",
      "args": [
        "tool",
        "run",
        "--from",
        "awslabs.aurora-dsql-mcp-server@latest",
        "awslabs.aurora-dsql-mcp-server.exe"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR",
        "AWS_PROFILE": "your-aws-profile",
        "AWS_REGION": "us-east-1"
      }
    }
  }
}
```

查找 MCP 客户端配置文件

对于一些最常见的代理式开发工具，您可以在以下文件路径中找到 MCP 客户端配置：

- Kiro：
  - 用户配置：`~/.kiro/settings/mcp.json`
  - 工作区配置：`/path/to/workspace/.kiro/settings/mcp.json`
- Claude Code：有关详细的设置帮助，请参阅 Claude 代码安装
  - 用户配置："mcpServers" 中的 `~/.claude.json`
  - 项目配置：`/path/to/project/.mcp.json`
  - 本地配置：`"projects" -> "path/to/project" -> "mcpServers"` 中的 `~/.claude.json`
- 游标：
  - 全局：`~/.cursor/mcp.json`
  - 项目：`/path/to/project/.cursor/mcp.json`
- Codex：`~/.codex/config.toml`
  - 在配置文件中为每个 MCP 服务器配置了一个 `[mcp_servers.<server-name>]` 表。请参阅自定义 Codex 安装说明
- 包装：
  - 文件编辑：`~/.warp/mcp_settings.json`
  - 应用程序编辑器：`Settings > AI > Manage MCP Servers` 并粘贴 json
- Amazon Q 开发者版 CLI：`~/.aws/amazonq/mcp.json`
- Cline：通常是一个嵌套的 VS 代码路径，即 `~/.vscode-server/path/to/cline_mcp_settings.json`

Claude Code

先决条件

重要：MCP 服务器管理只能通过 Claude Code CLI 终端体验获得，而不是 VS Code 原生面板模式。

首先按照 Claude 的原生安装建议流程安装 Claude Code CLI。

选择正确的范围

Claude Code 提供 3 种不同的范围：本地（默认）、项目和用户，以及有关要根据凭证敏感度选择哪个范围和需要共享哪个范围的详细信息。有关更多详细信息，请参阅有关 MCP Installation Scopes 的 Claude Code 文档。

1. 本地范围的服务器代表默认配置级别，存储在项目路径的 `~/.claude.json` 下。它们都是您私有的，并且只能在当前项目目录中访问。这是创建 MCP 服务器时的默认 `scope`。
2. 项目范围的服务器支持团队协作，但仍然只能在项目目录中访问。项目范围的服务器在项目的根目录中添加一个 `.mcp.json` 文件。此文件旨在签入版本控制，确保所有团队成员都能访问相同的 MCP 工具和服务。添加项目范围的服务器时，Claude Code 会自动使用适当的配置结构创建或更新此文件。
3. 用户范围的服务器存储在 `~/.claude.json` 中并提供跨项目的可访问性，使它们可以在计算机上的所有项目中使用，同时对您的用户账户保持私有。

使用 Claude CLI（建议）

使用交互式 `claude` CLI 会话可以改善故障排除体验，因此这是建议的路径。

```
claude mcp add amazon-aurora-dsql \
  --scope [one of local, project, or user] \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

故障排除：在不同的 AWS 账户上将 Claude 代码与 Bedrock 结合使用

如果您使用 Bedrock AWS 账户或与连接到 dsql 集群所需的配置文件不同的配置文件配置了 Claude Code，则需要提供其它环境参数：

```
  --env AWS_PROFILE="[dsql profile, eg. default]" \
  --env AWS_REGION="[dsql cluster region, eg. us-east-1]" \
```

在配置文件中直接修改

Claude Code 需要字母数字命名，因此我们建议您为服务器命名：`aurora-dsql-mcp-server`。

本地范围

更新项目特定的 `mcpServers` 字段内的 `~/.claude.json`：

```
{
```

```
  "projects": {
    "/path/to/project": {
      "mcpServers": {}
    }
  }
}
```

项目范围

更新 `mcpServers` 字段中的 `/path/to/project/root/.mcp.json`：

```
{
  "mcpServers": {}
}
```

用户范围

更新项目特定的 `mcpServers` 字段内的 `~/.claude.json`：

```
{
  "mcpServers": {}
}
```

Codex

选项 1：Codex CLI

如果您安装了 Codex CLI，则可以使用 codex mcp 命令来配置 MCP 服务器。

```
codex mcp add amazon-aurora-dsql \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

选项 2：config.toml

要对 MCP 服务器选项进行更精细的控制，可以手动编辑 `~/.codex/config.toml` 配置文件。在配置文件中为每个 MCP 服务器配置了一个 `[mcp_servers.<server-name>]` 表。

```
[mcp_servers.amazon-aurora-dsql]
command = "uvx"
args = [
  "awslabs.aurora-dsql-mcp-server@latest",
  "--cluster_endpoint", "<DSQL_CLUSTER_ID>.dsql.<AWS_REGION>.on.aws",
  "--region", "<AWS_REGION>",
  "--database_user", "<DATABASE_USERNAME>"
]

[mcp_servers.amazon-aurora-dsql.env]
FASTMCP_LOG_LEVEL = "ERROR"
```

验证安装

对于 Amazon Q 开发者版 CLI、Kiro CLI、Claude CLI/TUI 或 Codex CLI/TUI，请运行 `/mcp` 以查看 MCP 服务器的状态。

对于 Kiro IDE，还可以导航到 Kiro 面板的 `MCP SERVERS` 选项卡，该选项卡显示了所有已配置的 MCP 服务器及其连接状态指示器。

服务器配置选项

`--allow-writes`

默认情况下，dsql MCP 服务器不允许写入操作（“只读模式”）。在此模式下，任何对 transact 工具的调用都将失败。要使用 transact 工具，需通过传递 `--allow-writes` 参数来允许写入。

建议在连接到 DSQL 时使用最低权限访问原则。例如，用户应尽可能使用只读角色。只读模式采用最优客户端强制策略来拒绝变更。

`--cluster_endpoint`

这是用于指定要连接到的集群的必需参数。这应是集群的完整端点，例如 `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws`

`--database_user`

这是用于指定用来连接的用户的必需参数。例如，`admin` 或 `my_user`。请注意，您使用的 AWS 凭证必须具有以该用户身份登录的权限。有关在 DSQL 中设置和使用数据库角色的更多信息，请参阅使用数据库角色和 IAM 角色。

--profile

您可以指定用于凭证的 AWS 配置文件。请注意，无法在 Docker 安装中使用此参数。

此外，还支持在 MCP 配置中使用 `AWS_PROFILE` 环境变量：

```
"env": {
  "AWS_PROFILE": "your-aws-profile"
}
```

如果两者均未提供，则 MCP 服务器将默认使用 AWS 配置文件中的“default”配置文件。

--region

这是用于指定 DSQL 数据库区域的必需参数。

--knowledge-server

可选参数，用于指定 DSQL 知识工具（文档搜索、读取和建议）的远程 MCP 服务器端点。默认情况下，它是预配置的。

示例：

```
--knowledge-server https://custom-knowledge-server.example.com
```

注意：为了安全起见，请仅使用受信任的知识服务器端点。服务器应为 HTTPS 端点。

--knowledge-timeout

可选参数，用于指定向知识服务器发出的请求的超时时间（以秒为单位）。

默认值：`30.0`

示例：

```
--knowledge-timeout 60.0
```

如果您在通过网速慢的网络访问文档时遇到超时，请增大此值。

Aurora DSQL 操控：技能和能力

本节介绍如何使用技能和能力为 Aurora DSQL 配置 AI 操控。这些基于 markdown 的配置文件提供了上下文和指导，AI 助手在生成代码时会自动应用这些上下文和指导，以提高代理式开发的质量。

概述

技能和能力是模块化功能，可扩展 Aurora DSQL 的 AI 助手功能。它们打包了 AI 助手在处理 Aurora DSQL 数据库时自动使用的指令、元数据和资源。

为什么要使用技能和能力

技能和能力为 Aurora DSQL 开发提供了几个主要优势：

- 专业的 AI 助手：为 Aurora DSQL 提供特定领域的专业知识，包括最佳实践、与 Postgres 兼容的 SQL 模式和分布式数据库优化。
- 减少重复性：创建一次，自动使用。不再需要在多个对话中反复提供相同的指导。
- 上下文效率：技能按需加载，而不是预先使用上下文。AI 根据需要分阶段加载信息。
- 持续学习：随着 Aurora DSQL 功能的发展，当更新技能时，AI 助手会自动访问更新的模式。

建议的设置路径

选择与您的开发环境相匹配的设置路径：

- the section called “Skills CLI”（与代理无关）
- the section called “Kiro 能力”
- the section called “Claude 技能”
- the section called “Gemini 技能”
- the section called “Codex 技能”

也可以通过将技能文件夹复制到工具的 `rules` 或 `skills` 目录中，将 DSQL 技能与其它 AI 编码代理一起使用。

Skills CLI

可以使用 Skills CLI 安装 DSQL 技能。这种与代理无关的设置方法适用于大多数 AI 编码助手，可让您同时将技能安装到多个代理上。

设置

运行以下命令以安装 Aurora DSQL 技能：

```
npx skills add awslabs/mcp --skill dsql
```

CLI 将指导您完成：

- 选择代理：选择要安装到哪些代理（Kiro、Claude Code、Cursor、Copilot、Gemini、Codex、Roo、Cline、OpenCode、Windsurf 等）
- 安装范围：在以下选项中进行选择：
  - 项目：安装在当前目录中（与您的项目一起提交）
  - 全局：安装在主目录中（适用于所有项目）
- 安装方法：在以下选项中进行选择：
  - 符号链接（建议）：单一事实来源，易于更新
  - 复制到所有代理：每个代理都有独立的副本

管理技能

可随时使用以下方法检查和更新技能：

```
npx skills check
npx skills update
```

Kiro 能力

Kiro 能力是统一的软件包，它将 MCP 工具与框架专业知识和操控说明捆绑在一起。每项能力都包括一个入口点文档，用于说明可用的 MCP 工具和激活触发器、MCP 服务器配置以及按需加载的其它特定于工作流程的指南。

能力会根据用户上下文动态激活。能力并不是预先加载所有工具，而是在相关关键字触发激活之前保持接近零的基线用量。

设置

要为 Aurora DSQL 设置 Kiro 能力，请执行以下操作：

1. 直接从 Kiro Powers Registry 进行安装
2. 在 IDE 中重定向到“能力”后，您可以：

- 选择试用能力按钮。建议希望让 AI 指导 MCP 服务器设置或使用 Aurora DSQL 进行交互式入门体验以创建新集群的用户使用。
- 打开一个新的 Kiro 聊天，并询问任何与 Aurora DSQL 相关的问题。（可选）使用现有集群详细信息更新 MCP 配置以测试 MCP 服务器连接，以便它可以与能力一起开箱即用。如果 Kiro 代理识别出能力对完成用户的任务有价值，它将自动激活该能力。

Claude 技能

Claude 技能是扩展 Claude 的功能的模块化能力。每项技能都会对 Claude 在相关时自动使用的指令、元数据和可选资源进行打包。技能基于文件系统并按需加载，以最大限度地减少上下文用量。

使用 Skills CLI 进行简单设置

可以使用 the section called “Skills CLI” 将技能安装到 Claude Code 中。要仅将 Claude Code 指定为要安装到的代理，请使用：

```
npx skills add awslabs/mcp --skill dsql --agent claude-code
```

备选方案：使用 Git 克隆直接设置

备选设置采用 dsql-skill 目录的稀疏克隆，并将此克隆符号链接到 `~/.claude/skills/` 文件夹。这样，就可以在需要更新技能时对技能进行更改。

先决条件

- Git 已安装

设置步骤

1. 创建基本存储库目录

```
mkdir -p .dsql_skill_repos
```

2. 对 MCP 存储库中的技能进行稀疏克隆

仅克隆 `dsql-skill` 文件夹（而不克隆其它文件）：

```
cd .dsql_skill_repos
git clone --filter=blob:none --no-checkout https://github.com/awslabs/mcp.git
cd mcp
```

```
git sparse-checkout init --cone
git sparse-checkout set src/aurora-dsql-mcp-server/skills/dsql-skill
git checkout
cd ../..
```

3. 将技能符号链接到 Skills 目录

添加技能目录（默认：global/user-scoped）：

```
mkdir -p ~/.claude/skills
```

如果您想让它成为项目范围的技能，请改用项目根目录的 `.claude/skills/` 目录。

添加符号链接：

```
ln -s "$(pwd)/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/dsql-skill"
 ~/.claude/skills/dsql-skill
```

4. 验证设置

```
# Should show SKILL.md and other skill files
ls -la ~/.claude/skills/dsql-skill/
```

5. 验证技能使用情况

配置技能后，您应该有一个新的技能命令：`/dsql`。您可能必须在添加技能后重启 Claude Code，才能检测到该技能。您可以根据需要从 Claude Code CLI 或面板中使用此命令。

更新技能

要从存储库中提取最新更改，请执行以下操作：

```
cd .dsql_skill_repos/mcp
git pull
```

目录结构

设置全局技能后，您应该会看到以下目录：

```
.dsql_skill_repos/
### mcp/                              # Sparse git checkout
    ### src/
        ### aurora-dsql-mcp-server/
            ### skills/
                ### dsql-skill/
                    ### SKILL.md
                    ### ...

~/.claude/
### skills/
    ### dsql-skill -> /path/to/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/
dsql-skill
```

> **Note**
>
> 如果您不想追踪该技能，请将 `.dsql_skill_repos/` 添加到您的 `.gitignore`。稀疏签出仅保留技能文件夹，从而最大限度地减少了磁盘用量。

Gemini 技能

要直接在 Gemini 中添加 Aurora DSQL 技能，请确定范围：`workspace`（包含在项目中）或 `user`（默认，全局），然后使用技能安装程序。

设置

```
gemini skills install https://github.com/awslabs/mcp.git --path src/aurora-dsql-mcp-
server/skills/dsql-skill --scope $SCOPE
```

将 `$SCOPE` 替换为 `workspace` 或 `user`。

然后，可以通过 Gemini 使用 `/dsql` 技能命令，Gemini 会自动检测何时应该使用该技能。

Codex 技能

通过 `$skill-installer` 技能使用 Codex CLI 或 TUI 中的技能安装程序。

设置

```
$skill-installer install dsql skill: https://github.com/awslabs/mcp/tree/main/src/
aurora-dsql-mcp-server/skills/dsql-skill
```

重启 Codex 以获得技能。然后可以使用 `$dsql` 激活该技能。

开始使用 Aurora DSQL 查询编辑器

借助 Aurora DSQL 查询编辑器，无需安装或配置外部客户端，即可直接从 AWS 管理控制台安全地连接到 Aurora DSQL 集群并运行 SQL 查询。它提供了一个直观的工作区，内置语法高亮、自动补全和智能代码辅助功能。您可以在单个界面中快速浏览架构对象、开发并运行 SQL 查询以及查看结果。

本主题将引导您完成连接到集群、运行查询、查看结果和浏览高级功能（例如，执行计划）所需的步骤。

> **Note**
>
> 查询编辑器已在所有支持 Aurora DSQL 的区域推出。有关区域可用性的更多详细信息，请参阅 AWS 区域服务。

先决条件

在您开始之前，确保您满足以下要求：

- 您拥有至少一个可用的 Aurora DSQL 集群。有关创建集群的更多详细信息，请参阅步骤 1：创建 Aurora DSQL 单区域集群。
- 集群端点可公开访问。查询编辑器不支持公共访问受到基于资源的策略阻止的集群或通过 VPC 端点管理的集群。有关访问限制的更多详细信息，请参阅在 Aurora DSQL 中使用基于资源的策略屏蔽公共访问权限和使用 AWS PrivateLink 管理和连接到 Amazon Aurora DSQL 集群。
- 您的 IAM 用户或角色具有访问和连接到集群所需的权限。有关权限的更多详细信息，请参阅使用数据库角色和 IAM 身份验证。

使用查询编辑器

打开查询编辑器

要打开查询编辑器，请执行以下操作：

1. 打开 Aurora DSQL 控制台。
2. 在导航窗格中，选择查询编辑器。

或者，在集群页面中，选择要查询的集群，然后选择使用查询编辑器进行连接以直接启动该编辑器。

> **Note**
> 未保存工作和连接状态。如果您退出 Aurora DSQL 控制台、关闭浏览器标签页或注销，则您的连接、查询文本和结果都将丢失。

连接到集群

要连接到集群，请执行以下操作：

1. 如果不存在集群连接，编辑器将显示未连接任何集群。选择连接或在集群资源管理器窗格中选择 +（添加）以连接到现有集群。
2. （可选）使用不同的角色连接到多个集群或同一集群。

浏览集群对象

集群资源管理器会显示所有可用的集群连接，并可让您浏览数据库、架构、表和视图等对象。它还提供常用操作（例如刷新、创建表）以及其他特定于上下文的选项。

运行查询

运行查询

1. 在查询编辑器选项卡窗格中，输入您的 SQL 语句。例如：

```
SELECT * FROM public.orders LIMIT 10;
```

2. 验证查询选项卡右上角显示的活动集群上下文。这表示与当前查询选项卡关联的集群连接。

3. （可选）使用连接下拉列表查看所有可用连接或切换至其他集群。更改连接时将更新该选项卡中的查询执行位置。
4. 选择运行以运行查询。

> **Note**
>
> 每个查询在结果窗格中最多可返回 10000 行数据。对于大型数据集，请使用筛选条件或限制来优化查询。

查看结果和执行计划

查询运行后，可在编辑器底部的“结果”面板中查看输出。默认情况下，每次执行查询都会显示结果（表）选项卡，并以表格形式呈现查询输出。

要获取查询执行计划，请运行 `EXPLAIN ANALYZE` 或 `EXPLAIN ANALYZE VERBOSE` 以获取有关查询性能的更多见解。有关执行计划的更多详细信息，请参阅阅读 Aurora DSQL EXPLAIN 计划。

> **Tip**
>
> `EXPLAIN ANALYZE VERBOSE` 命令可用于显示 DPU 使用量估算值（包括计算、读取、写入和总 DPU 值），以便您即时了解单个 SQL 语句消耗的资源。

查询编辑器：将 JupyterLab 与 Aurora DSQL 结合使用

本指南提供了有关如何将 JupyterLab 与 Python 结合使用以连接并查询 Amazon Aurora DSQL 的分步操作说明。JupyterLab 是一类常用的交互式计算环境，可在单个文档中整合代码、文本和可视化内容，被广泛用于数据科学与研究应用场景。

以下说明将介绍在本地安装的 JupyterLab 与使用 Amazon SageMaker AI（一项完全托管的机器学习服务，可提供带数据工作流 UI 的托管环境）这两种场景中，Aurora DSQL 的基本用法。

开始使用

要求

- 一个 Aurora DSQL 集群

- 已配置的 AWS 凭证（仅限本地安装）
- Python 版本 3.9 或更高版本（仅限本地安装）

使用本地 JupyterLab

要开始使用 JupyterLab，用户必须先使用 Python 的 pip 安装该应用程序：

```
pip install jupyterlab
```

随后，可以通过运行 **jupyter lab** 来打开 JupyterLab。这将在 localhost:8888 打开 JupyterLab 应用程序，可通过浏览器进行访问。继续操作之前，请确保已在本地环境中配置 AWS 凭证。

使用 Amazon SageMaker AI

在 AWS 管理控制台中，转至 Amazon SageMaker AI 控制台页面，然后转至应用程序和 IDE 下的笔记本部分。您可以在此部分中选择创建笔记本实例以开始创建 SageMaker 环境。在单击创建笔记本实例之前，请选择实例类型和平台。

有关设置和实例选项的更多信息，请参阅 Amazon SageMaker AI 设置文档。

> **Note**
>
> 警告：使用 Amazon SageMaker AI 可能会导致您的 AWS 账户产生费用。

在 SageMaker 实例变为活动状态后，您可以在笔记本实例部分中使用打开 JupyterLab 将其打开。在笔记本中开始使用 Aurora DSQL 之前，您必须以 SageMaker 实例的 IAM 角色身份提供对 DSQL 集群的访问权限。要执行此操作，最简单的方法是单击笔记本实例页面中的 IAM 角色的链接。您可以在该处编辑附加到 SageMaker IAM 角色的策略。有关配置 IAM 策略以允许访问 Aurora DSQL 的更多信息，请参阅身份验证和授权。

使用 JupyterLab 连接到 Aurora DSQL

设置 JupyterLab 实例后，用于在本地环境和 SageMaker AI 中连接到 Aurora DSQL 的步骤相同。创建一个空白 Python 3 笔记本，可在其中添加带 Python 代码的单元。

在 Python 单元中，从官方信任存储下载 Amazon 根证书：

```
import urllib.request
```

```
urllib.request.urlretrieve('https://www.amazontrust.com/repository/AmazonRootCA1.pem',
 'root.pem')
```

要连接到 Aurora DSQL，请先在 Python 单元中安装适用于 Python 的 Aurora DSQL 连接器和 Psycopg 驱动程序，然后将其导入：

```
pip install aurora_dsql_python_connector psycopg
```

```
import aurora_dsql_psycopg as dsql
```

导入该连接器后，您可以创建 DSQL 配置并进行连接。Aurora DSQL Python 连接器会在每次建立连接时自动创建身份验证令牌。

```
config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin"
    }

conn = dsql.connect(**config)
```

运行代码后，您现在应已建立与 Aurora DSQL 的 Psycopg 连接。随后，您可以使用 Psycopg 游标并提供您的 SQL 查询来运行查询。有关如何将 Psycopg 用于兼容 Postgres 的数据库的更多信息，请参阅 Psycopg 文档。此查询的结果将以元组列表的形式存储在 `results_list` 中。

```
with conn:
    with conn.cursor() as cur:
        cur.execute("SELECT * FROM table")
        results_list = cur.fetchall()
```

随后，您可以使用 Python 框架（如 Pandas）来分析或可视化您的查询结果，例如：

```
pip install pandas

import pandas as pd

df = pd.DataFrame(tuples_list)
print(df)
```

```
print(f"Total records: {len(df)}")
```

示例笔记本

Aurora DSQL 示例存储库包含使用 Aurora DSQL 的示例笔记本。

延伸阅读

Amazon SageMaker AI 设置文档

适用于 Python 的 Aurora DSQL 连接器

Pandas 文档

Amazon Aurora DSQL 的备份和还原

Amazon Aurora DSQL 可以通过与 AWS Backup 集成来协助您满足监管合规和业务连续性要求，后者是一项完全托管式数据保护服务，可以轻松地在 AWS 服务、云中和本地集中管理和自动执行备份。该服务可简化单区域和多区域 Aurora DSQL 集群的备份创建、管理和还原过程。

主要功能包括以下方面:

- 通过 AWS 管理控制台、SDK 或 AWS CLI 进行集中化备份管理
- 完全集群备份
- 自动备份计划和保留策略
- 跨区域和跨账户功能
- 针对您存储的所有备份进行 WORM（一次写入、多次读取）配置

有关 AWS Backup 备份保管库锁的功能的更多信息以及 Aurora DSQL 的可用 AWS Backup 功能的详细列表，请参阅《AWS Backup Developer Guide》中的 Vault lock benefits 和 AWS Backup feature availability。

开始使用 AWS Backup

AWS Backup 创建 Aurora DSQL 集群的完整副本。您可以按照 Getting started with AWS Backup 中的步骤开始将 AWS Backup 用于 Aurora DSQL：

1. 创建按需备份，以实现即时保护。
2. 制定备份计划以进行自动、计划的备份。
3. 配置保留期和跨区域复制。
4. 为备份活动设置监控和通知。

还原备份

还原 Aurora DSQL 集群时，AWS Backup 始终创建新的集群来保留源数据。

配置多区域集群

要还原 Aurora DSQL 单区域集群，请使用控制台（https://console.aws.amazon.com/backup）或 CLI 来选择要还原的恢复点（备份）。为将从备份创建的新集群配置设置。有关详细说明，请参阅 Restore a single-Region Aurora DSQL cluster。

还原多区域集群

可通过控制台（https://console.aws.amazon.com/backup）和 AWS CLI 支持还原 Aurora DSQL 多区域集群：有关详细说明，请参阅 Restore a multi-Region Aurora DSQL cluster。

要还原到多区域 Aurora DSQL 集群，您可以使用在单个 AWS 区域中制作的备份。但是，在启动还原过程之前，必须确保多区域集群的所有 AWS 区域中都有完全相同的备份副本。如果还没有这些副本，必须首先将备份复制到支持多区域集群的另一个 AWS 区域。

我们建议在关键的 AWS 区域中创建备份副本，以启用强大的灾难恢复选项并满足合规要求。要查看 Aurora DSQL 的可用 AWS 区域，请参阅 the section called “AWS 区域可用性”。

有关这些步骤的详细说明，请参阅 Amazon Aurora DSQL restore 文档。

监控和合规性

AWS Backup 通过以下资源让用户全面地了解备份和还原操作。

- 用于跟踪备份和还原作业的集中式控制面板
- 与 CloudWatch 和 CloudTrail 集成。
- 用于合规报告和审计的 AWS Backup Audit Manager。

请参阅使用 AWS CloudTrail 记录 Aurora DSQL 操作，以了解有关用户、角色或 AWS 服务 在使用 Aurora DSQL 时所采取操作的日志记录的更多信息。

其他资源

要了解有关 AWS Backup 功能以及将其与 Aurora DSQL 结合使用的更多信息，请参阅以下资源：

- Managed policies for AWS Backup
- Amazon Aurora DSQL restore
- Supported services by AWS 区域

- Encryption for backups in AWS Backup

通过将 AWS Backup 用于 Aurora DSQL，您可以实施强大、合规和自动化的备份策略，以便保护关键的数据库资源，同时最大限度地减少管理开销。无论您是管理单个集群还是复杂的多区域部署，AWS Backup 都可提供确保数据保持安全和可恢复所需的工具。

Aurora DSQL 的监控和日志记录

监控和日志记录是保持 Amazon Aurora DSQL 资源的可靠性、可用性和性能的重要方面。您应从 Aurora DSQL 资源的所有部分监控和收集日志记录数据，以便轻松地调试多点故障。

- Amazon CloudWatch 实时监控您的 AWS 资源以及在 AWS 上运行的应用程序。您可以收集和跟踪指标，创建自定义的控制面板，以及设置警报以在指定的指标达到您指定的阈值时通知您或采取措施。例如，您可以使用 CloudWatch 跟踪 Amazon EC2 实例的 CPU 使用率或其他指标并且在需要时自动启动新实例。有关更多信息，请参阅《Amazon CloudWatch 用户指南》https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/。
- AWS CloudTrail 捕获由您的 AWS 账户 或代表该账户发出的 API 调用和相关事件，并将日志文件传输到您指定的 Amazon S3 桶。您可以标识哪些用户和账户调用了 AWS、发出调用的源 IP 地址以及调用的发生时间。有关更多信息，请参阅《AWS CloudTrail 用户指南》。

使用 Amazon CloudWatch 监控 Aurora DSQL

使用 CloudWatch 监控 Aurora DSQL，CloudWatch 会收集原始数据并将其处理为易读且近乎实时的指标。CloudWatch 将这些统计数据保留 15 个月，有助于您更好地了解 Web 应用程序或服务性能。设置警报以监视特定阈值，并在达到阈值时发送通知或采取行动。查看以下可用于 Aurora DSQL 的使用情况和可观测性指标。

有关更多信息，请参阅《Amazon CloudWatch 用户指南》。

可观测性和性能

此表概述了 Aurora DSQL 的可观测性指标。它包括用于跟踪只读事务数和总事务数的指标，以提供总体工作负载特征。包括查询超时和 OCC 冲突率等可操作指标，有助于识别性能问题和并发冲突。与会话相关的指标，包括有关活动状态和总数方面的指标，可供深入了解系统上的当前负载。

CloudWatch 指标名称	指标	单位	描述
ReadOnlyTransactions	Read-only transactions	none	The number of read-only transactions
TotalTransactions	Total transactions	none	The total number of transactions executed on the system,

CloudWatch 指标名称	指标	单位	描述
			including read-only transactions.
QueryTimeouts	Query timeouts	none	The number of queries which have timed out due to hitting the maximum transaction time
OccConflicts	OCC conflicts	none	The number of transactions aborted due to key level OCC
CommitLatency	Commit Latency	milliseconds	Time spent by commit phase of query execution (P50)
BytesWritten	Bytes Written	bytes	Bytes written to storage
BytesRead	Bytes Read	bytes	Bytes read from storage
ComputeTime	QP compute time	milliseconds	QP wall clock time
ClusterStorageSize	Cluster Storage Size	bytes	Cluster size

使用情况指标

Aurora DSQL 使用名为分布式处理单元（DPU）的单个标准化计费单位，来衡量所有基于请求的活动，例如查询处理、读取和写入。

CloudWatch 指标名称	指标	维度：ResourceId	单位	描述
WriteDPU	Write Units	<cluster-id>	DPU	Approximates the write active-

CloudWatch 指标名称	指标	维度：ResourceId	单位	描述
				use component of your Aurora DSQL cluster DPU usage.
MultiRegionWriteDPU	Multi-Region Write Units	<cluster-id>	DPU	Applicable for Multi-Region clusters: Approximates the multi-Region write active-use component of your Aurora DSQL cluster DPU usage.
ReadDPU	Read Units	<cluster-id>	DPU	Approximates the read active-use component of your Aurora DSQL cluster DPU usage.
ComputeDPU	Compute Units	<cluster-id>	DPU	Approximates the compute active-use component of your Aurora DSQL cluster DPU usage.

CloudWatch 指标名称	指标	维度：ResourceId	单位	描述
TotalDPU	Total Units	<cluster-id>	DPU	Approximates the total active-use component of your Aurora DSQL cluster DPU usage.

使用 AWS CloudTrail 记录 Aurora DSQL 操作

Amazon Aurora DSQL 与 AWS CloudTrail 集成，后者是一项提供用户、角色或 AWS 服务 所采取操作的记录的服务。CloudTrail 中有两种类型的事件：管理事件和数据事件。发出管理事件以审计 AWS 资源配置更改。数据事件通常会在服务数据面板中捕获 AWS 资源使用情况。

CloudTrail 将 Aurora DSQL 的所有 API 调用作为事件捕获。Aurora DSQL 将控制台活动记录为管理事件。它还会将经过身份验证的集群连接尝试捕获为数据事件。

使用 CloudTrail 收集的信息，您可以确定向 Aurora DSQL 发出的请求、从中发出请求的 IP 地址、发出请求的时间、发出请求的用户身份以及其它详细信息。

当您创建 AWS 账户时，CloudTrail 会在账户中默认启用，并且您可以访问 CloudTrail 事件历史记录。CloudTrail 事件历史记录提供对 AWS 区域 中过去 90 天的已记录管理事件的可查看、可搜索、可下载和不可变记录。有关更多信息，请参见《AWS CloudTrail 用户指南》的 使用 CloudTrail 事件历史记录。记录事件历史记录不会收取 CloudTrail 费用。

要在您的 AWS 账户中创建持续的事件记录，包括 Aurora DSQL 的事件，请创建跟踪或 AWS CloudTrail Lake 事件数据存储（AWS CloudTrail 事件的集中存储和分析解决方案）。有关创建跟踪的更多信息，请参阅 Working with CloudTrail trails。要了解有关设置和管理事件数据存储的信息，请参阅 CloudTrail Lake event data stores。

CloudTrail 中的 Aurora DSQL 管理事件

CloudTrail Management events 提供有关对您 AWS 账户中的资源执行的管理操作的信息。这些也称为控制面板操作。默认情况下，CloudTrail 会在事件历史记录中捕获管理事件。

Amazon Aurora DSQL 将所有 Aurora DSQL 控制面板操作记录为管理事件。有关 Aurora DSQL 记录到 CloudTrail 的 Amazon Aurora DSQL 控制面板操作的列表，请参阅 Aurora DSQL API 参考。

控制面板日志

Amazon Aurora DSQL 将以下 Aurora DSQL 控制面板操作作为管理事件记录到 CloudTrail。

- CreateCluster
- DeleteCluster
- GetCluster
- GetVpcEndpointServiceName
- ListClusters
- ListTagsForResource
- TagResource
- UntagResource
- UpdateCluster

备份和还原日志

Amazon Aurora DSQL 将以下 Aurora DSQL 备份和还原操作作为管理事件记录到 CloudTrail。

- `StartBackupJob`
- `StopBackupJob`
- `GetBackupJob`
- `StartRestoreJob`
- `StopRestoreJob`
- `GetRestoreJob`

有关使用 AWS Backup 保护 Aurora DSQL 集群的更多信息，请参阅 Amazon Aurora DSQL 的备份和还原。

AWS KMS 日志

Amazon Aurora DSQL 将以下 AWS KMS 操作作为管理事件记录到 CloudTrail。

- `GenerateDataKey`
- `Decrypt`

要详细了解 CloudTrail 日志如何跟踪 Aurora DSQL 代表您发送到 AWS KMS 的请求，请参阅监控 Aurora DSQL 与 AWS KMS 的交互。

CloudTrail 中的 Aurora DSQL 数据事件

CloudTrail Data events 通常提供有关对资源或在资源中执行的资源操作的信息。此类事件还用于捕获服务的数据面板操作。数据事件通常是高容量活动。默认情况下，CloudTrail 不记录数据事件。CloudTrail 事件历史记录不记录数据事件。

有关如何记录数据事件的更多信息，请参阅《AWS CloudTrail 用户指南》中的使用 AWS 管理控制台记录数据事件和使用 AWS Command Line Interface 记录数据事件。

记录数据事件将收取额外费用。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。

对于 Aurora DSQL，CloudTrail 会将与 Aurora DSQL 集群进行的任何连接尝试作为数据事件捕获。下表列出了可以记录其数据事件的 Aurora DSQL 资源类型。资源类型（控制台）列显示可从 CloudTrail 控制台上的资源类型列表中选择的值。resources.type 值列显示了您在使用 AWS CLI 或 CloudTrail API 配置高级事件选择器时需要指定的 `resources.type` 值。记录到 CloudTrail 的数据 API 列显示了针对该资源类型记录到 CloudTrail 的 API 调用。

资源类型（控制台）	resources.type 值	记录至 CloudTrail 的数据 API
Amazon Aurora DSQL	`AWS::DSQL::Cluster`	• `DbConnect` • `DbConnectAdmin`

您可以将高级事件选择器配置为根据 `eventName` 和 `resources.ARN` 字段进行筛选，以仅记录筛选出的事件。有关这些字段的更多信息，请参阅《AWS CloudTrail API 参考》中的 AdvancedFieldSelector。

以下示例说明如何使用 AWS CLI 配置 `dsql-data-events-trail` 来接收 Aurora DSQL 的数据事件。

```
aws cloudtrail put-event-selectors \
--region us-east-1 \
--trail-name dsql-data-events-trail \
--advanced-event-selectors '[{
"Name": "Log DSQL Data Events",
    "FieldSelectors": [
       { "Field": "eventCategory", "Equals": ["Data"] },
```

```
        { "Field": "resources.type", "Equals": ["AWS::DSQL::Cluster"] } ]}]'
```

Amazon Aurora DSQL 中的安全性

AWS 的云安全性的优先级最高。为了满足对安全性最敏感的组织的需求，我们打造了具有超高安全性的数据中心和网络架构。作为 AWS 的客户，您也可以从这些数据中心和网络架构受益。

安全性是 AWS 和您的共同责任。责任共担模式将其描述为云的安全性和云中的安全性：

- 云的安全性:AWS 负责保护在 AWS Cloud 中运行 AWS 服务的基础结构。AWS 还向您提供可安全使用的服务。第三方审核员定期测试和验证我们的安全性的有效性，作为 AWS Compliance Programs 的一部分。要了解适用于 Amazon Aurora DSQL 的合规性计划，请参阅 AWS 按合规性计划提供的范围内服务。
- 云中的安全性：您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档有助于您了解如何在使用 Aurora DSQL 时应用责任共担模式。以下主题说明如何配置 Aurora DSQL 以实现安全性和合规性目标。您还会了解如何使用其它 AWS 服务来协助您监控和保护 Aurora DSQL 资源。

主题

- Amazon Aurora DSQL 的 AWS 托管式策略
- Amazon Aurora DSQL 中的数据保护
- Amazon Aurora DSQL 的数据加密
- Aurora DSQL 的身份和访问管理
- Aurora DSQL 的基于资源的策略
- 使用 Aurora DSQL 中的服务相关角色
- 将 IAM 条件键与 Amazon Aurora DSQL 结合使用
- Amazon Aurora DSQL 中的事件响应
- Amazon Aurora DSQL 的合规性验证
- Amazon Aurora DSQL 中的韧性
- Amazon Aurora DSQL 中的基础设施安全性
- Amazon Aurora DSQL 中的配置和漏洞分析
- 防止跨服务混淆座席
- Aurora DSQL 的安全最佳实践

Amazon Aurora DSQL 的 AWS 托管式策略

AWS 托管式策略是由 AWS 创建和管理的独立策略。AWS 托管式策略旨在为许多常见使用案例提供权限，以便您可以开始为用户、组和角色分配权限。

请记住，AWS 托管式策略可能不会为您的特定使用案例授予最低权限，因为它们可供所有 AWS 客户使用。我们建议通过定义特定于使用案例的客户管理型策略来进一步减少权限。

您无法更改 AWS 托管式策略中定义的权限。如果 AWS 更新在 AWS 托管式策略中定义的权限，则更新会影响该策略所附加到的所有主体身份（用户、组和角色）。当新的 AWS 服务 启动或新的 API 操作可用于现有服务时，AWS 最有可能更新 AWS 托管式策略。

有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管式策略。

AWS 托管式策略：AmazonAuroraDSQLFullAccess

您可以将 `AmazonAuroraDSQLFullAccess` 附加到您的用户、组和角色。

此策略授予支持对 Aurora DSQL 进行完全管理访问的权限。拥有这些权限的主体可以：

- 创建、删除和更新 Aurora DSQL 集群，包括多区域集群
- 管理集群内联策略（创建、查看、更新和删除策略）
- 在集群中添加和移除标签
- 列出集群并查看有关各个集群的信息
- 查看附加到 Aurora DSQL 集群的标签
- 以任何用户（包括管理员）身份连接到数据库
- 对 Aurora DSQL 集群执行备份和还原操作，包括启动、停止以及监控备份和还原作业
- 使用客户自主管理型 AWS KMS 密钥对集群加密
- 查看其账户中来自 CloudWatch 的任何指标
- 使用 AWS Fault Injection Service（AWS FIS）向 Aurora DSQL 集群注入故障，进行容错能力测试
- 为 `dsql.amazonaws.com` 服务创建服务相关角色，这是创建集群所必需的

权限详细信息

该策略包含以下权限。

- `dsql`：向主体授予对 Aurora DSQL 的完全访问权限。
- `cloudwatch`：授予将指标数据点发布到 Amazon CloudWatch 的权限。
- `iam`：授予创建服务相关角色的权限。
- `backup and restore`：授予启动、停止和监控 Aurora DSQL 集群的备份和还原作业的权限。
- `kms`：授予所需的权限，以便在创建、更新或连接到集群时，验证对用于 Aurora DSQL 集群加密的客户自主管理型密钥的访问权限。
- `fis`：授予使用 AWS Fault Injection Service（AWS FIS）的权限，以便向 Aurora DSQL 集群注入故障，进行容错能力测试。

您可以在 IAM 控制台中和《AWS Managed Policy Reference Guide》中找到 `AmazonAuroraDSQLFullAccess` 策略。

AWS 托管式策略：AmazonAuroraDSQLReadOnlyAccess

您可以将 `AmazonAuroraDSQLReadOnlyAccess` 附加到您的用户、组和角色。

支持对 Aurora DSQL 进行读取访问。拥有这些权限的主体可以列出集群并查看有关各个集群的信息。这些主体可以查看附加到 Aurora DSQL 集群的标签，并查看集群内联策略。这些主体可以在您的账户中检索和查看来自 CloudWatch 的任何指标。

权限详细信息

该策略包含以下权限。

- `dsql`：授予对 Aurora DSQL 中所有资源的只读权限。
- `cloudwatch`：授予检索批量 CloudWatch 指标数据以及对检索到的数据执行指标数学运算的权限

您可以在 IAM 控制台中和《AWS Managed Policy Reference Guide》中找到 `AmazonAuroraDSQLReadOnlyAccess` 策略。

AWS 托管式策略：AmazonAuroraDSQLConsoleFullAccess

您可以将 `AmazonAuroraDSQLConsoleFullAccess` 附加到您的用户、组和角色。

支持通过 AWS 管理控制台对 Amazon Aurora DSQL 进行完全管理访问。拥有这些权限的主体可以：

- 使用控制台，创建、删除和更新 Aurora DSQL 集群，包括多区域集群
- 通过控制台管理集群内联策略（创建、查看、更新和删除策略）
- 列出集群并查看有关各个集群的信息
- 查看您账户中任何资源的标签
- 以任何用户（包括管理员）身份连接到数据库
- 对 Aurora DSQL 集群执行备份和还原操作，包括启动、停止以及监控备份和还原作业
- 使用客户自主管理型 AWS KMS 密钥对集群加密
- 从 AWS 管理控制台启动 AWS CloudShell
- 查看您账户中 CloudWatch 的所有指标
- 使用 AWS Fault Injection Service（AWS FIS）向 Aurora DSQL 集群注入故障，进行容错能力测试
- 为 `dsql.amazonaws.com` 服务创建服务相关角色，这是创建集群所必需的

您可以在 IAM 控制台上找到 `AmazonAuroraDSQLConsoleFullAccess` 策略，并在《AWS Managed Policy Reference Guide》中找到 AmazonAuroraDSQLConsoleFullAccess。

权限详细信息

该策略包含以下权限。

- `dsql`：通过 AWS 管理控制台授予对 Aurora DSQL 中所有资源的完全管理权限。
- `cloudwatch`：授予检索批量 CloudWatch 指标数据以及对检索到的数据执行指标数学运算的权限。
- `tag`：授予权限，以返回当前在指定的 AWS 区域中用于调用账户的标签键和值。
- `backup and restore`：授予启动、停止和监控 Aurora DSQL 集群的备份和还原作业的权限。
- `kms`：授予所需的权限，以便在创建、更新或连接到集群时，验证对用于 Aurora DSQL 集群加密的客户自主管理型密钥的访问权限。
- `cloudshell`：授予启动 AWS CloudShell 来与 Aurora DSQL 进行交互的权限。
- `ec2`：授予查看 Aurora DSQL 连接所需的 Amazon VPC 端点信息的权限。
- `fis`：授予使用 AWS FIS 的权限，以便向 Aurora DSQL 集群注入故障来进行容错能力测试。
- `access-analyzer:ValidatePolicy` 在策略编辑器中授予对 linter 的权限，用于提供有关当前策略中错误、警告和安全问题的实时反馈。

- `fis`：授予使用 AWS Fault Injection Service（AWS FIS）的权限，以便向 Aurora DSQL 集群注入故障，进行容错能力测试。

您可以在 IAM 控制台中和《AWS Managed Policy Reference Guide》中找到 `AmazonAuroraDSQLConsoleFullAccess` 策略。

AWS 托管式策略：AuroraDSQLServiceRolePolicy

您无法将 AuroraDSQLServiceRolePolicy 策略附加至 IAM 实体。此策略附加至服务相关角色，该角色支持 Aurora DSQL 访问账户资源。

您可以在 IAM 控制台上找到 `AuroraDSQLServiceRolePolicy` 策略，并在《AWS Managed Policy Reference Guide》中找到 AuroraDSQLServiceRolePolicy。

Aurora DSQL 对 AWS 托管式策略的更新

查看有关自此服务开始跟踪这些更改起，适用于 Aurora DSQL 的 AWS 托管式策略更新的详细信息。有关此页面更改的自动提醒，请订阅 Aurora DSQL 文档历史记录页面上的 RSS 源。

更改	描述	日期
AmazonAuroraDSQLFullAccess 和 AmazonAuroraDSQLConsoleFullAccess 更新	增加了对 AWS Fault Injection Service（AWS FIS）与 Aurora DSQL 集成的支持。通过此集成，您可以向单区域和多区域 Aurora DSQL 集群注入故障，来测试应用程序的容错能力。您可以在 AWS FIS 控制台中创建实验模板来定义故障场景，并针对特定 Aurora DSQL 集群进行测试。 有关这些策略的更多信息，请参阅 AmazonAuroraDSQLFu	2025 年 8 月 19 日

更改	描述	日期
	llAccess 和 AmazonAuroraDSQLConsoleFullAccess。	
AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess 更新	添加了基于资源的策略（RBP）支持，新增了权限：PutClusterPolicy 、GetClusterPolicy 和 DeleteClusterPolicy 。这些权限支持管理附加到 Aurora DSQL 集群的内联策略，以实现精细的访问控制。 有关更多信息，请参阅 AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess。	2025 年 10 月 15 日
AmazonAuroraDSQLFullAccess 更新	添加了对 Aurora DSQL 集群执行备份和还原操作的功能，包括启动、停止和监控作业。它还添加了使用客户自主管理型 KMS 密钥进行集群加密的功能。 有关更多信息，请参阅 AmazonAuroraDSQLFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 21 日

更改	描述	日期
AmazonAuroraDSQLConsoleFullAccess 更新	添加了通过 AWS Console Home对 Aurora DSQL 集群执行备份和还原操作的功能。这包括启动、停止和监控作业。它还支持使用客户自主管理型 KMS 密钥进行集群加密和启动 AWS CloudShell。 有关更多信息，请参阅 AmazonAuroraDSQLConsoleFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 21 日

更改	描述	日期
AmazonAuroraDSQLFullAccess 更新	该策略添加了四个新权限，用于跨多个 AWS 区域创建和管理数据库集群：`PutMultiRegionProperties` `PutWitnessRegion` 、`AddPeerCluster` 和 `RemovePeerCluster` 。这些权限包括资源级控制措施和条件键，因此您可以控制您可以修改哪些集群用户。 该策略还添加了 `GetVpcEndpointServiceName` 权限，有助于您通过 AWS PrivateLink 连接到 Aurora DSQL 集群。 有关更多信息，请参阅 AmazonAuroraDSQLFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日

更改	描述	日期
AmazonAuroraDSQLReadOnlyAccess 更新	包括在通过 AWS PrivateLink 连接到 Aurora DSQL 集群时确定正确的 VPC 端点服务名称的功能。Aurora DSQL 为每个单元创建唯一的端点，因此，此 API 有助于确保您可以为集群识别正确的端点并避免连接错误。 有关更多信息，请参阅 AmazonAuroraDSQLReadOnlyAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日
AmazonAuroraDSQLConsoleFullAccess 更新	向 Aurora DSQL 添加新的权限，以支持多区域集群管理和 VPC 端点连接。新权限包括：`PutMultiRegionProperties`、`PutWitnessRegion`、`AddPeerCluster`、`RemovePeerCluster`、`GetVpcEndpointServiceName` 有关更多信息，请参阅 AmazonAuroraDSQLConsoleFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日

更改	描述	日期
AuroraDsqlServiceLinkedRolePolicy 更新	向策略中添加了将指标发布到 `AWS/AuroraDSQL` 和 `AWS/Usage` CloudWatch 命名空间的功能。这样，关联的服务或角色就可以向 CloudWatch 环境发送更全面的使用情况和性能数据。 有关更多信息，请参阅 AuroraDsqlServiceLinkedRolePolicy 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 8 日
页面已创建	已开始跟踪与 Amazon Aurora DSQL 相关的 AWS 托管式策略	2024 年 12 月 3 日

Amazon Aurora DSQL 中的数据保护

责任共担模式适用于数据保护。如该模式中所述，Amazon 负责保护全面运行 AWS Cloud的全球基础设施。您负责维护对托管在此基础结构上的内容的控制。您还负责您所使用的 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 Security Blog 上的 Shared Responsibility Model and GDPR 博客文章。

出于数据保护目的，我们建议您使用 AWS IAM Identity Center 或 AWS Identity and Access Management 来保护凭证和设置各个用户。这样，每个用户只获得履行其工作职责所需的权限。还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证（MFA）。
- 使用 SSL/TLS 与 资源进行通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用 AWS CloudTrail 设置 API 和用户活动日记账记录。有关使用跟踪来捕获活动的信息，请参阅《User Guide》中的 Working with trails。
- 使用加密解决方案以及 AWS 服务中的所有默认安全控制。

- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。

强烈建议您切勿将机密信息或敏感信息（如客户电子邮件地址）放入标签或自由格式文本字段（如名称字段）中。这包括使用控制台、API、AWS CLI 或 AWS SDK 处理其它内容时。在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供 URL，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

数据加密

Amazon Aurora DSQL 提供了专为任务关键型和主要数据存储设计的高度持久的存储基础设施。在 Aurora DSQL 区域中，数据以冗余方式存储在多个设施间的多个设备中。

传输中加密

默认情况下，为您配置了传输中加密。Aurora DSQL 使用 TLS 来加密 SQL 客户端和 Aurora DSQL 之间的所有流量。

对 AWS CLI、SDK 或 API 客户端与 Aurora DSQL 端点之间的传输中数据进行加密和签名：

- Aurora DSQL 为加密传输中数据提供了 HTTPS 端点。
- 为了保护向 Aurora DSQL 发出的 API 请求的完整性，API 调用必须由调用方签名。调用由 X.509 证书或客户的 AWS 秘密访问密钥根据前面版本 4 签名流程 (Sigv4) 签名。有关更多信息，请参阅《AWS 一般参考》中的签名版本 4 签名流程。
- 使用 AWS CLI 或 AWS 开发工具包之一向 AWS 发出请求。这些工具会自动使用您在配置工具时指定的访问密钥为您签署请求。

FIPS 合规性

默认情况下，Aurora DSQL 数据面板端点（用于数据库连接的集群端点）使用经 FIPS 140-2 验证的加密模块。集群连接不需要单独的 FIPS 端点。

对于控制面板操作，Aurora DSQL 在支持的区域中提供专用 FIPS 端点。有关控制面板 FIPS 端点的详细信息，请参阅《AWS 一般参考》中的 Aurora DSQL endpoints and quotas。

有关静态加密，请参阅 Aurora DSQL 中的静态加密。

互联网络流量隐私

Aurora DSQL 与本地应用程序之间以及 Aurora DSQL 与同一 AWS 区域内的其它 AWS 资源之间的连接均受到保护。

在您的私有网络和 AWS 之间有两个连接选项：

- 一个AWS Site-to-Site VPN 连接。有关更多信息，请参阅什么是 AWS Site-to-Site VPN？
- 一个 Direct Connect 连接。有关更多信息，请参阅什么是 Direct Connect？

使用 AWS 发布的 API 操作通过网络获取 Aurora DSQL 的访问权限。客户端必须支持以下内容：

- 传输层安全性协议（TLS）。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密（PFS）的密码套件，例如 DHE（临时 Diffie-Hellman）或 ECDHE（临时椭圆曲线 Diffie-Hellman）。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

见证区域中的数据保护

创建多区域集群时，见证区域通过参与加密事务的同步复制来协助实现自动故障恢复。如果对等集群变为不可用，则见证区域保持可用于验证和处理数据库写入，从而确保不丢失可用性。

见证区域通过以下这些设计功能保护和保障您的数据：

- 见证区域仅接收和存储加密的事务日志。它从不托管、存储或传输您的加密密钥。
- 见证区域只重点关注写入事务日志记录和仲裁函数。根据设计，它无法读取您的数据。
- 见证区域在没有集群连接端点或查询处理器的情况下运行。这会阻止用户访问数据库。

有关见证区域的更多信息，请参阅配置多区域集群。

为 Aurora DSQL 连接配置 SSL/TLS 证书

Aurora DSQL 要求所有连接均使用传输层安全性协议（TLS）加密。要建立安全连接，客户端系统必须信任 Amazon 根证书颁发机构（Amazon Root CA 1）。此证书预安装在许多操作系统上。本节提供在各种操作系统上验证预安装的 Amazon Root CA 1 证书的说明，并指导您完成手动安装证书的过程（如果证书尚不存在）。

建议使用 PostgreSQL 版本 17。

Important

对于生产环境，建议使用 `verify-full` SSL 模式，以确保最高级别的连接安全性。此模式验证服务器证书是否由受信任的证书颁发机构签名，以及服务器主机名是否与证书匹配。

验证预安装证书

在大多数操作系统中，已经预安装了 Amazon Root CA 1。要验证这一点，您可以按照以下步骤操作。

Linux（RedHat/CentOS/Fedora）

在终端中运行以下命令：

```
trust list | grep "Amazon Root CA 1"
```

如果证书已安装，将显示以下输出：

```
label: Amazon Root CA 1
```

macOS

1. 打开 Spotlight 搜索（Command + 空格）
2. 搜索 Keychain Access
3. 在系统密钥链下选择系统根
4. 在证书列表中查找 Amazon Root CA 1

Windows

Note

由于 psql Windows 客户端存在一个已知问题，因此使用系统根证书 (`sslrootcert=system`) 可能会返回以下错误：`SSL error: unregistered scheme`。您可以采用从 Windows 进行连接作为使用 SSL 连接到集群的替代方法。

如果操作系统中未安装 Amazon Root CA 1，请按照以下步骤操作。

安装证书

如果操作系统上未预安装 `Amazon Root CA 1` 证书，则需要手动安装该证书，以便与 Aurora DSQL 集群建立安全连接。

Linux 证书安装

按照以下步骤在 Linux 系统上安装 Amazon 根 CA 证书。

1. 下载根证书：

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 将证书复制到信任存储：

```
sudo cp ./AmazonRootCA1.pem /etc/pki/ca-trust/source/anchors/
```

3. 更新 CA 信任存储：

```
sudo update-ca-trust
```

4. 验证安装：

```
trust list | grep "Amazon Root CA 1"
```

macOS 证书安装

这些证书安装步骤是可选的。Linux 证书安装也适用于 macOS。

1. 下载根证书：

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 将证书添加到系统密钥链：

```
sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain
 AmazonRootCA1.pem
```

3. 验证安装：

```
security find-certificate -a -c "Amazon Root CA 1" -p /Library/Keychains/
System.keychain
```

使用 SSL/TLS 验证进行连接

在配置 SSL/TLS 证书以便与 Aurora DSQL 集群建立安全连接之前，请确保满足以下先决条件。

- 已安装 PostgreSQL 版本 17
- 使用适当的凭证配置了 AWS CLI
- Aurora DSQL 集群端点信息

从 Linux 进行连接

1. 生成并设置身份验证令牌：

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --hostname your-cluster-endpoint)
```

2. 使用系统证书（如果已预安装）进行连接：

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. 或者，使用下载的证书进行连接：

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

Note

有关 PGSSLMODE 设置的更多信息，请参阅 PostgresQL 17 Database Connection Control Functions 文档中的 sslmode。

从 macOS 进行连接

1. 生成并设置身份验证令牌：

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --hostname your-cluster-endpoint)
```

2. 使用系统证书（如果已预安装）进行连接：

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. 或者，下载根证书并将其另存为 `root.pem`（如果未预安装证书）

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

4. 使用 psql 进行连接：

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

从 Windows 进行连接

使用命令提示符

1. 生成身份验证令牌：

```
aws dsql generate-db-connect-admin-auth-token ^
--region=your-cluster-region ^
--expires-in=3600 ^
--hostname=your-cluster-endpoint
```

2. 设置密码环境变量：

```
set "PGPASSWORD=token-from-above"
```

3. 设置 SSL 配置：

```
set PGSSLROOTCERT=C:\full\path\to\root.pem
set PGSSLMODE=verify-full
```

4. 连接到数据库：

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres ^
--username admin ^
--host your-cluster-endpoint
```

使用 PowerShell

1. 生成并设置身份验证令牌：

```
$env:PGPASSWORD = (aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --expires-in=3600 --hostname=your-cluster-endpoint)
```

2. 设置 SSL 配置：

```
$env:PGSSLROOTCERT='C:\full\path\to\root.pem'
$env:PGSSLMODE='verify-full'
```

3. 连接到数据库：

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres `
```

```
--username admin `
--host your-cluster-endpoint
```

其他资源

- PostgreSQL SSL 文档
- Amazon Trust Services

Amazon Aurora DSQL 的数据加密

Amazon Aurora DSQL 可对所有用户静态数据进行加密。为了增强安全性，此加密使用 AWS Key Management Service（AWS KMS）。此功能减少保护敏感数据时涉及的操作负担和复杂性。静态加密有助于：

- 减少保护敏感数据的运营负担
- 构建符合严格的加密合规性和法规要求的安全敏感型应用程序
- 通过始终在加密集群中保护数据，增加额外的一层数据保护
- 遵守组织策略、行业或政府法规以及合规性要求

使用 Aurora DSQL，可以构建符合严格的加密合规性和法规要求的安全敏感型应用程序。以下各节说明如何为新的和现有 Aurora DSQL 数据库配置加密以及如何管理加密密钥。

主题

- Aurora DSQL 的 KMS 密钥类型
- Aurora DSQL 中的静态加密
- 将 AWS KMS 和数据密钥与 Aurora DSQL 结合使用
- 授权将 AWS KMS key用于 Aurora DSQL
- Aurora DSQL 加密上下文
- 监控 Aurora DSQL 与 AWS KMS 的交互
- 创建加密的 Aurora DSQL 集群
- 移除或更新 Aurora DSQL 集群的密钥
- 使用 Aurora DSQL 进行加密的注意事项

Aurora DSQL 的 KMS 密钥类型

Aurora DSQL 与 AWS KMS 集成，以管理集群的加密密钥。要了解有关密钥类型和状态的更多信息，请参阅《AWS Key Management Service Developer Guide》中的 AWS Key Management Service concepts。创建新集群时，可以从以下 KMS 密钥类型中进行选择来对集群进行加密：

AWS 拥有的密钥

默认加密类型。Aurora DSQL 拥有密钥，不向您收取额外费用。当您访问加密集群时，Amazon Aurora DSQL 会透明地解密集群数据。您无需更改代码或应用程序即可使用或管理加密集群，并且所有 Aurora DSQL 查询都将处理加密的数据。

客户自主管理型密钥

您可以在 AWS 账户中创建、拥有和管理密钥。您对 KMS 密钥拥有完全控制权。将收取 AWS KMS 费用。

使用 AWS 拥有的密钥进行静态加密不另行收费。但是，使用客户自主管理型密钥需支付 AWS KMS 费用。有关更多信息，请参阅 AWS KMS 定价页面。

您可以随时在这些密钥类型之间切换。有关密钥类型的更多信息，请参阅《AWS Key Management Service Developer Guide》中的 Customer managed keys 和 AWS 拥有的密钥。

Aurora DSQL 静态加密可在所有提供 Aurora DSQL 的 AWS 区域中使用。

Aurora DSQL 中的静态加密

Amazon Aurora 使用 256 位高级加密标准（AES-256）对静态数据进行加密。这种加密功能有助于保护您的数据，来防止对底层存储进行未经授权的访问。AWS KMS 管理集群的加密密钥。您可以使用默认 AWS 拥有的密钥，也可以选择使用您自己的 AWS KMS 客户管理密钥。要了解有关为 Aurora DSQL 集群指定和管理密钥的更多信息，请参阅创建加密的 Aurora DSQL 集群和移除或更新 Aurora DSQL 集群的密钥。

主题

- AWS 拥有的密钥
- 客户管理密钥

AWS 拥有的密钥

默认情况下，Aurora DSQL 使用 AWS 拥有的密钥对所有集群进行加密。这些密钥可免费使用，并且每年轮换，以保护您的账户资源。您无需查看、管理、使用或审计这些密钥，因此无需采取任何措施来保护数据。有关 AWS 拥有的密钥 的更多信息，请参阅《AWS Key Management Service 开发人员指南》中的 AWS 拥有的密钥。

客户管理密钥

您可以在 AWS 账户中创建、拥有和管理客户自主管理型密钥。您完全控制这些 KMS 密钥，包括其策略、加密材料、标签和别名。有关管理权限的更多信息，请参阅《AWS Key Management Service Developer Guide》中的 Customer managed keys。

当您指定客户自主管理型密钥来进行集群级加密时，Aurora DSQL 使用该密钥对集群及其所有区域数据进行加密。为了防止数据丢失和维护集群访问权限，Aurora DSQL 需要访问您的加密密钥。如果您禁用客户自主管理型密钥、计划删除密钥或具有限制服务访问权限的策略，则集群的加密状态将更改为 `KMS_KEY_INACCESSIBLE`。当 Aurora DSQL 无法访问密钥时，用户无法连接到集群，集群的加密状态更改为 `KMS_KEY_INACCESSIBLE`，而服务将无法访问集群数据。

对于多区域集群，客户可以单独配置每个区域的 AWS KMS 加密密钥，而每个区域集群均使用自己的集群级加密密钥。如果 Aurora DSQL 无法访问多区域集群中某个对等集群的加密密钥，则该对等集群的状态将变为 `KMS_KEY_INACCESSIBLE`，而不可用于读取和写入操作。其它对等集群继续正常运行。

> **Note**
>
> 如果 Aurora DSQL 无法访问客户自主管理型密钥，则集群加密状态将变为 `KMS_KEY_INACCESSIBLE`。还原密钥访问权限后，服务将在 15 分钟内自动检测到还原情况。有关更多信息，请参阅“集群闲置”。
> 对于多区域集群，如果长时间丢失密钥访问权限，则集群还原时间取决于当密钥无法访问时写入了多少数据。

将 AWS KMS 和数据密钥与 Aurora DSQL 结合使用

Aurora DSQL 静态加密功能使用 AWS KMS key和数据密钥的层次结构来保护集群数据。

建议您在 Aurora DSQL 中实施集群之前先制定加密策略计划。如果您要在 Aurora DSQL 中存储敏感或机密数据，请考虑在计划中包括客户端加密。这样，您就可以尽量靠近数据源来加密数据，确保其在整个生命周期中受到保护。

主题

- 将 AWS KMS key与 Aurora DSQL 结合使用
- 将集群密钥与 Aurora DSQL 结合使用
- 集群密钥缓存

将 AWS KMS key与 Aurora DSQL 结合使用

静态加密使用 AWS KMS key保护 Aurora DSQL 集群。默认情况下，Aurora DSQL 使用 AWS 拥有的密钥，即在 Aurora DSQL 服务账户中创建并管理的多租户加密密钥。但是，您可以使用 AWS 账户中的客户自主管理型密钥对 Aurora DSQL 集群进行加密。您可以为每个集群选择不同的 KMS 密钥，即使集群参与多区域设置。

您可以在创建或更新集群时为集群选择 KMS 密钥。您可以通过以下方式随时更改集群的 KMS 密钥：在 Aurora DSQL 控制台中或使用 `UpdateCluster` 操作。切换密钥的过程不需要停机或降低服务质量。

Important

Aurora DSQL 仅支持对称 KMS 密钥。不能使用非对称 KMS 密钥来加密 Aurora DSQL 集群。

客户自主管理型密钥提供以下优势。

- 您可以创建和管理 KMS 密钥，包括设置密钥策略和 IAM 策略来控制对 KMS 密钥的访问。您可以启用和禁用 KMS 密钥、启用和禁用自动密钥轮换，以及当 KMS 密钥不再使用时删除 KMS 密钥。
- 您可以使用具有导入的密钥材料的客户托管密钥，或者您拥有和管理的自定义密钥存储中的客户托管密钥。
- 您可以通过检查 AWS CloudTrail 日志中对 AWS KMS 的 Aurora DSQL API 调用来审计 Aurora DSQL 集群的加密和解密。

但是，AWS 拥有的密钥是免费的，其使用不会计入 AWS KMS 资源或请求配额。客户自主管理型密钥会针对每次 API 调用产生费用，并且对于此类密钥具有 AWS KMS 配额。

将集群密钥与 Aurora DSQL 结合使用

Aurora DSQL 对集群使用 AWS KMS key来为集群生成并加密一个唯一的数据密钥，称为集群密钥。

该集群密钥用作密钥加密密钥。Aurora DSQL 使用此集群密钥来保护用于加密集群数据的数据加密密钥。Aurora DSQL 为集群中的每个底层结构生成唯一的数据加密密钥，但多个集群项目可能受相同的数据加密密钥保护。

为了解密集群密钥，Aurora DSQL 会在您首次访问加密集群时向 AWS KMS 发送请求。为保持集群可用，Aurora DSQL 会定期验证对 KMS 密钥的解密访问权限，即使您没有积极地访问集群也是如此。

Aurora DSQL 在 AWS KMS 外部存储和使用集群密钥与数据加密密钥。它会借助高级加密标准 (AES) 加密和 256 位加密密钥保护所有密钥。然后，它存储加密密钥及加密数据，以便它们可根据需要用于解密集群数据。

如果您更改集群的 KMS 密钥，Aurora DSQL 会使用新的 KMS 密钥重新加密现有集群密钥。

集群密钥缓存

为了避免针对每个 Aurora DSQL 操作调用 AWS KMS，Aurora DSQL 会针对每个调用方将明文集群密钥缓存在内存中。如果 Aurora DSQL 在处于不活动状态 15 分钟后获取对缓存集群密钥的请求，它会向 AWS KMS 发送新请求来解密集群密钥。此调用将捕获在上次请求解密集群密钥之后对 AWS KMS 或 AWS Identity and Access Management（IAM）中 AWS KMS key的访问策略所做的任何更改。

授权将 AWS KMS key用于 Aurora DSQL

如果您使用您账户中的客户自主管理型密钥来保护 Aurora DSQL 集群，则该密钥的策略必须向 Aurora DSQL 授予代表您使用该密钥的权限。

您可以全面控制有关客户自主管理型密钥的策略。Aurora DSQL 无需额外授权，即可使用默认 AWS 拥有的密钥来保护您 AWS 账户中的 Aurora DSQL 集群。

客户托管密钥的密钥策略

当您选择客户自主管理型密钥来保护 Aurora DSQL 集群时，Aurora DSQL 需要相应的权限，以便代表做出选择的主体使用 AWS KMS key。该主体（用户或角色）必须对 Aurora DSQL 所需的 AWS KMS key拥有权限。您可以在密钥策略或 IAM 策略中提供这些权限。

Aurora DSQL 对客户自主管理型密钥至少需要具备以下权限：

- `kms:Encrypt`
- `kms:Decrypt`
- `kms:ReEncrypt*`（适用于 kms:ReEncryptFrom 和 kms:ReEncryptTo）
- `kms:GenerateDataKey`

- kms:DescribeKey

例如，以下示例密钥策略仅提供所需的权限。该策略具有以下效果：

- 支持 Aurora DSQL 在加密操作中使用 AWS KMS key，但仅当它代表账户中有权使用 Aurora DSQL 的主体行事时才可如此。如果在策略语句中指定的主体无权使用 Aurora DSQL，调用将失败，即使调用来自 Aurora DSQL 服务也是如此。
- kms:ViaService 条件密钥仅当 Aurora DSQL 代表策略语句中所列主体发出请求时，才支持此类权限。这些主体不能直接调用这些操作。

在使用示例密钥策略之前，请将示例委托人替换为 AWS 账户 中的实际委托人。

```
{
  "Sid": "Enable dsql IAM User Permissions",
  "Effect": "Allow",
  "Principal": {
    "Service": "dsql.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey",
    "kms:Encrypt",
    "kms:ReEncryptFrom",
    "kms:ReEncryptTo"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:aws:dsql:ClusterId": "w4abucpbwuxx",
      "aws:SourceArn": "arn:aws:dsql:us-east-2:111122223333:cluster/w4abucpbwuxx"
    }
  }
},
{
  "Sid": "Enable dsql IAM User Describe Permissions",
  "Effect": "Allow",
  "Principal": {
    "Service": "dsql.amazonaws.com"
  },
  "Action": "kms:DescribeKey",
  "Resource": "*",
```

```
  "Condition": {
    "StringLike": {
      "aws:SourceArn": "arn:aws:dsql:us-east-2:111122223333:cluster/w4abucpbwuxx"
    }
  }
}
```

Aurora DSQL 加密上下文

加密上下文 是一组包含任意非机密数据的键值对。在请求中包含加密上下文以加密数据时，AWS KMS 以加密方式将加密上下文绑定到加密的数据。要解密数据，您必须传入相同的加密上下文。

Aurora DSQL 在所有 AWS KMS 加密操作中使用相同的加密上下文。如果您使用客户自主管理型密钥来保护 Aurora DSQL 集群，则可使用加密上下文在审计记录和日志中确定 AWS KMS key的使用情况。它也以明文形式显示在日志中，例如 AWS CloudTrail 中的那些日志。

加密上下文还可用作在策略中进行授权的条件。

在向 AWS KMS 发出的请求中，Aurora DSQL 使用具有密钥/值对的加密上下文：

```
"encryptionContext": {
  "aws:dsql:ClusterId": "w4abucpbwuxx"
},
```

密钥/值对标识 Aurora DSQL 要加密的集群。键是 `aws:dsql:ClusterId`。值是集群的标识符。

监控 Aurora DSQL 与 AWS KMS 的交互

如果您使用客户自主管理型密钥来保护 Aurora DSQL 集群，则可以使用 AWS CloudTrail 日志来跟踪 Aurora DSQL 代表您发送到 AWS KMS 的请求。

展开以下各节，了解 Aurora DSQL 如何使用 AWS KMS 操作 `GenerateDataKey` 和 `Decrypt`。

GenerateDataKey

当您对集群启用静态加密时，Aurora DSQL 会创建一个唯一的集群密钥。它向 AWS KMS 发送 `GenerateDataKey` 请求，以便为集群指定 AWS KMS key。

记录 `GenerateDataKey` 操作的事件与以下示例事件类似。该用户是 Aurora DSQL 服务账户。参数包括 AWS KMS key的 Amazon 资源名称（ARN）、需要 256 位密钥的密钥说明符以及标识集群的加密上下文。


```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "type": "AWSService",
        "invokedBy": "dsql.amazonaws.com"
    },
    "eventTime": "2025-05-16T18:41:24Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "dsql.amazonaws.com",
    "userAgent": "dsql.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "aws:dsql:ClusterId": "w4abucpbwuxx"
        },
        "keySpec": "AES_256",
        "keyId": "arn:aws:kms:us-east-1:982127530226:key/8b60dd9f-2ff8-4b1f-8a9c-
bf570cbfdb5e"
    },
    "responseElements": null,
    "requestID": "2da2dc32-d3f4-4d6c-8a41-aff27cd9a733",
    "eventID": "426df0a6-ba56-3244-9337-438411f826f4",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:982127530226:key/8b60dd9f-2ff8-4b1f-8a9c-
bf570cbfdb5e"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "sharedEventID": "f88e0dd8-6057-4ce0-b77d-800448426d4e",
    "vpcEndpointId": "AWS Internal",
    "vpcEndpointAccountId": "vpce-1a2b3c4d5e6f1a2b3",
    "eventCategory": "Management"
```

```
}
```

Decrypt

当您访问加密的 Aurora DSQL 集群时，Aurora DSQL 需要解密集群密钥，以便它可以解密层次结构中位于其下方的密钥。然后，它解密集群中的数据。为了解密集群密钥，Aurora DSQL 向 AWS KMS 发送 `Decrypt` 请求，以便为集群指定 AWS KMS key。

记录 `Decrypt` 操作的事件与以下示例事件类似。用户是您的 AWS 账户中正在访问集群的主体。参数包括加密的集群密钥（作为加密文字 blob）以及标识集群的加密上下文。AWS KMS 从加密文字中得出 AWS KMS key的 ID。


```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AWSService",
    "invokedBy": "dsql.amazonaws.com"
  },
  "eventTime": "2018-02-14T16:42:39Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "Decrypt",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "dsql.amazonaws.com",
  "userAgent": "dsql.amazonaws.com",
  "requestParameters": {
    "keyId": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    "encryptionContext": {
      "aws:dsql:ClusterId": "w4abucpbwuxx"
    },
    "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
  },
  "responseElements": null,
  "requestID": "11cab293-11a6-11e8-8386-13160d3e5db5",
  "eventID": "b7d16574-e887-4b5b-a064-bf92f8ec9ad3",
  "readOnly": true,
  "resources": [
    {
      "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
      "accountId": "AWS Internal",
      "type": "AWS::KMS::Key"
```

```
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
  "sharedEventID": "d99f2dc5-b576-45b6-aa1d-3a3822edbeeb",
  "vpcEndpointId": "AWS Internal",
  "vpcEndpointAccountId": "vpce-1a2b3c4d5e6f1a2b3",
  "eventCategory": "Management"
}
```

创建加密的 Aurora DSQL 集群

所有 Aurora DSQL 集群都已静态加密。默认情况下，集群免费使用 AWS 拥有的密钥，您也可以指定自定义 AWS KMS 密钥。按照以下步骤从 AWS 管理控制台或 AWS CLI 创建加密集群。

Console

在 AWS 管理控制台中创建加密集群

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 在控制台左侧的导航窗格中，选择集群。
3. 选择右上角的创建集群，然后选择单区域。
4. 在集群加密设置中，选择以下选项之一。
    - 接受默认设置，无需支付额外费用即可使用 AWS 拥有的密钥进行加密。
    - 选择自定义加密设置（高级）以指定自定义 KMS 密钥。然后，搜索或输入 KMS 密钥的 ID 或别名。或者，选择创建 AWS KMS 密钥以便在 AWS KMS 控制台中创建新密钥。
5. 选择创建集群。

要确认集群的加密类型，请导航到集群页面，并选择集群的 ID 以查看集群详细信息。查看集群设置选项卡，集群 KMS 密钥设置会对使用 AWS 拥有的密钥的集群显示 Aurora DSQL 默认密钥，或对于其它加密类型显示密钥 ID。

Note

如果您选择拥有和管理您自己的密钥，请确保适当地设置 KMS 密钥策略。有关示例和更多信息，请参阅 the section called “客户托管密钥的密钥策略”。

CLI

创建使用默认 AWS 拥有的密钥加密的集群

- 使用以下命令创建 Aurora DSQL 集群。

```
aws dsql create-cluster
```

如以下加密详细信息所示，默认情况下，集群的加密状态为启用，默认加密类型为 AWS 拥有的密钥。现在，该集群已使用 Aurora DSQL 服务账户中的默认 AWS 拥有的密钥进行加密。

```
"encryptionDetails": {
  "encryptionType" : "AWS_OWNED_KMS_KEY",
  "encryptionStatus" : "ENABLED"
}
```

创建使用客户自主管理型密钥加密的集群

- 使用以下命令创建 Aurora DSQL 集群，同时将红色文本的密钥 ID 替换为客户自主管理型密钥的 ID。

```
aws dsql create-cluster \
--kms-encryption-key d41d8cd98f00b204e9800998ecf8427e
```

如以下加密详细信息所示，默认情况下，集群的加密状态为启用，加密类型为客户自主管理型 KMS 密钥。集群现在已使用您的密钥进行加密。

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:kms:us-east-1:111122223333:key/
d41d8cd98f00b204e9800998ecf8427e",
  "encryptionStatus" : "ENABLED"
```

```
}
```

移除或更新 Aurora DSQL 集群的密钥

可以使用 AWS 管理控制台或 AWS CLI 在 Amazon Aurora DSQL 中更新或移除现有集群上的加密密钥。如果您移除密钥而不替换它，Aurora DSQL 将使用默认的 AWS 拥有的密钥。按照以下步骤从 Aurora DSQL 控制台或 AWS CLI 更新现有集群的加密密钥。

Console

在 AWS 管理控制台中更新或移除加密密钥

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 在控制台左侧的导航窗格中，选择集群。
3. 从列表视图中，查找并选择要更新的集群所对应的行。
4. 选择操作菜单，然后选择修改。
5. 在集群加密设置中，选择以下选项之一来修改您的加密设置。
   - 如果要从自定义密钥切换到 AWS 拥有的密钥，请取消选择自定义加密设置（高级）选项。将应用默认设置，并使用 AWS 拥有的密钥免费加密您的集群。
   - 如果要从一个自定义 KMS 密钥切换到另一个自定义 KMS 密钥，或要从 AWS 拥有的密钥切换到 KMS 密钥，请选择自定义加密设置（高级）选项（如果尚未选择）。然后，搜索并选择您要使用的密钥的 ID 或别名。或者，选择创建 AWS KMS 密钥以便在 AWS KMS 控制台中创建新密钥。
6. 选择保存。

CLI

以下示例显示了如何使用 AWS CLI 来更新加密的集群。

使用默认 AWS 拥有的密钥更新加密的集群

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key "AWS_OWNED_KMS_KEY"
```

集群描述的 EncryptionStatus 设置为 ENABLED，而 EncryptionType 为 AWS_OWNED_KMS_KEY。

```
"encryptionDetails": {
  "encryptionType" : "AWS_OWNED_KMS_KEY",
  "encryptionStatus" : "ENABLED"
}
```

现在，此集群已使用 Aurora DSQL 服务账户中默认的 AWS 拥有的密钥进行加密。

在 Aurora DSQL 中使用客户自主管理型密钥更新加密的集群

更新加密的集群，如下例所示：

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234a1b234
```

集群描述的 EncryptionStatus 转换为 UPDATING，而 EncryptionType 为 CUSTOMER_MANAGED_KMS_KEY。Aurora DSQL 完成通过平台传播新密钥后，加密状态将转换为 ENABLED

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:us-east-1:kms:key/abcd1234-abcd-1234-a123-ab1234a1b234",
  "encryptionStatus" : "ENABLED"
}
```

Note

如果您选择拥有和管理您自己的密钥，请确保适当地设置 KMS 密钥策略。有关示例和更多信息，请参阅 the section called “客户托管密钥的密钥策略”。

使用 Aurora DSQL 进行加密的注意事项

- Aurora DSQL 对所有集群静态数据进行加密。您不能禁用此加密，也不能仅加密集群中的某些项目。
- AWS Backup 对您的备份以及从这些备份中还原的任何集群进行加密。您可以在 AWS Backup 中使用 AWS 拥有的密钥或客户自主管理型密钥加密备份数据。
- Aurora DSQL 已启用以下数据保护状态：
  - 静态数据：Aurora DSQL 对持久存储介质上的所有静态数据进行加密
  - 传输中数据：Aurora DSQL 默认情况下使用传输层安全性协议（TLS）对所有通信进行加密
- 当您转换为其它密钥时，我们建议您保持原始密钥处于启用状态，直到转换过程完成。AWS 在使用新密钥加密数据之前，需要使用原始密钥来解密数据。当集群的 `encryptionStatus` 为 `ENABLED` 并且您看到新的客户自主管理型密钥的 `kmsKeyArn` 时，该过程就完成了。
- 当您禁用客户自主管理型密钥或撤销 Aurora DSQL 使用您的密钥的访问权限时，集群将进入 `IDLE` 状态。
- AWS 管理控制台和 Amazon Aurora DSQL API 对加密类型使用不同的术语：
  - AWS 管理控制台：在控制台中，当使用客户自主管理型密钥时，您将看到 `KMS`；当使用 AWS 拥有的密钥时，您将看到 `DEFAULT`。
  - API：Amazon Aurora DSQL API 使用 `CUSTOMER_MANAGED_KMS_KEY` 来表示客户自主管理型密钥，而使用 `AWS_OWNED_KMS_KEY` 来表示 AWS 拥有的密钥。
- 如果您在创建集群期间未指定加密密钥，Aurora DSQL 会自动使用 AWS 拥有的密钥加密数据。
- 您可以随时在 AWS 拥有的密钥和客户自主管理型密钥之间切换。使用 AWS 管理控制台、AWS CLI 或 Amazon Aurora DSQL API 进行此更改。

Aurora DSQL 的身份和访问管理

AWS Identity and Access Management（IAM）是一项，AWS 服务可以帮助管理员安全地控制对 AWS 资源的访问。IAM 管理员控制谁可以通过身份验证（登录）和获得授权（具有权限）来使用 Aurora DSQL 资源。IAM 是一项无需额外费用即可使用的 AWS 服务。

主题

- 受众
- 使用身份进行身份验证
- 使用策略管理访问
- Amazon Aurora DSQL 如何与 IAM 协同工作
- Amazon Aurora DSQL 的基于身份的策略示例
- 对 Amazon Aurora DSQL 身份和访问权限问题进行故障排除

受众

您使用 AWS Identity and Access Management（IAM）的方式因您的角色而异：

- 服务用户：如果您无法访问功能，请从管理员处请求权限（请参阅对 Amazon Aurora DSQL 身份和访问权限问题进行故障排除）
- 服务管理员：确定用户访问权限并提交权限请求（请参阅Amazon Aurora DSQL 如何与 IAM 协同工作）
- IAM 管理员：编写用于管理访问权限的策略（请参阅Amazon Aurora DSQL 的基于身份的策略示例）

使用身份进行身份验证

身份验证是您使用身份凭证登录 AWS 的方法。您必须作为 AWS 账户根用户、IAM 用户或通过担任 IAM 角色进行身份验证。

您可以使用来自 AWS IAM Identity Center（IAM Identity Center）等身份源的凭证、单点登录身份验证或 Google/Facebook 凭证，以联合身份进行登录。有关登录的更多信息，请参阅《AWS 登录 用户指南》中的如何登录您的 AWS 账户。

对于编程访问，AWS 提供了 SDK 和 CLI 来对请求进行加密签名。有关更多信息，请参阅《IAM 用户指南》中的适用于 API 请求的 AWS 签名版本 4。

AWS 账户 根用户

当您创建 AWS 账户 时，最初使用的是一个对所有 AWS 服务和资源拥有完全访问权限的登录身份（称为 AWS 账户根用户）。我们强烈建议不要使用根用户进行日常任务。有关需要根用户凭证的任务，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，请要求人类用户必须使用带有身份提供者的联合身份验证才能使用临时凭证访问 AWS 服务。

联合身份是来自企业目录、Web 身份提供者的用户，或 Directory Service 中的用户（这些用户使用来自身份源的凭证访问 AWS 服务）。联合身份代入可提供临时凭证的角色。

要集中管理访问权限，建议使用。AWS IAM Identity Center有关更多信息，请参阅《AWS IAM Identity Center 用户指南》中的什么是 IAM Identity Center？。

IAM 用户和群组

IAM 用户是对某个人员或应用程序具有特定权限的一个身份。建议使用临时凭证，而非具有长期凭证的 IAM 用户。有关更多信息，请参阅《IAM 用户指南》中的要求人类用户使用带有身份提供商的联合身份验证才能使用临时凭证访问 AWS。

IAM 组指定一组 IAM 用户，便于更轻松地对大量用户进行权限管理。有关更多信息，请参阅《IAM 用户指南》中的 IAM 用户使用案例。

IAM 角色

IAM 角色是具有特定权限的身份，可提供临时凭证。您可以通过从用户切换到 IAM 角色（控制台）或调用 AWS CLI 或 AWS API 操作来担任角色。有关更多信息，请参阅《IAM 用户指南》中的担任角色的方法。

IAM 角色对于联合用户访问、临时 IAM 用户权限、跨账户访问、跨服务访问以及在 Amazon EC2 上运行的应用程序非常有用。有关更多信息，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

使用策略管理访问

您将创建策略并将其附加到 AWS 身份或资源，以控制 AWS 中的访问。策略在与身份或资源关联时定义权限。当主体发出请求时，AWS 会评估这些策略。大多数策略在 AWS 中存储为 JSON 文档。有关 JSON 策略文档的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概述。

管理员使用策略，通过定义哪个主体可以在什么条件下对哪些资源执行哪些操作来指定谁有权访问什么。

默认情况下，用户和角色没有权限。IAM 管理员创建 IAM 策略并将其添加到角色中，然后用户可以担任这些角色。IAM 策略定义权限，与执行操作所用的方法无关。

基于身份的策略

基于身份的策略是您附加到身份（用户、组或角色）的 JSON 权限策略文档。这些策略控制身份可以执行什么操作、对哪些资源执行以及在什么条件下执行。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

基于身份的策略可以是内联策略（直接嵌入到单个身份中）或托管策略（附加到多个身份的独立策略）。要了解如何在托管策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管策略与内联策略之间进行选择。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。您必须在基于资源的策略中指定主体。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用来自 IAM 的 AWS 托管策略。

其他策略类型

AWS 支持额外的策略类型，这些策略类型可以设置由更常用的策略类型授予的最大权限：

- 权限边界 – 设置基于身份的策略可以授予 IAM 实体的最大权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM 实体的权限边界。
- 服务控制策略（SCP）– 指定 AWS Organizations 中组织或组织单元的最大权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的服务控制策略。
- 资源控制策略（RCP）– 设置对账户中资源的最大可用权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的资源控制策略（RCP）。
- 会话策略 – 在为角色或联合用户创建临时会话时，作为参数传递的高级策略。有关更多信息，请参阅《IAM 用户指南》中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解 AWS 如何确定在涉及多种策略类型时是否允许请求，请参阅《IAM 用户指南》中的策略评估逻辑。

Amazon Aurora DSQL 如何与 IAM 协同工作

在使用 IAM 来管理对 Aurora DSQL 的访问权限之前，了解哪些 IAM 功能可与 Aurora DSQL 结合使用。

可以与 Amazon Aurora DSQL 结合使用的 IAM 功能

IAM 功能	Aurora DSQL 支持
基于身份的策略	是
基于资源的策略	是
策略操作	是
策略资源	是
策略条件键	是
ACL	否
ABAC（策略中的标签）	是
临时凭证	是
主体权限	是
服务角色	是
服务关联角色	是

要大致了解 Aurora DSQL 和其它 AWS 服务如何与大多数 IAM 功能结合使用，请参阅《IAM 用户指南》中的使用 IAM 的 AWS 服务。

Aurora DSQL 的基于身份的策略

支持基于身份的策略：是

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。要了解可在 JSON 策略中使用的所有元素，请参阅《IAM 用户指南》中的 IAM JSON 策略元素引用。

Aurora DSQL 的基于身份的策略示例

要查看 Aurora DSQL 基于身份的策略示例，请参阅 Amazon Aurora DSQL 的基于身份的策略示例。

Aurora DSQL 内基于资源的策略

支持基于资源的策略：是

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。主体可以包括账户、用户、角色、联合用户或亚马逊云科技服务。基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用来自 IAM 的 AWS 托管式策略。

要了解如何为 Aurora DSQL 集群创建和管理基于资源的策略，请参阅 Resource-based policies for Aurora DSQL。

Aurora DSQL 的策略操作

支持策略操作：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

JSON 策略的 `Action` 元素描述可用于在策略中允许或拒绝访问的操作。在策略中包含操作以授予执行关联操作的权限。

要查看 Aurora DSQL 操作的列表，请参阅《Service Authorization Reference》中的 Actions Defined by Amazon Aurora DSQL。

Aurora DSQL 中的策略操作在操作前使用以下前缀：

```
dsql
```

要在单个语句中指定多项操作，请使用逗号将它们隔开。

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

要查看 Aurora DSQL 基于身份的策略示例，请参阅 Amazon Aurora DSQL 的基于身份的策略示例。

Aurora DSQL 的策略资源

支持策略资源：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

Resource JSON 策略元素指定要向其应用操作的一个或多个对象。作为最佳实践，请使用其 Amazon 资源名称（ARN）指定资源。对于不支持资源级权限的操作，请使用通配符 (*) 指示语句应用于所有资源。

```
"Resource": "*"
```

要查看 Aurora DSQL 资源类型及其 ARN 的列表，请参阅《Service Authorization Reference》中的 Resources Defined by Amazon Aurora DSQL。要了解您可以使用哪些操作指定每个资源的 ARN，请参阅 Actions Defined by Amazon Aurora DSQL。

要查看 Aurora DSQL 基于身份的策略示例，请参阅 Amazon Aurora DSQL 的基于身份的策略示例。

Aurora DSQL 的策略条件键

支持特定于服务的策略条件键：是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

Condition 元素根据定义的条件指定语句何时执行。您可以创建使用条件运算符（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。要查看所有 AWS 全局条件键，请参阅《IAM 用户指南》中的 AWS 全局条件上下文键。

有关 Aurora DSQL 条件密钥的列表，请参阅《Service Authorization Reference》中的 Condition keys for Amazon Aurora DSQL。要了解您可以对哪些操作和资源使用条件密钥，请参阅 Actions defined by Amazon Aurora DSQL。

要查看 Aurora DSQL 基于身份的策略示例，请参阅 Amazon Aurora DSQL 的基于身份的策略示例。

Aurora DSQL 中的 ACL

支持 ACL：否

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，但它们不使用 JSON 策略文档格式。

ABAC 与 Aurora DSQL

支持 ABAC（策略中的标签）：是

基于属性的访问权限控制（ABAC）是一种授权策略，该策略基于称为标签的属性来定义权限。您可以将标签附加到 IAM 实体和 AWS 资源，然后设计 ABAC 策略，以支持在主体的标签与资源上的标签匹配时执行操作。

要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name` `aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。

如果某个服务对于每种资源类型都支持所有这三个条件键，则对于该服务，该值为是。如果某个服务仅对于部分资源类型支持所有这三个条件键，则该值为部分。

有关 ABAC 的更多信息，请参阅《IAM 用户指南》中的使用 ABAC 授权定义权限。要查看设置 ABAC 步骤的教程，请参阅《IAM 用户指南》中的使用基于属性的访问权限控制（ABAC）。

将临时凭证与 Aurora DSQL 结合使用

支持临时凭证：是

临时凭证提供对 AWS 资源的短期访问权限，并且是在您使用联合身份验证或切换角色时自动创建的。AWS 建议您动态生成临时凭证，而不是使用长期访问密钥。有关更多信息，请参阅《IAM 用户指南》中的 IAM 中的临时安全凭证和使用 IAM 的。AWS 服务

Aurora DSQL 的跨服务主体权限

支持转发访问会话（FAS）：是

转发访问会话（FAS）使用调用 AWS 服务 的主体的权限，与发出请求的 AWS 服务 结合，向下游服务发出请求。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。

Aurora DSQL 的服务角色

支持服务角色：是

服务角色是由一项服务担任、代表您执行操作的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务 委派权限的角色。

Warning

更改服务角色的权限可能会破坏 Aurora DSQL 的功能。仅当 Aurora DSQL 提供相关指导时，才编辑服务角色。

Aurora DSQL 的服务相关角色

支持服务关联角色：是

服务关联角色是一种与 AWS 服务 关联的服务角色。服务可以代入代表您执行操作的角色。服务关联角色显示在您的 AWS 账户 中，并由该服务拥有。IAM 管理员可以查看但不能编辑服务关联角色的权限。

有关创建或管理 Aurora DSQL 的服务相关角色的详细信息，请参阅使用 Aurora DSQL 中的服务相关角色。

Amazon Aurora DSQL 的基于身份的策略示例

默认情况下，用户和角色不具备创建或修改 Aurora DSQL 资源的权限。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。

要了解如何使用这些示例 JSON 策略文档创建基于 IAM 身份的策略，请参阅《IAM 用户指南》中的创建 IAM 策略（控制台）。

有关 Aurora DSQL 定义的操作和资源类型的详细信息，包括每种资源类型的 ARN 格式，请参阅《Service Authorization Reference》中的 Actions, Resources, and Condition Keys for Amazon Aurora DSQL。

主题

• 策略最佳实践

- 使用 Aurora DSQL 控制台
- 允许用户查看他们自己的权限
- 支持集群管理和数据库连接
- Aurora DSQL 基于标签的资源访问权限

策略最佳实践

基于身份的策略确定某个人是否可以创建、访问或删除您账户中的 Aurora DSQL 资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下指南和建议：

- AWS 托管式策略及转向最低权限许可入门：要开始向用户和工作负载授予权限，请使用 AWS 托管式策略来为许多常见使用场景授予权限。您可以在 AWS 账户 中找到这些策略。建议通过定义特定于您的使用场景的 AWS 客户托管式策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管策略或工作职能的 AWS 托管策略。
- 应用最低权限：在使用 IAM 策略设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的 IAM 中的策略和权限。
- 使用 IAM 策略中的条件进一步限制访问权限：您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果通过特定 AWS 服务（例如 CloudFormation）使用服务操作，您还可以使用条件来授予对服务操作的访问权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。
- 使用 IAM Access Analyzer 验证您的 IAM 策略，以确保权限的安全性和功能性：IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM 策略语言（JSON）和 IAM 最佳实践。IAM Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的使用 IAM Access Analyzer 验证策略。
- 需要多重身份验证（MFA）：如果您所处的场景要求您的 AWS 账户 中有 IAM 用户或根用户，请启用 MFA 来提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的使用 MFA 保护 API 访问。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实践。

使用 Aurora DSQL 控制台

要访问 Amazon Aurora DSQL 控制台，您必须具有一组最低的权限。这些权限必须支持您列出和查看有关您的 AWS 账户中 Aurora DSQL 资源的详细信息。如果创建比必需的最低权限更为严格的基于身份的策略，对于附加了该策略的实体（用户或角色），控制台将无法按预期正常运行。

对于只需要调用 AWS CLI 或 AWS API 的用户，无需为其提供最低控制台权限。相反，只允许访问与其尝试执行的 API 操作相匹配的操作。

为确保用户和角色仍可使用 Aurora DSQL 控制台，还要将 Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess` 或 `AmazonAuroraDSQLReadOnlyAccess` AWS 托管式策略附加到实体。有关更多信息，请参阅《IAM 用户指南》中的为用户添加权限。

允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管式策略。此策略包括在控制台上完成此操作或者以编程方式使用 AWS CLI 或 AWS API 所需的权限。

```
{
    "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": "*"
        }
    ]
```

```
}
```

支持集群管理和数据库连接

以下策略向 IAM 用户授予管理和连接特定 Aurora DSQL 集群的权限。该策略将集群管理和连接操作的范围限定为单个集群 Amazon 资源名称（ARN），同时支持对所有资源执行 `dsql:ListClusters`，因为此操作不支持资源级权限。

此示例使用 `dsql:DbConnectAdmin` 通过 `admin` 角色进行连接。要改为使用自定义数据库角色进行连接，请将 `dsql:DbConnectAdmin` 替换为 `dsql:DbConnect`。有关更多信息，请参阅 Aurora DSQL 的身份验证和授权。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowClusterManagement",
            "Effect": "Allow",
            "Action": [
                "dsql:GetCluster",
                "dsql:UpdateCluster",
                "dsql:DeleteCluster",
                "dsql:DbConnectAdmin",
                "dsql:TagResource",
                "dsql:ListTagsForResource",
                "dsql:UntagResource"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/my-cluster-id"
        },
        {
            "Sid": "AllowListClusters",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        }
    ]
}
```

Aurora DSQL 基于标签的资源访问权限

您可以在基于身份的策略中使用条件，以便基于标签控制对 Aurora DSQL 资源的访问权限。以下示例显示了如何可以创建支持查看集群的策略。但是，仅当集群标签 Owner 的值为该用户的用户名时，此策略才授予权限。此策略还授予在控制台上完成此操作的必要权限。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListClustersInConsole",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        },
        {
            "Sid": "ViewClusterIfOwner",
            "Effect": "Allow",
            "Action": "dsql:GetCluster",
            "Resource": "arn:aws:dsql:*:*:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

您可以将该策略附加到您账户中的 IAM 用户。如果名为 richard-roe 的用户尝试查看 Aurora DSQL 集群，则必须将集群标记为 Owner=richard-roe 或 owner=richard-roe。否则，IAM 会拒绝访问。条件标签键 Owner 匹配 Owner 和 owner，因为条件键名称不区分大小写。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。

以下策略支持用户创建集群，前提是他们将自己的用户名作为 Owner 来标记集群。它还支持仅在用户已拥有的集群上进行标记。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateTaggedCluster",
            "Effect": "Allow",
            "Action": "dsql:CreateCluster",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "${aws:username}"
                }
            }
        },
        {
            "Sid": "AllowTagOwnedClusters",
            "Effect": "Allow",
            "Action": "dsql:TagResource",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

对 Amazon Aurora DSQL 身份和访问权限问题进行故障排除

使用以下信息有助于您诊断和修复在使用 Aurora DSQL 和 IAM 时可能遇到的常见问题。

主题

- 我无权在 Aurora DSQL 中执行操作
- 我无权执行 iam:PassRole
- 我希望支持我的 AWS 账户以外的人员访问我的 Aurora DSQL 资源

我无权在 Aurora DSQL 中执行操作

如果您收到错误提示，指明您无权执行某个操作，则必须更新策略以允许执行该操作。

当 `mateojackson` 尝试使用控制台来查看有关 *`my-dsql-cluster`* 资源的详细信息，但不具有 *`GetCluster`* 权限时，会发生以下示例错误。

```
User: iam:::user/mateojackson is not authorized to perform: GetCluster on resource: my-
dsql-cluster
```

在此情况下，必须更新 `mateojackson` 用户的策略，以允许使用 *`GetCluster`* 操作访问 *`my-dsql-cluster`* 资源。

如果您需要帮助，请联系 管理员。您的管理员是提供登录凭证的人。

我无权执行 iam:PassRole

如果您收到一个错误，表明您无权执行 `iam:PassRole` 操作，则必须更新策略以支持您将角色传递给 Aurora DSQL。

有些 AWS 服务 允许将现有角色传递到该服务，而不是创建新服务角色或服务关联角色。为此，您必须具有将角色传递到服务的权限。

当名为 `marymajor` 的 IAM 用户尝试使用控制台在 Aurora DSQL 中执行操作时，会发生以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 `iam:PassRole` 操作。

如果您需要帮助，请联系 AWS 管理员。您的管理员是提供登录凭证的人。

我希望支持我的 AWS 账户以外的人员访问我的 Aurora DSQL 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以代入角色。对于支持基于资源的策略或访问控制列表（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解 Aurora DSQL 是否支持这些功能，请参阅 Amazon Aurora DSQL 如何与 IAM 协同工作。
- 要了解如何为您拥有的 AWS 账户 中的资源提供访问权限，请参阅《IAM 用户指南》中的为您拥有的另一个 AWS 账户 中的 IAM 用户提供访问权限。
- 要了解如何为第三方 AWS 账户 提供您的资源的访问权限，请参阅《IAM 用户指南》中的为第三方拥有的 AWS 账户 提供访问权限。
- 要了解如何通过身份联合验证提供访问权限，请参阅《IAM 用户指南》中的为经过外部身份验证的用户（身份联合验证）提供访问权限。
- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

Aurora DSQL 的基于资源的策略

使用 Aurora DSQL 的基于资源的策略，通过直接附加到集群资源的 JSON 策略文档来限制或授予对集群的访问权限。这些策略可以精细地控制谁可以在什么条件下访问您的集群。

默认情况下，Aurora DSQL 集群可通过公共互联网进行访问，并将 IAM 身份验证作为主要的安全控制措施。基于资源的策略可让您添加访问限制，特别是屏蔽来自公共互联网的访问。

基于资源的策略与 IAM 基于身份的策略协同工作。AWS 评估这两种类型的策略，以确定对集群的任何访问请求的最终权限。默认情况下，可以在账户内访问 Aurora DSQL 集群。如果 IAM 用户或角色拥有 Aurora DSQL 权限，则他们可以在未附加任何基于资源的策略的情况下访问集群。

Note

对基于资源的策略所做的更改会最终一致，并且通常在一分钟内生效。

有关基于身份的策略与基于资源的策略之间的差别的更多信息，请参阅《IAM 用户指南》中的基于身份的策略和基于资源的策略。

何时使用基于资源的策略

基于资源的策略在以下情况下特别有用：

- 基于网络的访问控制：根据从中发出请求的 VPC 或 IP 地址来限制访问，或者完全阻止公共互联网访问。使用诸如 `aws:SourceVpc` 和 `aws:SourceIp` 之类的条件键来控制网络访问。
- 多个团队或应用程序：为多个团队或应用程序授予对同一个集群的访问权限。您可以在集群上定义一次访问规则，而不是为每个主体管理单独的 IAM 策略。

- 复杂的条件访问：根据网络属性、请求上下文和用户属性等多种因素控制访问权限。您可以在单个策略中组合多个条件。
- 集中式安全治理：使集群所有者能够使用与现有安全实践集成的熟悉的 AWS 策略语法来控制访问权限。

> **Note**
>
> Aurora DSQL 基于资源的策略尚不支持跨账户访问，但将在未来的版本中提供。

当有人尝试连接到 Aurora DSQL 集群时，AWS 会在授权上下文中评估基于资源的策略以及任何相关的 IAM 策略，以确定是应允许还是拒绝该请求。

基于资源的策略可以向与集群相同的 AWS 账户中的主体授予访问权限。对于多区域集群，每个区域集群都有自己的基于资源的策略，支持在需要时进行区域特定的访问控制。

> **Note**
>
> 条件上下文键可能因区域而异（例如 VPC ID）。

主题

- 使用基于资源的策略创建集群
- 为集群添加和编辑基于资源的策略
- 查看基于资源的策略
- 移除基于资源的策略
- 常见的基于资源的策略示例
- 在 Aurora DSQL 中使用基于资源的策略屏蔽公共访问权限
- Aurora DSQL API 操作和基于资源的策略

使用基于资源的策略创建集群

您可以在创建新集群时附加基于资源的策略，以确保从一开始就实施访问控制。每个集群都可以有一个直接附加到该集群的内联策略。

AWS 管理控制台

在创建集群期间添加基于资源的策略

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 选择创建集群。
3. 根据需要配置集群名称、标签和多区域设置。
4. 在集群设置部分，找到基于资源的策略选项。
5. 开启添加基于资源的策略。
6. 在 JSON 编辑器中输入您的策略文档。例如，要屏蔽公共互联网访问，请执行以下操作：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

7. 您可以使用编辑语句或添加新语句来构建您的策略。
8. 完成剩余的集群配置，然后选择创建集群。

AWS CLI

创建集群时使用 `--policy` 参数来附加内联策略：

```
aws dsql create-cluster --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
        }
    }]
}'
```

AWS SDK

Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
        }
    }]
}

response = client.create_cluster(
    policy=json.dumps(policy)
)

print(f"Cluster created: {response['identifier']}")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
    }
  }]
}
""";

CreateClusterRequest request = CreateClusterRequest.builder()
    .policy(policy)
    .build();

CreateClusterResponse response = client.createCluster(request);
System.out.println("Cluster created: " + response.identifier());
```

为集群添加和编辑基于资源的策略

AWS 管理控制台

向现有集群添加基于资源的策略

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 从集群列表中选择集群以打开集群详细信息页面。
3. 选择权限选项卡。

4. 在基于资源的策略部分中，选择添加策略。
5. 在 JSON 编辑器中输入您的策略文档。您可以使用编辑语句或添加新语句来构建您的策略。
6. 选择添加策略。

编辑现有的基于资源的策略

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 从集群列表中选择集群以打开集群详细信息页面。
3. 选择权限选项卡。
4. 在基于资源的策略部分中，选择编辑。
5. 在 JSON 编辑器中修改策略文档。您可以使用编辑语句或添加新语句来更新您的策略。
6. 选择保存更改。

AWS CLI

使用 `put-cluster-policy` 命令在集群上附加新策略或更新现有策略：

```
aws dsql put-cluster-policy --identifier your_cluster_id --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": { "aws:SourceVpc": "true" }
        }
    }]
}'
```

AWS SDK

Python

```
import boto3
import json
```

```
client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
    }]
}

response = client.put_cluster_policy(
    identifier='your_cluster_id',
    policy=json.dumps(policy)
)
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.PutClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "Null": {"aws:SourceVpc": "true"}
    }
  }]
}
```

```
""";

PutClusterPolicyRequest request = PutClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .policy(policy)
    .build();

client.putClusterPolicy(request);
```

查看基于资源的策略

您可以查看附加到集群的基于资源的策略，以了解当前已实施的访问控制。

AWS 管理控制台

查看基于资源的策略

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。
2. 从集群列表中选择集群以打开集群详细信息页面。
3. 选择权限选项卡。
4. 在基于资源的策略部分查看附加的策略。

AWS CLI

使用 `get-cluster-policy` 命令来查看集群的基于资源的策略：

```
aws dsql get-cluster-policy --identifier your_cluster_id
```

AWS SDK

Python

```

import boto3
import json

client = boto3.client('dsql')
```

```
response = client.get_cluster_policy(
    identifier='your_cluster_id'
)

# Parse and pretty-print the policy
policy = json.loads(response['policy'])
print(json.dumps(policy, indent=2))
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyResponse;

DsqlClient client = DsqlClient.create();

GetClusterPolicyRequest request = GetClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

GetClusterPolicyResponse response = client.getClusterPolicy(request);
System.out.println("Policy: " + response.policy());
```

移除基于资源的策略

您可以从集群中移除基于资源的策略以更改访问控制。

Important

当您从集群中移除所有基于资源的策略时，访问将完全由 IAM 基于身份的策略控制。

AWS 管理控制台

移除基于资源的策略

1. 登录 AWS 管理控制台并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql/。

2. 从集群列表中选择集群以打开集群详细信息页面。
3. 选择权限选项卡。
4. 在基于资源的策略部分中，选择删除。
5. 在确认对话框中，键入 **confirm** 以确认删除。
6. 选择删除。

AWS CLI

使用 `delete-cluster-policy` 命令从集群中移除策略。

```
aws dsql delete-cluster-policy --identifier your_cluster_id
```

AWS SDK

Python

```
import boto3

client = boto3.client('dsql')

response = client.delete_cluster_policy(
    identifier='your_cluster_id'
)

print("Policy deleted successfully")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

DeleteClusterPolicyRequest request = DeleteClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

client.deleteClusterPolicy(request);
System.out.println("Policy deleted successfully");
```

常见的基于资源的策略示例

这些示例显示了用于控制对 Aurora DSQL 集群的访问权限的常见模式。您可以组合和修改这些模式以满足您的特定访问要求。

屏蔽公共互联网访问

此策略屏蔽从公共互联网（非 VPC）连接到您的 Aurora DSQL 集群。该策略未指定客户可以从哪些 VPC 进行连接，只规定他们必须从 VPC 进行连接。要限制对特定 VPC 的访问，请将 `aws:SourceVpc` 与 `StringEquals` 条件运算符结合使用。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

> **Note**
>
> 此示例仅使用 `aws:SourceVpc` 来检查 VPC 连接。`aws:VpcSourceIp` 和 `aws:SourceVpce` 条件键提供了更高的粒度，但对于仅限 VPC 的基本访问控制来说并不是必需的。

要为特定角色提供例外情况，请改用以下策略：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyAccessFromOutsideVPC",
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        },
        "StringNotEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:role/ExceptionRole",
            "arn:aws:iam::123456789012:role/AnotherExceptionRole"
          ]
        }
      }
    }
  ]
}
```

限制对 AWS 组织的访问权限

此策略将访问权限限制为 AWS 组织内部的主体：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
```

```
        "dsql:DbConnectAdmin"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalOrgID": "o-exampleorgid"
        }
      }
    }
  ]
}
```

限制对组织单元的访问权限

此策略将访问权限限制为 AWS 组织中特定组织单元（OU）内的主体，提供比组织范围的访问权限更精细的控制：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotLike": {
          "aws:PrincipalOrgPaths": "o-exampleorgid/r-examplerootid/ou-exampleouid/*"
        }
      }
    }
  ]
}
```

多区域集群策略

对于多区域集群，每个区域集群都维护其自己的资源策略，支持特定于区域的控制。以下是每个区域不同策略的示例：

us-east-1 策略：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-east1-id"
        },
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

us-east-2 策略：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
```

```
          "dsql:DbConnect"
        ],
        "Condition": {
          "StringEquals": {
            "aws:SourceVpc": "vpc-east2-id"
          }
        }
      }
    ]
}
```

> **Note**
>
> 条件上下文键可能因 AWS 区域而异（例如 VPC ID）。

在 Aurora DSQL 中使用基于资源的策略屏蔽公共访问权限

屏蔽公共访问权限（BPA）功能可识别并防止附加基于资源的策略，这些策略授予跨您的 AWS 账户对 Aurora DSQL 集群进行公共访问的权限。使用 BPA，您可以阻止对 Aurora DSQL 资源的公共访问。BPA 会在创建或修改基于资源的策略期间执行检查，并通过 Aurora DSQL 协助改善您的安全状况。

BPA 使用自动推理来分析您的基于资源的策略授予的访问权限，并且如果在管理基于资源的策略时发现此类权限，为您发送提醒。该分析可验证策略中使用的所有基于资源的策略语句、操作和条件键集的访问权限。

> **Important**
>
> BPA 通过直接附加到 Aurora DSQL 资源（例如集群）的基于资源的策略来阻止授予公共访问权限，从而有助于保护您的资源。除了使用 BPA 之外，还要仔细检查以下策略，来确认它们不授予公有访问权限：
>
> - 附加到关联 AWS 主体（例如 IAM 角色）的基于身份的策略
> - 附加到关联 AWS 资源 [例如 AWS Key Management Service（KMS）密钥] 的基于资源的策略

您必须确保主体不包含 * 条目，或者指定的条件键之一限制主体对资源的访问。如果基于资源的策略授予跨 AWS 对集群进行公共访问的权限，则 Aurora DSQL 将阻止您创建或修改策略，直到策略中的规范得到更正并被视为非公开为止。

您可以通过在 Principal 块内指定一个或多个主体来将策略设为非公有。以下基于资源的策略示例通过指定两个主体来阻止公有访问。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id"
}
```

通过指定特定条件键来限制访问的策略也不被视为公有策略。除了评估基于资源的策略中指定的主体外，还使用以下可信条件键来完成就非公有访问对基于资源的策略的评估：

- 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`

此外，要将基于资源的策略设为非公有策略，Amazon 资源名称（ARN）和字符串键的值不得包含通配符或变量。如果您的基于资源的策略使用 `aws:PrincipalIsAWSService` 键，则必须确保已将键值设置为 true。

以下策略限制指定账户中的用户 `Ben` 的访问权限。该条件使 `Principal` 受限制且不被视为公有。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/Ben"
    }
  }
}
```

以下基于非公有资源的策略示例使用 `StringEquals` 运算符限制 `sourceVPC`。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dsql:*",
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
```

```
      }
    }
  ]
}
```

Aurora DSQL API 操作和基于资源的策略

Aurora DSQL 中基于资源的策略控制对特定 API 操作的访问权限。以下各节列出了按类别整理的所有 Aurora DSQL API 操作，并指明了哪些操作支持基于资源的策略。

支持 RBP 列指示在将策略附加到集群时，API 操作是否需要接受基于资源的策略评估。

标签 API

API 操作	说明	支持 RBP
ListTagsForResource	列出 Aurora DSQL 资源的标签	是
TagResource	向 Aurora DSQL 资源添加标签	是
UntagResource	从 Aurora DSQL 资源移除标签	是

集群管理 API

API 操作	说明	支持 RBP
CreateCluster：	创建新集群	否
DeleteCluster	删除集群	是
GetCluster	检索有关集群的信息	是
GetVpcEndpointServiceName	检索集群的 VPC 端点服务名称	是
ListClusters	列出您账户中的集群	否
UpdateCluster	更新集群的配置	是

多区域属性 API

API 操作	说明	支持 RBP
AddPeerCluster	将对等集群添加到多区域配置中	是
PutMultiRegionProperties	为集群设置多区域属性	是
PutWitnessRegion	为多区域集群设置见证区域	是

基于资源的策略 API

API 操作	说明	支持 RBP
DeleteClusterPolicy	从集群中删除基于资源的策略	是
GetClusterPolicy	检索集群的基于资源的策略	是
PutClusterPolicy	创建或更新集群的基于资源的策略	是

AWS Fault Injection Service API

API 操作	说明	支持 RBP
InjectError	注入错误以进行故障注入测试	否

备份与还原 API

API 操作	说明	支持 RBP
GetBackupJob	检索有关备份作业的信息	否
GetRestoreJob	检索有关还原作业的信息	否
StartBackupJob	启动集群的备份作业	是

API 操作	说明	支持 RBP
StartRestoreJob	从备份启动还原作业	否
StopBackupJob	停止正在运行的备份作业	否
StopRestoreJob	停止正在运行的还原作业	否

使用 Aurora DSQL 中的服务相关角色

Aurora DSQL 使用 AWS Identity and Access Management（IAM）服务相关角色。服务相关角色是一种独特类型的 IAM 角色，它与 Aurora DSQL 直接关联。服务相关角色由 Aurora DSQL 预定义，并包含服务代表 Aurora DSQL 集群调用 AWS 服务所需的所有权限。

服务相关角色可让设置过程变得更为容易，因为您不必手动添加使用 Aurora DSQL 所需的权限。创建集群时，Aurora DSQL 将自动为您创建服务相关角色。只有在删除所有集群之后，才可删除服务相关角色。这将保护您的 Aurora DSQL 资源，因为您不会无意中移除访问资源所需的权限。

有关支持服务相关角色的其它服务的信息，请参阅使用 IAM 的 AWS 服务，并查找服务相关角色列中显示为是的服务。请选择是与查看该服务的服务关联角色文档的链接。

服务相关角色适用于所有受支持的 Aurora DSQL 区域。

Aurora DSQL 的服务相关角色权限

Aurora DSQL 使用名为 `AWSServiceRoleForAuroraDsql` 的服务相关角色：支持 Amazon Aurora DSQL 代表您创建和管理 AWS 资源。此服务相关角色附加到以下托管式策略：AuroraDsqlServiceLinkedRolePolicy。

> **Note**
>
> 您必须配置权限，允许 IAM 实体（如用户、组或角色）创建、编辑或删除服务关联角色。您可能会遇到以下错误消息：`You don't have the permissions to create an Amazon Aurora DSQL service-linked role`。如果您看到此消息，请确保您已启用以下权限：
>
> JSON
>
> ```
> {
> ```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateDsqlServiceLinkedRole",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "dsql.amazonaws.com"
                }
            }
        }
    ]
}
```

有关更多信息，请参阅服务相关角色权限。

创建服务相关角色

您无需手动创建 AuroraDSQLServiceLinkedRolePolicy 服务相关角色。Aurora DSQL 为您创建此服务相关角色。如果已从您的账户中删除 AuroraDSQLServiceLinkedRolePolicy 服务相关角色，Aurora DSQL 将在您创建新的 Aurora DSQL 集群时创建该角色。

编辑服务相关角色

Aurora DSQL 不支持您编辑 AuroraDSQLServiceLinkedRolePolicy 服务相关角色。在创建服务相关角色后，您将无法更改角色的名称，因为可能有多种实体引用该角色。不过，您可以使用 IAM 控制台、AWS Command Line Interface (AWS CLI) 或 IAM API 编辑角色描述。

删除服务相关角色

如果不再需要使用某个需要服务关联角色的功能或服务，我们建议您删除该角色。这样，您就没有未受主动监控或维护的未使用实体。

在删除账户的服务相关角色之前，必须删除该账户中的所有集群。

您可以使用 IAM 控制台、AWS CLI 或 IAM API 删除服务相关角色。有关更多信息，请参阅《IAM 用户指南》中的创建服务相关角色。

Aurora DSQL 服务相关角色的受支持区域

Aurora DSQL 支持在该服务可用的所有区域中使用服务相关角色。有关更多信息，请参阅 AWS 区域和端点。

将 IAM 条件键与 Amazon Aurora DSQL 结合使用

在 Aurora DSQL 中授予权限时，可以指定确定权限策略如何生效的条件。以下示例说明了如何在 Aurora DSQL 权限策略中使用条件键。

示例 1：授予在特定 AWS 区域中创建集群的权限

以下策略授予在美国东部（弗吉尼亚州北部）和美国东部（俄亥俄州）区域中创建集群的权限。此策略使用资源 ARN 来限制支持的区域，因此，仅当在该策略的 Resource 部分中指定该 ARN 时，Aurora DSQL 才能创建集群。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": ["dsql:CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

示例 2：授予在特定 AWS 区域中创建多区域集群的权限

以下策略授予在美国东部（弗吉尼亚州北部）和美国东部（俄亥俄州）区域中创建多区域集群的权限。此策略使用资源 ARN 来限制支持的区域，因此，仅当在该策略的 Resource 部分中指定此 ARN 时，Aurora DSQL 才能创建多区域集群。请注意，创建多区域集群还需要在每个指定的区域中具有 PutMultiRegionProperties、PutWitnessRegion 和 AddPeerCluster 权限。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "dsql:CreateCluster",
          "dsql:PutMultiRegionProperties",
          "dsql:PutWitnessRegion",
          "dsql:AddPeerCluster"
        ],
        "Resource": [
           "arn:aws:dsql:us-east-1:123456789012:cluster/*",
           "arn:aws:dsql:us-east-2:123456789012:cluster/*"
        ]
      }
    ]
}
```

示例 3：授予创建具有特定见证区域的多区域集群的权限

以下策略使用 Aurora DSQL `dsql:WitnessRegion` 条件键，并让用户创建在美国西部（俄勒冈州）具有见证区域的多区域集群。如果您未指定 `dsql:WitnessRegion` 条件，则可以使用任何区域作为见证区域。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dsql:CreateCluster",
                "dsql:PutMultiRegionProperties",
                "dsql:AddPeerCluster"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "dsql:PutWitnessRegion"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": [
                        "us-west-2"
                    ]
                }
            }
        }
    ]
}
```

Amazon Aurora DSQL 中的事件响应

AWS 非常重视安全性。作为 AWS 云责任共担模式的一部分，AWS 负责管理数据中心、网络和软件架构，以满足对安全最为敏感的组织的要求。AWS 负责与 Amazon Aurora DSQL 服务本身相关的任何事件响应。此外，作为 AWS 客户，您也有责任维护云端的安全。这意味着，您可以控制从您有权使用的 AWS 工具和功能中选择并实施的安全措施。此外，您还需要负责您在责任共担模式中的事件响应部分。

通过建立符合云端运行应用程序目标的安全基准，您可以检测出可以响应的偏差。为了帮助您了解事件响应和您的选择对企业目标的影响，建议您查看以下资源：

- AWS 安全事件响应指南
- AWS Best Practices for Security, Identity, and Compliance
- 《Security Perspective of the AWS Cloud Adoption Framework (CAF)》白皮书

Amazon GuardDuty 是一项托管式威胁检测服务，可以持续监控恶意或未经授权的行为，来协助客户保护 AWS 账户和工作负载，并识别潜在的可疑活动，防止其升级为事件。Amazon GuardDuty 可以监控异常的 API 调用或可能未经授权的部署等活动，这些活动表明账户或资源可能遭到破坏或者有恶意行为者正在进行侦察。例如，Amazon GuardDuty 能够检测到 Amazon Aurora DSQL API 中的可疑活动，例如用户从新位置登录并创建新集群。

Amazon Aurora DSQL 的合规性验证

要了解某个 AWS 服务是否在特定合规性计划范围内，请参阅按合规性计划划分的范围内的 AWS 服务，然后选择您感兴趣的合规性计划。有关常规信息，请参阅 AWS 合规性计划、、。

您可以使用 AWS Artifact 下载第三方审计报告。有关更多信息，请参阅在 AWS Artifact 中下载报告。

您在使用 AWS 服务 时的合规性责任由您的数据的敏感性、您公司的合规性目标以及适用的法律法规决定。有关您在使用 AWS 服务 时的合规责任的更多信息，请参阅 AWS 安全性文档。

Amazon Aurora DSQL 中的韧性

AWS 全球基础设施围绕 AWS 区域和可用区（AZ）构建。AWS 区域提供多个在物理上独立且隔离的可用区，这些可用区与延迟低、吞吐量高且冗余性高的网络连接在一起。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错能力和可扩展性。Aurora DSQL 的设计使您可以利用 AWS 区域基础设施，同时提供最高的数据库可用性。默认情况下，Aurora DSQL 中的单区域集群具有多可用区可用性，可以容忍可能影响对完整可用区进行访问的重大组件故障和基础设施中断。多区域集群提供了多可用区韧性的所有优势，同时仍能提供强一致性数据库可用性，即使在应用程序客户端无法访问 AWS 区域的情况下也是如此。

有关 AWS 区域 和可用区的更多信息，请参阅 AWS 全球基础设施。

除了 AWS 全球基础设施之外，Aurora DSQL 还提供了多种功能，有助于支持您的数据韧性和备份需求。

备份与还原

Aurora DSQL 支持使用 AWS Backup 控制台进行备份和还原。您可以对单区域和多区域集群执行完整备份和还原。有关更多信息，请参阅 Amazon Aurora DSQL 的备份和还原。

复制

根据设计，Aurora DSQL 将所有写入事务提交到分布式事务日志，并将所有提交的日志数据同步复制到三个可用区中的用户存储副本。多区域集群在读取区域和写入区域之间提供完整的跨区域复制功能。

指定的见证区域支持仅事务日志写入，并且不占用任何存储空间。见证区域没有端点。这意味着见证区域仅存储加密的事务日志，无需管理或配置，并且用户无法访问。如果见证区域受损，则集群可用性不会受到影响。在见证区域恢复之前，写入事务的延迟可能会略有增加。

Aurora DSQL 事务日志和用户存储空间分布在各处，所有数据都作为单个逻辑卷呈现给 Aurora DSQL 查询处理器。Aurora DSQL 根据数据库主键范围和访问模式，自动拆分、合并和复制数据。Aurora DSQL 根据读取访问频率，自动纵向扩展和缩减只读副本。

集群存储副本分布在多租户存储实例集之间。如果组件或可用区受损，Aurora DSQL 会自动将访问重定向到依然正常运行的组件，并异步修复缺失的副本。一旦 Aurora DSQL 修复了受损副本，Aurora DSQL 就会自动将其添加回存储仲裁，并使其可供您的集群使用。

高可用性

默认情况下，Aurora DSQL 中的单区域和多区域集群处于主动-主动状态，您无需手动预置、配置或重新配置任何集群。Aurora DSQL 可实现集群恢复的完全自动化，从而无需进行传统的主要-辅助失效转移操作。复制始终是同步的，并在多个可用区中完成，因此，在故障恢复期间，不会由于复制滞后或失效转移到异步辅助数据库而导致丢失数据的风险。

单区域集群提供多可用区冗余端点，该端点可自动实现跨三个可用区的并发访问，并具有很强的数据一致性。这意味着，这三个可用区中任何一个可用区上的用户存储副本始终向一个或多个读取器返回相同的结果，并且始终可以接收写入。可以跨 Aurora DSQL 多区域集群的所有区域提供这种强一致性和多可用区韧性。这意味着多区域集群提供了两个强一致性区域端点，因此客户端可以不加区分地对任一区域进行读取或写入，提交时复制滞后为零。

Aurora DSQL 为单区域集群提供 99.99% 的可用性，并为多区域集群提供 99.999% 的可用性。

故障注入测试

Amazon Aurora DSQL 与 AWS Fault Injection Service（AWS FIS）集成，后者是完全托管式服务，用于运行受控的故障注入实验来提高应用程序的韧性。通过使用 AWS FIS，您可以：

- 创建定义特定故障场景的实验模板
- 注入故障（集群连接错误率提升），来验证应用程序错误处理和恢复机制
- 测试多区域应用程序行为，验证当一个 AWS 区域出现高连接错误率时，流量在 AWS 区域之间的转移情况

例如，在横跨美国东部（弗吉尼亚州北部）和美国东部（俄亥俄州）的多区域集群中，您可以在美国东部（俄亥俄州）运行实验来进行故障测试，同时美国东部（弗吉尼亚州北部）继续正常运行。这种受控测试有助于确定并解决潜在的问题，以防这些问题影响到生产工作负载。

有关 AWS FIS 所支持操作的完整列表，请参阅《AWS FIS User Guide》中的 Action targets。

有关在 AWS FIS 中可用的 Amazon Aurora DSQL 操作的信息，请参阅《AWS FIS User Guide》中的 Aurora DSQL actions reference。

要开始运行故障注入实验，请参阅《AWS FIS 用户指南》中的 Planning your AWS FIS experiments。

Amazon Aurora DSQL 中的基础设施安全性

作为一项托管式服务，Amazon Aurora DSQL 受到 AWS 全局网络安全程序保护，请参阅安全、身份与合规性的最佳实践。

您可以使用 AWS 发布的 API 调用通过网络访问 Aurora DSQL。客户端必须支持传输层安全性协议（TLS）1.2 或更高版本。客户端还必须支持具有完全向前保密 (PFS) 的密码套件，例如 DHE (Ephemeral Diffie-Hellman) 或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 主体关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

使用 AWS PrivateLink 管理和连接到 Amazon Aurora DSQL 集群

借助适用于 Amazon Aurora DSQL 的 AWS PrivateLink，您可以在 Amazon Virtual Private Cloud 中预置接口 Amazon VPC 端点（接口端点）。这些端点可从位于本地（通过 Amazon VPC 及 Direct Connect）或其它 AWS 区域（通过 Amazon VPC 对等连接）中的应用程序直接访问。使用 AWS PrivateLink 和接口端点，您可以简化应用程序与 Aurora DSQL 之间的私有网络连接。

Amazon VPC 中的应用程序无需公有 IP 地址，即可使用 Amazon VPC 接口端点访问 Aurora DSQL。

接口端点由一个或多个弹性网络接口（ENI）表示，这些接口是从 Amazon VPC 中的子网分配的私有 IP 地址。通过接口端点向 Aurora DSQL 发出的请求仍留在 AWS 网络上。有关如何将 Amazon VPC 与本地网络连接的更多信息，请参阅 Direct Connect User Guide 和《AWS Site-to-Site VPN VPN User Guide》。

有关接口端点的一般信息，请参阅《AWS PrivateLink User Guide》中的 Access an AWS service using an interface Amazon VPC endpoint。

适用于 Aurora DSQL 的 Amazon VPC 端点类型

Aurora DSQL 需要两种不同类型的 AWS PrivateLink 端点。

1. 管理端点：此端点用于 Aurora DSQL 集群上的管理操作，例如 `get`、`create`、`update`、`delete` 和 `list`。请参阅使用 AWS PrivateLink 管理 Aurora DSQL 集群。

2. 连接端点：此端点用于通过 PostgreSQL 客户端连接到 Aurora DSQL 集群。请参阅使用 AWS PrivateLink 连接到 Aurora DSQL 集群。

使用适用于 Aurora DSQL 的 AWS PrivateLink 时的注意事项

Amazon VPC 注意事项面向适用于 Aurora DSQL 的 AWS PrivateLink。有关更多信息，请参阅《AWS PrivateLink 指南》中的使用接口 VPC 端点访问 AWS 服务和 AWS PrivateLink 配额。

使用 AWS PrivateLink 管理 Aurora DSQL 集群

您可以使用 AWS Command Line Interface或 AWS 软件开发工具包（SDK）通过 Aurora DSQL 接口端点管理 Aurora DSQL 集群。

创建 Amazon VPC 端点

要创建 Amazon VPC 接口端点，请参阅《AWS PrivateLink Guide》中的 Create an Amazon VPC endpoint。

```
aws ec2 create-vpc-endpoint \
--region region \
--service-name com.amazonaws.region.dsql \
--vpc-id your-vpc-id \
--subnet-ids your-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id \
```

要对 Aurora DSQL API 请求使用默认区域 DNS 名称，请在创建 Aurora DSQL 接口端点时不要禁用私有 DNS。启用私有 DNS 后，从 Amazon VPC 内向 Aurora DSQL 服务发出的请求将自动解析到 Amazon VPC 端点的私有 IP 地址，而不是公有 DNS 名称。启用私有 DNS 后，在 Amazon VPC 内发出的 Aurora DSQL 请求将自动解析到 Amazon VPC 端点。

如果未启用私有 DNS，请使用 `--region` 和 `--endpoint-url` 参数以及 AWS CLI 命令，通过 Aurora DSQL 接口端点来管理 Aurora DSQL 集群。

使用端点 URL 列出集群

在以下示例中，将 AWS 区域 `us-east-1` 和 Amazon VPC 端点 ID `vpce-1a2b3c4d-5e6f.dsql.us-east-1.vpce.amazonaws.com` 的 DNS 名称替换为您自己的信息。

```
aws dsql --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsql.us-
east-1.vpce.amazonaws.com list-clusters
```

API 操作

有关在 Aurora DSQL 中管理资源的文档，请参阅 Aurora DSQL API 参考。

管理端点策略

通过全面测试和配置 Amazon VPC 端点策略，有助于确保 Aurora DSQL 集群安全、合规，并符合组织的特定访问控制和治理要求。

示例：完整的 Aurora DSQL 访问策略

以下策略将授予通过指定的 Amazon VPC 端点访问所有 Amazon DSQL 操作和资源的完全访问权限。

```
aws ec2 modify-vpc-endpoint \
    --vpc-endpoint-id vpce-xxxxxxxxxxxxxxxxx \
    --region region \
    --policy-document '{
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": "*",
          "Action": "dsql:*",
          "Resource": "*"
        }
      ]
    }'
```

示例：受限的 Aurora DSQL 访问策略

以下策略仅允许这些 Aurora DSQL 操作。

- `CreateCluster`
- `GetCluster`
- `ListClusters`

所有其它 Aurora DSQL 操作均会遭拒绝。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "*"
    }
  ]
}
```

使用 AWS PrivateLink 连接到 Aurora DSQL 集群

AWS PrivateLink 端点设置完毕并处于活动状态后，就可以使用 PostgreSQL 客户端连接到 Aurora DSQL 集群了。以下连接说明概述了为通过 AWS PrivateLink 端点进行连接而构造正确的主机名的步骤。

设置 AWS PrivateLink 连接端点

步骤 1：获取集群的服务名称

在创建用于连接到集群的 AWS PrivateLink 端点时，首先需要获取特定于集群的服务名称。

AWS CLI

```
aws dsql get-vpc-endpoint-service-name \
--region us-east-1 \
--identifier your-cluster-id
```

响应示例

```
{
    "serviceName": "com.amazonaws.us-east-1.dsql-fnh4"
```

```
}
```

服务名称包含标识符，如示例中的 dsql-fnh4。在构造用于连接到集群的主机名时，也需要此标识符。

AWS SDK for Python (Boto3)

```
import boto3

dsql_client = boto3.client('dsql', region_name='us-east-1')
response = dsql_client.get_vpc_endpoint_service_name(
    identifier='your-cluster-id'
)
service_name = response['serviceName']
print(f"Service Name: {service_name}")
```

AWS SDK for Java 2.x

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetVpcEndpointServiceNameRequest;
import software.amazon.awssdk.services.dsql.model.GetVpcEndpointServiceNameResponse;

String region = "us-east-1";
String clusterId = "your-cluster-id";

DsqlClient dsqlClient = DsqlClient.builder()
    .region(Region.of(region))
    .credentialsProvider(DefaultCredentialsProvider.create())
    .build();

GetVpcEndpointServiceNameResponse response = dsqlClient.getVpcEndpointServiceName(
    GetVpcEndpointServiceNameRequest.builder()
        .identifier(clusterId)
        .build()
);
String serviceName = response.serviceName();
System.out.println("Service Name: " + serviceName);
```

步骤 2：创建 Amazon VPC 端点

使用在上一步中获得的服务名称，创建 Amazon VPC 端点。

> **Important**
>
> 以下连接说明仅适用于在启用私有 DNS 时连接到集群。创建端点时请勿使用 `--no-private-dns-enabled` 标志，因为这会使下面的连接说明无法正常发挥作用。如果您禁用私有 DNS，则需要创建自己的通配符私有 DNS 记录，该记录指向已创建的端点。

AWS CLI

```
aws ec2 create-vpc-endpoint \
    --region us-east-1 \
    --service-name service-name-for-your-cluster \
    --vpc-id your-vpc-id \
    --subnet-ids subnet-id-1 subnet-id-2  \
    --vpc-endpoint-type Interface \
    --security-group-ids security-group-id
```

示例响应：

```
{
    "VpcEndpoint": {
        "VpcEndpointId": "vpce-0123456789abcdef0",
        "VpcEndpointType": "Interface",
        "VpcId": "vpc-0123456789abcdef0",
        "ServiceName": "com.amazonaws.us-east-1.dsql-fnh4",
        "State": "pending",
        "RouteTableIds": [],
        "SubnetIds": [
            "subnet-0123456789abcdef0",
            "subnet-0123456789abcdef1"
        ],
        "Groups": [
            {
                "GroupId": "sg-0123456789abcdef0",
                "GroupName": "default"
            }
        ],
        "PrivateDnsEnabled": true,
        "RequesterManaged": false,
        "NetworkInterfaceIds": [
```

```
            "eni-0123456789abcdef0",
            "eni-0123456789abcdef1"
        ],
        "DnsEntries": [
            {
                "DnsName": "*.dsql-fnh4.us-east-1.vpce.amazonaws.com",
                "HostedZoneId": "Z7HUB22UULQXV"
            }
        ],
        "CreationTimestamp": "2025-01-01T00:00:00.000Z"
    }
}
```

SDK for Python

```
import boto3

ec2_client = boto3.client('ec2', region_name='us-east-1')
response = ec2_client.create_vpc_endpoint(
    VpcEndpointType='Interface',
    VpcId='your-vpc-id',
    ServiceName='com.amazonaws.us-east-1.dsql-fnh4',  # Use the service name from
 previous step
    SubnetIds=[
        'subnet-id-1',
        'subnet-id-2'
    ],
    SecurityGroupIds=[
        'security-group-id'
    ]
)

vpc_endpoint_id = response['VpcEndpoint']['VpcEndpointId']
print(f"VPC Endpoint created with ID: {vpc_endpoint_id}")
```

SDK for Java 2.x

将端点 URL 用于 Aurora DSQL API

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ec2.Ec2Client;
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointRequest;
```

```
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointResponse;
import software.amazon.awssdk.services.ec2.model.VpcEndpointType;

String region = "us-east-1";
String serviceName = "com.amazonaws.us-east-1.dsql-fnh4";  // Use the service name
 from previous step
String vpcId = "your-vpc-id";

Ec2Client ec2Client = Ec2Client.builder()
    .region(Region.of(region))
    .credentialsProvider(DefaultCredentialsProvider.create())
    .build();

CreateVpcEndpointRequest request = CreateVpcEndpointRequest.builder()
    .vpcId(vpcId)
    .serviceName(serviceName)
    .vpcEndpointType(VpcEndpointType.INTERFACE)
    .subnetIds("subnet-id-1", "subnet-id-2")
    .securityGroupIds("security-group-id")
    .build();

CreateVpcEndpointResponse response = ec2Client.createVpcEndpoint(request);
String vpcEndpointId = response.vpcEndpoint().vpcEndpointId();
System.out.println("VPC Endpoint created with ID: " + vpcEndpointId);
```

通过 Direct Connect 或 Amazon VPC 对等连接进行连接时的额外设置

可能需要进行一些额外的设置，才能使用本地设备上的 AWS PrivateLink 连接端点通过 Amazon VPC 对等连接或 Direct Connect 连接到 Aurora DSQL 集群。如果您的应用程序与 AWS PrivateLink 端点在同一个 Amazon VPC 中运行，则无需此设置。上面创建的私有 DNS 条目无法在端点的 Amazon VPC 之外正确解析，但您可以创建自己的私有 DNS 记录，这些记录将解析为您的 AWS PrivateLink 连接端点。

创建指向 AWS PrivateLink 端点的完全限定域名的私有 CNAME DNS 记录。所创建的 DNS 记录的域名应由以下各个部分构造而成：

1. 服务名称中的服务标识符。例如：`dsql-fnh4`
2. 这些区域有：AWS 区域

使用以下格式的域名创建 CNAME DNS 记录：`*.`*`service-identifier`*`.`*`region`*`.on.aws`

域名的格式之所以重要，有两个原因：

1. 使用 `verify-full` SSL 模式时，用于连接到 Aurora DSQL 的主机名必须与 Aurora DSQL 的服务器证书相匹配。这可确保最高级别的连接安全性。
2. Aurora DSQL 使用用于连接到 Aurora DSQL 的主机名的集群 ID 部分来标识连接的集群。

如果无法创建私有 DNS 记录，您仍然可以连接到 Aurora DSQL。请参阅在无私有 DNS 的情况下使用 AWS PrivateLink 端点连接到 Aurora DSQL 集群。

使用 AWS PrivateLink 连接端点连接到 Aurora DSQL 集群

AWS PrivateLink 端点设置完毕并处于活动状态（检查 `State` 是否为 `available`）后，就可以使用 PostgreSQL 客户端连接到 Aurora DSQL 集群了。有关使用 AWS SDK 的说明，您可以按照 Programming with Aurora DSQL 中的指导进行操作。您必须更改集群端点以匹配主机名格式。

构造主机名

通过 AWS PrivateLink 进行连接的主机名不同于公有 DNS 主机名。您需要使用以下组件来构造它。

1. `Your-cluster-id`
2. 服务名称中的服务标识符。例如：`dsql-fnh4`
3. AWS 区域。例如：`us-east-1`

采用以下格式：`*cluster-id*.*service-identifier*.*region*.on.aws`

示例：使用 PostgreSQL 进行连接

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export SERVICE_IDENTIFIER=dsql-fnh4  # This should match the identifier in your service
 name

# Construct the hostname
export HOSTNAME="$CLUSTERID.$SERVICE_IDENTIFIER.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $HOSTNAME)
```

```
# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

在无私有 DNS 的情况下使用 AWS PrivateLink 端点连接到 Aurora DSQL 集群

上面的连接说明依赖于私有 DNS 记录。如果您的应用程序与您的 AWS PrivateLink 端点在同一 Amazon VPC 中运行，则会为您创建 DNS 记录。或者，如果您通过 Amazon VPC 对等连接或 Direct Connect 从本地设备进行连接，则可以创建自己的私有 DNS 记录。但是，由于您的安全团队施加了网络限制，因此并非始终可以设置 DNS 记录。如果您的应用程序必须使用 Direct Connect 或从对等连接的 Amazon VPC 进行连接，并且无法设置 DNS 记录，那么您仍然可以连接到 Aurora DSQL。

Aurora DSQL 使用主机名的集群 ID 部分来标识连接的集群，但是如果无法设置 DNS 记录，Aurora DSQL 支持使用 `amzn-cluster-id` 连接选项指定目标集群。使用此选项，可以在连接时使用 AWS PrivateLink 端点的完全限定域名作为主机名。

> **Important**
>
> 使用 AWS PrivateLink 端点的完全限定域名或 IP 地址进行连接时，不支持 `verify-full` SSL 模式。因此，最好设置私有 DNS。

示例：使用 PostgreSQL 指定集群 ID 连接选项

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export HOSTNAME=vpce-04037adb76c111221-d849uc2p.dsql-fnh4.us-east-1.vpce.amazonaws.com
 # This should match your endpoint's fully-qualified domain name

# Construct the hostname used to generate the authentication token
export AUTH_HOSTNAME="$CLUSTERID.dsql.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $AUTH_HOSTNAME)

# Specify the amzn-cluster-id connection option
export PGOPTIONS="-c amzn-cluster-id=$CLUSTERID"

# Connect using psql
```

```
psql -d postgres -h $HOSTNAME -U admin
```

排查 AWS PrivateLink 问题

常见问题和解决方案

下表列出了将 AWS PrivateLink 与 Aurora DSQL 结合使用时相关的常见问题和解决方案。

问题	可能的原因	解决方案
连接超时	安全组配置不正确	使用 Amazon VPC Reachability Analyzer 可确保网络设置支持端口 5432 上的流量。
DNS 解析失败	未启用私有 DNS	确认 Amazon VPC 端点是在启用私有 DNS 的情况下创建的。
身份验证失败	凭证不正确或令牌过期	生成新的身份验证令牌并验证用户名。
找不到服务名称	集群 ID 不正确	在获取服务名称时，请仔细检查您的集群 ID 和 AWS 区域。

相关资源

有关更多信息，请参阅以下资源：

- Amazon Aurora DSQL 用户指南
- AWS PrivateLink 文档
- Access AWS services through AWS PrivateLink

Amazon Aurora DSQL 中的配置和漏洞分析

AWS 负责处理基本安全任务，如来宾操作系统（OS）和数据库补丁、防火墙配置和灾难恢复等。这些流程已通过相应第三方审核和认证。有关更多详细信息，请参阅以下资源：

- 责任共担模式
- 亚马逊云科技：安全流程概览（白皮书）

防止跨服务混淆座席

混淆代理问题是一个安全性问题，即不具有操作执行权限的实体可能会迫使具有更高权限的实体执行该操作。在 AWS 中，跨服务模拟可能会导致混淆代理问题。一个服务（呼叫服务）调用另一项服务（所谓的服务）时，可能会发生跨服务模拟。可以操纵调用服务，使用其权限以在其他情况下该服务不应有访问权限的方式对另一个客户的资源进行操作。为防止这种情况，AWS 提供可帮助您保护所有服务的数据的工具，而这些服务中的服务主体有权限访问账户中的资源。

我们建议在资源策略中使用 `aws:SourceArn` 和 `aws:SourceAccount` 全局条件上下文键，来限制 Amazon Aurora DSQL 为其它服务提供的访问资源的权限。如果您只希望将一个资源与跨服务访问相关联，请使用。`aws:SourceArn`如果您想允许该账户中的任何资源与跨服务使用操作相关联，请使用。`aws:SourceAccount`

防范混淆代理问题最有效的方法是使用 `aws:SourceArn` 全局条件上下文键和资源的完整 ARN。如果不知道资源的完整 ARN，或者正在指定多个资源，请针对 ARN 未知部分使用带有通配符字符（`*`）的 `aws:SourceArn` 全局上下文条件键。例如 `arn:aws:dsql:*:123456789012:*`。

如果 `aws:SourceArn` 值不包含账户 ID，例如 Amazon S3 存储桶 ARN，您必须使用两个全局条件上下文键来限制权限。

`aws:SourceArn` 的值必须为 ResourceDescription。

以下示例演示如何在 Aurora DSQL 中使用 `aws:SourceArn` 和 `aws:SourceAccount` 全局条件上下文键来防范混淆代理问题。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "backup.amazonaws.com"
    },
    "Action": "dsql:GetCluster",
    "Resource": [
      "arn:aws:dsql:*:123456789012:cluster/*"
    ],
    "Condition": {
      "ArnLike": {
```

```
          "aws:SourceArn": "arn:aws:backup:*:123456789012:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
}
```

Aurora DSQL 的安全最佳实践

Aurora DSQL 提供了在您开发和实施自己的安全策略时需要考虑的许多安全功能。以下最佳实践是一般指导原则，并不代表完整安全解决方案。这些最佳实践可能不适合环境或不满足环境要求，请将其视为有用的考虑因素而不是惯例。

主题

- Aurora DSQL 的检测性安全最佳实践
- Aurora DSQL 的预防性安全最佳实践

Aurora DSQL 的检测性安全最佳实践

除了以下安全使用 Aurora DSQL 的方法外，请参阅 AWS Well-Architected Tool 中的安全性，以了解云技术如何提高安全性。

Amazon CloudWatch 警报

使用 Amazon CloudWatch 警报，您可以在指定时间段内监控某个指标。如果指标超过给定阈值，则会向 Amazon SNS 主题或 AWS Auto Scaling 策略发送通知。CloudWatch 警报将不会调用操作，因为这些操作处于特定状态。而是必须在状态已改变并在指定的若干个时间段内保持不变后才调用。

标记 Aurora DSQL 资源以进行标识和自动化

可以将自己的元数据以标签形式分配给 AWS 资源。每个标签都是一个标注，包含一个客户定义的密钥和一个可选值，方便管理、搜索和筛选资源。

标签可实现分组控制。尽管没有固有类型的标签，但利用标签，可以根据用途、所有者、环境或其他条件分类资源。下面是一些示例：

- 安全性 – 用于确定加密等要求。

- 机密性 – 资源支持的特定数据机密等级的标识符。
- 环境 – 用于区分开发、测试和生产基础设施。

可以将自己的元数据以标签形式分配给 AWS 资源。每个标签都是一个标注，包含一个客户定义的密钥和一个可选值，方便管理、搜索和筛选资源。

标签可实现分组控制。尽管没有固有类型的标签，但利用标签，您可以根据用途、所有者、环境或其他标准来将 资源分类。下面是一些示例。

- 安全性：用于确定加密等要求。
- 机密性：资源支持的特定数据机密等级的标识符。
- 环境：用于区分开发、测试和生产基础设施。

有关更多信息，请参阅 Best Practices for Tagging AWS Resources。

Aurora DSQL 的预防性安全最佳实践

除了以下安全使用 Aurora DSQL 的方法外，请参阅 AWS Well-Architected Tool 中的安全性，以了解云技术如何提高安全性。

使用 IAM 角色对 Aurora DSQL 的访问进行身份验证。

访问 Aurora DSQL 的用户、应用程序和其它 AWS 服务都必须在 AWS API 和 AWS CLI 请求中包含有效的 AWS 凭证。您不应直接在应用程序或 EC2 实例中存储 AWS 凭证。这些是长期凭证，不会自动轮换。如果这些凭证遭到泄露，则会对业务产生重大影响。利用 IAM 角色，您可以获得可用于访问 AWS 服务和资源的临时访问密钥。

有关更多信息，请参阅 Aurora DSQL 的身份验证和授权。

使用 IAM 策略进行 Aurora DSQL 基本授权。

当您授予权限时，您将决定谁会获得这些权限，获得对于哪些 Aurora DSQL API 的权限，以及支持对这些资源执行的具体操作。实施最低权限对于减小安全风险以及错误或恶意意图造成的影响至关重要。

将权限策略附加到 IAM 角色，并授予对 Aurora DSQL 资源执行操作的权限。IAM 实体的权限边界也可用，借助该边界，您可以设置基于身份的策略可向 IAM 实体授予的最大权限。

与您的 AWS 账户的根用户最佳实践类似，请勿使用 Aurora DSQL 中的 `admin` 角色来执行日常操作。相反，我们建议您创建自定义数据库角色来管理和连接到集群。有关更多信息，请参阅访问 Aurora DSQL 和了解 Aurora DSQL 的身份验证和授权。

在生产环境中使用 **`verify-full`**。

此设置验证服务器证书是否由受信任的证书颁发机构签名，以及服务器主机名是否与证书匹配。

更新 PostgreSQL 客户端

定期将 PostgreSQL 客户端更新到最新版本，以受益于安全性方面的改进。建议使用 PostgreSQL 版本 17。

在 Aurora DSQL 中为资源添加标签

在 AWS 中，标签是用户定义的键值对，您可以定义这些键值对，并将其与 Aurora DSQL 资源（例如集群）相关联。标签是可选的。如果您提供键，则值是可选的。

您可以使用 AWS 管理控制台、AWS CLI 或 AWS SDK 在 Aurora DSQL 集群上添加、列出和删除标签。您可以在集群创建期间和之后使用 AWS 管理控制台添加标签。要在创建集群后使用 AWS CLI 为其添加标签，请使用 `TagResource` 操作。

使用名称为集群添加标签

Aurora DSQL 创建具有全局唯一标识符的集群，该标识符被指定为 Amazon 资源名称（ARN）。如果您想为集群分配一个用户友好名称，我们建议您使用标签。

如果您使用 Aurora DSQL 控制台来创建控制台，Aurora DSQL 会自动创建一个标签。此标签具有一个键（即名称）和一个自动生成的表示集群名称的值。此值是可配置的，因此您可以为集群分配更友好的名称。如果集群具有“名称”标签以及关联的值，则您可以在整个 Aurora DSQL 控制台中看到该值。

标记要求

标签具有以下要求：

- 键不得以 `aws:` 作为前缀。
- 每个标签集中的各个键必须是独一无二的。
- 键的长度必须介于 1 到 128 个允许的字符之间。
- 值的长度必须介于 0 到 256 个允许的字符之间。
- 每个标签集中的值不需要是唯一的。
- 可用作键和值的字符包括字母、数字、空格及以下任何符号：_ . : / = + - @。
- 键和值区分大小写。

标记使用说明

在 Aurora DSQL 中使用标签时，应考虑以下事项。

- 使用 AWS CLI 或 Aurora DSQL API 操作时，确保提供要使用的 Aurora DSQL 资源的 Amazon 资源名称（ARN）。有关更多信息，请参阅适用于 Aurora DSQL 资源的 Amazon 资源名称（ARN）格式。
- 每个资源都有一个标签集，它是分配给该资源的一个或多个标签的集合。
- 每个资源的每个标签集内最多有 50 个标签。
- 如果删除资源，则会删除所有关联的标签。
- 您可以在创建资源时添加标签，也可以使用以下 API 操作来查看和修改标签：`TagResource`、`UntagResource` 和 `ListTagsForResource`。
- 可以将标签与 IAM 策略结合使用。可以使用这些标签来管理对 Aurora DSQL 集群的访问，并控制可将什么操作应用于这些资源。要了解更多信息，请参阅使用标签控制对 AWS 资源的访问。
- 您可以将标签用于跨 AWS 的各种其它活动。要了解更多信息，请参阅常见标记策略。

使用 Amazon Aurora DSQL 的注意事项

使用 Amazon Aurora DSQL 时请考虑以下行为。有关 PostgreSQL 兼容性和支持的更多信息，请参阅 Aurora DSQL 中的 SQL 功能兼容性。有关配额和限制，请参阅 Amazon Aurora DSQL 中的集群配额和数据库限制。

- 运行 DROP TABLE 命令后，存储限制计算可能需要一些时间才能反映已释放的存储空间。如果您需要额外的存储容量，请参阅集群配额以请求配额更新。
- 对于 Aurora DSQL 中的大型表，请使用系统目录而不是 COUNT(*) 操作来检索表行计数。有关更多信息，请参阅使用 Aurora DSQL 中的系统表和命令。
- Aurora DSQL 通过架构级别的授权来管理权限。管理员用户使用 CREATE SCHEMA 创建架构，并使用 GRANT USAGE ON SCHEMA 向其它角色授予访问权限。管理员用户管理公有架构中的对象，而非管理员用户则在用户创建的架构中创建对象。管理员角色可以向自己授予任何其它角色，来获得对用户创建的对象的权限。有关更多信息，请参阅 授权数据库角色在数据库中使用 SQL。
- 当驱动程序调用 PG_PREPARED_STATEMENTS 时，Aurora DSQL 提供缓存的预准备语句的集群范围视图。对于同一个集群和 IAM 角色，您看到的每个连接的预准备语句数量可能会超过预期。Aurora DSQL 在准备过程中动态管理语句名称。
- 从仅限 IPv4 的实例进行连接时，请确保客户端已配置为进行 IPv4 连接。某些 PostgreSQL 客户端在双堆栈模式下，会同时尝试进行 IPv4 和 IPv6 连接。如果 IPv4 连接遇到节流，则客户端可能会尝试 IPv6，并在仅限 IPv4 的主机上返回 NetworkUnreachable 错误。将您的客户端配置为显式使用 IPv4 以避免这种行为。
- 管理员用户创建新架构后，GRANT 和 REVOKE 更改将在连接生命周期（最长一小时）内传播到现有连接。为了立即生效，请在权限更改后建立新的连接。
- 在罕见的多区域链接集群恢复场景中，自动集群恢复操作可保持高可用性，但您可能会遇到暂时的并发控制或连接错误。在大多数情况下，只有一定百分比的工作负载受到影响。当您遇到这些暂时错误时，请重试事务或重新连接您的客户端。
- 某些 SQL 客户端（例如 Datagrip）会请求广泛的系统元数据来填充架构信息。Aurora DSQL 为 SQL 查询功能提供核心元数据。与其完整功能集相比，这些客户端中的架构显示所展示的信息可能有限。
- 为确保查询能够识别新创建的架构和表，请在创建或删除数据库对象后刷新连接。这包括在删除架构后或查询在其它连接中创建的对象时看到 Schema Already Exists 错误的情形。断开连接并重新连接，或者再次运行 SET search_path 以刷新目录缓存。

- 对于复杂的查询，使用 `EXPLAIN ANALYZE VERBOSE` 来识别高延迟操作并优化查询计划。涵盖索引可通过启用仅索引扫描而不是全表扫描来显著降低 DPU 成本。有关更多信息，请参阅 使用 Aurora DSQL EXPLAIN 解释计划。
- 连接限制在集群级别进行管理。请参阅集群配额来请求配额更新。

Amazon Aurora DSQL 中的集群配额和数据库限制

以下各节介绍 Aurora DSQL 的集群配额和数据库限制。

集群配额

您的 AWS 账户在 Aurora DSQL 中具有以下集群配额。要请求增加特定 AWS 区域内单区域和多区域集群的服务配额，请使用服务配额控制台页面。如需增加其它配额，请联系 AWS 支持。

说明	默认限制	是否可配置？	Aurora DSQL 错误代码
每个 AWS 账户的最大单区域集群数	20 个集群	是	API 错误代码 `ServiceQuotaExceededException`
每个 AWS 账户的最大多区域集群数	5 个集群	是	API 错误代码 `ServiceQuotaExceededException`
每个集群的最大存储空间	默认限制为 10 TiB，经批准提高限制后可高达 256 TiB	是	`DISK_FULL(53100)`
每个集群的最大连接数	10000 个连接	是	`TOO_MANY_CONNECTIONS(53300)`
每个集群的最大连接速率	每秒 100 个连接	否	`CONFIGURED_LIMIT_EXCEEDED(53400)`
每个集群的最大连接容量爆增	1000 个连接	否	无错误代码
最大并发还原作业数	4	否	无错误代码

说明	默认限制	是否可配置？	Aurora DSQL 错误代码
连接重新填充速率	每秒 100 个连接	否	无错误代码

Aurora DSQL 中的数据库限制

下表列出了 Aurora DSQL 中的数据库限制。

说明	默认限制	是否可配置？	Aurora DSQL 错误代码	错误消息
主键中使用的列的最大组合大小	1 KiB	否	54000	`ERROR: key size too large`
二级索引中列的最大组合大小	1 KiB	否	54000	`ERROR: key size too large`
表中一行的最大大小	2 MiB	否	54000	`ERROR: maximum row size excee`
不属于索引一部分的列的最大大小	1 MiB	否	54000	`ERROR: maximum column size ex`
主键或二级索引中的最大列数	8	否	54011	`ERROR: more than 8 column key are not supported`
表中的最大列数	255	否	54011	`ERROR: tables can have at mos`
表中的最大索引数	24	否	54000	`ERROR: more than 24 indexes p allowed`
在一个写入事务中修改的所有数据的最大大小	10 MiB	否	54000	`ERROR: transaction size limit DETAIL: Current transaction s 10mb`

说明	默认限制	是否可配置？	Aurora DSQL 错误代码	错误消息
事务块中可能突变的表行的最大数量	每个事务 3000 行。请参阅 Aurora DSQL 有关 PostgreSQL 兼容性的注意事项。	否	54000	ERROR: transaction row limit
查询操作可以使用的最大基本内存量	每个事务 128 MiB	否	53200	ERROR: query requires too muc out of memory.
数据库中定义的最大架构数	10	否	54000	ERROR: more than 10 schemas n
数据库中的最大表数	1000 个表	否	54000	ERROR: creating more than 100 allowed
集群中的最大数据库数	1	否	无错误代码	ERROR: unsupported statement
最长事务时间	5 分钟	否	54000	ERROR: transaction age limit exceeded
最大连接持续时间	60 分钟	否	无错误代码	无错误消息
数据库中的最大视图数	5000	否	54000	ERROR: creating more than 500 allowed
最大视图定义大小	2 MiB	否	54000	ERROR: view definition too la

说明	默认限制	是否可配置？	Aurora DSQL 错误代码	错误消息
最大序列数量	5000	否	`54000`	`ERROR: creating more than 500` `not allowed`

有关特定于 Aurora DSQL 的数据类型限制，请参阅 Aurora DSQL 中支持的数据类型。

Aurora DSQL API 参考

除了 AWS 管理控制台和 AWS Command Line Interface（AWS CLI）之外，Aurora DSQL 还提供了一个 API 接口。可以使用 API 操作来管理 Aurora DSQL 中的资源。

有关 API 操作的字母顺序列表，请参阅操作。

有关数据类型的字母顺序列表，请参阅数据类型。

有关常用查询参数的列表，请参阅常用参数。

有关错误代码的描述，请参阅常见错误。

有关 AWS CLI 的更多信息，请参阅适用于 Aurora DSQL 的 AWS Command Line Interface参考。

排查 Aurora DSQL 中的问题

Note

以下主题为您在使用 Aurora DSQL 时可能遇到的错误和问题提供故障排除建议。如果您发现某个问题未在此处列出，请联系 AWS 支持人员

主题

- 连接错误故障排除
- 身份验证错误故障排除
- 授权错误故障排除
- SQL 错误故障排除
- OCC 错误故障排除
- SSL/TLS 连接故障排除

连接错误故障排除

error: unrecognized SSL error code: 6 或 unable to accept connection, sni was not received

可能您使用的 psql 版本早于 version 14，不支持服务器名称指示（SNI）。连接到 Aurora DSQL 时需要 SNI。

您可以使用 `psql --version` 来检查客户端版本。

error: NetworkUnreachable

尝试连接时出现 `NetworkUnreachable` 错误可能表示客户端不支持 IPv6 连接，而不是表示存在实际的网络问题。此错误通常发生在仅限 IPv4 的实例上，这是因为 PostgreSQL 客户端处理双堆栈连接的方式所致。当服务器支持双堆栈模式时，这些客户端首先将主机名解析为 IPv4 和 IPv6 地址。它们首先尝试 IPv4 连接，如果初始连接失败，则尝试 IPv6。如果系统不支持 IPv6，您将看到一条常规 `NetworkUnreachable` 错误，而不是一条明确的“IPv6 not supported”消息。

身份验证错误故障排除

IAM authentication failed for user "..."

在生成 Aurora DSQL IAM 身份验证令牌时，您可以设置的最长持续时间为 1 周。一周后，您将无法使用该令牌进行身份验证。

此外，如果您代入的角色已过期，Aurora DSQL 会拒绝您的连接请求。例如，如果您尝试使用临时 IAM 角色进行连接，即使您的身份验证令牌尚未过期，Aurora DSQL 也将拒绝连接请求。

要了解 IAM 如何与 Aurora DSQL 结合使用的更多信息，请参阅了解 Aurora DSQL 的身份验证和授权和 Aurora DSQL 中的 AWS Identity and Access Management。

An error occurred (InvalidAccessKeyId) when calling the GetObject operation: The AWS Access Key ID you provided does not exist in our records

IAM 拒绝了您的请求。有关更多信息，请参阅为什么签署请求。

IAM role <role> does not exist

Aurora DSQL 找不到您的 IAM 角色。有关更多信息，请参阅 IAM 角色。

IAM role must look like an IAM ARN

有关更多信息，请参阅 IAM 标识符 - IAM ARN。

用户与操作的映射错误

当身份验证令牌类型与数据库角色不匹配时，会发生此错误。Aurora DSQL 使用两种令牌类型：对于 `admin` 角色为 `DbConnectAdmin`，对于自定义数据库角色为 `DbConnect`。

- 如果您看到 `Wrong user to action mapping. user: admin, action: DbConnect`，请使用 `generate-db-connect-admin-auth-token` 来替代 `generate-db-connect-auth-token`。
- 如果您看到 `Wrong user to action mapping. user: myusername, action: DbConnectAdmin`，请使用 `generate-db-connect-auth-token` 来替代 `generate-db-connect-admin-auth-token`。

授权错误故障排除

Role <role> not supported

Aurora DSQL 不支持 `GRANT` 操作。请参阅 Aurora DSQL 中支持的 SQL 命令子集。

Cannot establish trust with role <role>

Aurora DSQL 不支持 GRANT 操作。请参阅 Aurora DSQL 中支持的 SQL 命令子集。

Role <role> does not exist

Aurora DSQL 找不到指定的数据库用户。请参阅授权自定义数据库角色连接到集群。

ERROR: permission denied to grant IAM trust with role <role>

要向数据库角色授予访问权限，您必须使用管理员角色连接到集群。要了解更多信息，请参阅授权数据库角色在数据库中使用 SQL。

ERROR: role <role> must have the LOGIN attribute

您创建的任何数据库角色都必须具有 LOGIN 权限。

要解决此问题，请确保您已创建具有 LOGIN 权限的 PostgreSQL 角色。有关更多信息，请参阅 PostgreSQL 文档中的 CREATE ROLE 和 ALTER ROLE。

ERROR: role <role> cannot be dropped because some objects depend on it

如果您删除具有 IAM 关系的数据库角色，Aurora DSQL 会返回错误，直到您使用 AWS IAM REVOKE 撤销该关系。要了解更多信息，请参阅撤销授权。

SQL 错误故障排除

Error: Not supported

Aurora DSQL 并不支持所有基于 PostgreSQL 的方言。要了解支持的内容，请参阅 Aurora DSQL 中支持的 PostgreSQL 功能。

Error: use **CREATE INDEX ASYNC** instead

要在包含现有行的表上创建索引，必须使用 CREATE INDEX ASYNC 命令。要了解更多信息，请参阅在 Aurora DSQL 中异步创建索引。

OCC 错误故障排除

OC000 “ERROR: mutation conflicts with another transaction, retry as needed”

该事务尝试修改另一个并发事务正在处理的相同元组。这表明被修改的元组存在资源争用情况。要了解更多信息，请参阅 Aurora DSQL 中的并发控制。

OC001 “ERROR: schema has been updated by another transaction, retry as needed”

您的 PostgreSQL 会话具有架构目录的一个缓存副本。该缓存副本在加载时是有效的。我们称为时间 T1 和版本 V1。

另一个事务在时间 T2 更新目录。我们称之为 V2。

当原始会话在时间 T2 尝试从存储中读取时，它仍在使用目录版本 V1。Aurora DSQL 的存储层拒绝该请求，因为 T2 时的最新目录版本是 V2。

当您在时间 T3 从原始会话中重试时，Aurora DSQL 会刷新目录缓存。T3 时的事务使用的是目录 V2。只要自时间 T2 以来没有发生其它目录更改，Aurora DSQL 就会完成事务。

SSL/TLS 连接故障排除

SSL error: certificate verify failed

此错误表示客户端无法验证服务器的证书。请确保：

1. 已正确安装 Amazon Root CA 1 证书。有关如何验证和安装此证书的说明，请参阅为 Aurora DSQL 连接配置 SSL/TLS 证书。
2. `PGSSLROOTCERT` 环境变量指向正确的证书文件。
3. 证书文件具有正确的权限。

Unrecognized SSL error code: 6

低于版本 14 的 PostgreSQL 客户端会发生此错误。要解决此问题，请将 PostgreSQL 客户端升级到版本 17。

SSL error: unregistered scheme (Windows)

这是使用系统证书时 Windows psql 客户端的一个已知问题。使用从 Windows 进行连接说明中描述的下载证书文件方法。

提供有关 Amazon Aurora DSQL 的反馈

如果您遇到对迁移至关重要但 Aurora DSQL 目前不支持的功能，AWS 会提供多种反馈渠道：

反馈渠道

Aurora DSQL Discord 服务器

加入 Aurora DSQL Discord 服务器，以便与 AWS 团队和社区建立联系。分享功能请求，讨论迁移挑战，并获得实时反馈。

AWS Support

如果您有 AWS Support 计划，请创建一个支持案例来讨论您的具体要求和时间表需求。

AWS re:Post

使用 AWS re:Post 可向社区和 AWS 专家提问和分享反馈。

有效的功能请求

请求功能时，请提供：

- 使用案例描述：解释您想实现什么目标以及原因
- 当前解决方法：描述您尝试过的所有替代方案
- 业务影响：解释缺失的功能如何影响您的迁移时间表或应用程序功能
- 优先级：说明这是会阻碍您的迁移，还是锦上添花的改进

《Amazon Aurora DSQL 用户指南》的文档历史记录

下表介绍了 Aurora DSQL 的文档版本。

变更	说明	日期
新内容：适用于 .NET Npgsql 的 Aurora DSQL 连接器	添加了适用于 .NET Npgsql 的 Aurora DSQL 连接器的文档，该连接器使用自动 IAM 身份验证封装 Npgsql。该连接器处理 .NET 应用程序的令牌生成、SSL 配置和连接管理。有关更多信息，请参阅使用 .NET Npgsql 连接器连接到 Aurora DSQL 集群。	2026 年 3 月 20 日
更新了内容：SQL 命令参考和系统查询	在事务控制命令参考中添加了 START TRANSACTION 和 ROLLBACK，以 END 和 ABORT 作为别名。为检索 Aurora DSQL 和 PostgreSQL 版本信息添加了有用的系统查询。有关更多信息，请参阅 PostgreSQL 兼容性参考。	2026 年 3 月 13 日
更新了内容：将数据加载到 Aurora DSQL 中	更新了数据加载指南，加入了客户端 \copy 用法、INSERT 最佳实践和加载前预先创建表的指南。有关更多信息，请参阅将数据加载到 Aurora DSQL 中。	2026 年 3 月 13 日
新内容：适用于 Ruby pg 的 Aurora DSQL 连接器	添加了适用于 Ruby pg 的 Aurora DSQL 连接器的文档，该连接器使用自动 IAM 身份验证封装 pg gem。该连接器处理 Ruby 应用程序的令牌生成	2026 年 3 月 12 日

	、SSL 配置和连接管理。有关更多信息，请参阅使用 Ruby pg 连接器连接到 Aurora DSQL 集群。	
更新了内容：异步 DDL 作业	更新了 sys.jobs 文档，加入了有关监控和管理异步 DDL 操作的详细信息。有关更多信息，请参阅 PostgreSQL 兼容性参考。	2026 年 3 月 6 日
更新了内容：PHP 身份验证令牌生成	向身份验证令牌生成页面添加了 PHP SDK 选项卡。有关更多信息，请参阅生成身份验证令牌。	2026 年 3 月 5 日
新内容：将数据加载到 Aurora DSQL 中	添加了将数据加载到 Aurora DSQL 集群的指南，包括使用 Aurora DSQL 加载器实用程序。有关更多信息，请参阅将数据加载到 Aurora DSQL 中。	2026 年 3 月 5 日
新内容：序列和标识列	添加了对序列和标识列的支持。有关 CREATE SEQUENCE、ALTER SEQUENCE、DROP SEQUENCE 和序列操作函数的新 SQL 命令参考页面。更新了 CREATE TABLE 和 ALTER TABLE 以包含标识列语法。添加了有关选择标识符类型和缓存大小的新指南。有关更多信息，请参阅序列和标识列。	2026 年 2 月 11 日

新内容：适用于 Go 的 Aurora DSQL 连接器	添加了适用于 Go 的 Aurora DSQL 连接器的文档，该连接器使用自动 IAM 身份验证封装 pgx。该连接器处理 Go 应用程序的令牌生成、SSL 配置和连接管理。有关更多信息，请参阅使用 Go 连接器连接到 Aurora DSQL 集群。	2026 年 2 月 5 日
更新了内容：Amazon Aurora DSQL 集群连接工具	重组了集群连接工具文档，以阐明 AWS 提供的连接器、适配器和第三方工具之间的区别。添加了缺少的指向代码示例的链接。有关更多信息，请参阅 Amazon Aurora DSQL 集群连接工具。	2026 年 1 月 26 日
新内容：适用于 DBeaver 社区版的 Aurora DSQL 插件	添加了适用于 DBeaver 社区版的 Aurora DSQL 插件的文档，该插件支持 IAM 身份验证并简化了 Aurora DSQL 集群的连接设置。包括安装说明、连接配置和故障排除指南。有关更多信息，请参阅使用 DBeaver 访问 Aurora DSQL。	2026 年 1 月 26 日
新内容：适用于 SQLTools 的 Aurora DSQL 驱动程序	添加了适用于 SQLTools 的 Aurora DSQL 驱动程序的文档，这是一个 Visual Studio 代码扩展，可让开发人员通过自动 IAM 身份验证直接从 VS Code 连接和查询 Aurora DSQL 数据库。有关更多信息，请参阅使用适用于 SQLTools 的 Aurora DSQL 驱动程序。	2026 年 1 月 26 日

更新了内容：Aurora DSQL 操控：技能和能力	添加了文档以支持使用 Skills CLI 进行安装，从而获得与代理无关的支持。Skills CLI 提供了一种简化的设置方法，适用于多个 AI 编码助手，包括 Claude Code、Cursor、Copilot、Gemini 等。有关更多信息，请参阅 Aurora DSQL 操控：技能和能力。	2026 年 1 月 23 日
新内容：适用于 Tortoise ORM 的 Aurora DSQL 适配器	添加了对 Tortoise ORM（一个 Python 异步 ORM 框架）的支持。适用于 Tortoise ORM 的 Aurora DSQL 适配器使开发人员能够将 Tortoise ORM 与 Aurora DSQL 集群结合使用。有关更多信息，请参阅 Aurora DSQL 适配器和方言。	2026 年 1 月 23 日
新内容：Aurora DSQL 操控：技能和能力	添加了有关通过 Aurora DSQL 使用技能和能力配置 AI 操控的新文档。包括 Kiro Powers、Claude Skills、Gemini Skills 和 Codex Skills 的设置说明。有关更多信息，请参阅 Aurora DSQL 操控：技能和能力。	2026 年 1 月 16 日
数值数据类型索引支持	在 Aurora DSQL 中为 `numeric` 数据类型添加了索引支持。现在，您可以使用 `numeric` 列作为主键和二级索引。有关更多信息，请参阅 Aurora DSQL 中支持的数据类型。	2026 年 1 月 13 日

更新了内容：支持的 SQL 命令子集	已将 SQL 命令文档重组为单独的页面，以改善导航和清晰性。现在，每个命令（CREATE TABLE、ALTER TABLE、CREATE VIEW、ALTER VIEW、DROP VIEW）都有其自己的专用页面。有关更多信息，请参阅支持的 SQL 命令子集。	2026 年 1 月 6 日
更新了内容：AWS Labs Aurora DSQL MCP 服务器	更新了 MCP 服务器文档，具有适用于 Claude Code 和 Codex 的详细安装方法，包括基于 CLI 的设置和配置文件示例。添加了有关在不同开发工具中查找 MCP 客户端配置文件的全面指南。有关更多信息，请参阅 AWS Labs Aurora DSQL MCP 服务器。	2025 年 12 月 19 日
更新了内容：从 PostgreSQL 迁移到 Aurora DSQL	重新起草了 PostgreSQL 兼容性部分，作为全面的迁移指南。包括框架兼容性信息、常见迁移模式、架构差异和 AI 辅助迁移指南。添加了旨在提供有关 Aurora DSQL 的反馈的新章节。有关更多信息，请参阅从 PostgreSQL 迁移到 Aurora DSQL。	2025 年 12 月 16 日

更新了内容：使用 AWS PrivateLink 连接到 Aurora DSQL	添加了有关私有 DNS 设置和集群 ID 连接选项的文档，以支持客户将 AWS PrivateLink 与 Direct Connect 或 Amazon VPC 对等连接结合使用。包括通过 `amzn-cluster-id` 连接选项在不使用私有 DNS 的情况下进行连接的指南。有关更多信息，请参阅使用 AWS PrivateLink 管理和连接到 Aurora DSQL 集群。	2025 年 12 月 11 日
更新了内容：Aurora DSQL 集群生命周期	更新了有关 Aurora DSQL 集群生命周期管理的文档。解释集群状态定义、状态转换以及在空闲和非活动状态期间可用的操作。有关更多信息，请参阅 Aurora DSQL 集群生命周期。	2025 年 12 月 4 日
新内容：适用于 Python 和 Node.js 的 Aurora DSQL 连接器	添加了有关适用于 Python 的 Aurora DSQL 连接器（psycopg、psycopg2、asyncpg）和适用于 Node.js 的 Aurora DSQL 连接器（node-postgres、Postgres.js）的文档。这些连接器集成了 IAM 身份验证，用于将应用程序连接到 Aurora DSQL 集群。有关更多信息，请参阅 Aurora DSQL 连接器。	2025 年 11 月 21 日

新内容：将 JupyterLab 和 Aurora DSQL 结合使用	添加了有关使用带有 Python 的 JupyterLab 连接和查询 Aurora DSQL 的分步指南。包括有关本地 JupyterLab 安装和 Amazon SageMaker AI 环境的说明。有关更多信息，请参阅将 JupyterLab 与 Aurora DSQL 结合使用。	2025 年 11 月 20 日
更新了内容：Aurora DSQL 的配额	已将最大集群存储配额从 128 TiB 更新为 256 TiB。有关更多信息，请参阅 Aurora DSQL 的配额。	2025 年 11 月 19 日
新内容：Aurora DSQL 查询编辑器入门	添加了有关在 AWS 管理控制台中使用 Aurora DSQL 查询编辑器的文档。包括先决条件、连接设置和查询执行说明。有关更多信息，请参阅 Aurora DSQL 查询编辑器入门。	2025 年 11 月 18 日

Amazon Aurora DSQL 的基于资源的策略支持	添加了基于资源的策略（RBP）支持，新增了权限：`PutClusterPolicy` 、`GetClusterPolicy` 和 `DeleteClusterPolicy` 。这些权限支持管理附加到 Aurora DSQL 集群的内联策略，以实现精细的访问控制。更新了托管式策略 AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess，以包括 RBP 功能。有关更多信息，请参阅 Amazon Aurora DSQL 的 AWS 托管式策略。	2025 年 10 月 15 日
Aurora DSQL JDBC 连接器	增加了有关 Aurora DSQL JDBC 连接器的文档，这是一款 PgJDBC 连接器，集成了 IAM 身份验证，用于将 Java 应用程序连接到 Amazon Aurora DSQL 集群。有关更多信息，请参阅使用 JDBC 连接器连接到 Aurora DSQL 集群。	2025 年 9 月 2 日

针对 AWS FIS 集成的 AWS 托管式策略更新	更新了 AmazonAuroraDSQLFullAccess 和 AmazonAuroraDSQLConsoleFullAccess 策略，用于支持 AWS Fault Injection Service与 Aurora DSQL 的集成。通过此集成，您可以向单区域和多区域 Aurora DSQL 集群注入故障，来测试应用程序的容错能力。有关这些策略的更多信息，请参阅 AWS 托管式策略的更新。	2025 年 8 月 19 日
Amazon Aurora DSQL 的正式发布（GA）	Amazon Aurora DSQL 现已正式发布，新增了对 CloudWatch 监控、增强型数据保护功能和 AWS Backup 集成的支持。有关更多信息，请参阅 Monitoring Aurora DSQL with CloudWatch、Backup and restore for Amazon Aurora DSQL 和 Data encryption for Amazon Aurora DSQL。	2025 年 5 月 27 日
AmazonAuroraDSQLFullAccess 更新	添加了对 Aurora DSQL 集群执行备份和还原操作的功能，包括启动、停止和监控作业。它还添加了使用客户自主管理型 KMS 密钥进行集群加密的功能。有关更多信息，请参阅 AmazonAuroraDSQLFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 21 日

AmazonAuroraDSQLConsoleFullAccess 更新	添加了通过 AWS Console Home对 Aurora DSQL 集群执行备份和还原操作的功能。这包括启动、停止和监控作业。它还支持使用客户自主管理型 KMS 密钥进行集群加密和启动 AWS CloudShell。有关更多信息，请参阅 AmazonAuroraDSQLConsoleFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 21 日
AmazonAuroraDSQLReadOnlyAccess 更新	包括在通过 AWS PrivateLink 连接到 Aurora DSQL 集群时确定正确的 VPC 端点服务名称的功能。Aurora DSQL 为每个单元创建唯一的端点，因此，此 API 有助于确保您可以为集群识别正确的端点并避免连接错误。有关更多信息，请参阅 AmazonAuroraDSQLReadOnlyAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日

AmazonAuroraDSQLFullAccess 更新	该策略添加了四个新权限，用于跨多个 AWS 区域创建和管理数据库集群：PutMultiRegionProperties PutWitnessRegion 、AddPeerCluster 和 RemovePeerCluster 。这些权限包括资源级控制措施和条件键，因此您可以控制您可以修改哪些集群用户。该策略还添加了 GetVpcEndpointServiceName 权限，有助于您通过 AWS PrivateLink 连接到 Aurora DSQL 集群。有关更多信息，请参阅 AmazonAuroraDSQLConsoleFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日
AmazonAuroraDSQLConsoleFullAccess 更新	向 Aurora DSQL 添加新的权限，以支持多区域集群管理和 VPC 端点连接。新权限包括：PutMultiRegionProperties 、PutWitnessRegion 、AddPeerCluster 、RemovePeerCluster 、GetVpcEndpointServiceName 。请参阅 AmazonAuroraDSQLConsoleFullAccess 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 13 日

AuroraDsqlServiceLinkedRolePolicy 更新	向策略中添加了将指标发布到 `AWS/AuroraDSQL` 和 `AWS/Usage` CloudWatch 命名空间的功能。这样，关联的服务或角色就可以向 CloudWatch 环境发送更全面的使用情况和性能数据。有关更多信息，请参阅 AuroraDsqlServiceLinkedRolePolicy 和使用 Aurora DSQL 中的服务相关角色。	2025 年 5 月 8 日
AWS PrivateLink适用于 Amazon Aurora DSQL 的	Aurora DSQL 现在支持 AWS PrivateLink。借助 AWS PrivateLink，您可以使用接口 Amazon VPC 端点和私有 IP 地址，来简化虚拟私有云（VPC）、Aurora DSQL 和本地数据中心之间的私有网络连接。有关更多信息，请参阅使用 AWS PrivateLink 管理和连接到 Amazon Aurora DSQL 集群。	2025 年 5 月 8 日
初始版本。	《Amazon Aurora DSQL 用户指南》的初始版本	2024 年 12 月 3 日