aws

用户指南

Amazon Aurora DSQL

Amazon Aurora DSQL: 用户指南

Table of Contents

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

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 数据库的兼容性

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，并支持一部分 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。

支持的 AWS 区域和端点

区域名称	区域	端点	协议
美国东部（弗吉尼亚州北部）	us-east-1	dsql.us-east-1.api.aws	HTTPS
美国东部（俄亥俄州）	us-east-2	dsql.us-east-2.api.aws	HTTPS
美国西部（俄勒冈州）	us-west-2	dsql.us-west-2.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
亚太地区（东京）	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS
亚太地区（首尔）	ap-northeast-2	dsql.ap-northeast-2.api.aws	HTTPS
亚太地区（大阪）	ap-northeast-3	dsql.ap-northeast-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（爱尔兰）

- 欧洲地区（伦敦）
- 欧洲地区（巴黎）

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
- 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、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 Management Console访问集群，并使用 `psql` 实用程序与数据库进行交互。最后，您将设置好正常运行的 Aurora DSQL 集群，并可用于测试或生产工作负载。

主题

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 sign in to the AWS Management Console 的权限。
- 您的 IAM 身份必须满足以下任一条件：
  - 对 AWS 账户中的任何资源执行任何操作的访问权限
  - IAM 权限 `iam:CreateServiceLinkedRole` 和能够访问 IAM 策略操作 `dsql:*`
- 如果您在类似 Unix 的环境中使用 AWS CLI，请确保安装 Python 版本 3.8+ 和 `psql` 版本 14+。要检查应用程序版本，请运行以下命令。

  ```
  python3 --version
  psql --version
  ```

  如果您在不同的环境中使用 AWS CLI，请务必手动设置 Python 版本 3.8+ 和 `psql` 版本 14+。
- 如果您打算使用 AWS CloudShell 访问 Aurora DSQL，则所提供的 Python 版本 3.8+ 和 `psql` 版本 14+ 无需额外的设置。有关 AWS CloudShell 的更多信息，请参阅什么是 AWS CloudShell？
- 如果您打算使用 GUI 访问 Aurora DSQL，请使用 DBeaver 或 JetBrains DataGrip。有关更多信息，请参阅使用 DBeaver 访问 Aurora DSQL和使用 JetBrains DataGrip 访问 Aurora DSQL。

访问 Aurora DSQL

您可以通过以下方法访问 Aurora DSQL。要了解如何使用 CLI、API 和 SDK，请参阅访问 Aurora DSQL。

主题

通过 AWS Management Console访问 Aurora DSQL

您可以访问 Aurora DSQL 的 AWS Management Console，网址为 https://console.aws.amazon.com/dsql。

使用 SQL 客户端访问 Aurora DSQL

Aurora DSQL 使用 PostgreSQL 协议。在连接到集群时，通过提供签名的 IAM authentication token 作为密码，使用您首选的交互式客户端。身份验证令牌是 Aurora DSQL 使用 AWS 签名版本 4 动态生成的唯一字符串。

Aurora DSQL 仅将此令牌用于身份验证。建立连接后，令牌不会影响连接。如果您尝试使用过期的令牌重新连接，则连接请求将遭到拒绝。有关更多信息，请参阅 在 Amazon Aurora DSQL 中生成身份验证令牌。

主题

使用 psql 访问 Aurora DSQL（PostgreSQL 交互式终端）

`psql` 实用程序是 PostgreSQL 的基于终端的前端。它使您能够以交互方式键入查询，向 PostgreSQL 发出查询，然后查看查询结果。要缩短查询响应时间，请使用 PostgreSQL 版本 17 客户端。

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 https://www.postgresql.org/docs/current/app-psql.htm。

如果您已经安装了 AWS CLI，请使用以下示例连接到集群。您可以使用预安装了 psql 的 AWS CloudShell，也可以直接安装 psql。

```
# 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 DBeaver 访问 Aurora DSQL

DBeaver 是一个基于 GUI 的开源数据库工具。可以使用它来连接和管理数据库。要下载 DBeaver，请访问 DBeaver 社区网站上的下载页面。以下步骤解释了如何使用 DBeaver 连接到集群。

在 DBeaver 中设置新的 Aurora DSQL 连接

1. 选择新建数据库连接。
2. 在新建数据库连接窗口中，选择 PostgreSQL。
3. 在连接设置/主要选项卡中，选择连接方式：主机，然后输入以下信息。

   - 主机：使用您的集群端点。

     数据库：输入 `postgres`

     身份验证：选择 `Database Native`

     用户名：输入 `admin`

     密码：生成身份验证令牌。复制生成的令牌并将其用作密码。

4. 忽略所有警告，并将身份验证令牌粘贴到 DBeaver 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 `SSLMODE=require`。Aurora DSQL 在服务器端强制执行 SSL 通信，并拒绝非 SSL 连接。

5. 您应该已连接到集群，并可以开始运行 SQL 语句。

> **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。

DBeaver 的身份验证凭证到期

已建立的会话会在最多 1 小时内保持身份验证状态，或者直至 DBeaver 断开连接或超时。要建立新连接，请在连接设置的密码字段中提供有效的身份验证令牌。尝试打开新会话（例如，列出新表或新的 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 DBeaver 使所有先前打开的会话变为无效。使用 `expires-in` 选项选择 IAM 身份验证令牌的持续时间时，请记住这一点。

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，请前往 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`。Aurora DSQL 在服务器端强制执行 SSL 通信，并将拒绝非 SSL 连接。

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。

身份验证凭证到期

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，则必须在数据来源属性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。

将 PostgreSQL 协议与 Aurora DSQL 结合使用

PostgreSQL 使用基于消息的协议在客户端和服务器之间进行通信。可通过 TCP/IP 以及 Unix 域套接字支持该协议。下表显示了 Aurora DSQL 如何支持 PostgreSQL protocol。

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，请参阅配置自定义数据库角色。
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。

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 Management Console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）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。
- 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。

4. （可选）在标签中，选择或输入此集群的标签。
5. 选择创建集群。

步骤 2：连接到 Aurora DSQL 集群

当您根据 Aurora DSQL 集群的集群 ID 和区域创建集群时，会自动生成集群端点。命名格式为 *clusterid*.dsql.*region*.on.aws。客户端使用端点来创建与集群的网络连接。

身份验证是使用 IAM 进行管理的，因此您不需要在数据库中存储凭证。身份验证令牌是自动生成的唯一字符串。令牌仅用于身份验证，在建立连接后不会影响连接。在尝试连接之前，请确保 IAM 身份具有 dsql:DbConnectAdmin 权限，如先决条件中所述。

> **Note**
>
> 要优化数据库连接速度，请使用 PostgreSQL 版本 17 客户端并将 PGSSLNEGOTIATION 设置为 direct：PGSSLNEGOTIATION=direct。

使用身份验证令牌连接到集群

1. 在 Aurora DSQL 控制台中，选择要连接到的集群。
2. 选择连接。
3. 从端点（主机）复制端点。
4. 确保在身份验证令牌（密码）部分中选择以管理员身份连接。
5. 复制生成的身份验证令牌。此令牌的有效期为 15 分钟。
6. 在操作系统命令行上，使用以下命令来启动 psql 并连接到集群。将 *your_cluster_endpoint* 替换为您之前复制的集群端点。

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

当提示输入密码时，请输入您之前复制的身份验证令牌。如果您尝试使用过期的令牌重新连接，则连接请求将遭到拒绝。有关更多信息，请参阅 在 Amazon Aurora DSQL 中生成身份验证令牌。

7. 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？

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。以下示例语句需要名为 `department-insert-multirow.sql` 和 `invoice.csv` 的数据文件，您可以从 GitHub 上的 aws-samples/aurora-dsql-samples 存储库中下载这些文件。

在 Aurora DSQL 中运行示例 SQL 命令

1. 创建名为 `example` 的架构。

```
CREATE SCHEMA example;
```

2. 创建使用自动生成的 UUID 作为主键的 invoice 表。

```
CREATE TABLE example.invoice(
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    created timestamp,
    purchaser int,
    amount float);
```

3. 创建使用空表的二级索引。

```
CREATE INDEX ASYNC invoice_created_idx on example.invoice(created);
```

4. 创建 department 表。

```
CREATE TABLE example.department(id INT PRIMARY KEY UNIQUE, name text, email text);
```

5. 使用命令 `psql \include` 加载您从 GitHub 上的 aws-samples/aurora-dsql-samples 存储库下载的名为 `department-insert-multirow.sql` 的文件。将 *my-path* 替换为本地副本的路径。

```
\include my-path/department-insert-multirow.sql
```

6. 使用命令 `psql \copy` 加载您从 GitHub 上的 aws-samples/aurora-dsql-samples 存储库下载的名为 `invoice.csv` 的文件。将 *my-path* 替换为本地副本的路径。

```
\copy example.invoice(created, purchaser, amount) from my-path/invoice.csv csv
```

7. 查询各部门并按总销售额对各部门进行排序。

```
SELECT name, sum(amount) AS sum_amount
FROM example.department LEFT JOIN example.invoice ON
 department.id=invoice.purchaser
GROUP BY name
HAVING sum(amount) > 0
ORDER BY sum_amount DESC;
```

以下示例输出显示 Department One 的销售额最高。

```
           name            |     sum_amount
---------------------------+--------------------
 Example Department One    |  32628.75608634601
 Example Department Three  |  32427.43955110429
 Example Department Eight  | 32256.810987098102
 Example Department Five   |  31391.14891163639
 Example Department Seven  | 31253.236846746757
 Example Department Six    |  29699.06014910414
 Example Department Two    |  29465.58360076501
 Example Department Four   |  28764.19185819191
(8 rows)
```

步骤 4：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。

以下示例说明如何创建初始集群，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 AWS Management Console并打开 Aurora DSQL 控制台，网址为 https://console.aws.amazon.com/dsql。
2. 在导航窗格中，选择集群。
3. 选择创建集群，然后选择多区域。
4. （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。
   - 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。
5. 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。
6. （可选）为初始集群选择标签。
7. 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。
8. 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。
9. 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。

10. 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。
11. （可选）为第二个集群选择集群设置。例如，可以选择 AWS KMS key。
12. 选择创建集群以创建第二个集群。
13. 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。
14. 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 Clusters (集群)。
3. 选择多区域集群中的第二个集群所对应的行。
4. 在操作中，选择在 CloudShell 中打开。
5. 选择以管理员身份进行连接。
6. 选择启动 CloudShell。
7. 选择运行。
8. 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。

   示例事务

   Example

   ```
   CREATE SCHEMA example;
   CREATE TABLE example.invoice(id UUID PRIMARY KEY DEFAULT gen_random_uuid(), created
    timestamp, purchaser int, amount float);
   CREATE INDEX ASYNC invoice_created_idx on example.invoice(created);
   CREATE TABLE example.department(id INT PRIMARY KEY UNIQUE, name text, email text);
   ```

9. 使用 `psql` `copy` 和 `include` 命令来加载示例数据。有关更多信息，请参阅 步骤 3：在 Aurora DSQL 中运行示例 SQL 命令。

```
\copy example.invoice(created, purchaser, amount) from samples/invoice.csv csv
\include samples/department-insert-multirow.sql
```

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 Clusters (集群)。
3. 选择多区域集群中的第二个集群所对应的行。
4. 在操作中，选择在 CloudShell 中打开。
5. 选择以管理员身份进行连接。
6. 选择启动 CloudShell。
7. 选择运行。
8. 查询您插入到第二个集群的数据。

   Example

```
SELECT name, sum(amount) AS sum_amount
FROM example.department
LEFT JOIN example.invoice ON department.id=invoice.purchaser
GROUP BY name
HAVING sum(amount) > 0
ORDER BY sum_amount DESC;
```

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 Management Console、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。

授权

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 管理控制台创建身份验证令牌，默认情况下，该令牌将在一小时后自动过期。如果您使用 AWS CLI 或 SDK 创建令牌，则默认值为 15 分钟。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 Management Console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：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 Management Console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：

- 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

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。

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

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

JavaScript 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 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

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.create())
                .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

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::SharedCredentials.new()

  begin
      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
      })
  rescue => error
```

```
    puts error.full_message
  end
end
```

.NET

> **Note**
>
> 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;
using Amazon.Runtime;

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

AWSCredentials credentials = FallbackCredentialsFactory.GetCredentials();

var token = DSQLAuthTokenGenerator.GenerateDbConnectAdminAuthToken(credentials,
 RegionEndpoint.USEast1, yourClusterEndpoint);

Console.WriteLine(token);
```

Golang

> **Note**
>
> Golang SDK 未提供用于生成预签名令牌的内置方法。您必须手动构造签名请求，如以下代码示例中所示。

在以下代码示例中，根据 PostgreSQL 用户指定 `action`：

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

除了 *`yourClusterEndpoint`* 和 *`region`* 外，以下示例还使用了 *`action`*。请根据 PostgreSQL 用户指定 *`action`*。

```
func GenerateDbConnectAdminAuthToken(yourClusterEndpoint string, region
 string, action string) (string, error) {
 // Fetch credentials
 sess, err := session.NewSession()
 if err != nil {
  return "", err
 }

 creds, err := sess.Config.Credentials.Get()
 if err != nil {
  return "", err
 }
 staticCredentials := credentials.NewStaticCredentials(
  creds.AccessKeyID,
  creds.SecretAccessKey,
  creds.SessionToken,
 )

 // The scheme is arbitrary and is only needed because validation of the URL
 requires one.
 endpoint := "https://" + yourClusterEndpoint
 req, err := http.NewRequest("GET", endpoint, nil)
 if err != nil {
  return "", err
 }
 values := req.URL.Query()
 values.Set("Action", action)
 req.URL.RawQuery = values.Encode()

 signer := v4.Signer{
  Credentials: staticCredentials,
 }
 _, err = signer.Presign(req, nil, "dsql", region, 15*time.Minute, time.Now())
 if err != nil {
  return "", err
 }

 url := req.URL.String()[len("https://"):]

 return url, nil
}
```

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 Management Console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、分布式应用程序所需的性能和可扩展性。

PostgreSQL 兼容性亮点

Aurora DSQL 目前基于 PostgreSQL 版本 16。主要兼容性包括以下各项：

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 兼容性

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。

主要架构差异

Aurora DSQL 的分布式、无共享设计导致与传统的 PostgreSQL 具有一些基本差异。这些差异是 Aurora DSQL 架构不可或缺的一部分，并提供了许多性能和可扩展性优势。主要差异包括以下各项：

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

Aurora DSQL 和 PostgreSQL 对所有 SQL 查询返回完全相同的结果。请注意，Aurora DSQL 与没有 `ORDER BY` 子句的 PostgreSQL 不同。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
- Aurora DSQL 中不支持的 PostgreSQL 功能

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

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。	否

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	描述	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	描述	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	描述	Range	解析	存储大小	索引支持
`time[(p)][without time zone]`	`time`	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`	`time`	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`	`timetz`	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	描述	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	描述	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。

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	
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。 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	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 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
COMMIT	
BEGIN	[WORK | TRANSACTION] [READ ONLY | READ WRITE]

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

此 PostgreSQL 部分提供有关支持的表达式的详细信息，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。本节介绍 Aurora DSQL 对于这些命令支持的 PostgreSQL 语法元素。

主题

- CREATE TABLE
- ALTER TABLE
- 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 |
  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 [, ... ] ) ]
```

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
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

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 ]
```

描述

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，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。支持以下检查选项。
- 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。例如，直接在视图中调用 CURRENT_USER 将始终返回进行调用的用户，而不是视图所有者。这不受视图的 security_invoker 设置影响，因此将 security_invoker 设置为 false 的视图不等同于 SECURITY DEFINER 函数。
- 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。但请注意，只有在创建或替换视图时才会进行这种查找。因此，即使对于安全调用程序视图，视图的用户也只需要对包含视图的架构拥有 USAGE 权限，而不需要对视图查询中引用的架构拥有此权限。
- 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。

对于可自动更新的视图，系统会将视图上的任何 INSERT、UPDATE 或 DELETE 语句转换为底层基本关系上的相应语句。完全支持带有 ON CONFLICT UPDATE 子句的 INSERT 语句。

如果可自动更新的视图包含 WHERE 条件，则该条件将限制基本关系的哪些行可供视图上的 UPDATE 和 DELETE 语句修改。但是，UPDATE 可以更改某行以使其不再满足 WHERE 条件，从而使其在视图中不可见。同样，INSERT 命令可能会插入不满足 WHERE 条件的基本关系行，从而使它们在视图中不可见。ON CONFLICT UPDATE 可能同样会影响视图中不可见的现有行。

可以使用 CHECK OPTION 来防止 INSERT 和 UPDATE 命令创建在视图中不可见的行。

如果使用 security_barrier 属性标记了可自动更新的视图，则该视图的所有 WHERE 条件（以及任何使用标记为 LEAKPROOF 的运算符的条件）始终在视图用户已添加的任何条件之前进行评估。请注意，由于这一原因，最终未返回的行（因为它们没有通过用户的 WHERE 条件）可能最终仍会被锁定。可以使用 EXPLAIN 来查看哪些条件应用于关系级别（因此不会锁定行），哪些不适用。

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';
```

这将创建一个视图，其中包含在创建视图时位于 film 表中的列。虽然使用 * 来创建视图，但之后添加到表中的列不会成为视图的一部分。

使用 LOCAL CHECK OPTION 创建视图。

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

这将创建一个同时检查新行的 kind 和 classification 的视图。

创建一个混合了可更新和不可更新的列的视图。

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

此视图将支持 INSERT、UPDATE 和 DELETE。films 表中的所有列都将是可更新的，而计算列 country 和 avg_rating 将是只读的。

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

> **Note**
>
> 虽然递归视图的名称在此 CREATE 中是架构限定的，但其内部自引用不是架构限定的。这是因为隐式创建的公用表表达式（CTE）的名称不能由架构限定。

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 [, ... ] )
```

描述

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。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视图来完成的事情。

参数

ALTER VIEW 参数

name

现有视图的名称（可选择架构限定）。

column_name

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。视图的默认值将优先于基本关系中的任何默认值。

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] [, ...]), **RESET (view_option_name [, ...])**

设置或重置视图选项。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。该值必须是布尔值，例如 true 或 false。
- security_invoker (boolean)：更改视图的 security-barrier 属性。该值必须是布尔值，例如 true 或 false。

备注

出于 PostgreSQL 的历史原因，ALTER TABLE 也可以与视图一起使用；但视图中支持的 ALTER TABLE 的仅有变体与前面所示的变体等效。

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 ]
```

描述

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）。

Aurora DSQL 中不支持的 PostgreSQL 功能

Aurora DSQL 与 PostgreSQL 兼容。这意味着 Aurora DSQL 支持核心关系功能，例如 ACID 事务、二级索引、联接、插入和更新。有关支持的 SQL 功能的概述，请参阅支持的 SQL 表达式。

以下各节重点介绍 Aurora DSQL 中目前不支持哪些 PostgreSQL 功能。

不支持的对象

Aurora DSQL 不支持的对象包括：

- 单个 Aurora DSQL 集群上的多个数据库
- 临时表
- 触发
- 类型（部分支持）
- 表空间
- 用 SQL 以外的语言编写的函数
- 序列
- 分区

不支持的约束

- 外键
- 排他性约束

不受支持的命令

- `ALTER SYSTEM`
- `TRUNCATE`
- `SAVEPOINT`
- `VACUUM`

> **Note**
>
> Aurora DSQL 不需要 vacuum 操作。系统无需手动 vacuum 命令，即可自动维护统计数据并管理存储优化。

不支持的扩展

Aurora DSQL 不支持 PostgreSQL 扩展。下表显示了不受支持的扩展：

- PL/pgSQL
- PostGIS
- PGVector
- PGAudit
- Postgres_FDW
- PGCron
- pg_stat_statements

不支持的 SQL 表达式

下表描述了 Aurora DSQL 中不支持的子句。

类别	主要子句	不支持的子句
CREATE	INDEX ASYNC	ASC | DESC
CREATE	INDEX [1]	
TRUNCATE		
ALTER	SYSTEM	所有 ALTER SYSTEM 命令都被阻止。
CREATE	TABLE	COLLATE, AS SELECT, INHERITS, PARTITION

类别	主要子句	不支持的子句
CREATE	FUNCTION	LANGUAGE *non-sql-lang* ，其中 *non-sql-lang* 是除 SQL 之外的任何语言
CREATE	TEMPORARY	TABLES
CREATE	EXTENSION	
CREATE	SEQUENCE	
CREATE	MATERIALIZED	VIEW
CREATE	TABLESPACE	
CREATE	TRIGGER	
CREATE	TYPE	
CREATE	DATABASE	您无法创建其它数据库。

[1] 请参阅 Aurora DSQL 中的异步索引，以便在指定表的列上创建索引。

Aurora DSQL 有关 PostgreSQL 兼容性的注意事项

使用 Aurora DSQL 时，请考虑以下兼容性限制。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 使用单个名为 `postgres` 的内置数据库。您无法创建其它数据库，也无法重命名或删除 `postgres` 数据库。
- `postgres` 数据库使用 UTF-8 字符编码。您不能更改编码。
- 数据库仅使用 C 排序规则。
- Aurora DSQL 使用 UTC 作为系统时区。您无法使用参数或 SQL 语句（如 `SET TIMEZONE`）修改时区。
- PostgreSQL `Repeatable Read` 的事务隔离级别是固定的。
- 事务具有以下约束：
  - 事务不能混合 DDL 和 DML 操作

  - 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 目前不让您运行 `GRANT [permission] ON DATABASE`。如果您尝试运行该语句，Aurora DSQL 会返回错误消息 `ERROR: unsupported object type in GRANT`。
- Aurora DSQL 不让非管理员用户角色运行 `CREATE SCHEMA` 命令。您无法运行 `GRANT [permission] on DATABASE` 命令并授予对数据库的 `CREATE` 权限。如果非管理员用户角色尝试创建架构，Aurora DSQL 会返回错误消息 `ERROR: permission denied for database postgres`。
- 非管理员用户无法在公有架构中创建对象。只有管理员用户才能在公有架构中创建对象。管理员用户角色有权向非管理员用户授予对这些对象的读取、写入和修改权限，但不能授予对公有架构本身的 `CREATE` 权限。非管理员用户必须使用不同的、用户创建的架构来创建对象。
- Aurora DSQL 不支持命令 `ALTER ROLE [] CONNECTION LIMIT`。如果您需要提高连接限制，请联系 AWS 支持人员。
- Aurora DSQL 不支持 asyncpg，这是一款适用于 Python 的异步 PostgreSQL 数据库驱动程序。

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。

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
```

Aurora DSQL 返回与下面类似的响应。

```
           job_id           |   status   | details
----------------------------+------------+---------
 vs3kcl3rt5ddpk3a6xcq57cmcy | completed  |
 ihbyw2aoirfnrdfoc4ojnlamoq | processing |
```

状态列可以是以下值之一。

submitted	processing	failed	completed
任务已提交，但是 Aurora DSQL 尚未开始处理该任务。	Aurora DSQL 正在处理该任务。	任务失败。有关更多信息，请参阅详细信息列。如果 Aurora	Aurora DSQL

submitted	processing	failed	completed
		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。	

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。

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
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

名称	适用于 Aurora DSQL
pg_extension	否
pg_foreign_data_wrapper	否
pg_foreign_server	否
pg_foreign_table	否
pg_index	是
pg_inherits	是
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

名称	适用于 Aurora DSQL
pg_publication_rel	否
pg_range	是
pg_replication_origin	否
pg_rewrite	否
pg_seclabel	否
pg_sequence	否
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

名称	适用于 Aurora DSQL
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

名称	适用于 Aurora DSQL
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
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

名称	适用于 Aurora DSQL
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
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

名称	适用于 Aurora DSQL
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
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

名称	适用于 Aurora DSQL
pg_statio_sys_tables	否
pg_statio_user_indexes	否

sys.jobs 和 sys.iam_pg_role_mappings 视图

Aurora DSQL 支持以下系统视图：

sys.jobs

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 WHERE job_id = 'example_job_uuid';

        job_id        |   status   | details
----------------------+------------+---------
 example_job_uuid | processing |
(1 row)
```

sys.iam_pg_role_mappings

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;
```

pg_class 表

pg_class 表存储有关数据库对象的元数据。要获取表中行数的近似计数，请运行以下命令。

```
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
```

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。

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

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

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

本章介绍如何跨多个 AWS 区域配置和管理集群。

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 配置多区域集群

您可以使用相同的 AWS 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
```

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。

AWS CloudFormation

AWS::DSQL::Cluster 是一种 AWS 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。AWS 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 AWS 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。

Amazon Aurora DSQL SDK、驱动程序和示例代码

AWS 软件开发工具包（SDK）适用于许多常用编程语言。每个软件开发工具包都提供 API、代码示例和文档，使开发人员能够更轻松地以其首选语言构建应用程序。

适配器与方言

下表列出了 Aurora DSQL 可用的 ORM 适配器和数据库方言。

编程语言	框架	存储库链接
Java	Hibernate	https://github.com/awslabs/aurora-dsql-hibernate/
Python	Django	https://github.com/awslabs/aurora-dsql-django/
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-sqlalchemy/

代码示例

使用 AWS SDK 进行集群管理

下表显示了使用 AWS SDK 的不同编程语言的集群管理代码示例。

编程语言	存储库链接示例
C++	https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/cluster_management
C# (.NET)	https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/cluster_management
Go	https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/cluster_management
Java	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/cluster_management
JavaScript	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/cluster_management
Python	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/cluster_management

编程语言	存储库链接示例
Ruby	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/cluster_management
Rust	https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/cluster_management

驱动程序和对象关系映射（ORM）示例

下表显示了不同编程语言的数据库驱动程序和 ORM 框架代码示例。

编程语言	驱动程序或框架	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	Hibernate	https://github.com/awslabs/aurora-dsql-hibernate/tree/main/examples/pet-clinic-app
Java	pgJDBC	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc
JavaScript	node-postgres	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/node-postgres

编程语言	驱动程序或框架	存储库链接示例
JavaScript	Postgres.js	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js
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
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-sqlalchemy/tree/main/examples/pet-clinic-app
Ruby	Rails	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails
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
TypeScript	Sequelize	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize
TypeScript	TypeORM	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm

使用 AWS CLI 的 Aurora DSQL

请参阅以下各节，了解如何使用 AWS CLI 管理集群。

CreateCluster

要创建集群，请使用 `create-cluster` 命令。

> **Note**
>
> 集群创建是异步进行的。调用 `GetCluster` API，直到状态为 `ACTIVE`。一旦集群变为 `ACTIVE`，您就可以连接到该集群。

示例命令

```
aws dsql create-cluster --region us-east-1
```

> **Note**
>
> 如果要在创建时禁用删除保护，请添加 `--no-deletion-protection-enabled` 标志。

示例响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "CREATING",
    "creationTime": "2025-05-22T14:03:26.631000-07:00",
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    },
    "deletionProtectionEnabled": true
}
```

GetCluster

要描述集群，请使用 `get-cluster` 命令。

示例命令

```
aws dsql get-cluster \
  --region us-east-1 \
  --identifier <your_cluster_id>
```

示例响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2025-05-22T14:03:26.631000-07:00",
    "deletionProtectionEnabled": true,
    "tags": {},
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    }
}
```

UpdateCluster

要更新现有集群，请使用 `update-cluster` 命令。

> **Note**
>
> 更新是异步进行的。调用 `GetCluster` API 直到状态为 `ACTIVE`，您将观察到更改。

示例命令

```
aws dsql update-cluster \
  --region us-east-1 \
  --no-deletion-protection-enabled \
  --identifier your_cluster_id
```

示例响应

```
{
```

```
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "UPDATING",
    "creationTime": "2024-05-24T09:15:32.708000-07:00"
}
```

DeleteCluster

要删除现有集群，请使用 `delete-cluster` 命令。

> **Note**
>
> 您只能删除禁用了删除保护的集群。在创建新集群时，默认启用删除保护。

示例命令

```
aws dsql delete-cluster \
  --region us-east-1 \
  --identifier your_cluster_id
```

示例响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "DELETING",
    "creationTime": "2024-05-24T09:16:43.778000-07:00"
}
```

ListClusters

要获取集群的列表，请使用 `list-clusters` 命令。

示例命令

```
aws dsql list-clusters --region us-east-1
```

示例响应

```
{
    "clusters": [
        {
            "identifier": "abc0def1baz2quux3quux4quuux",
            "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
abc0def1baz2quux3quux4quuux"
        },
        {
            "identifier": "abc0def1baz2quux3quux4quuuux",
            "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
abc0def1baz2quux3quux4quuuux"
        }
    ]
}
```

多区域集群上的 GetCluster

要获取有关多区域集群的信息，请使用 `get-cluster` 命令。对于多区域集群，响应将包括关联的集群 ARN。

示例命令

```
aws dsql get-cluster \
  --region us-east-1 \
  --identifier your_cluster_id
```

示例响应

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2025-05-22T13:56:18.716000-07:00",
    "deletionProtectionEnabled": true,
    "multiRegionProperties": {
        "witnessRegion": "us-west-2",
        "clusters": [
            "arn:aws:dsql:us-east-1:842685632318:cluster/fuabuc7d3szkr37uqd5znkjynu"
        ]
    },
    "tags": {},
```

```
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    }
}
```

创建、读取、更新、删除 Aurora DSQL 集群

为单区域和多区域部署提供了创建、读取、更新、删除（CRUD）示例。对于每种编程语言都有一个专门的 `cluster_management` 章节，用于演示这些关键管理任务。

单区域部署非常适合为特定地理区域的用户提供服务的应用程序，可简化管理并减少延迟。多区域部署通过将数据库分布到多个 AWS 区域，来协助您实现更高的可用性和灾难恢复能力。

选择与应用程序对可用性、性能和地理分布的要求相一致的部署类型。

主题

创建集群

要了解如何在 Aurora DSQL 中创建单区域和多区域集群，请参阅以下信息。

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()
```

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={
```

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;
}
```

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);

    // We can only set the witness region for the first cluster
    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;

    // For the second cluster we can set witness region and designate cluster1 as a
peer
    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();
```

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 updateClusterCommand1 = new UpdateClusterCommand(
            {
                identifier: response1.identifier,
                multiRegionProperties: {
                    witnessRegion: witnessRegion,
                    clusters: [response2.arn]
                }
            }
        );

        await client1.send(updateClusterCommand1);
        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 1 ${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 2 ${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

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);
        }
    }
}
```

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

使用以下示例在单个 AWS 区域中创建集群。

```
use aws_config::load_defaults;
```

```
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};
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_cluster(region: &'static str) -> GetClusterOutput {
    let client = dsql_client(region).await;
    let tags = HashMap::from([(
        String::from("Name"),
        String::from("rust single region cluster"),
    )]);

    let create_cluster_output = client
        .create_cluster()
        .set_tags(Some(tags))
        .deletion_protection_enabled(true)
        .send()
        .await
        .unwrap();
    println!("Created {}", create_cluster_output.arn);
```

```
    println!("Waiting for cluster to become ACTIVE");
    client
        .wait_until_cluster_active()
        .identifier(&create_cluster_output.identifier)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap()
        .into_result()
        .unwrap()
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";
    let output = create_cluster(region).await;
    println!("{:#?}", output);
    Ok(())
}
```

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

使用以下示例在单个 AWS 区域中创建集群。

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

def create_cluster(region)
  client = Aws::DSQL::Client.new(region: region)
  cluster = client.create_cluster(
    deletion_protection_enabled: true,
    tags: {
      Name: "ruby single region cluster"
    }
  )
  puts "Created #{cluster.arn}"

  # The DSQL SDK offers built-in waiters to poll for a cluster's
  # transition to ACTIVE.
  puts "Waiting for cluster to become ACTIVE"
```

```
  client.wait_until(:cluster_active, 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 create cluster in #{region}: #{e.message}"
end

def main
  region = "us-east-1"
  cluster = create_cluster(region)
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

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__
```

.NET

使用以下示例在单个 AWS 区域中创建集群。

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

namespace DSQLExamples.examples
{
    public class CreateSingleRegionCluster
    {
        /// <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 a cluster without delete protection and a name.
        /// </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
                };

                CreateClusterResponse response = await
 client.CreateClusterAsync(createClusterRequest);
                Console.WriteLine($"Initiated creation of {response.Arn}");

                return response;
            }
        }

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

            await Create(region);
        }
    }
}
```

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}");
        }
    }
}
```

Golang

使用以下示例在单个 AWS 区域中创建集群。

```
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 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)
 }

 // Create a DSQL client
 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("error creating cluster: %w", err)
 }

 fmt.Printf("Created cluster: %s\n", *clusterProperties.Arn)

 // 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
 })

 id := clusterProperties.Identifier

 // Create the input for the clusterProperties
 getInput := &dsql.GetClusterInput{
  Identifier: id,
 }
```

```
 // Wait for the cluster to become active
 fmt.Println("Waiting for cluster to become ACTIVE")
 err = waiter.Wait(ctx, getInput, 5*time.Minute)
 if err != nil {
  return fmt.Errorf("error waiting for cluster to become active: %w", err)
 }

 fmt.Printf("Cluster %s is now active\n", *id)
 return nil
}

// Example usage in main function
func main() {

 region := "us-east-1"

 // Set up context with timeout
 ctx, cancel := context.WithTimeout(context.Background(), 6*time.Minute)
 defer cancel()

 if err := CreateCluster(ctx, region); err != nil {
  log.Fatalf("Failed to create cluster: %v", err)
 }
}
```

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)
 }

}
```

获取集群

要了解如何在 Aurora DSQL 中返回有关集群的信息，请参阅以下信息。

Python

要获取有关单区域或多区域集群的信息，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单区域或多区域集群的信息。

```
#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

要获取有关单区域或多区域集群的信息，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

以下示例可让您获取有关单区域或多区域集群的信息。

```
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

以下示例可让您获取有关单区域或多区域集群的信息。

```
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

以下示例可让您获取有关单区域或多区域集群的信息。

```
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

以下示例可让您获取有关单区域或多区域集群的信息。

```
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

以下示例可让您获取有关单区域或多区域集群的信息。

```
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

要了解如何在 Aurora DSQL 中更新集群，请参阅以下信息。更新集群可能需要一两分钟的时间。我们建议您稍等片刻，然后运行 get cluster 以获取集群的状态。

Python

要更新单区域或多区域集群，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单区域或多区域集群。

```
#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

要更新单区域或多区域集群，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单区域或多区域集群。

```
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单区域或多区域集群。

```
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单区域或多区域集群。

```
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单区域或多区域集群。

```
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单区域或多区域集群。

```
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

要了解如何在 Aurora DSQL 中删除集群，请参阅以下信息。

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()
```

要删除多区域集群，请使用以下示例。

```
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;
}
```

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();
```

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);
        }
    }
}
```

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(())
}
```

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

要在单个 AWS 区域中删除集群，请使用以下示例。

```
equire "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__
```

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);
        }
    }
}
```

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)
 }
```

```
}
```

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)
 }
}
```

教程

GitHub 上的以下教程和示例代码有助于您在 Aurora DSQL 中执行常见任务。

- Using Benchbase with Aurora DSQL：Benchbase 开源基准测试实用程序的一个分支，经验证可与 Aurora DSQL 结合使用。
- Aurora DSQL loader：这个开源 Python 脚本可让您更轻松地根据自己的用例将数据加载到 Aurora DSQL 中，例如填充用于测试的表或将数据传输到 Aurora DSQL 中。
- Aurora DSQL samples：GitHub 上的 `aws-samples/aurora-dsql-samples` 存储库包含代码示例，这些代码示例介绍了如何在各种编程语言中使用 AWS SDK、对象关系映射器（ORM）和 Web 框架来连接和使用 Aurora DSQL。这些示例演示了如何执行常见任务，例如安装客户端、处理身份验证和执行 CRUD 操作。

将 AWS Lambda 与 Amazon Aurora DSQL 结合使用

以下教程介绍了如何将 Lambda 与 Aurora DSQL 结合使用

先决条件

- 用于创建 Lambda 函数的授权。有关更多信息，请参阅 Lambda 入门。
- 用于创建或修改由 Lambda 创建的 IAM 策略的授权。您需要权限 `iam:CreatePolicy` 和 `iam:AttachRolePolicy`。有关更多信息，请参阅 Actions, resources, and condition keys for IAM。
- 您必须安装了 npm v8.5.3 或更高版本。
- 您必须安装了 zip v3.0 或更高版本。

在 AWS Lambda 中创建新的函数。

1. 通过以下网址登录 AWS Management Console 并打开 AWS Lambda 控制台：https://console.aws.amazon.com/lambda/。
2. 选择创建函数。
3. 提供名称，例如 `dsql-sample`。
4. 请勿编辑默认设置，以确保 Lambda 创建具有基本 Lambda 权限的新角色。
5. 选择创建函数。

授权您的 Lambda 执行角色连接到集群

1. 在 Lambda 函数中，选择配置 > 权限。
2. 选择角色名称以在 IAM 控制台中打开执行角色。
3. 选择添加权限 > 创建内联策略并使用 JSON 编辑器。
4. 在操作中，粘贴以下操作，以授权您的 IAM 身份使用管理员数据库角色进行连接。

```
"Action": ["dsql:DbConnectAdmin"],
```

> **Note**
>
> 我们使用管理员角色来尽量减少入门的先决条件步骤。不应为生产应用程序使用管理员数据库角色。要了解如何使用对数据库具有最少权限的授权来创建自定义数据库角色，请参阅使用数据库角色和 IAM 身份验证。

5. 在资源中，添加集群的 Amazon 资源名称（ARN）。还可以使用通配符。

```
"Resource": ["*"]
```

6. 选择下一步。
7. 输入策略的名称，例如 dsql-sample-dbconnect。
8. 选择创建策略。

创建一个要上传到 Lambda 的包。

1. 创建名为 myfunction 的文件夹。
2. 在该文件夹中，创建一个名为 package.json 的新文件，其中包含以下内容。

```
{
  "dependencies": {
    "@aws-sdk/dsql-signer": "^3.705.0",
    "assert": "2.1.0",
    "pg": "^8.13.1"
  }
}
```

3. 在该文件夹中，在目录中创建一个名为 index.mjs 的文件，其中包含以下内容。

```
import { DsqlSigner } from "@aws-sdk/dsql-signer";
import pg from "pg";
import assert from "node:assert";
const { Client } = pg;


async function dsql_sample(clusterEndpoint, region) {
  let client;
  try {
    // The token expiration time is optional, and the default value 900 seconds
    const signer = new DsqlSigner({
      hostname: clusterEndpoint,
      region,
    });
    const token = await signer.getDbConnectAdminAuthToken();
    // <https://node-postgres.com/apis/client>
    // By default `rejectUnauthorized` is true in TLS options
    // <https://nodejs.org/api/tls.html#tls_tls_connect_options_callback>
```

```
    // The config does not offer any specific parameter to set sslmode to verify-
full
    // Settings are controlled either via connection string or by setting
    // rejectUnauthorized to false in ssl options
    client = new Client({
      host: clusterEndpoint,
      user: "admin",
      password: token,
      database: "postgres",
      port: 5432,
      // <https://node-postgres.com/announcements> for version 8.0
      ssl: true,
      rejectUnauthorized: false
    });

    // Connect
    await client.connect();

    // Create a new table
    await client.query(`CREATE TABLE IF NOT EXISTS owner (
      id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
      name VARCHAR(30) NOT NULL,
      city VARCHAR(80) NOT NULL,
      telephone VARCHAR(20)
    )`);

    // Insert some data
    await client.query("INSERT INTO owner(name, city, telephone) VALUES($1, $2,
 $3)",
      ["John Doe", "Anytown", "555-555-1900"]
    );

    // Check that data is inserted by reading it back
    const result = await client.query("SELECT id, city FROM owner where name='John
 Doe'");
    assert.deepEqual(result.rows[0].city, "Anytown")
    assert.notEqual(result.rows[0].id, null)

    await client.query("DELETE FROM owner where name='John Doe'");

  } catch (error) {
    console.error(error);
    throw new Error("Failed to connect to the database");
  } finally {
```

```
    client?.end();
  }
  Promise.resolve();
}


// https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html
export const handler = async (event) => {
  const endpoint = event.endpoint;
  const region = event.region;
  const responseCode = await dsql_sample(endpoint, region);

  const response = {
    statusCode: responseCode,
    endpoint: endpoint,
  };
  return response;
};
```

4. 使用以下命令来创建包。

```
npm install
zip -r pkg.zip .
```

上传代码包并测试 Lambda 函数

1. 在 Lambda 函数的代码选项卡中，选择上传自 > .zip 文件
2. 上传您创建的 `pkg.zip`。有关更多信息，请参阅使用 .zip 文件归档部署 Node.js Lambda 函数。
3. 在 Lambda 函数的测试选项卡中，粘贴以下 JSON 有效载荷，然后对其进行修改以使用您的集群 ID。
4. 在 Lambda 函数的测试选项卡中，使用以下修改过的事件 JSON 来指定集群的端点。

```
{"endpoint": "replace_with_your_cluster_endpoint"}
```

5. 输入事件名称，例如 `dsql-sample-test`。选择保存。
6. 选择测试。
7. 选择详细信息以展开执行响应和日志输出。
8. 如果成功，Lambda 函数执行响应应返回 200 状态代码。

```
{"statusCode": 200, "endpoint": "your_cluster_endpoint"}
```

如果数据库返回错误或数据库连接失败，Lambda 函数执行响应将返回 500 状态代码。

```
{"statusCode": 500,"endpoint": "your_cluster_endpoint"}
```

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 Management Console、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，创建自定义的控制平面，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》。
- 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》。

查看 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 Management Console、AWS CLI 或 Aurora DSQL API 查看集群和集群实例的状态。

Aurora DSQL 集群状态和定义

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。

状态	描述
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。
活跃	集群正在运行，可供使用。
Idle (空闲)	当集群空闲的时间足够长，致使 Aurora DSQL 可以回收为其配置的资源时，集群就变为空闲状态。当您连接到空闲的集群时，Aurora DSQL 会将集群转换回活跃状态。
非活跃	当集群上长时间没有活动时，集群将变为非活跃状态。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。
正在更新	当您更改集群配置时，集群会变为正在更新状态。

状态	描述
Deleting	当您提交删除集群请求时，集群会变为正在删除状态。
Deleted	已成功删除了集群。
已失败	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。一旦您删除最后一个对等集群，多区域集群就会变为正在删除状态。

查看 Aurora DSQL 集群状态

要查看集群的状态，请使用 AWS Management Console、AWS CLI 或 Aurora DSQL API。

控制台

按照以下步骤在 AWS Management Console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 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, including read-only transactions.

CloudWatch 指标名称	指标	单位	描述
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-use component of your Aurora

CloudWatch 指标名称	指标	维度：ResourceId	单位	描述
				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.
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 Management Console 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 合规性计划 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

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场景提供权限，以便您可以开始为用户、组和角色分配权限。

请记住，AWS 托管策略可能不会为您的特定使用场景授予最低权限，因为它们可供所有 AWS 客户使用。我们建议通过定义特定于您的使用场景的客户管理型策略来进一步减少权限。

您无法更改 AWS 托管策略中定义的权限。如果 AWS 更新在 AWS 托管策略中定义的权限，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 托管策略。

有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管策略。

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。这些主体可以在集群中添加和移除标签。这些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。这些主体能够以任何用户（包括管理员）身份连接到数据库。这些主体可以对 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。策略包括 AWS KMS 权限，这些权限支持对用于集群加密的客户自主管理型密钥执行操作。这些主体可以在您的账户中查看来自 CloudWatch 的任何指标。这些主体还有权限为 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。

您可以在 IAM 控制台上找到 `AmazonAuroraDSQLFullAccess` 策略，并在《AWS Managed Policy Reference Guide》中找到 AmazonAuroraDSQLFullAccess。

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。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 控制台上找到 `AmazonAuroraDSQLReadOnlyAccess` 策略，并在《AWS Managed Policy Reference Guide》中找到 AmazonAuroraDSQLReadOnlyAccess。

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 Management Console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。这些主体可以列出集群并查看有关各个集群的信息。这些主体可以看到您账户中任何资源的标签。这些主体能够以任何用户（包括管理员）身份连接到数据库。这些主体可以对 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。策略包括 AWS KMS 权限，这些权限支持对用于集群加密的客户自主管理型密钥执行操作。这些主体可以从 AWS Management Console启动 AWS

CloudShell。这些主体可以在您的账户中查看来自 CloudWatch 的任何指标。这些主体还有权限为 `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 Management Console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。

您可以在 IAM 控制台上找到 `AmazonAuroraDSQLReadOnlyAccess` 策略，并在《AWS Managed Policy Reference Guide》中找到 AmazonAuroraDSQLReadOnlyAccess。

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 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
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	2025 年 5 月 13 日

更改	描述	日期
	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。	
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

更改	描述	日期
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
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 URL 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。

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？
- AWS Direct Connect 连接。有关更多信息，请参阅什么是 AWS 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。

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 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。

> **Note**
>
> 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 客户托管密钥。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
- 客户托管密钥

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。

客户托管密钥

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

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。
- 向 AWS KMS key管理员（可以代入 `db-team` 角色的用户）授予对 AWS KMS key的只读访问权限

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 Management Console或 AWS CLI 创建加密集群。

Console

在 AWS Management Console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 Management Console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 Management Console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 Management Console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 Management Console、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 AWS Identity and Access Management（IAM）的方式因您可以在 Aurora DSQL 中执行的工作而异。

服务用户：如果您使用 Aurora DSQL 服务来执行工作，则管理员会为您提供您所需的凭证和权限。当您使用更多 Aurora DSQL 功能来完成工作时，您可能需要额外的权限。了解如何管理访问权限有助于您向管理员请求适合的权限。如果您无法访问 Aurora DSQL 中的功能，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：如果您在公司负责 Aurora DSQL 资源，则您可能具有 Aurora DSQL 的完全访问权限。您有责任确定您的服务用户应访问哪些 Aurora DSQL 功能和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要了解有关贵公司如何将 IAM 与 Aurora DSQL 结合使用的更多信息，请参阅 Amazon Aurora DSQL 如何与 IAM 协同工作。

IAM 管理员：如果您是 IAM 管理员，您可能希望了解有关如何编写策略以管理对 Aurora DSQL 的访问权限的详细信息。要查看您可在 IAM 中使用的 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。

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 用户或通过代入 IAM 角色进行身份验证（登录到 AWS）。

您可以使用通过身份源提供的凭证以联合身份登录到 AWS。AWS IAM Identity Center（IAM Identity Center）用户、您公司的单点登录身份验证以及您的 Google 或 Facebook 凭证都是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当您使用联合身份验证访问 AWS 时，您就是在间接代入角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录到 AWS 的更多信息，请参阅《AWS 登录 User Guide》中的 How to sign in to your AWS 账户。

如果您以编程方式访问 AWS，则 AWS 将提供软件开发工具包（SDK）和命令行界面（CLI），以便使用您的凭证以加密方式签署您的请求。如果您不使用 AWS 工具，则必须自行对请求签名。有关使用推荐的方法自行签署请求的更多信息，请参阅《IAM 用户指南》中的用于签署 API 请求的 AWS 签名版本 4。

无论使用何种身份验证方法，您都可能需要提供其他安全信息。例如，AWS 建议您使用多重身份验证（MFA）来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的多重身份验证和《IAM 用户指南》中的 IAM 中的 AWS 多重身份验证。

AWS 账户根用户

当您创建 AWS 账户时，最初使用的是一个对账户中所有 AWS 服务和资源拥有完全访问权限的登录身份。此身份称为 AWS 账户根用户，使用您创建账户时所用的电子邮件地址和密码登录，即可获得该身

份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关要求您以根用户身份登录的任务的完整列表，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，要求人类用户（包括需要管理员访问权限的用户）结合使用联合身份验证和身份提供程序，以使用临时凭证来访问 AWS 服务。

联合身份是来自企业用户目录、Web 身份提供程序、AWS Directory Service、Identity Center 目录的用户，或任何使用通过身份源提供的凭证来访问 AWS 服务的用户。当联合身份访问 AWS 账户时，他们代入角色，而角色提供临时凭证。

要集中管理访问权限，建议您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中创建用户和组，也可以连接并同步到您自己的身份源中的一组用户和组以跨所有 AWS 账户和应用程序使用。有关 IAM Identity Center 的信息，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 AWS 账户内对某个人员或应用程序具有特定权限的一个身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的对于需要长期凭证的用例，应在需要时更新访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅《IAM 用户指南》中的 IAM 用户的使用案例。

IAM 角色

IAM 角色是 AWS 账户中具有特定权限的身份。它类似于 IAM 用户，但与特定人员不关联。要在 AWS Management Console中临时代入 IAM 角色，可以从用户切换到 IAM 角色（控制台）。您可以调用 AWS CLI 或 AWS API 操作或使用自定义网址以代入角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的代入角色的方法。

具有临时凭证的 IAM 角色在以下情况下很有用：

- 联合用户访问：要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关用于联合身份验证的角色的信息，请参阅《IAM 用户指南》中的针对第三方身份提供商创建角色（联合身份验证）。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。
- 临时 IAM 用户权限：IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。
- 跨账户存取：您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些 AWS 服务，您可以将策略直接附加到资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅 IAM 用户指南中的 IAM 中的跨账户资源访问。
- 跨服务访问：某些 AWS 服务使用其他 AWS 服务中的特征。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon EC2 中运行应用程序或在 Simple Storage Service（Amazon S3）中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 转发访问会话（FAS）：当您使用 IAM 用户或角色在 AWS 中执行操作时，您将被视为主体。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用主体调用 AWS 服务的权限，结合请求的 AWS 服务，向下游服务发出请求。只有在服务收到需要与其他 AWS 服务或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两项操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。
  - 服务角色 - 服务角色是服务代表您在您的账户中执行操作而分派的 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。
  - 服务相关角色：服务相关角色是与 AWS 服务 关联的一种服务角色。服务可以代入代表您执行操作的角色。服务相关角色显示在您的 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相关角色的权限。
- 在 Amazon EC2 上运行的应用程序：您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时凭证。这优先于在 EC2 实例中存储访问密钥。要将 AWS 角色分配给 EC2 实例并使其对该实例的所有应用程序可用，您可以创建一个附加到实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的使用 IAM 角色向在 Amazon EC2 实例上运行的应用程序授予权限。

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。策略是 AWS 中的对象；在与身份或资源相关联时，策略定义它们的权限。在主体（用户、根用户或角色会话）发出请求时，AWS

将评估这些策略。策略中的权限确定是允许还是拒绝请求。大多数策略在 AWS 中存储为 JSON 文档。有关 JSON 策略文档的结构和内容的更多信息，请参阅 IAM 用户指南中的 JSON 策略概览。

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。

默认情况下，用户和角色没有权限。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。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

IAM 策略定义操作的权限，无关乎您使用哪种方法执行操作。例如，假设您有一个允许 `iam:GetRole` 操作的策略。具有该策略的用户可以从 AWS Management Console、AWS CLI 或 AWS API 获取角色信息。

基于身份的策略

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》中的使用客户托管策略定义自定义 IAM 权限。

基于身份的策略可以进一步归类为内联策略或托管式策略。内联策略直接嵌入单个用户、组或角色中。托管策略是可以附加到 AWS 账户 中的多个用户、组和角色的独立策略。托管式策略包括 AWS 托管式策略和客户托管策略。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。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 AWS 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。

访问控制列表（ACL）

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，尽管它们不使用 JSON 策略文档格式。

Amazon S3、AWS WAF 和 Amazon VPC 是支持 ACL 的服务示例。要了解有关 ACL 的更多信息，请参阅《Amazon Simple Storage Service 开发人员指南》中的访问控制列表（ACL）概览。

其他策略类型

AWS 支持额外的、不太常用的策略类型。这些策略类型可以设置更常用的策略类型向您授予的最大权限。

- 权限边界：权限边界是一个高级特征，用于设置基于身份的策略可以为 IAM 实体（IAM 用户或角色）授予的最大权限。您可为实体设置权限边界。这些结果权限是实体基于身份的策略及其权限边界的交集。在 `Principal` 中指定用户或角色的基于资源的策略不受权限边界限制。任一项策略中的显式拒绝将覆盖允许。有关权限边界的更多信息，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）– SCP 是 JSON 策略，指定了组织或组织单元（OU）在 AWS Organizations 中的最大权限。AWS Organizations 服务可以分组和集中管理您的企业拥有的多个 AWS 账户 账户。如果在组织内启用了所有特征，则可对任意或全部账户应用服务控制策略（SCP）。SCP 限制成员账户中实体（包括每个 AWS 账户根用户）的权限。有关组织和 SCP 的更多信息，请参阅《AWS Organizations 用户指南》中的服务控制策略。
- 资源控制策略（RCP）– RCP 是 JSON 策略，您可以使用它们设置账户中资源的最大可用权限，而无需更新附加到您拥有的每个资源的 IAM 策略。RCP 限制了成员账户中资源的权限，并可能影响身份（包括 AWS 账户根用户）的有效权限，无论这些身份是否属于您的组织。有关 Organizations 和 RCP（包括支持 RCP 的 AWS 服务列表）的更多信息，请参阅《AWS Organizations User Guide》中的 Resource control policies (RCPs)。
- 会话策略：会话策略是当您以编程方式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。结果会话的权限是用户或角色的基于身份的策略和会话策略的交集。权限也可以来自基于资源的策略。任一项策略中的显式拒绝将覆盖允许。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	否
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
服务相关角色	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。您无法在基于身份的策略中指定主体，因为它适用于其附加的用户或角色。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：否

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 AWS 服务。

要启用跨账户访问，您可以将整个账户或其他账户中的 IAM 实体指定为基于资源的策略中的主体。将跨账户主体添加到基于资源的策略只是建立信任关系工作的一半而已。当主体和资源处于不同的 AWS 账户 中时，则信任账户中的 IAM 管理员还必须授予主体实体（用户或角色）对资源的访问权限。他们通过将基于身份的策略附加到实体以授予权限。但是，如果基于资源的策略向同一个账户中的主体授予访问权限，则不需要额外的基于身份的策略。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。

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。策略操作通常与关联的 AWS API 操作同名。有一些例外情况，例如没有匹配 API 操作的仅限权限 操作。还有一些操作需要在策略中执行多个操作。这些附加操作称为相关操作。

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。语句必须包含 `Resource` 或 `NotResource` 元素。作为最佳实践，请使用其 Amazon 资源名称（ARN）指定资源。对于支持特定资源类型（称为资源级权限）的操作，您可以执行此操作。

对于不支持资源级权限的操作（如列出操作），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。

在 Condition 元素（或 Condition 块）中，可以指定语句生效的条件。Condition 元素是可选的。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。

如果您在一个语句中指定多个 Condition 元素，或在单个 Condition 元素中指定多个键，则 AWS 使用逻辑 AND 运算评估它们。如果您为单个条件键指定多个值，则 AWS 使用逻辑 OR 运算来评估条件。在授予语句的权限之前必须满足所有的条件。

在指定条件时，您也可以使用占位符变量。例如，只有在使用 IAM 用户名标记 IAM 用户时，您才能为其授予访问资源的权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM 策略元素：变量和标签。

AWS 支持全局条件键和特定于服务的条件键。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

基于属性的访问控制（ABAC）是一种授权策略，该策略基于属性来定义权限。在 AWS 中，这些属性称为标签。您可以将标签附加到 IAM 实体（用户或角色）以及 AWS 资源。标记实体和资源是 ABAC 的第一步。然后设计 ABAC 策略，以在主体的标签与他们尝试访问的资源标签匹配时允许操作。

ABAC 在快速增长的环境中非常有用，并在策略管理变得繁琐的情况下可以提供帮助。

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

某些 AWS 服务 在您使用临时凭证登录时无法正常工作。有关更多信息，包括 AWS 服务 与临时凭证配合使用，请参阅 IAM 用户指南中的使用 IAM 的 AWS 服务。

如果您不使用用户名和密码而用其他方法登录到 AWS Management Console，可使用临时凭证。例如，当您使用贵公司的单点登录（SSO）链接访问 AWS 时，该过程将自动创建临时凭证。当您以用户身份登录控制台，然后切换角色时，您还会自动创建临时凭证。有关切换角色的更多信息，请参阅《IAM 用户指南》中的从用户切换到 IAM 角色（控制台）。

您可以使用 AWS CLI 或者 AWS API 创建临时凭证。之后，您可以使用这些临时凭证访问 AWS。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 中的临时安全凭证。

Aurora DSQL 的跨服务主体权限

支持转发访问会话（FAS）：是

当您使用 IAM 用户或角色在 AWS 中执行操作时，您将被视为主体。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用主体调用 AWS 服务的权限，结合请求的 AWS 服务，向下游服务发出请求。只有在服务收到需要与其他 AWS 服务或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两项操作的权限。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

支持服务相关角色：是

服务相关角色是一种与 AWS 服务 相关的服务角色。服务可以代入代表您执行操作的角色。服务相关角色显示在您的 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相关角色的权限。

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。他们也无法使用 AWS Management Console、AWS Command Line Interface (AWS CLI) 或 AWS API 执行任务。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。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

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

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。我们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 托管式策略或工作职能的 AWS 托管式策略。

- 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 AWS 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": "*"
        }
    ]
}
```

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相关角色。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。

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他服务的信息，请参阅与 IAM 配合使用的 AWS 服务，并查找 Service-Linked Role（服务相关角色）列中显示为 Yes（是）的服务。选择是和链接，查看该服务的服务相关角色文档。

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、编辑或删除服务相关角色。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": [
        {
            "Action": ["dsql:CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

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相关角色的特征或服务，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 Security Incident Response Guide》
- 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，请参阅合规性计划范围内的 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、。

您使用 AWS 服务的合规性责任取决于您数据的敏感度、贵公司的合规性目标以及适用的法律法规。AWS 提供以下资源来帮助满足合规性：

- Security Compliance & Governance：这些解决方案实施指南讨论了架构考虑因素，并提供了部署安全性和合规性功能的步骤。
- 符合 HIPAA 要求的服务参考：列出符合 HIPAA 要求的服务。并非所有 AWS 服务 都符合 HIPAA 要求。
- AWS 合规性资源：此业务手册和指南集合可能适用于您的行业和位置。
- AWS 客户合规指南：从合规角度了解责任共担模式。这些指南总结了保护 AWS 服务 的最佳实践，并将指南映射到跨多个框架的安全控制，包括美国国家标准与技术研究院（NIST）、支付卡行业安全标准委员会（PCI）和国际标准化组织（ISO）。
- AWS Config 开发人员指南中的使用规则评估资源 - 此 AWS Config 服务评测您的资源配置对内部实践、行业指南和法规的遵循情况。
- AWS Security Hub：此 AWS 服务 向您提供 AWS 中安全状态的全面视图。Security Hub 通过安全控制措施评估您的 AWS 资源并检查其是否符合安全行业标准和最佳实践。有关受支持服务及控制措施的列表，请参阅 Security Hub 控制措施参考。
- Amazon GuardDuty：该 AWS 服务 通过监控您的环境中是否存在可疑和恶意活动，来检测您的 AWS 账户、工作负载、容器和数据面临的潜在威胁。GuardDuty 可以通过满足某些合规性框架规定的入侵检测要求，来协助您满足各种合规性要求，如 PCI DSS。
- AWS Audit Manager：此 AWS 服务 可帮助您持续审核您的 AWS 使用情况，以简化管理风险以及与相关法规和行业标准的合规性的方式。

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结构相比，可用区具有更高的可用性、容错性和可扩展性。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 全球基础结构。

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。

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。

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（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 AWS 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 AWS 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 Guide》中的 Access an AWS service using an interface VPC endpoint 和 AWS PrivateLink quotas。

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.dynamodb.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-xxxxxxxxxxxxxxxx \
    --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
```

响应示例

```
{
    "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);
```

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. 这些区域有：AWS 区域

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
```

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。

事务	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 Management Console、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。

- Aurora DSQL 不会在大型表的事务超时之前完成 `COUNT(*)` 操作。要从系统目录中检索表行计数，请参阅 Using systems tables and commands in Aurora DSQL。
- 调用 `PG_PREPARED_STATEMENTS` 的驱动程序可能会为集群提供不一致的缓存预处理语句视图。对于同一个集群和 IAM 角色，您看到的每个连接的预处理语句数量可能会超过预期。Aurora DSQL 不会保留您预处理的语句名称。
- 在罕见的多区域关联集群损坏情景中，恢复事务提交可用性所需的时间可能比预期的要更长。通常，自动集群恢复操作可能会导致暂时并发控制或连接错误。在大多数情况下，您将只会看到一定百分比的工作负载受到影响。当您看到这些传输错误时，请重试事务或重新连接您的客户端。
- 某些 SQL 客户端（例如 Datagrip）会广泛调用系统元数据来填充架构信息。Aurora DSQL 并不支持所有这些信息，并返回错误。此问题不影响 SQL 查询功能，但可能会影响架构显示。
- 管理员角色拥有一组与数据库管理任务相关的权限。默认情况下，这些权限不会扩展到其他用户创建的对象。管理员角色无法向其他用户授予或撤销对这些用户创建的对象的权限。管理员用户可以向自己授予任何其它角色，来获得对这些对象的必要权限。

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。

描述	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 128 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

描述	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。

描述	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

描述	默认限制	是否可配置？	Aurora DSQL 错误代码	错误消息
事务块中可以突变的表和索引行的最大数量	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

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 Management Console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

error: unrecognized SSL error code: 6

原因：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。

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 PostgreSQL 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 PostgreSQL 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: SELECT FOR UPDATE in a read-only transaction is a no-op

您正在尝试只读事务中不支持的操作。要了解更多信息，请参阅了解 Aurora DSQL 中的并发控制。

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”

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。

《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
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	2025 年 5 月 21 日

	和使用 Aurora DSQL 中的服务相关角色。	
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
适用于 Amazon Aurora DSQL 的 AWS PrivateLink	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