aws

User Guide

Amazon Aurora DSQL



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

Amazon Aurora DSQL: User Guide

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

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

Table of Contents

What is Amazon Aurora DSQL? ... 1
When to use ... 1
Key features ... 1
AWS Region availability ... 2
Multi-Region clusters ... 5
Pricing ... 6
What's next? ... 6
Getting started ... 8
Prerequisites ... 8
Create a single-Region cluster ... 8
Connect to a cluster ... 9
Run SQL commands ... 10
Create a multi-Region cluster ... 11
Troubleshooting ... 12
How billing works ... 13
How metering works ... 13
DPU component metering ... 14
ComputeDPU ... 14
WriteDPU ... 14
ReadDPU ... 15
Stream DPU ... 16
Billing examples ... 16
Multi-Region billing ... 21
Monitoring DPU usage ... 22
Available DPU metrics ... 22
Viewing DPU metrics ... 22
Additional observability metrics ... 23
Finding high-load queries ... 23
EXPLAIN ANALYZE VERBOSE for cost awareness ... 24
Example 1: SELECT query ... 25
Example 2: INSERT query ... 26
Using DPU information for optimization ... 27
Interpreting DPU information ... 27
Cost estimation best practices ... 28

Authentication and authorization 29
Managing your cluster 29
Connecting to your cluster 29
PostgreSQL and IAM roles 30
Using IAM policy actions with Aurora DSQL 31
Using IAM policy actions to connect to clusters 31
Using IAM policy actions to manage clusters 32
Revoking authorization using IAM and PostgreSQL 33
Generate an authentication token 34
Console 34
AWS CloudShell 35
AWS CLI 36
Aurora DSQL SDKs 37
Database roles and IAM authentication 46
IAM roles 46
IAM users 47
Connect 47
Query 47
View mappings 48
Revoke 48
Aurora DSQL and PostgreSQL 50
Compatibility highlights 50
Distributed architecture benefits 51
SQL compatibility 51
Supported data types 52
Supported SQL features 58
Supported subsets of SQL commands 63
Migration guide 116
Concurrency control 123
Transaction snapshots 123
Concurrency control responses 123
Data conflict types 124
Guidelines for optimizing transaction performance 125
DDL and distributed transactions 126
Primary keys 127
Data structure and storage 128

Guidelines for choosing a primary key ... 128
Sequences and identity columns ... 129
Sequence manipulation functions ... 129
Identity columns ... 132
Working with sequences and identity columns ... 133
Foreign key constraints ... 135
How Aurora DSQL maintains referential integrity ... 135
Foreign key example scenarios ... 136
Best practices with foreign keys in Aurora DSQL ... 137
Asynchronous indexes ... 138
Usage notes ... 139
Creating an index ... 139
Querying an index ... 140
Unique index build failures ... 141
Uniqueness violations ... 142
System tables and commands ... 144
System tables ... 144
Useful system queries ... 154
The ANALYZE command ... 155
EXPLAIN plans ... 156
PostgreSQL EXPLAIN plans ... 156
Key elements ... 157
Filtering ... 158
Reading EXPLAIN plans ... 159
Batched nested-loop joins ... 162
DPUs in EXPLAIN ANALYZE ... 165
Managing Aurora DSQL clusters ... 169
Single-Region clusters ... 169
Using AWS SDKs ... 169
Using AWS CLI ... 208
Multi-Region clusters ... 211
Using AWS SDKs ... 211
Using AWS CLI ... 265
CloudFormation ... 272
Initial configuration ... 272
Find clusters ... 272

Update configuration ... 273
Aurora DSQL Cluster lifecycle ... 273
Cluster statuses ... 274
Viewing cluster statuses ... 276
Programming with Aurora DSQL ... 278
Connectors ... 278
JDBC connector ... 279
Python connector ... 283
Go connector ... 295
Node.js connectors ... 302
Ruby connector ... 310
PHP connector ... 316
.NET connector ... 320
Rust connector ... 325
Adapters and dialects ... 332
Hibernate dialect ... 332
Django adapter ... 345
SQLAlchemy dialect ... 356
Tortoise ORM adapter ... 368
Accessing Aurora DSQL ... 378
Session parameters ... 380
SQL clients ... 382
DBeaver ... 382
JetBrains DataGrip ... 386
Psql ... 387
VSCode ... 388
Troubleshooting ... 390
Database Connectivity Tools ... 390
Aurora DSQL Adapters ... 390
Database Driver Samples ... 391
ORM and Framework Samples ... 392
Loading data ... 393
Choosing a loading approach ... 393
Aurora DSQL Loader ... 394
Migration pathways ... 399
Using PostgreSQL \copy ... 401

Additional resources ... 402
Generative AI ... 402
AWS Labs Aurora DSQL MCP Server ... 403
Aurora DSQL steering: Plugins, skills, and powers ... 411
Query Editor ... 415
Prerequisites ... 415
Working with the Query Editor ... 415
Query Editors: Using JupyterLab with Aurora DSQL ... 417
Getting started ... 418
Example notebook ... 420
Further reading ... 420
Backup and restore ... 421
Getting started with AWS Backup ... 421
Restoring your backups ... 421
Restoring single-Region clusters ... 422
Restoring multi-Region clusters ... 422
Monitoring and compliance ... 422
Additional resources ... 423
CDC streams ... 424
How it works ... 424
Topics on this page ... 425
Related topics ... 425
Ordering and delivery semantics ... 425
Delivery guarantees ... 425
Ordering ... 425
Consumer strategies ... 426
Multi-Region CDC stream configuration ... 427
Processing CDC records downstream ... 427
Getting started ... 428
Prerequisites ... 428
Step 1: Create an Amazon Kinesis data stream ... 429
Step 2: Create an IAM role for Aurora DSQL ... 430
Step 3: Create the CDC stream ... 433
Step 4: Verify that records are flowing ... 434
Step 5: Consume records with a Python script ... 435
Managing CDC streams ... 439

Configuring IAM ... 439
Caller permissions ... 440
Service role ... 441
Service role trust policy ... 441
Service role permissions policy ... 442
Data protection ... 444
Understanding CDC records ... 444
CDC envelope versioning ... 444
How records map to Amazon Kinesis ... 445
Primary key in the payload ... 445
Record payload ... 445
Payload fields ... 448
Format details ... 450
Write-set compaction ... 451
Oversized records ... 452
Data type serialization ... 454
Schema evolution in CDC records ... 459
Monitoring streams ... 459
Stream lifecycle ... 460
Troubleshooting an impaired or failed stream ... 460
Error code reference ... 461
Recovering an impaired stream ... 465
Monitoring stream health ... 466
CloudWatch metrics for CDC streams ... 466
Monitoring best practices ... 468
Monitoring and logging ... 470
Monitoring with CloudWatch ... 470
Observability ... 471
Usage ... 473
CDC streams ... 475
Cost alarms ... 477
Aurora DSQL Database Insights ... 480
What is DASH? ... 480
DASH metrics and wait events ... 481
Accessing DASH data ... 486
Related resources ... 489

Logging with CloudTrail ... 490
Management events ... 490
Data events ... 492
Security ... 494
AWS managed policies ... 495
AmazonAuroraDSQLFullAccess ... 495
AmazonAuroraDSQLReadOnlyAccess ... 497
AmazonAuroraDSQLConsoleFullAccess ... 497
AuroraDsqlServiceLinkedRolePolicy ... 499
Policy updates ... 499
Data protection ... 506
Data encryption ... 507
Data Protection in witness Regions ... 508
SSL/TLS certificates ... 508
Data encryption ... 507
KMS key types ... 515
Encryption at rest ... 515
Using KMS and data keys ... 517
Authorizing your KMS key ... 519
Encryption context ... 521
Monitoring AWS KMS ... 521
Creating an encrypted cluster ... 524
Removing or updating a key ... 526
Considerations ... 528
Identity and access management ... 529
Audience ... 530
Authenticating with identities ... 530
Managing access using policies ... 531
How Aurora DSQL works with IAM ... 533
Identity-based policy examples ... 538
Troubleshooting ... 543
Resource-based policies ... 545
When to Use ... 546
Create with policies ... 547
Add and edit policies ... 550
View Policy ... 554

Remove Policy ... 555
Policy examples ... 556
Block public access ... 561
API Operations ... 564
Using a service-linked role ... 567
Service-linked role permissions for Aurora DSQL ... 567
Create a service-linked role ... 568
Edit a service-linked role ... 569
Delete a service-linked role ... 569
Supported Regions for Aurora DSQL service-linked roles ... 569
Using IAM condition keys ... 569
Available condition keys ... 570
Restrict witness Region ... 570
Require tags on create ... 571
Restrict fault injection ... 572
Incident response ... 573
Compliance validation ... 573
Resilience ... 574
Backup and restore ... 574
Replication ... 575
High availability ... 575
Fault injection testing ... 576
Infrastructure Security ... 576
Managing clusters using AWS PrivateLink ... 577
Configuration and vulnerability analysis ... 588
Cross-service confused deputy prevention ... 588
CDC stream service role ... 590
Security best practices ... 591
Detective security best practices ... 591
Preventative security best practices ... 592
Tagging resources ... 594
Name tag ... 594
Tagging requirements ... 594
CDC stream tags ... 595
Tagging usage notes ... 595
Considerations ... 596

Quotas and limits ... **598**
Cluster quotas ... 598
Database limits ... 599
API reference ... **604**
Troubleshooting ... **384**
Connection errors ... 605
Authentication errors ... 606
Authorization errors ... 607
SQL errors ... 607
Error: server unavailable (SQLSTATE XX000) ... 608
Concurrency control responses ... 608
SSL/TLS connections ... 609
Database Insights visibility ... 609
Aurora DSQL cluster not appearing in the Amazon CloudWatch Database Insights console ... 609
Providing feedback ... **611**
Feedback channels ... 611
Effective feature requests ... 611
Release notes ... **612**
2026 ... 612
October 2026 ... 612
September 2026 ... 612
August 2026 ... 613
July 2026 ... 615
June 2026 ... 616
May 2026 ... 616
April 2026 ... 617
March 2026 ... 618
February 2026 ... 619
January 2026 ... 621
2025 ... 621
December 2025 ... 621
November 2025 ... 622
October 2025 ... 623
August 2025 ... 623
July 2025 ... 623

May 2025 624
2024 624
December 2024 624
Document history 625

What is Amazon Aurora DSQL?

Amazon Aurora DSQL is a serverless, distributed relational database service optimized for transactional workloads. Aurora DSQL offers virtually unlimited scale and doesn't require you to manage infrastructure. The active-active highly available architecture provides 99.99% single-Region and 99.999% multi-Region availability.

When to use Aurora DSQL

Aurora DSQL is optimized for transactional workloads that benefit from ACID transactions and a relational data model. Because it's serverless, Aurora DSQL is ideal for application patterns of microservice, serverless, and event-driven architectures. Aurora DSQL is PostgreSQL-compatible, so you can use familiar drivers, object-relational mappings (ORMs), frameworks, and SQL features.

Aurora DSQL automatically manages system infrastructure and scales compute, I/O, and storage based on your workload. Because you have no servers to provision or manage, you don't have to worry about maintenance downtime related to provisioning, patching, or infrastructure upgrades.

Aurora DSQL helps you to build and maintain enterprise applications that are always available at any scale. The active-active serverless design automates failure recovery, so you don't need to worry about traditional database failover. Your applications benefit from Multi-AZ and multi-Region availability, and you don't have to be concerned about eventual consistency or missing data related to failovers.

Key features in Aurora DSQL

The following key features help you create a serverless distributed database to support your high-availability applications:

Distributed architecture

Aurora DSQL is composed of the following multi-tenant components:

- Relay and connectivity
- Compute and databases
- Transaction log, concurrency control, and isolation
- Storage

A control plane coordinates the preceding components. Each component provide redundancy across three Availability Zones (AZs), with automatic cluster scaling and self-healing in case of component failures. To learn more about how this architecture supports high availability, see Resilience in Amazon Aurora DSQL.

Single-Region and multi-Region clusters

Aurora DSQL clusters provide the following benefits:

- Synchronous data replication
- Consistent read operations
- Automatic failure recovery
- Data consistency across multiple AZs or Regions

If an infrastructure component fails, Aurora DSQL automatically routes requests to healthy infrastructure without manual intervention. Aurora DSQL provides *atomicity, consistency, isolation, and durability (ACID) transactions* with strong consistency, snapshot isolation, atomicity, and cross-AZ and cross-Region durability.

Multi-Region peered clusters provide the same resilience and connectivity as single-Region clusters. But they improve availability by offering two Regional endpoints, one in each peered cluster Region. Both endpoints of a peered cluster present a single logical database. They are available for concurrent read and write operations, and provide strong data consistency. You can build applications that run in multiple Regions at the same time for performance and resilience—and know that readers always see the same data.

Compatibility with PostgreSQL

The distributed database layer (compute) in Aurora DSQL is based on a current major version of PostgreSQL. You can connect to Aurora DSQL with familiar PostgreSQL drivers and tools, such as `psql`. Aurora DSQL is currently compatible with PostgreSQL version 16 and supports a wide range of PostgreSQL features, expressions, and data types. For more information about the supported SQL features, see SQL feature compatibility in Aurora DSQL.

Region availability for Aurora DSQL

With Amazon Aurora DSQL, you can deploy database instances across multiple AWS Regions to support global applications and meet data residency requirements. Region availability determines where you can create and manage Aurora DSQL database clusters. Database administrators and application architects who need to design highly available, globally distributed database systems

often need to understand Region support for their workloads. Common use cases include setting up cross-Region disaster recovery, serving users from geographically closer database instances to reduce latency, and maintaining data copies in specific locations for compliance.

The following table shows the AWS Regions where Aurora DSQL is currently available and the endpoint for each AWS Region.

Region Name	Region	Endpoint	Protocol
US East (Ohio)	us-east-2	dsql.us-east-2.api.aws	HTTPS
		dsql-fips.us-east-2.api.aws	HTTPS
US East (N. Virginia)	us-east-1	dsql.us-east-1.api.aws	HTTPS
		dsql-fips.us-east-1.api.aws	HTTPS
US West (Oregon)	us-west-2	dsql.us-west-2.api.aws	HTTPS
		dsql-fips.us-west-2.api.aws	HTTPS
Asia Pacific (Hong Kong)	ap-east-1	dsql.ap-east-1.api.aws	HTTPS
Asia Pacific (Melbourne)	ap-southeast-4	dsql.ap-southeast-4.api.aws	HTTPS
Asia Pacific (Mumbai)	ap-south-1	dsql.ap-south-1.api.aws	HTTPS
Asia Pacific (Osaka)	ap-northeast-3	dsql.ap-northeast-3.api.aws	HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Seoul)	ap-northeast-2	dsql.ap-northeast-2.api.aws	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	dsql.ap-southeast-1.api.aws	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	dsql.ap-southeast-2.api.aws	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS
Canada (Central)	ca-central-1	dsql.ca-central-1.api.aws dsql-fips.ca-central-1.api.aws	HTTPS HTTPS
Canada West (Calgary)	ca-west-1	dsql.ca-west-1.api.aws dsql-fips.ca-west-1.api.aws	HTTPS HTTPS
Europe (Frankfurt)	eu-central-1	dsql.eu-central-1.api.aws	HTTPS
Europe (Ireland)	eu-west-1	dsql.eu-west-1.api.aws	HTTPS
Europe (London)	eu-west-2	dsql.eu-west-2.api.aws	HTTPS

Region Name	Region	Endpoint	Protocol
Europe (Paris)	eu-west-3	dsql.eu-west-3.api.aws	HTTPS
Europe (Spain)	eu-south-2	dsql.eu-south-2.api.aws	HTTPS
Europe (Stockholm)	eu-north-1	dsql.eu-north-1.api.aws	HTTPS
South America (São Paulo)	sa-east-1	dsql.sa-east-1.api.aws	HTTPS

Multi-Region cluster availability for Aurora DSQL

You can create Aurora DSQL multi-Region clusters within specific AWS Region sets. Each Region set groups geographically related Regions that can work together in a multi-Region cluster.

North American Regions

- US East (N. Virginia)
- US East (Ohio)
- US West (Oregon)
- Canada (Central)
- Canada West (Calgary)

Asia Pacific Regions

- Asia Pacific (Hong Kong)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)

- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Tokyo)

European Regions

- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Europe (Spain)
- Europe (Stockholm)

Important Limitations

You must create multi-Region clusters within a single Region set. For example, if you include US East (N. Virginia) in a cluster, you can only add other Regions from the North American set.

> **Important**
>
> Aurora DSQL currently doesn't support cross-continent multi-Region clusters.

Pricing for Aurora DSQL

For cost information, see Aurora DSQL pricing.

What's next?

For information about the core components in Aurora DSQL and to get started with the service, see the following:

- Getting started with Aurora DSQL
- SQL feature compatibility in Aurora DSQL
- Accessing Aurora DSQL with PostgreSQL-compatible clients

- Aurora DSQL and PostgreSQL

Getting started with Aurora DSQL

Amazon Aurora DSQL is a serverless, fully managed, distributed relational database optimized for transactional workloads. In the following sections, you'll learn how to create single-Region and multi-Region Aurora DSQL clusters, connect to them, and create and load a sample schema. You will access clusters with the AWS Console and optionally interact with your database using other PostgreSQL clients. By the end, you'll have a working Aurora DSQL cluster set up that is ready to use for test or production workloads.

Topics

- Prerequisites
- Step 1: Create an Aurora DSQL single-Region cluster
- Step 2: Connect to your Aurora DSQL cluster
- Step 3: Run sample SQL commands in Aurora DSQL
- Step 4 (Optional): Create a multi-Region cluster
- Troubleshooting

Prerequisites

Before you can begin using Aurora DSQL, make sure you meet the following prerequisites:

- Your IAM identity must have permission to sign in to the console.
- Your IAM identity must meet the following criteria:
  - Access to perform any action on any resource in your AWS account
  - `AmazonAuroraDSQLConsoleFullAccess` AWS managed policy is attached.

Step 1: Create an Aurora DSQL single-Region cluster

The basic unit of Aurora DSQL is the cluster, which is where you store your data. In this task, you create a cluster in a single AWS Region.

To create a single-Region cluster in Aurora DSQL

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql.

2. Choose **Create cluster**.
3. In **Cluster topology**, choose **Single Region**.
4. In **Cluster configuration**, choose one of the following:

   - **Quick create** to create the cluster with recommended settings. You can change some settings, such as **Tag: Name** and deletion protection, before you create the cluster.
   - **Advanced configuration** to control all settings, including encryption and resource-based policies.

5. (Optional) If you chose **Advanced configuration**, do any of the following:

   - In **Name and tags**, change the value of the default **Name** tag, or choose **Add additional tags**.
   - In **Cluster settings**, select **Customize encryption settings (advanced)** to choose or create an AWS KMS key. If you use a customer managed key, make sure that the key policy grants Aurora DSQL the required permissions. For more information, see Key policy for a customer managed key.
   - In **Cluster settings**, keep **Enable deletion protection** selected to prevent a delete operation from removing your cluster. The console selects it by default.
   - In **Cluster settings**, select **Add resource-based policy** to specify access control policies for this cluster.

6. Choose **Create cluster**. The console opens the details page for your new cluster.

Step 2: Connect to your Aurora DSQL cluster

Aurora DSQL supports multiple ways to connect to your cluster, including the DSQL Query Editor, AWS CloudShell, the local psql client, and other PostgreSQL-compatible tools. In this step, you connect using the Aurora DSQL Query Editor, which provides a quick way to begin interacting with your new cluster.

To connect using the Query Editor

1. In the Aurora DSQL Console (https://console.aws.amazon.com/dsql), open the **Clusters** page and confirm that your cluster creation has completed and its status is Active.
2. Select your cluster from the list, or choose the **Cluster ID** to open the Cluster details page.
3. Choose **Connect with Query editor**.

4. Choose Connect as **admin** for the cluster that was just created.
   - Optionally you can connect with a custom role, see Using database roles and IAM authentication.

Step 3: Run sample SQL commands in Aurora DSQL

Test your Aurora DSQL cluster by running SQL statements. After opening the cluster in the Query Editor, select and run each sample query step by step.

Run sample SQL commands in Aurora DSQL

1. Create a schema named `test`.

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

2. Create a hello_world table that uses an automatically generated UUID as the primary key.

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

3. Insert a sample row.

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

4. Read the inserted values.

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

5. Optionally clean up.

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

Step 4 (Optional): Create a multi-Region cluster

When you create a multi-Region cluster, you specify the following Regions:

Remote Region

This is the Region in which you create a second cluster. You create a second cluster in this Region and peer it to your initial cluster. Aurora DSQL replicates all writes on the initial cluster to the remote cluster. You can read and write on any cluster.

Witness Region

This Region receives all data that is written to the multi-Region cluster. However, witness Regions don't host client endpoints and don't provide user data access. A limited window of the encrypted transaction log is maintained in witness Regions. This log facilitates recovery and supports transactional quorum if a Region becomes unavailable.

Use the following procedure to create a multi-Region cluster. The console creates a cluster in each of the two Regions and peers them for you. The procedures after it show cross-Region write replication and consistent reads from both Regional endpoints.

To create a multi-Region cluster

1. Sign in to the Aurora DSQL console.
2. In the navigation pane, choose **Clusters**.
3. Choose **Create cluster**.
4. In **Cluster topology**, choose **2 active Regions + Witness**.
5. In **Cluster configuration**, choose one of the following:
   - **Quick create** to create the clusters with recommended settings. **Peer cluster 1** is in your current Region. To change the Region for **Peer cluster 2** or **Witness peer**, choose a Region in the **Region** row of the table.
   - **Advanced configuration** to control all settings for each Region. In **Regions**, choose the **Peer cluster 2 Region** and the **Witness Region**.
6. (Optional) If you chose **Advanced configuration**, change the settings for each Region in **Name and tags**, **Encryption key**, **Deletion protection**, and **Resource-based policy - optional**. If you use a customer managed key, make sure that the key policy grants Aurora DSQL the required permissions. For more information, see Key policy for a customer managed key.

7. Choose **Create cluster**. The console creates the cluster in your current Region and opens its details page. A notification shows the progress while the console creates the second cluster and peers the two clusters.

To connect to your second cluster

1. Open the Aurora DSQL console and choose the Region for your second cluster.
2. Choose **Clusters**.
3. Select the row for the second cluster in your multi-Region cluster.
4. Choose **Connect with Query editor**.
5. Choose **Connect as admin**.
6. Create a sample schema and table, and insert data by following the steps in Step 3: Run sample SQL commands in Aurora DSQL.

To query data in the second cluster from the Region hosting your initial cluster

1. In the Aurora DSQL console, choose the Region for your initial cluster.
2. Choose **Clusters**.
3. Select the row for the second cluster in your multi-Region cluster.
4. Choose **Connect with Query editor**.
5. Choose **Connect as admin**.
6. Query the data that you inserted into the second cluster.

   Example

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

Troubleshooting

See the Troubleshooting section of the Aurora DSQL documentation.

How billing works in Aurora DSQL

With Amazon Aurora DSQL, you only pay for what you use with no upfront costs. This section explains how Aurora DSQL meters your database activity and translates it into charges on your AWS bill. For current pricing by Region, see the Aurora DSQL pricing page.

Topics

- How metering works
- DPU component metering explained
- Multi-Region billing
- Monitoring DPU usage with CloudWatch
- Finding high-load queries with Aurora DSQL Database Insights
- Using EXPLAIN ANALYZE VERBOSE for cost awareness
- Cost estimation best practices

How metering works

Aurora DSQL charges only for work performed, including work done by transactions that are aborted or fail. Aurora DSQL meters two primary components: database activity, measured in Distributed Processing Units (DPUs), and storage, measured in GiB-month.

DPUs measure how much work the system does to run your SQL workload and are comprised of three components for single-Region clusters: Compute DPUs, Read DPUs, and Write DPUs. Multi-Region clusters incur an additional MultiRegion Write DPU component. For more information about Multi-Region billing, see Multi-Region billing. Clusters with change data capture (CDC) enabled incur an additional Stream DPU component. For more information about CDC billing, see Stream DPU.

The following table summarizes the components that Aurora DSQL uses to meter your database activity. On your bill, you see only two line items: one for storage and one for DPU, which is the sum of each of the individual components.

Metering unit	Activity type	Measurement
Compute DPU	Query processing	CPU time

Metering unit	Activity type	Measurement
Read DPU	Read data from your database	Bytes read from storage
Write DPU	Write data to your database	Bytes written to storage
Stream DPU	CDC streaming (when enabled)	Bytes streamed
Storage	Table storage	GiB-month

DPU component metering explained

For every transaction, Aurora DSQL calculates the total DPU as the sum of three components: Compute DPU, Read DPU, and Write DPU. The following sections explain how Aurora DSQL meters each component.

```
Total DPU = ComputeDPU + ReadDPU + WriteDPU
```

ComputeDPU

Compute DPUs are measured using the total processing time spent executing your query, including joins, functions, aggregations, sorting, and query planning. Because parts of your query can be processed in parallel, the Compute DPU reflects the sum of all processing time — not the wall-clock time of the query.

The following formula summarizes how to calculate Compute DPUs:

```
ComputeDPU = Total Compute time (in seconds)
```

WriteDPU

For every transaction, Aurora DSQL measures Write DPUs by the total bytes written to storage. Write DPUs include the total data written to your base table as well as any secondary indexes. Aurora DSQL bills each row written to your base table and secondary indexes that is smaller than 128 bytes as if it is 128 bytes. Aurora DSQL bills a write transaction that writes less than 1,024 bytes as if it wrote 1,024 bytes.

> **Note**
>
> Write operations also incur ReadDPU charges because Aurora DSQL reads the primary key index to verify uniqueness before writing.

The following formulas show the steps to calculate Write DPUs:

Step 1: Calculate bytes written

```
Bytes Written = Sum of max(size of each row, 128 bytes) for all rows written
```

Step 2: Calculate WriteDPU

```
WriteDPU = max(Bytes Written, 1024) × 0.00004883
```

ReadDPU

For every transaction, Aurora DSQL measures Read DPUs by the total bytes read from storage. Read DPUs include data read from your base table as well as any secondary indexes.

Per-partition minimum: Aurora DSQL measures read bytes per storage partition, not per row. If a read request to a storage partition returns less than 128 bytes, Aurora DSQL rounds it up to 128 bytes. For example, if your query reads from 4 partitions—200 bytes from one partition and 50 bytes from each of the other three—the three 50-byte reads are each rounded up to 128 bytes, resulting in a total of 200 + 128 + 128 + 128 = 584 bytes billed.

Transaction minimum: Aurora DSQL bills a read transaction that reads less than 2,048 bytes in total as if it read 2,048 bytes.

The following formulas show the steps to calculate Read DPUs:

Step 1: Calculate bytes read

```
Bytes Read = # of rows read × size of each row
```

> **Note**
>
> The actual bytes read depends on how your data is distributed across storage partitions, because the 128-byte per-partition minimum is applied per partition. If all of your row sizes

are above 128 bytes, you can simply multiply the number of rows read by the size of each row.

Step 2: Calculate ReadDPU

```
ReadDPU = max(Bytes Read, 2048) × 0.00000183105
```

Stream DPU

Stream DPUs apply only when you enable change data capture (CDC) on your cluster. Aurora DSQL measures Stream DPUs by the total bytes streamed to your target Amazon Kinesis Data Streams stream. When there are no changes to stream, you incur no Stream DPU charges.

The following formula shows how to calculate Stream DPUs:

```
StreamDPU = Bytes Streamed × 0.0000023283
```

Aurora DSQL bills Stream DPUs at the same rate as all other DPU components. For more information about CDC streams, see *CDC streams*.

Billing examples

The following examples demonstrate how Aurora DSQL calculates DPUs for common operations. Cost values in these examples use the us-east-1 Region price. For pricing in other Regions, see the Aurora DSQL pricing page.

Example: Simple point lookup (read)

This example demonstrates a point lookup ReadDPU calculation where the transaction minimum applies.

Schema:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id VARCHAR(50) NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
);
-- Average row size: ~100 bytes
```

Query:

```
SELECT * FROM orders WHERE customer_id = 'cust-12345';
```

Scenario: The query returns 5 rows, each approximately 100 bytes. Assuming all rows reside in one storage partition, the total bytes read is 5 × 100 = 500 bytes. Since 500 bytes exceeds the 128-byte per-partition minimum, no per-partition minimum applies.

Calculate ReadDPU:

```
ReadDPU = max(500, 2048) × 0.00000183105 = 2048 × 0.00000183105 = 0.00375
```

The transaction minimum of 2,048 bytes applies since 500 < 2,048.

Total transaction cost:

Assuming query execution time of 3 ms (0.003 seconds):

```
ComputeDPU: 0.003
ReadDPU:    0.00375
WriteDPU:   0.0
-------------------
Total DPU:  0.00675
```

Example: Filtered range scan (read)

Schema:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id VARCHAR(50) NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
);
```

```
-- Average row size: ~100 bytes
-- Table contains 100 orders for customer 'cust-12345'
```

Query:

```
SELECT * FROM orders
WHERE customer_id = 'cust-12345'
  AND total_amount > 500.00;
```

Scenario: The query scans 100 rows for customer 'cust-12345', but the `total_amount > 500.00` filter reduces the result to only 10 rows returned. Aurora DSQL bills for all 100 rows scanned. Assuming all rows reside in one storage partition, the total bytes read is 100 × 100 = 10,000 bytes.

Calculate ReadDPU:

```
ReadDPU = max(10000, 2048) × 0.00000183105 = 10000 × 0.00000183105 = 0.01831
```

Since 10,000 bytes exceeds the 2,048-byte transaction minimum, the actual bytes read are used.

Total transaction cost:

Assuming query execution time of 8 ms (0.008 seconds):

```
ComputeDPU: 0.008
ReadDPU:    0.01831
WriteDPU:   0.0
--------------------
Total DPU:  0.02631
```

> **Important**
>
> To minimize ReadDPU costs, design queries and indexes to scan only the rows you need. In this example, adding an index on `(customer_id, total_amount)` could allow the query to scan fewer rows.

Example: Single insert (read and write)

Schema:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id VARCHAR(50) NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
);
-- Average row size: ~100 bytes
```

Query:

```
INSERT INTO orders (customer_id, total_amount, status)
VALUES ('cust-67890', 150.00, 'pending');
```

Scenario: Insert 1 row, approximately 100 bytes.

WriteDPU calculation:

Step 1 - Calculate bytes written:

```
1 row × max(100 bytes, 128 bytes) = 1 × 128 = 128 bytes
```

Step 2 - Calculate WriteDPU:

```
WriteDPU = max(128, 1024) × 0.00004883 = 1024 × 0.00004883 = 0.05
```

The transaction minimum of 1,024 bytes applies since 128 < 1,024.

ReadDPU (primary key check):

Aurora DSQL reads the primary key index to verify uniqueness before writing. This incurs the transaction minimum read charge.

```
ReadDPU = 0.00375 (transaction minimum)
```

Total transaction cost:

Assuming query execution time of 8 ms (0.008 seconds):

```
ComputeDPU: 0.008
ReadDPU:    0.00375
WriteDPU:   0.05
-------------------
Total DPU:  0.06175
```

Example: Bulk insert (read and write)

Schema:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id VARCHAR(50) NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
);
-- Average row size: ~100 bytes
```

Query:

```
INSERT INTO orders (customer_id, total_amount, status)
VALUES
    ('cust-001', 100.00, 'pending'),
    ('cust-002', 150.00, 'pending'),
    ... -- 100 rows total
    ('cust-100', 200.00, 'pending');
```

Scenario: Insert 100 rows, each approximately 100 bytes.

WriteDPU calculation:

Step 1 - Calculate bytes written:

```
100 rows × max(100 bytes, 128 bytes) = 100 × 128 = 12,800 bytes
```

Step 2 - Calculate WriteDPU:

```
WriteDPU = max(12800, 1024) × 0.00004883 = 12800 × 0.00004883 = 0.625
```

ReadDPU (primary key checks):

Aurora DSQL reads the primary key index for each row to verify uniqueness. Assuming all 100 key lookups reside in one storage partition, the total bytes read is 100 × 16 bytes (UUID) = 1,600 bytes:

```
ReadDPU = max(1600, 2048) × 0.00000183105 = 2048 × 0.00000183105 = 0.00375
```

The transaction minimum of 2,048 bytes applies since 1,600 < 2,048.

Total transaction cost:

Assuming query execution time of 80 ms (0.08 seconds):

```
ComputeDPU: 0.08
ReadDPU:    0.00375
WriteDPU:   0.625
-------------------
Total DPU:  0.70875
```

Multi-Region billing

Multi-Region clusters incur an additional MultiRegion Write DPU component on top of the standard Compute, Read, and Write DPUs. This section applies only to multi-Region clusters. Single-Region clusters do not incur this charge.

Multi-Region Write DPUs measure the total bytes written to the peered Region. Aurora DSQL charges this DPU in the Region where the write originated, not in the peered Region. For example, in a two-active Region and Witness peering configuration, Aurora DSQL charges the write DPU of the originating transaction for each active peered Region when calculating the `MultiRegionWriteDPU`. This means a transaction that accrues 8 `WriteDPU` incurs an additional 8 `MultiRegionWriteDPU` for this configuration. This is 8 for the originating Region, 8 for the peered Region, and no charge for the Witness Region. Aurora DSQL bills all charges to the Region that originated the transaction.

All clusters in the multi-Region configuration incur storage size charge. Because Aurora DSQL synchronously replicates the data you write to the peered Region, the storage size charge applies to all clusters in the multi-Region configuration individually. This also applies to clusters in **Pending Delete** state. For more information about the **Pending Delete** state, see the Aurora DSQL cluster lifecycle page.

```
MultiRegionWriteDPU = WriteDPU
```

Monitoring DPU usage with CloudWatch

Aurora DSQL publishes usage metrics to Amazon CloudWatch, allowing you to monitor consumption in near real-time.

Available DPU metrics

DPU metrics

CloudWatch metric	Description	Dimension
`WriteDPU`	Write usage component	ClusterId
`ReadDPU`	Read usage component	ClusterId
`ComputeDPU`	Query processing component	ClusterId
`MultiRegionWriteDPU`	Multi-Region replication (multi-Region clusters only)	ClusterId
`StreamDPU`	CDC streaming (clusters with CDC enabled only)	`ClusterId` (all streams); `ClusterId` , `StreamId` (per stream)
`TotalDPU`	Sum of all DPU components	ClusterId

Viewing DPU metrics

To view DPU metrics in CloudWatch

1. Open the CloudWatch console.
2. Navigate to **Metrics**, then **AuroraDSQL**, then **ClusterId**.
3. Select your cluster and the DPU metrics you want to monitor.

> **Tip**
>
> Use the **Sum** statistic for DPU metrics to see total usage over a time period. Add the **LAST** label to see the most recent value.

Additional observability metrics

For a complete list of Aurora DSQL metrics and monitoring capabilities, see *Monitoring and logging*.

Observability metrics

Metric	Description
`ClusterStorageSize`	Current storage size in bytes
`TotalTransactions`	Total transactions executed
`ReadOnlyTransactions`	Read-only transactions executed
`QueryTimeouts`	Queries that exceeded time limit
`OccConflicts`	Transactions aborted due to OCC conflicts. Aurora DSQL bills for the work done by these transactions.
`BytesWritten`	Raw bytes written to storage
`BytesRead`	Raw bytes read from storage

Finding high-load queries with Aurora DSQL Database Insights

CloudWatch DPU metrics show how many DPUs a cluster uses, but not which SQL statements use them. Aurora DSQL Database Insights breaks down database load by SQL statement and by wait event. Use it to see which statements account for the most load on your cluster. You can use Database Insights on every cluster with no setup and at no additional cost. For more information, see Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights.

Keep the following in mind when you use Database Insights to understand your costs:

- **Compare with typical usage:** When DPU usage rises, compare the top SQL statements for that time range with those from a period of typical usage. Statements that are new to the list, or that account for a larger share of the load, are the first ones to investigate.
- **Load measures session time:** Database load is the average number of active sessions (AAS), where an active session is one with an open transaction. A statement's load grows with how often it runs and with how long each run takes. A statement that takes milliseconds but runs many times per second can add more load than one that takes seconds but runs rarely.
- **Wait events show where time goes:** A wait event shows what sessions spend time on, such as CPU or storage reads, but not why. For example, a well-indexed statement that runs often can spend most of its time in storage read wait events.
- **Attribute load to workloads:** On a shared cluster, use Prometheus Query Language (PromQL) to break down load by IAM role or by application name. To see load by application, set an application name in your connection parameters.

To see a statement's execution plan and how many DPUs one run uses, run the statement with `EXPLAIN ANALYZE VERBOSE`. For details, see the section called “EXPLAIN ANALYZE VERBOSE for cost awareness”.

Using EXPLAIN ANALYZE VERBOSE for cost awareness

After you find a statement to investigate, use `EXPLAIN ANALYZE VERBOSE` to examine it in detail. Aurora DSQL extends `EXPLAIN ANALYZE VERBOSE` to include a statement-level DPU usage estimate at the end of the output. This provides immediate visibility into query cost, helping you identify workload cost drivers, tune query performance, and better forecast resource usage.

Database Insights shows SQL text with placeholders instead of literal values. To run a statement that you found in Database Insights, replace the placeholders with values that are typical for your workload. The amount of data that a statement reads can depend on these values.

> **Note**
>
> You must use `EXPLAIN ANALYZE VERBOSE` (with VERBOSE) to see DPU estimates. A plain `EXPLAIN ANALYZE` without VERBOSE does not show DPU information.

Warning

EXPLAIN ANALYZE VERBOSE runs the statement that it explains. To examine an INSERT, UPDATE, or DELETE statement without changing your data, run it in a transaction and then roll back the transaction. Replace *write-statement* with the statement that you want to examine.

```
BEGIN;
EXPLAIN ANALYZE VERBOSE write-statement;
ROLLBACK;
```

Example 1: SELECT query

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

```
QUERY PLAN
------------------------------------------------------
Index Only Scan using test_table_pkey on public.test_table  (cost=125100.05..171100.05
 rows=1000000 width=36) (actual time=2.973..4.482 rows=120 loops=1)
  Output: id, context
  -> Storage Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000 width=36)
 (actual rows=120 loops=1)
       Projections: id, context
       -> B-Tree Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000
 width=36) (actual rows=120 loops=1)
Query Identifier: qymgw1m77maoe
Planning Time: 11.415 ms
Execution Time: 4.528 ms
Statement DPU Estimate:
  Compute: 0.01607 DPU
  Read: 0.04312 DPU
  Write: 0.00000 DPU
  Total: 0.05919 DPU
```

In this example, the SELECT statement performs an index-only scan, so most of the cost comes from Read DPU (0.04312), representing the data retrieved from storage and Compute DPU (0.01607), which reflects the compute resources used to process and return the results. There is no

Write DPU since the query does not modify data. The total DPU (0.05919) is the sum of Compute + Read + Write.

Example 2: INSERT query

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

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

This statement primarily performs writes, so most of the cost is associated with Write DPU. The Compute DPU (0.01550) represents the work done to process and insert the values. The Read DPU (0.00307) reflects minor system reads (for catalog lookups or index checks).

Notice the transaction minimums shown in parentheses next to Read and Write DPUs. These minimums apply at the transaction level, meaning the total Read or Write DPU for an entire transaction is never less than these values. If you are using `EXPLAIN ANALYZE VERBOSE` to forecast costs and this is the only statement in the transaction, use the transaction minimum values rather than the raw statement estimates. If the transaction contains multiple statements, the minimums apply to the aggregate across all statements. Because `EXPLAIN ANALYZE VERBOSE` reports statement-level estimates while billing applies transaction-level minimums, the values may not exactly match CloudWatch metrics or billing data.

Using DPU information for optimization

Per-statement DPU estimates give you a powerful way to optimize queries beyond just execution time. Common use cases include:

- **Cost awareness:** Understand how expensive a query is relative to others.
- **Schema optimization:** Compare the impact of indexes or schema changes on both performance and resource efficiency.
- **Budget planning:** Estimate workload cost based on observed DPU usage.
- **Query comparison:** Evaluate alternative query approaches by their relative DPU consumption.

Interpreting DPU information

Keep the following best practices in mind when using DPU data from `EXPLAIN ANALYZE VERBOSE`:

- **Use it directionally:** Treat the reported DPU as a way to understand the *relative* cost of a query rather than an exact match with CloudWatch metrics or billing data. Differences are expected because `EXPLAIN ANALYZE VERBOSE` reports statement-level cost, while CloudWatch aggregates transaction-level activity. CloudWatch also includes background operations (such as asynchronous ANALYZE or compactions) and transaction overhead (`BEGIN`/`COMMIT`) that `EXPLAIN ANALYZE VERBOSE` intentionally excludes.
- **Test with representative data for proof of concepts:** When running a proof of concept to evaluate costs, ensure your tables contain data volumes and distributions similar to your expected production workload. DPU estimates — whether from `EXPLAIN ANALYZE VERBOSE` or CloudWatch metrics — that are based on empty or sparsely populated tables will not reflect real-world costs.
- **DPU variability across runs is normal** in distributed systems and does not indicate errors. Factors such as caching, execution plan changes, concurrency, background operations like asynchronous ANALYZE, or shifts in data distribution can all cause the same query to consume different resources from one run to the next.
- **Batch small operations:** If your workload issues many small statements, consider batching them into larger write operations within a single transaction (modifications must not exceed 10 MB per transaction, though reads are only limited by the 5-minute transaction timeout). This amortizes transaction minimums across more work and produces more meaningful cost estimates.

- **Use for tuning, not billing:** DPU data in `EXPLAIN ANALYZE VERBOSE` is designed for cost awareness, query tuning, and optimization. It is not a billing-grade metric. Always rely on CloudWatch metrics or monthly billing reports for authoritative cost and usage data.

Cost estimation best practices

- **Monitor before optimizing:** Use CloudWatch metrics to understand your current usage pattern before making optimization decisions. For details, see the section called “Monitoring DPU usage”.
- **Find the statements that drive load:** Use Aurora DSQL Database Insights to find the statements that account for the most load, and to see which statements changed when usage rose. For details, see the section called “Finding high-load queries”.
- **Examine specific statements with EXPLAIN ANALYZE VERBOSE:** Run `EXPLAIN ANALYZE VERBOSE` on the statements that Database Insights identifies, and on critical queries during development, to understand their cost characteristics. When running a proof of concept to evaluate costs, test against tables with representative data volumes and distributions — estimates based on empty or sparsely populated tables will not reflect production costs. For details, see the section called “EXPLAIN ANALYZE VERBOSE for cost awareness”.
- **Focus on transaction efficiency:** Because minimums apply at the transaction level, batch related operations together to amortize minimum charges.
- **Set CloudWatch alarms:** Create alarms on DPU metrics to get notified of unexpected usage spikes.

Authentication and authorization for Aurora DSQL

Aurora DSQL uses IAM roles and policies for cluster authorization. You associate IAM roles with PostgreSQL database roles for database authorization. This approach combines benefits from IAM with PostgreSQL privileges. Aurora DSQL uses these features to provide a comprehensive authorization and access policy for your cluster, database, and data.

Managing your cluster using IAM

To manage your cluster, use IAM for authentication and authorization:

IAM authentication

To authenticate your IAM identity when you manage Aurora DSQL clusters, you must use IAM. You can provide authentication using the AWS Management Console, AWS CLI, or the AWS SDK.

IAM authorization

To manage Aurora DSQL clusters, grant authorization using IAM actions for Aurora DSQL. For example, to describe a cluster, make sure that your IAM identity has permissions for the IAM action `dsql:GetCluster`, as in the following sample policy action.

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

For more information, see Using IAM policy actions to manage clusters.

Connecting to your cluster using IAM

To connect to your cluster, use IAM for authentication and authorization:

IAM authentication

Generate a temporary authentication token using an IAM identity with authorization to connect to your cluster. To learn more, see Generating an authentication token in Amazon Aurora DSQL.

IAM authorization

Grant the following IAM policy actions to the IAM identity you're using to establish the connection to your cluster's endpoint:

- Use `dsql:DbConnectAdmin` if you're using the `admin` role. Aurora DSQL creates and manages this role for you. The following sample IAM policy action permits `admin` to connect to *`my-cluster`*.

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

- Use `dsql:DbConnect` if you're using a custom database role. You create and manage this role by using SQL commands in your database. The following sample IAM policy action permits a custom database role to connect to *`my-cluster`* for up to one hour.

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

After you establish a connection, your role is authorized for up to one hour for the connection.

Interacting with your database using PostgreSQL database roles and IAM roles

PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. You create PostgreSQL roles using SQL commands. To manage database-level authorization, grant PostgreSQL permissions to your PostgreSQL database roles.

Aurora DSQL supports two types of database roles: an `admin` role and custom roles. Aurora DSQL automatically creates a predefined `admin` role for you in your Aurora DSQL cluster. You can't modify the `admin` role. When you connect to your database as `admin`, you can issue SQL to

create new database-level roles to associate with your IAM roles. To let IAM roles connect to your database, associate your custom database roles with your IAM roles.

Authentication

Use the `admin` role to connect to your cluster. After you connect your database, use the command `AWS IAM GRANT` to associate a custom database role with the IAM identity authorized to connect to the cluster, as in the following example.

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

To learn more, see Authorizing database roles to connect to your cluster.

Authorization

Use the `admin` role to connect to your cluster. Run SQL commands to set up custom database roles and grant permissions. To learn more, see PostgreSQL database roles and PostgreSQL privileges in the PostgreSQL documentation.

Using IAM policy actions with Aurora DSQL

The IAM policy action you use depends on the role you use to connect to your cluster: either `admin` or a custom database role. The policy also depends on the IAM actions required for this role.

Using IAM policy actions to connect to clusters

When you connect to your cluster with the default database role of `admin`, use an IAM identity with authorization to perform the following IAM policy action.

```
"dsql:DbConnectAdmin"
```

When you connect to your cluster with a custom database role, first associate the IAM role with the database role. The IAM identity you use to connect to your cluster must have authorization to perform the following IAM policy action.

```
"dsql:DbConnect"
```

To learn more about custom database roles, see Using database roles and IAM authentication.

Using IAM policy actions to manage clusters

When managing your Aurora DSQL clusters, specify policy actions only for the actions that your role needs to perform. For example, if your role only needs to get cluster information, you might limit role permissions to only the `GetCluster` and `ListClusters` permissions, as in the following sample policy

JSON

```
{
"Version":"2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
    }
  ]
}
```

The following example policy shows all available IAM policy actions for managing clusters.

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" : "*"
    }
  ]
}
```

Revoking authorization using IAM and PostgreSQL

You can revoke permissions for your IAM roles to access your database-level roles:

Revoking admin authorization to connect to clusters

To revoke authorization to connect to your cluster with the `admin` role, revoke the IAM identity's access to `dsql:DbConnectAdmin`. Either edit the IAM policy or detach the policy from the identity.

After revoking connection authorization from the IAM identity, Aurora DSQL rejects all new connection attempts from that IAM identity. Any active connections that use the IAM identity might stay authorized for the duration of the connection. For more information on connection durations, see Quotas and limits.

Revoking custom role authorization to connect to clusters

To revoke access to database roles other than `admin`, revoke the IAM identity's access to `dsql:DbConnect`. Either edit the IAM policy or detach the policy from the identity.

You can also remove the association between the database role and IAM by using the command `AWS IAM REVOKE` in your database. To learn more about revoking access from database roles, see Revoking database authorization from an IAM role.

You can't manage permissions of the predefined `admin` database role. To learn how to manage permissions for custom database roles, see PostgreSQL privileges. Modifications to privileges take effect on the next transaction after Aurora DSQL successfully commits the modification transaction.

Generating an authentication token in Amazon Aurora DSQL

To connect to Amazon Aurora DSQL with a SQL client, generate an authentication token to use as the password. This token is used only for authenticating the connection. After the connection is established, the connection remains valid even if the authentication token expires.

If you create an authentication token using the AWS console, the AWS CLI, or SDKs, the token automatically expires in 15 minutes by default. The maximum duration is 604,800 seconds, which is one week. To connect to Aurora DSQL from your client again, you can use the same authentication token if it hasn't expired, or you can generate a new one.

To get started with generating a token, create an IAM policy and a cluster in Aurora DSQL. Then use the AWS console , AWS CLI, or the AWS SDKs to generate a token.

At a minimum, you must have the IAM permissions listed in Connecting to your cluster using IAM, depending on which database role you use to connect.

Topics

- Use the AWS console to generate an authentication token in Aurora DSQL
- Use AWS CloudShell to generate an authentication token in Aurora DSQL
- Use the AWS CLI to generate an authentication token in Aurora DSQL
- Use the SDKs to generate a token in Aurora DSQL

Use the AWS console to generate an authentication token in Aurora DSQL

Aurora DSQL authenticates users with a token rather than a password. You can generate the token from the console.

To generate an authentication token

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql.
2. Choose the cluster ID of the cluster for which you want to generate an authentication token. If you haven't yet created a cluster, follow the steps in Step 1: Create an Aurora DSQL single-Region cluster or Step 4 (Optional): Create a multi-Region cluster.
3. Choose **Connect** and then select **Get Token**.

4. Choose whether you want to connect as an `admin` or with a custom database role.
5. Copy the generated authentication token and use it for Access Aurora DSQL using SQL clients.

To learn more about custom database roles and IAM in Aurora DSQL, see *Authentication and authorization*.

Use AWS CloudShell to generate an authentication token in Aurora DSQL

Before you can generate an authentication token using AWS CloudShell, make sure that you Create an Aurora DSQL cluster.

To generate an authentication token using AWS CloudShell

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql.
2. At the bottom left of the AWS console, choose AWS CloudShell.
3. Run the following command to generate an authentication token for the `admin` role. Replace *`us-east-1`* with your Region and *`your_cluster_endpoint`* with the endpoint of your own cluster.

   > **Note**
   >
   > If you're not connecting as `admin`, use `generate-db-connect-auth-token` instead.

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

   If you run into issues, see Troubleshoot IAM and How can I troubleshoot access denied or unauthorized operation errors with an IAM policy?.
4. Use the following command to use `psql` to start a connection to your cluster.

   ```
   PGSSLMODE=require \
   psql --dbname postgres \
   ```

```
  --username admin \
  --host cluster_endpoint
```

5. You should see a prompt to provide a password. Copy the token that you generated, and make sure you don't include any additional spaces or characters. Paste it into the following prompt from `psql`.

```
Password for user admin:
```

6. Press **Enter**. You should see a PostgreSQL prompt.

```
postgres=>
```

   If you get an access denied error, make sure that your IAM identity has the `dsql:DbConnectAdmin` permission. If you have the permission and continue to get access deny errors, see Troubleshoot IAM and How can I troubleshoot access denied or unauthorized operation errors with an IAM policy?.

To learn more about custom database roles and IAM in Aurora DSQL, see *Authentication and authorization*.

Use the AWS CLI to generate an authentication token in Aurora DSQL

When your cluster is `ACTIVE`, you can generate an authentication token on the CLI by using the `aws dsql` command. Use either of the following techniques:

> **Note**
>
> Token generation is a local operation that signs the request using your current IAM credentials. It does not contact AWS to validate the credentials. If your credentials are expired or invalid, the token generation still succeeds, but the connection attempt fails. Ensure that your IAM credentials are valid before generating a token.

- If you are connecting with the `admin` role, use the `generate-db-connect-admin-auth-token` option.
- If you are connecting with a custom database role, use the `generate-db-connect-auth-token` option.

The following example uses the following attributes to generate an authentication token for the admin role.

- *your_cluster_endpoint* – The endpoint of the cluster. It follows the format *your_cluster_identifier*.dsql.*region*.on.aws, as in the example 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws.
- *region* – The AWS Region, such as us-east-2 or us-east-1.

The following examples set the expiration time for the token to expire in 3600 seconds (1 hour).

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
```

Use the SDKs to generate a token in Aurora DSQL

You can generate an authentication token for your cluster when it is in ACTIVE status. The SDK examples use the following attributes to generate an authentication token for the admin role:

- *your_cluster_endpoint* (or *yourClusterEndpoint*) – The endpoint of your Aurora DSQL cluster. The naming format is *your_cluster_identifier*.dsql.*region*.on.aws, as in the example 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws.
- *region* (or *RegionEndpoint*) – The AWS Region in which your cluster is located, such as us-east-2 or us-east-1.

Python SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for Python, which handles token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `generate_db_connect_admin_auth_token`.
- If you are connecting with a custom database role, use `generate_connect_auth_token`.

```
import boto3

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

C++ SDK

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `GenerateDBConnectAdminAuthToken`.
- If you are connecting with a custom database role, use `GenerateDBConnectAuthToken`.

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

using namespace Aws;
using namespace Aws::DSQL;

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

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

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

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

JavaScript SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connectors for Node.js, which handle token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `getDbConnectAdminAuthToken`.
- If you are connecting with a custom database role, use `getDbConnectAuthToken`.

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

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

Java SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for Java JDBC, which handles token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the admin role, use generateDbConnectAdminAuthToken.
- If you are connecting with a custom database role, use generateDbConnectAuthToken.

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

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

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

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

Rust SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for Rust SQLx, which handles token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the admin role, use db_connect_admin_auth_token.
- If you are connecting with a custom database role, use db_connect_auth_token.

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

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

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

Ruby SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for Ruby pg, which handles token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `generate_db_connect_admin_auth_token`.
- If you are connecting with a custom database role, use `generate_db_connect_auth_token`.

```
require 'aws-sdk-dsql'

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

  token_generator = Aws::DSQL::AuthTokenGenerator.new({
```

```
    :credentials => credentials
  })

  # if you're not using admin role, use generate_db_connect_auth_token instead
  token = token_generator.generate_db_connect_admin_auth_token({
    :endpoint => your_cluster_endpoint,
    :region => region
  })
end
```

PHP SDK

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for PHP PDO_PGSQL, which handles token generation automatically.

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `generateDbConnectAdminAuthToken`.
- If you are connecting with a custom database role, use `generateDbConnectAuthToken`.

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

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

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

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

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

.NET

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for .NET Npgsql, which handles token generation automatically.

> **Note**
>
> The official SDK for .NET doesn't include a built-in API call to generate an authentication token for Aurora DSQL. Instead, you must use `DSQLAuthTokenGenerator`, which is a utility class. The following code sample shows how to generate the authentication token for .NET.

You can generate the token in the following ways:

- If you are connecting with the `admin` role, use `DbConnectAdmin`.
- If you are connecting with a custom database role, use `DbConnect`.

The following example uses the `DSQLAuthTokenGenerator` utility class to generate the authentication token for a user with the `admin` role. Replace *`insert-dsql-cluster-endpoint`* with your cluster endpoint.

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

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

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

Console.WriteLine(token);
```

Go

> **Tip**
>
> AWS recommends using the Aurora DSQL Connector for Go pgx, which handles token generation automatically.

The AWS SDK for Go v2 provides a built-in method for generating authentication tokens in the `github.com/aws/aws-sdk-go-v2/feature/dsql/auth` package.

- If you are connecting with the `admin` role, use `auth.GenerateDBConnectAdminAuthToken`.
- If you are connecting with a custom database role, use `auth.GenerateDbConnectAuthToken`.

```
package main

import (
 "context"
 "fmt"

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

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

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

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

 fmt.Println(token)
}
```

Using database roles and IAM authentication

Aurora DSQL supports authentication using both IAM roles and IAM users. You can use either method to authenticate and access Aurora DSQL databases.

IAM roles

An IAM role is an identity within your AWS account that has specific permissions but is not associated with a specific person. Using IAM roles provide temporary security credentials. You can temporarily assume an IAM role in several ways:

- By switching roles in the AWS Management Console

- By calling an AWS CLI or AWS API operation
- By using a custom URL

After assuming a role, you can access Aurora DSQL using the role's temporary credentials. For more information about methods for using roles, see IAM Identities in the *IAM user guide*.

IAM users

An IAM user is an identity within your AWS account that has specific permissions and is associated with a single person or application. IAM users have long-term credentials such as passwords and access keys that can be used to access Aurora DSQL.

> **Note**
>
> To run SQL commands with IAM authentication, you can use either IAM role ARNs or IAM user ARNs in the examples below.

Authorizing database roles to connect to your cluster

Create an IAM role and grant connection authorization with the IAM policy action: `dsql:DbConnect`.

The IAM policy must also grant permission to access the cluster resources. Use a wildcard (*) or follow the instructions in Using IAM condition keys with Amazon Aurora DSQL.

Authorizing database roles to use SQL in your database

You must use an IAM role with authorization to connect to your cluster.

1. Connect to your Aurora DSQL cluster using a SQL utility.

   Use the `admin` database role with an IAM identity that is authorized for IAM action `dsql:DbConnectAdmin` to connect to your cluster.
2. Create a new database role, making sure to specify the `WITH LOGIN` option.

   ```
   CREATE ROLE example WITH LOGIN;
   ```
3. Associate the database role with the IAM role ARN.

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

4. Grant database-level permissions to the database role

   The following examples use the GRANT command to provide authorization within the database.

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

For more information, see PostgreSQL GRANT and PostgreSQL Privileges in the PostgreSQL documentation.

Viewing IAM to database role mappings

To view the mappings between IAM roles and database roles, query the `sys.iam_pg_role_mappings` system table.

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

Example output:

```
 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)
```

This table shows all the mappings between IAM roles (identified by their ARN) and PostgreSQL database roles.

Revoking database authorization from an IAM role

To revoke database authorization, use the `AWS IAM REVOKE` operation.

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

To learn more about revoking authorization, see Revoking authorization using IAM and PostgreSQL.

Aurora DSQL and PostgreSQL

Aurora DSQL is a PostgreSQL-compatible, distributed relational database designed for transactional workloads. Aurora DSQL uses core PostgreSQL components such as the parser, planner, optimizer, and type system.

The Aurora DSQL design ensures that all supported PostgreSQL syntax provides compatible behavior and yields identical query results. For example, Aurora DSQL provides type conversions, arithmetic operations, and numerical precision and scale that are identical to PostgreSQL. Any deviations are documented.

Aurora DSQL also introduces advanced capabilities such as optimistic concurrency control and distributed schema management. With these features, you can use the familiar tooling of PostgreSQL while benefiting from the performance and scalability of a modern, cloud-native, distributed applications.

PostgreSQL compatibility highlights

Aurora DSQL is currently based on PostgreSQL version 16. Key highlights include the following:

Wire protocol

Aurora DSQL uses the standard PostgreSQL v3 wire protocol. This enables integration with standard PostgreSQL clients, drivers, and tools. For example, Aurora DSQL is compatible with `psql`, `pgjdbc`, and `psycopg`.

SQL syntax

Aurora DSQL supports a wide range of standard PostgreSQL expressions and functions commonly used in transactional workloads. Supported SQL expressions yield identical results to PostgreSQL, including the following:

- Handling of nulls
- Sort order behavior
- Scale and precision for numeric operations
- Equivalence for string operations

For more information, see SQL feature compatibility in Aurora DSQL.

Transaction management

Aurora DSQL preserves the primary characteristics of PostgreSQL, such as ACID transactions and an isolation level equivalent to PostgreSQL Repeatable Read. For more information, see Concurrency control in Aurora DSQL.

Distributed architecture benefits

The distributed, shared-nothing design of Aurora DSQL provides performance and scalability benefits beyond traditional single-node databases. Key capabilities include the following:

Optimistic Concurrency Control (OCC)

Aurora DSQL uses an optimistic concurrency control model. This lock-free approach prevents transactions from blocking one another, eliminates deadlocks, and enables high-throughput parallel execution. These features make Aurora DSQL particularly valuable for applications requiring consistent performance at scale. For more example, see Concurrency control in Aurora DSQL.

Asynchronous DDL operations

Aurora DSQL runs DDL operations asynchronously, which allows uninterrupted reads and writes during schema changes. Its distributed architecture allows Aurora DSQL to perform the following actions:

- Run DDL operations as background tasks, minimizing disruption.
- Coordinate catalog changes as strongly consistent distributed transactions. This ensures atomic visibility across all nodes, even during failures or concurrent operations.
- Operate in a fully distributed, leaderless manner across multiple Availability Zones with decoupled compute and storage layers.

For more on using the EXPLAIN command in PostgreSQL, see DDL and distributed transactions in Aurora DSQL.

SQL feature compatibility in Aurora DSQL

In the following sections, learn about Aurora DSQL support for PostgreSQL data types and SQL commands.

Topics

- Supported data types in Aurora DSQL
- Supported SQL for Aurora DSQL
- Supported subsets of SQL commands in Aurora DSQL
- Migrating from PostgreSQL to Aurora DSQL

Supported data types in Aurora DSQL

Aurora DSQL supports a subset of the common PostgreSQL types.

Topics

- Numeric data types
- Character data types
- Date and time data types
- Miscellaneous data types
- Query runtime data types

Numeric data types

Aurora DSQL supports the following PostgreSQL numeric data types.

Name	Aliases	Range and precision	Storage size	Index support
`smallint`	`int2`	-32768 to +32767	2 bytes	Yes
`integer`	`int`, `int4`	-2147483648 to +2147483647	4 bytes	Yes
`bigint`	`int8`	-9223372036854775808 to +9223372036854775807	8 bytes	Yes
`real`	`float4`	6 decimal digits precision	4 bytes	Yes
`double precision`	`float8`	15 decimal digits precision	8 bytes	Yes

Name	Aliases	Range and precision	Storage size	Index support

`numeric [ (p, s) ]`	`decimal [ (p, s) ]`
	
	`dec[ (p,s)]`
		Exact numeric of selectable precision. The maximum precision is 1000 and scale can be between -1000 and 1000. [1] The default is `numeric (18,6)`.	8 bytes + 2 bytes per group of 4 decimal digits. Maximum size is 510 bytes.	Yes

[1] – If you don't explicitly specify a size when you run `CREATE TABLE` or `ALTER TABLE ADD COLUMN`, Aurora DSQL enforces the defaults. Aurora DSQL applies limits when you run `INSERT` or `UPDATE` statements.

Character data types

Aurora DSQL supports the following PostgreSQL character data types.

Name	Aliases	Description	Aurora DSQL limit	Storage size	Index support
`character [ (n) ]`	`char [ (n) ]`	Fixed-length character string	4096 bytes[1,2]	Variable up to 4100 bytes	Yes
`character varying [ (n) ]`	`varchar [ (n) ]`	Variable-length character string	65535 bytes[1,2]	Variable up to 65539 bytes	Yes
`bpchar [ (n) ]`		If fixed length, this is an alias for `char`. If variable length, this is an alias for `varchar`, where trailing spaces are semantically insignificant.	4096 bytes[1,2]	Variable up to 4100 bytes	Yes

Name	Aliases	Description	Aurora DSQL limit	Storage size	Index support
`text`		Variable-length character string	1 MiB[1,2]	Variable up to 1 MiB	Yes

[1] – If you don't explicitly specify a size when you run `CREATE TABLE` or `ALTER TABLE ADD COLUMN`, then Aurora DSQL enforces the defaults. Aurora DSQL applies limits when you run `INSERT` or `UPDATE` statements.

[2] – Aurora DSQL automatically applies compression to large `text`, `varchar`, and `bpchar` values during `INSERT` and `UPDATE` operations. Aurora DSQL compresses these values only in columns that aren't part of a key. Values in primary key columns and in the key columns of a secondary index are always stored uncompressed.

To disable compression, use the `STORAGE` keyword. For more information, see `CREATE TABLE` and `ALTER TABLE`.

Date and time data types

Aurora DSQL supports the following PostgreSQL date and time data types.

Name	Aliase	Descripti on	Range	Resolution	Storag size	Index support
`date`		Calendar date (year, month, day)	4713 BC – 5874897 AD	1 day	4 bytes	Yes
`time [ (p) ] [ without time zone ]`		Time of day, with no time zone	00:00:00 – 24:00:00	1 microsecond	8 bytes	Yes
`time [ (p) ] with time zone`	`timet`	time of day,	00:00:00+1559 – 24:00:00 –1559	1 microsecond	12 bytes	No

Name	Aliase	Descripti on	Range	Resolution	Storag size	Index support
		including time zone				
`timestamp [(p)][ without time zone]`		Date and time, with no time zone	4713 BC – 294276 AD	1 microsecond	8 bytes	Yes
`timestamp [(p)] with time zone`	`times tz`	Date and time, including time zone	4713 BC – 294276 AD	1 microsecond	8 bytes	Yes
`interval [ fields ][ (p)]`		Time span	-178000000 years – 178000000 years	1 microsecond	16 bytes	No

Miscellaneous data types

Aurora DSQL supports the following miscellaneous PostgreSQL data types.

Name	Aliases	Description	Aurora DSQL limit	Storage size	Index support
`boolean`	`bool`	Logical Boolean (true/false)		1 byte	Yes
`bytea`		Binary data ("byte array")	1 MiB[1]	Variable up to 1 MiB limit	No
`UUID`		Universal ly unique identifier		16 bytes	Yes

Name	Aliases	Description	Aurora DSQL limit	Storage size	Index support
`json`		JSON data	1 MiB[2]	Variable up to 1 MiB limit.[2]	No
`jsonb`		Binary JSON data	1 MiB[2]	Variable up to 1 MiB limit.[2]	No

[1] – If you don't explicitly specify a size when you run `CREATE TABLE` or `ALTER TABLE ADD COLUMN`, then Aurora DSQL enforces the defaults. Aurora DSQL applies limits when you run `INSERT` or `UPDATE` statements.

[2] – Aurora DSQL automatically applies compression to large `json` and `jsonb` values during `INSERT` and `UPDATE` operations. The 1 MiB limit applies to the compressed size, so you can store `json` and `jsonb` values significantly larger than 1 MiB as long as they compress below the limit.

To disable compression, use the `STORAGE` keyword. For more information, see `CREATE TABLE` and `ALTER TABLE`.

JSON functions and operators

Aurora DSQL supports all PostgreSQL JSON functions and operators from section 9.16 JSON Functions and Operators with identical behavior.

> **Note**
>
> The functions `json_populate_record`, `json_populate_recordset`, `jsonb_populate_record`, and `jsonb_populate_recordset` work with table and view row types, but not with custom composite types as Aurora DSQL doesn't currently support `CREATE TYPE`.

The following examples show `json_populate_record` and `jsonb_populate_recordset` used with a table row type:

```
CREATE TABLE tt (c1 INT, c2 INT);
```

```
SELECT * FROM json_populate_record(null::tt, '{"c1": 1, "c2": 2}');
```

```
 c1 | c2
----+----
  1 |  2
(1 row)
```

```
SELECT * FROM jsonb_populate_recordset(null::tt, '[{"c1":1,"c2":2}, {"c1":3,"c2":4}]');
```

```
 c1 | c2
----+----
  1 |  2
  3 |  4
(2 rows)
```

Query runtime data types

Query runtime data types are internal data types used at query execution time. These types are distinct from the PostgreSQL-compatible types like `varchar` and `integer` that you define in your schema. Instead, these types are runtime representations that Aurora DSQL uses when processing a query.

The following data types are supported only during query runtime:

Array type

Aurora DSQL supports arrays of the supported data types. For example, you can have an array of integers. The function `string_to_array` splits a string into a PostgreSQL-style array with the comma delimiter (`,`) as shown in the following example. You can use arrays in expressions, function outputs, or temporary computations during query execution.

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

The function returns a response similar to the following:

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

inet type

The data type represents IPv4, IPv6 host addresses, and their subnets. This type is useful when parsing logs, filtering on IP subnets, or doing network calculations within a query. For more information, see inet in the PostgreSQL documentation.

Supported SQL for Aurora DSQL

Aurora DSQL supports a wide range of core PostgreSQL SQL features. In the following sections, you can learn about general PostgreSQL expression support. This list is not exhaustive.

SELECT command

Aurora DSQL supports the following clauses of the `SELECT` command.

Primary clause	Supported clauses
`FROM`	
`GROUP BY`	`ALL`, `DISTINCT`
`ORDER BY`	`ASC`, `DESC`, `NULLS`
`LIMIT`	
`DISTINCT`	
`HAVING`	
`USING`	
`WITH` (common table expressions)	
`INNER JOIN`	`ON`
`OUTER JOIN`	`LEFT`, `RIGHT`, `FULL`, `ON`
`CROSS JOIN`	`ON`
`UNION`	`ALL`

Primary clause	Supported clauses
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR { UPDATE | KEY SHARE }	Transaction limits apply. For more information, see Cluster quotas and database limits in Amazon Aurora DSQL. To understand how this impacts concurrent transactions, see Concurrency control in Aurora DSQL.

Data Definition Language (DDL)

Aurora DSQL supports the following PostgreSQL DDL commands.

Command	Primary Clause	Supported Clauses
CREATE	TABLE	For information about the supported syntax of the CREATE TABLE command, see CREATE TABLE.
ALTER	TABLE	For information about the supported syntax of the ALTER TABLE command, see ALTER TABLE.
DROP	TABLE	
CREATE	[UNIQUE] INDEX ASYNC	Index keys can be column names or expressions. You can also use INCLUDE, NULLS FIRST, NULLS LAST, and NULLS [NOT] DISTINCT. See CREATE INDEX for the supported syntax.
DROP	INDEX	

Command	Primary Clause	Supported Clauses
CREATE	STATISTICS	For information about the supported syntax of the CREATE STATISTICS command, see CREATE STATISTICS .
ALTER	STATISTICS	For information about the supported syntax of the ALTER STATISTICS command, see ALTER STATISTICS .
DROP	STATISTICS	For information about the supported syntax of the DROP STATISTICS command, see DROP STATISTICS .
CREATE	VIEW	For more information about the supported syntax of the CREATE VIEW command, see CREATE VIEW.
ALTER	VIEW	For information about the supported syntax of the ALTER VIEW command, see ALTER VIEW.
DROP	VIEW	For information about the supported syntax of the DROP VIEW command, see DROP VIEW.
CREATE	SEQUENCE	For information about the supported syntax of the CREATE SEQUENCE command, see CREATE SEQUENCE.
ALTER	SEQUENCE	For information about the supported syntax of the ALTER SEQUENCE command, see ALTER SEQUENCE.
DROP	SEQUENCE	For information about the supported syntax of the DROP SEQUENCE command, see DROP SEQUENCE.
CREATE	ROLE, WITH	

Command	Primary Clause	Supported Clauses
CREATE	FUNCTION	LANGUAGE SQL
CREATE	DOMAIN	
CREATE	SCHEMA	You can nest CREATE SEQUENCE and GRANT statements in the CREATE SCHEMA command.
ALTER	USER, ROLE, GROUP	RENAME TO
ALTER	ROUTINE	SET SCHEMA, OWNER TO, RENAME TO
COMMENT ON	ROUTINE	None
DROP	ROUTINE	None

Data Manipulation Language (DML)

Aurora DSQL supports the following PostgreSQL DML commands.

Command	Primary clause	Supported clauses
INSERT	INTO	VALUES SELECT [ON CONFLICT]
UPDATE	SET	WHERE (SELECT) FROM, WITH
DELETE	FROM	USING, WHERE

Data Control Language (DCL)

Aurora DSQL supports the following PostgreSQL DCL commands.

Command	Supported clauses
GRANT	ON, TO Supported object types include ROUTINE.
REVOKE	ON, FROM, CASCADE, RESTRICT Supported object types include ROUTINE.
ALTER DEFAULT PRIVILEGES	Configure default privileges for TABLE, SEQUENCE, FUNCTION, and TYPE objects.

Transaction Control Language (TCL)

Aurora DSQL supports the following PostgreSQL TCL commands.

Command	Supported clauses	Alias
COMMIT	[WORK | TRANSACTION] [AND NO CHAIN]	END
BEGIN	[WORK | TRANSACTION] [ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
START TRANSACTION	[ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
ROLLBACK	[WORK | TRANSACTION] [AND NO CHAIN]	ABORT
SET CONSTRAINTS	{ALL | *name* [, ...]}	

Command	Supported clauses	Alias
	{DEFERRED | IMMEDIATE }	

Utility commands

Aurora DSQL supports the following PostgreSQL utility commands:

- EXPLAIN
- ANALYZE (relation name only)

Supported subsets of SQL commands in Aurora DSQL

This section provides detailed information about supported SQL commands, focusing on commands with extensive parameter sets and subcommands. For example, CREATE TABLE in PostgreSQL offers many clauses and parameters, a subset of which are supported by Aurora DSQL. This section describes the supported subsets of common SQL commands using familiar PostgreSQL syntax elements that Aurora DSQL supports.

Topics

- CREATE TABLE
- ALTER TABLE
- CREATE INDEX
- CREATE STATISTICS
- ALTER STATISTICS
- DROP STATISTICS
- CREATE SEQUENCE
- ALTER SEQUENCE
- DROP SEQUENCE
- CREATE VIEW
- ALTER VIEW
- DROP VIEW
- SET CONSTRAINTS
- CREATE FUNCTION

- ALTER FUNCTION

CREATE TABLE

CREATE TABLE defines a new table.

Supported syntax

```
CREATE TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } ]
 [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY ( sequence_options ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and referential_action in a FOREIGN KEY/REFERENCES constraint is:
```

```
{ NO ACTION | RESTRICT | CASCADE | SET NULL [ ( column_name [, ... ] ) ] | SET DEFAULT
 [ ( column_name [, ... ] ) ] }

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 [, ... ] ) ]
```

Identity columns

> **Note**
>
> When using identity columns, the cache value should be carefully considered. For more information, see the Important callout on the CREATE SEQUENCE page.
> For guidance on how best to use identity columns based on workload patterns, see Working with sequences and identity columns.

The GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY (*sequence_options*) clause creates the column as an *identity column*. It will have an implicit sequence attached to it and in newly-inserted rows the column will automatically have values from the sequence assigned to it. Such a column is implicitly NOT NULL.

The clauses ALWAYS and BY DEFAULT determine how explicitly user-specified values are handled in INSERT and UPDATE commands.

In an INSERT command, if ALWAYS is selected, a user-specified value is only accepted if the INSERT statement specifies OVERRIDING SYSTEM VALUE. If BY DEFAULT is selected, then the user-specified value takes precedence.

In an UPDATE command, if ALWAYS is selected, any update of the column to any value other than DEFAULT will be rejected. If BY DEFAULT is selected, the column can be updated normally. (There is no OVERRIDING clause for the UPDATE command.)

The *sequence_options* clause can be used to override the parameters of the sequence. The available options include those shown for CREATE SEQUENCE, plus SEQUENCE NAME *name*. Without SEQUENCE NAME, the system chooses an unused name for the sequence.

Storage mode

The optional STORAGE clause sets the storage mode for the column. Use these options to control the behavior of compression for variable-length data types such as JSON, JSONB, TEXT, VARCHAR, and BPCHAR.

Amazon Aurora DSQL compresses some data types when they exceed a certain size. To disable this behavior, use the PLAIN or EXTERNAL options.

PLAIN

Aurora DSQL stores data inline without compression. This is the only option for fixed-length data types such as `integer`. Use this option to disable compression on some variable-length types.

MAIN | EXTENDED | DEFAULT

MAIN and EXTENDED allow optional compression of the column if the underlying data type supports compression. DEFAULT sets the storage mode to the default mode for the column's data type.

EXTERNAL

Aurora DSQL doesn't currently support TOAST tables, however EXTERNAL disables compression on data types that support compression.

Foreign key constraints

The REFERENCES and FOREIGN KEY clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If you omit the *refcolumn* list, Aurora DSQL uses the primary key of the *reftable*. Otherwise, the *refcolumn* list must refer to the columns of a non-deferrable unique or primary key constraint.

Aurora DSQL matches a value inserted into the referencing column(s) against the values of the referenced table and referenced columns using the given match type. There are two supported match types: MATCH FULL and MATCH SIMPLE (which is the default).

You can define a foreign key as either a column constraint or a table constraint:

- **Column constraint (REFERENCES)** – Use REFERENCES after the column data type for single-column foreign keys.

- **Table constraint (FOREIGN KEY)** – Use `FOREIGN KEY (...) REFERENCES ...` for single-column or multi-column foreign keys.

Referential actions

When you change data in the referenced columns, Aurora DSQL performs actions on the data in the referencing table's columns. The `ON DELETE` clause specifies the action to perform when a transaction deletes a referenced row in the referenced table. Likewise, the `ON UPDATE` clause specifies the action to perform when a transaction updates a referenced column to a new value. If a transaction updates the row but doesn't change the referenced column, Aurora DSQL takes no action.

Aurora DSQL supports the following referential actions:

`NO ACTION` (default)

Produces an error if the deletion or update would create a foreign key constraint violation. If the constraint is deferred, Aurora DSQL produces this error at constraint check time if any referencing rows still exist. This is the default action.

`RESTRICT`

Produces an error if a row to be deleted or updated matches a row in the referencing table. This prevents the action even if the state after the action wouldn't violate the foreign key constraint. In particular, it prevents updates of referenced rows to values that are distinct but compare as equal. Unlike `NO ACTION`, the `RESTRICT` check can't be deferred.

> ⚠ **Cascading actions count towards transaction modification limits**
>
> The `CASCADE`, `SET NULL`, and `SET DEFAULT` actions automatically modify rows in the referencing table when a referenced row is updated or deleted. The Aurora DSQL transaction row limit applies to these actions and can cause unexpected failures if not used carefully. Prefer `NO ACTION` or `RESTRICT` for foreign key relationships where child-row cardinality is unbounded or unpredictable. For more information, see Database limits in Aurora DSQL.

`CASCADE`

Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.

`SET NULL [ ( column_name [, ... ] ) ]`

Set all of the referencing columns, or a specified subset of the referencing columns, to null. A subset of columns can only be specified for `ON DELETE` actions.

`SET DEFAULT [ ( column_name [, ... ] ) ]`

Set all of the referencing columns, or a specified subset of the referencing columns, to their default values. A subset of columns can only be specified for `ON DELETE` actions. (There must be a row in the referenced table matching the default values, if they are not null, or the operation will fail.)

Match types

Aurora DSQL supports the following match types:

`MATCH SIMPLE` (default)

Allows any of the foreign key columns to be null. If any of them are null, the row isn't required to have a match in the referenced table.

`MATCH FULL`

Doesn't allow one column of a multi-column foreign key to be null unless all foreign key columns are null. If they are all null, the row isn't required to have a match in the referenced table.

You can apply `NOT NULL` constraints to the referencing columns to prevent these cases from arising.

Deferrability

You can control when a foreign key constraint is checked by specifying its deferrability:

`NOT DEFERRABLE` (default)

Aurora DSQL checks this constraint immediately after each statement. You can't change it to deferred with `SET CONSTRAINTS`.

`DEFERRABLE`

The constraint can be deferred to the end of the transaction using `SET CONSTRAINTS`. Without an `INITIALLY` clause, this defaults to `INITIALLY IMMEDIATE`.

`DEFERRABLE INITIALLY IMMEDIATE`

By default, Aurora DSQL checks this constraint after each statement, but you can defer it within a transaction by using `SET CONSTRAINTS ... DEFERRED`.

`DEFERRABLE INITIALLY DEFERRED`

By default, Aurora DSQL checks this constraint at transaction commit time. You can change it to immediate within a transaction by using `SET CONSTRAINTS ... IMMEDIATE`.

For more information about changing constraint check timing within a transaction, see [SET CONSTRAINTS].

> **⚠ Foreign key constraints only**
>
> In Aurora DSQL, the `DEFERRABLE` option applies to foreign key constraints only.

Foreign key constraint examples

Assume you have a table storing products:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Now you want a table storing orders of those products. You want to ensure that the orders table contains references to products that actually exist. Define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
```

```
    quantity integer
);
```

Now you can't create orders with non-NULL `product_no` entries that don't appear in the products table.

In this situation, the orders table is the *referencing* table and the products table is the *referenced* table. Similarly, there are referencing and referenced columns.

You can shorten the above command to:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

If you omit the column list, Aurora DSQL uses the primary key of the referenced table as the referenced column(s).

You can assign your own name for a foreign key constraint in the usual way:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer CONSTRAINT fk_product REFERENCES products,
    quantity integer
);
```

A foreign key can also constrain and reference a group of columns. It then needs to be written in table constraint form:

```
CREATE TABLE inventory (
    warehouse_id integer,
    product_no integer,
    quantity integer,
    PRIMARY KEY (warehouse_id, product_no)
);

CREATE TABLE shipments (
    shipment_id integer PRIMARY KEY,
    warehouse_id integer,
```

```
    product_no integer,
    FOREIGN KEY (warehouse_id, product_no) REFERENCES inventory (warehouse_id,
 product_no)
);
```

The number and types of the constrained columns need to be compatible with the number and types of the referenced columns.

A table can have more than one foreign key constraint. This is used to implement many-to-many relationships between tables:

```
CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

A foreign key constraint can reference the same table it belongs to. This is called a *self-referential* foreign key. For example, if you want rows of a table to represent nodes of a tree structure, you could write:

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text
);
```

A top-level node would have NULL `parent_id`, while non-NULL `parent_id` entries are constrained to reference valid rows of the table.

You can specify referential actions to control what happens when a referenced row is deleted or updated. The following example uses `ON DELETE RESTRICT` to prevent deletion of a product that is still referenced by an order:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products ON DELETE RESTRICT,
    quantity integer
);
```

With RESTRICT, attempting to delete a product that has orders referencing it produces an error immediately. With the default NO ACTION, the check can be deferred to the end of the transaction if the constraint is declared DEFERRABLE.

To create a foreign key that can be deferred to the end of a transaction, use the DEFERRABLE option. This is useful when you need to insert rows in both tables within the same transaction regardless of order:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products DEFERRABLE INITIALLY DEFERRED,
    quantity integer
);
```

With DEFERRABLE INITIALLY DEFERRED, the constraint isn't checked until commit time. You can insert the order row before the product row exists, as long as the product row is present when the transaction commits.

To use MATCH FULL with a composite foreign key, which requires that all referencing columns are null together or all non-null together:

```
CREATE TABLE shipments (
    shipment_id integer PRIMARY KEY,
    warehouse_id integer,
    product_no integer,
    FOREIGN KEY (warehouse_id, product_no)
        REFERENCES inventory (warehouse_id, product_no) MATCH FULL
);
```

ALTER TABLE

ALTER TABLE changes the definition of a table.

Supported syntax

```
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
ALTER TABLE ASYNC [ IF EXISTS ] [ ONLY ] name [ * ]
    VALIDATE CONSTRAINT constraint_name

where action is one of:

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ STORAGE { PLAIN | EXTERNAL
 | EXTENDED | MAIN | DEFAULT } ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name DROP NOT NULL
    ALTER [ COLUMN ] column_name DROP EXPRESSION [ IF EXISTS ]
    ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( sequence_options ) ]
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET
 sequence_option | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN |
 DEFAULT }
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
 DEFERRED | INITIALLY IMMEDIATE ]
    ADD table_constraint NOT VALID
    ADD table_constraint_using_index
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

and table_constraint is:

    [ CONSTRAINT constraint_name ]
    { CHECK ( expression ) |
      FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
 [, ... ] ) ]
        [ MATCH FULL | MATCH SIMPLE ]
        [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint_using_index is:

    [ CONSTRAINT constraint_name ]
    UNIQUE USING INDEX index_name
```

```
and referential_action in a FOREIGN KEY/REFERENCES constraint is:

    { NO ACTION | RESTRICT | CASCADE | SET NULL [ ( column_name [, ... ] ) ] | SET
 DEFAULT [ ( column_name [, ... ] ) ] }
```

Description

`ADD [ COLUMN ] [ IF NOT EXISTS ]`

This form adds a new column to the table, using the same syntax as `CREATE TABLE`. If `IF NOT EXISTS` is specified and a column already exists with this name, no error is thrown.

`DROP [ COLUMN ] [ IF EXISTS ]`

This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped except for primary key constraints. Dropping of primary key columns is not supported. Multivariate statistics referencing the dropped column will also be removed if the removal of the column would cause the statistics to contain data for only a single column. You will need to say `CASCADE` if anything outside the table depends on the column, for example, foreign key references or views. If `IF EXISTS` is specified and the column does not exist, no error is thrown. In this case a notice is issued instead.

`SET/DROP DEFAULT`

These forms set or remove the default value for a column (where removal is equivalent to setting the default value to NULL). The new default value will only apply in subsequent `INSERT` or `UPDATE` commands; it does not cause rows already in the table to change.

`DROP NOT NULL`

This form changes a column to allow null values.

`DROP EXPRESSION [ IF EXISTS ]`

This form turns a stored generated column into a normal base column. Existing data in the columns is retained, but future changes will no longer apply the generation expression. If `DROP EXPRESSION IF EXISTS` is specified and the column is not a generated column, no error is thrown. In this case a notice is issued instead.

`ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`, `SET GENERATED { ALWAYS | BY DEFAULT }`, `DROP IDENTITY [ IF EXISTS ]`

These forms change whether a column is an identity column or change the generation attribute of an existing identity column. See `CREATE TABLE` for details. Like `SET DEFAULT`, these forms

only affect the behavior of subsequent INSERT and UPDATE commands; they do not cause rows already in the table to change.

The *sequence_option* is an option supported by ALTER SEQUENCE such as INCREMENT BY. These forms alter the sequence that underlies an existing identity column.

> **Note**
>
> Amazon Aurora DSQL requires an explicit CACHE value when using ADD GENERATED AS IDENTITY. Additionally, identity columns are only supported on bigint columns.
> When using identity columns, the cache value should be carefully considered. For more information, see the Important callout on the CREATE SEQUENCE page.
> For guidance on how best to use identity columns based on workload patterns, see Working with sequences and identity columns.

SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }

This form sets the storage mode for a column. For details on the available storage modes, see Storage mode on the CREATE TABLE page.

ALTER CONSTRAINT

This form alters the attributes of an existing foreign key constraint. You can change a foreign key constraint between DEFERRABLE and NOT DEFERRABLE, and set whether it defaults to INITIALLY DEFERRED or INITIALLY IMMEDIATE. For a description of these options, see Deferrability on the CREATE TABLE page.

ADD *table_constraint* NOT VALID

This form adds a new CHECK or FOREIGN KEY constraint to a table. In Aurora DSQL, CHECK and FOREIGN KEY constraints added via ALTER TABLE ADD CONSTRAINT must use the NOT VALID option. Aurora DSQL creates the constraint but doesn't immediately validate it against existing data. This allows the constraint to be added without scanning the entire table. The constraint applies immediately to all new rows and updates.

After adding a constraint with NOT VALID, use ALTER TABLE ASYNC ... VALIDATE CONSTRAINT to validate that existing data also satisfies the constraint. The validation runs as an asynchronous DDL job. You can monitor its progress using sys.jobs.

ADD *table_constraint_using_index*

This form adds a new UNIQUE constraint to a table based on an existing unique index. All the columns of the index will be included in the constraint.

The index must be in a VALID state; adding a unique constraint using an index while the index is currently building is not supported.

If a constraint name is provided then the index will be renamed to match the constraint name. Otherwise the constraint will be named the same as the index.

After this command is executed, the index is "owned" by the constraint, in the same way as if the index had been built by a regular CREATE UNIQUE INDEX ASYNC command. In particular, dropping the constraint will make the index disappear too.

VALIDATE CONSTRAINT

This form validates a constraint that was previously created with the NOT VALID option. This command is an asynchronous DDL operation that doesn't block other transactions. When you run ALTER TABLE ASYNC ... VALIDATE CONSTRAINT, Aurora DSQL immediately returns a job_id.

You can monitor the status of this asynchronous job using the sys.jobs system view. You can also use sys.wait_for_job(*'job_id'*) to block the current session until the validation completes or fails.

The validation job scans the entire table to verify that all existing rows satisfy the constraint. Once validation completes successfully, Aurora DSQL marks the constraint as valid and the query planner enforces it for all queries. If validation fails because existing rows violate the constraint, the job fails and the constraint remains in the NOT VALID state.

This command validates only constraints that you created with the NOT VALID option. If you validate an already-valid constraint, Aurora DSQL succeeds without making changes.

DROP CONSTRAINT [IF EXISTS]

This form drops the specified constraint on a table, along with any index underlying the constraint. If IF EXISTS is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.

OWNER TO

This form changes the owner of the table to the specified user.

RENAME

The RENAME forms change the name of a table, the name of an individual column in a table, or the name of a constraint of the table. When renaming a constraint that has an underlying index, the index is renamed as well. There is no effect on the stored data.

SET SCHEMA

This form moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.

Parameters

IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing table to alter. If ONLY is specified before the table name, only that table is altered. If ONLY is not specified, the table and all its descendant tables (if any) are altered. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

column_name

Name of a new or existing column.

new_column_name

New name for an existing column.

new_name

New name for the table.

data_type

Data type of the new column.

table_constraint

A CHECK or FOREIGN KEY constraint definition. In Aurora DSQL, these constraints must be added with the NOT VALID option using ALTER TABLE ADD CONSTRAINT. See CREATE TABLE for the full constraint syntax.

constraint_name

Name of a new or existing constraint.

CASCADE

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column), and in turn all objects that depend on those objects.

RESTRICT

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

new_owner

The user name of the new owner of the table.

new_schema

The name of the schema to which the table will be moved.

Notes

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

If a dropped column is referenced as an `INCLUDE` column in the primary key, the primary key definition will be updated to remove the dropped column.

A table in Aurora DSQL can have at most 255 active columns at one time and a maximum of 1600 columns over the lifetime of the table. Dropping a column does not reclaim its attribute number. It removes it from the set of active columns but the dropped column continues to count against the lifetime limit of 1600 columns. For more information, see Database limits in Aurora DSQL.

CREATE INDEX

`CREATE INDEX` defines a new index. In Aurora DSQL, index creation is always asynchronous, so you must specify the `ASYNC` keyword. For information about how Aurora DSQL runs asynchronous index builds and how to monitor them, see Asynchronous indexes in Aurora DSQL.

Supported syntax

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

Description

`CREATE INDEX` constructs an index on the specified columns of the specified table. Indexes are primarily used to enhance database performance, though inappropriate use can result in slower performance.

You specify the key fields for the index as column names, or alternatively as expressions written in parentheses. You can specify multiple fields to create a multicolumn index.

An index field can be an expression computed from the values of one or more columns of the table row. Use this feature to obtain fast access to data based on some transformation of the basic data. For example, you can create an index on `upper(col)`, so a query with the condition `WHERE upper(col) = 'JIM'` can use that index.

All functions and operators used in an index definition must be immutable. That is, their results must depend only on their arguments and never on any outside influence, such as the contents of another table or the current time. This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression, remember to mark the function `IMMUTABLE` when you create it. If you change the definition of the user-defined function used by an index expression, be sure to drop and rebuilt the index.

The optional `WHERE` clause defines a *partial index*. A partial index contains entries for only the rows of the table that satisfy the predicate, rather than every row. When queries frequently target a well-defined subset of a table's rows, you can improve performance by creating an index on only that portion of the table. For example, a table might contain both active and archived records. If queries usually access only the active ones, you can index only the active rows.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when it creates the index, if data already exists, and each time you add data. Attempts to insert or update data that would result in duplicate entries generate an error.

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing index resembles the one that would have been created. The index name is required when `IF NOT EXISTS` is specified.

`INCLUDE`

The optional `INCLUDE` clause specifies a list of columns to include in the index as *non-key* columns. You can't use a non-key column in an index scan search qualification, and Aurora DSQL disregards it for purposes of any uniqueness or exclusion constraint that the index enforces. However, an index-only scan can return the contents of non-key columns without having to visit the index's table, because they are available directly from the index entry. Adding non-key columns therefore allows index-only scans for queries that otherwise couldn't use them.

Expressions aren't supported as included columns, because they can't be used in index-only scans.

`name`

The name of the index to create. You can't include a schema name here. Aurora DSQL always creates the index in the same schema as its parent table. The name of the index must be distinct from the name of any other relation, such as a table, sequence, index, or view, in that schema. If you omit the name, Aurora DSQL chooses a suitable name based on the parent table's name and the indexed column names.

`table_name`

The name, optionally schema-qualified, of the table to index.

`column_name`

The name of a column of the table.

expression

An expression based on one or more columns of the table. You usually must write the expression with surrounding parentheses, as shown in the syntax. However, you can omit the parentheses if the expression has the form of a function call.

NULLS FIRST

Specifies that nulls sort before non-nulls.

NULLS LAST

Specifies that nulls sort after non-nulls.

NULLS DISTINCT, NULLS NOT DISTINCT

Specifies whether null values are considered distinct, that is, not equal, for a unique index. The default is that they are distinct, so that a unique index can contain multiple null values in a column.

WHERE *predicate*

The optional `WHERE` clause specifies a Boolean expression, or predicate, that defines a partial index. The index includes only rows for which the predicate evaluates to true. The predicate can refer to any column of the table, not only the columns being indexed. As with index expressions, the predicate must contain only immutable functions, operators, and column references.

Aurora DSQL can use a partial index for a query only when it can prove that the query's `WHERE` conditions imply the index's predicate. If it can't, Aurora DSQL doesn't use the index for that query.

Examples

To create a unique index on the column `title` in the table `films`.

```
CREATE UNIQUE INDEX ASYNC title_idx ON films (title);
```

To create a unique index on the column `title` with included columns `director` and `rating` in the table `films`.

```
CREATE UNIQUE INDEX ASYNC title_idx ON films (title) INCLUDE (director, rating);
```

To create an index on the expression `lower(title)`, which allows efficient case-insensitive searches.

```
CREATE INDEX ASYNC ON films ((lower(title)));
```

This example omits the index name, so Aurora DSQL chooses a name, typically `films_lower_idx`.

To create an index with non-default sort ordering of nulls.

```
CREATE INDEX ASYNC title_idx_nulls_low ON films (title NULLS FIRST);
```

To create a partial index on `title` that indexes only the rows where `rating` is greater than 5:

```
CREATE INDEX ASYNC high_rating_idx ON films (title) WHERE rating > 5;
```

CREATE STATISTICS

Supported syntax

```
CREATE STATISTICS [ [ IF NOT EXISTS ] statistics_name ]
    ON ( expression )
    FROM table_name

CREATE STATISTICS [ [ IF NOT EXISTS ] statistics_name ]
    [ ( statistics_kind [, ... ] ) ]
    ON { column_name | ( expression ) }, { column_name | ( expression ) } [, ...]
    FROM table_name
```

Description

`CREATE STATISTICS` will create a new extended statistics object tracking data about the specified table. The statistics object will be created in the current database and will be owned by the user issuing the command.

The `CREATE STATISTICS` command has two basic forms. The first form allows univariate statistics for a single expression to be collected, providing benefits similar to an expression index without the overhead of index maintenance. This form does not allow the statistics kind to be specified, since the various statistics kinds refer only to multivariate statistics. The second form of the command allows multivariate statistics on multiple columns and/or expressions to be

collected, optionally specifying which statistics kinds to include. This form will also automatically cause univariate statistics to be collected on any expressions included in the list.

If a schema name is given (for example, `CREATE STATISTICS myschema.mystat ...`) then the statistics object is created in the specified schema. Otherwise it is created in the current schema. If given, the name of the statistics object must be distinct from the name of any other statistics object in the same schema.

Parameters

`IF NOT EXISTS`

Do not throw an error if a statistics object with the same name already exists. A notice is issued in this case. Note that only the name of the statistics object is considered here, not the details of its definition. Statistics name is required when `IF NOT EXISTS` is specified.

`statistics_name`

The name (optionally schema-qualified) of the statistics object to be created. If the name is omitted, Aurora DSQL chooses a suitable name based on the parent table's name and the defined column name(s) and/or expression(s).

`statistics_kind`

A multivariate statistics kind to be computed in this statistics object. Currently supported kinds are `ndistinct`, which enables n-distinct statistics, `dependencies`, which enables functional dependency statistics, and `mcv` which enables most-common values lists. If this clause is omitted, all supported statistics kinds are included in the statistics object. Univariate expression statistics are built automatically if the statistics definition includes any complex expressions rather than just simple column references.

`column_name`

The name of a table column to be covered by the computed statistics. This is only allowed when building multivariate statistics. At least two column names or expressions must be specified, and their order is not significant.

`expression`

An expression to be covered by the computed statistics. This may be used to build univariate statistics on a single expression, or as part of a list of multiple column names and/or expressions to build multivariate statistics. In the latter case, separate univariate statistics are built automatically for each expression in the list.

table_name

The name (optionally schema-qualified) of the table containing the column(s) the statistics are computed on.

Notes

You must be the owner of a table to create a statistics object reading it. Once created, however, the ownership of the statistics object is independent of the underlying table(s).

Expression statistics are per-expression and are similar to creating an index on the expression, except that they avoid the overhead of index maintenance. Expression statistics are built automatically for each expression in the statistics object definition.

Extended statistics are not currently used by the planner for selectivity estimations made for table joins.

In Aurora DSQL, you can create at most 5 extended statistics objects on a single table. If you exceed this limit, Aurora DSQL returns the error `more than 5 extended statistics per table are not allowed`. For more information, see Database limits in Aurora DSQL.

Examples

Create table `t1` with two functionally dependent columns, i.e., knowledge of a value in the first column is sufficient for determining the value in the other column. Then functional dependency statistics are built on those columns:

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- the number of matching rows will be drastically underestimated:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;
```

```
ANALYZE t1;

-- now the row count estimate is more accurate:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

Without functional-dependency statistics, the planner would assume that the two `WHERE` conditions are independent, and would multiply their selectivities together to arrive at a much-too-small row count estimate. With such statistics, the planner recognizes that the `WHERE` conditions are redundant and does not underestimate the row count.

Create table `t2` with two perfectly correlated columns (containing identical data), and an MCV list on those columns:

```
CREATE TABLE t2 (
    a   int,
    b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
                FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- valid combination (found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- invalid combination (not found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

The MCV list gives the planner more detailed information about the specific values that commonly appear in the table, as well as an upper bound on the selectivities of combinations of values that do not appear in the table, allowing it to generate better estimates in both cases.

ALTER STATISTICS

Supported syntax

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
```

```
ALTER STATISTICS name SET SCHEMA new_schema
ALTER STATISTICS name SET STATISTICS { new_target | DEFAULT }
```

Description

`ALTER STATISTICS` changes the parameters of an existing extended statistics object. Any parameters not specifically set in the `ALTER STATISTICS` command retain their prior settings.

You must own the statistics object to use `ALTER STATISTICS`. To change a statistics object's schema, you must also have `CREATE` privilege on the new schema. To alter the owner, you must be able to `SET ROLE` to the new owning role, and that role must have `CREATE` privilege on the statistics object's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the statistics object. However, a superuser can alter ownership of any statistics object anyway.)

Parameters

name

The name (optionally schema-qualified) of the statistics object to be altered.

new_owner

The user name of the new owner of the statistics object.

new_name

The new name for the statistics object.

new_schema

The new schema for the statistics object.

new_target

The statistic-gathering target for this statistics object for subsequent `ANALYZE` operations. In Aurora DSQL, the target can be set in the range `0` to `100`. Set it to `DEFAULT` to revert to using the system default statistics target (`default_statistics_target`). (Setting it to a value of `-1` is an obsolete spelling to get the same outcome.)

> **Important**
>
> If you set a target greater than `100`, Aurora DSQL returns the error `statistics target N exceeds maximum allowed value of 100`.

DROP STATISTICS

Supported syntax

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

Description

DROP STATISTICS removes statistics object(s) from the database. Only the statistics object's owner, the schema owner, or a superuser can drop a statistics object.

Parameters

IF EXISTS

Do not throw an error if the statistics object does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the statistics object to drop.

CASCADE, RESTRICT

These key words do not have any effect, since there are no dependencies on statistics.

Examples

To destroy two statistics objects in different schemas, without failing if they don't exist:

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

Compatibility

There is no DROP STATISTICS command in the SQL standard.

CREATE SEQUENCE

CREATE SEQUENCE — define a new sequence generator.

Important

In PostgreSQL, specifying CACHE is optional and defaults to 1. In a distributed system such as Amazon Aurora DSQL, sequence operations involve coordination, and a cache size of 1 can increase coordination overhead under high concurrency. While larger cache values allow sequence numbers to be served from locally preallocated ranges, improving throughput, unused reserved values can be lost, making gaps and ordering effects more visible. Because applications differ in their sensitivity to allocation ordering versus throughput, Amazon Aurora DSQL requires CACHE to be specified explicitly and currently supports `CACHE = 1` or `CACHE >= 65536`, providing a clear distinction between allocation behavior that is closer to strictly sequential generation and allocation optimized for highly concurrent workloads.

When `CACHE >= 65536`, sequence values remain guaranteed to be unique but might not be generated in strict increasing order across sessions, and gaps can occur, particularly when cached values are not fully consumed. These characteristics are consistent with PostgreSQL semantics for cached sequences under concurrent use, where both systems guarantee distinct values but do not guarantee strictly sequential ordering across sessions. Within a single client session, sequence values may not always appear strictly increasing, particularly outside explicit transactions. This behavior is similar to PostgreSQL deployments that use connection pooling. Allocation behavior closer to a single-session PostgreSQL environment can be achieved by using `CACHE = 1` or by obtaining sequence values within explicit transactions.

With `CACHE = 1`, sequence allocation follows PostgreSQL's non-cached sequence behavior. For guidance on how best to use sequences based on workload patterns, see Working with sequences and identity columns.

Supported syntax

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

where data_type is BIGINT
```

```
      and cache = 1 or cache >= 65536
```

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name *name*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. The sequence name must be distinct from the name of any other relation (table, sequence, index, view, materialized view, or foreign table) in the same schema.

After a sequence is created, you use the functions `nextval`, `currval`, and `setval` to operate on the sequence. These functions are documented in Sequence manipulation functions.

Although you cannot update a sequence directly, you can use a query like:

```
SELECT * FROM name;
```

to examine some of the parameters and current state of a sequence. In particular, the `last_value` field of the sequence shows the last value allocated by any session. (Of course, this value might be obsolete by the time it is printed, if other sessions are actively doing `nextval` calls.) Other parameters such as *increment* and *maxvalue* can be observed in the `pg_sequences` view.

Parameters

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing relation is anything like the sequence that would have been created — it might not even be a sequence.

name

The name (optionally schema-qualified) of the sequence to be created.

data_type

The optional clause AS *data_type* specifies the data type of the sequence. Valid types are `bigint`. `bigint` is the default. The data type determines the default minimum and maximum values of the sequence.

increment

The optional clause INCREMENT BY *increment* specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue **/ NO MINVALUE**

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If this clause is not supplied or NO MINVALUE is specified, then defaults will be used. The default for an ascending sequence is 1. The default for a descending sequence is the minimum value of the data type.

maxvalue **/ NO MAXVALUE**

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If this clause is not supplied or NO MAXVALUE is specified, then default values will be used. The default for an ascending sequence is the maximum value of the data type. The default for a descending sequence is -1.

CYCLE / NO CYCLE

The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

If NO CYCLE is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, NO CYCLE is the default.

start

The optional clause START WITH *start* allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The acceptable values for CACHE in Aurora DSQL are 1 or any number >= 65536. The minimum value is 1 (only one value can be generated at a time, meaning no cache).

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

The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped

as well. The specified table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Sequences are based on `bigint` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Because `nextval` and `setval` calls are never rolled back, sequence objects cannot be used if "gapless" assignment of sequence numbers is needed.

Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's `last_value` accordingly. Then, the next *`cache`*-1 uses of `nextval` within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a *`cache`* setting of 10, session A might reserve values 1..10 and return `nextval`=1, then session B might reserve values 11..20 and return `nextval`=11 before session A has generated `nextval`=2. Thus, with a *`cache`* setting of one it is safe to assume that `nextval` values are generated sequentially; with a *`cache`* setting greater than one you should only assume that the `nextval` values are all distinct, not that they are generated purely sequentially. Also, `last_value` will reflect the latest value reserved by any session, whether or not it has yet been returned by `nextval`.

Another consideration is that a `setval` executed on such a sequence will not be noticed by other sessions until they have used up any preallocated values they have cached.

Examples

Create an ascending sequence called `serial`, starting at 101:

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

Select the next number from this sequence:

```
SELECT nextval('serial');

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

Select the next number from this sequence:

```
SELECT nextval('serial');

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

Use this sequence in an `INSERT` command:

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

Reset the sequence to a specific value using `setval`:

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

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

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- Obtaining the next value is done using the `nextval()` function instead of the standard's `NEXT VALUE FOR` expression.
- The `OWNED BY` clause is a PostgreSQL extension.

`ALTER SEQUENCE`

`ALTER SEQUENCE` — change the definition of a sequence generator.

> **Important**
>
> When using sequences, the cache value should be carefully considered. For more information, see the Important callout on the CREATE SEQUENCE page.
> For guidance on how best to use sequences based on workload patterns, see Working with sequences and identity columns.

Supported syntax

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

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

Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameters not specifically set in the ALTER SEQUENCE command retain their prior settings.

You must own the sequence to use ALTER SEQUENCE. To change a sequence's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must be able to SET ROLE to the new owning role, and that role must have CREATE privilege on the sequence's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the sequence. However, a superuser can alter ownership of any sequence anyway.)

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

increment

The clause INCREMENT BY *increment* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue **/ NO MINVALUE**

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If NO MINVALUE is specified, the defaults of 1 and the minimum value of the data type for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue **/ NO MAXVALUE**

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If NO MAXVALUE is specified, the defaults of the maximum value of the data type and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

CYCLE

The optional CYCLE key word can be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

NO CYCLE

If the optional NO CYCLE key word is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, the old cycle behavior will be maintained.

start

The optional clause START WITH *start* changes the recorded start value of the sequence. This has no effect on the current sequence value; it simply sets the value that future ALTER SEQUENCE RESTART commands will use.

restart

The optional clause RESTART [WITH *restart*] changes the current value of the sequence. This is similar to calling the setval function with is_called = false: the specified value will be returned by the next call of nextval. Writing RESTART with no *restart* value is equivalent to supplying the start value that was recorded by CREATE SEQUENCE or last set by ALTER SEQUENCE START WITH.

In contrast to a setval call, a RESTART operation on a sequence is transactional and blocks concurrent transactions from obtaining numbers from the same sequence. If that's not the desired mode of operation, setval should be used.

cache

The clause CACHE *cache* enables sequence numbers to be preallocated and stored in memory for faster access. The value must be either 1 or some value >= 65536. If unspecified, the old cache value will be maintained. For more information about cache behavior, see the guidance under CREATE SEQUENCE.

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

The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying OWNED BY NONE removes any existing association, making the sequence "free-standing".

new_owner

The user name of the new owner of the sequence.

new_name

The new name for the sequence.

new_schema

The new schema for the sequence.

Notes

ALTER SEQUENCE will not immediately affect nextval results in backends, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current backend will be affected immediately.

ALTER SEQUENCE does not affect the currval status for the sequence.

ALTER SEQUENCE may cause other transactions to OCC.

For historical reasons, ALTER TABLE can be used with sequences too; but the only variants of ALTER TABLE that are allowed with sequences are equivalent to the forms shown above.

Examples

Restart a sequence called serial, at 105:

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

Compatibility

ALTER SEQUENCE conforms to the SQL standard, except for the AS, START WITH, OWNED BY, OWNER TO, RENAME TO, and SET SCHEMA clauses, which are PostgreSQL extensions.

DROP SEQUENCE

DROP SEQUENCE — remove a sequence.

Supported syntax

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

Description

DROP SEQUENCE removes sequence number generators. A sequence can only be dropped by its owner or a superuser.

Parameters

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of a sequence.

CASCADE

Automatically drop objects that depend on the sequence, and in turn all objects that depend on those objects.

RESTRICT

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

To remove the sequence `seq`:

```
DROP SEQUENCE seq;
```

Compatibility

`DROP SEQUENCE` conforms to the SQL standard, except that the standard only allows one sequence to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

CREATE VIEW

`CREATE VIEW` defines a new persistent view. Aurora DSQL does not support temporary views; only permanent views are supported.

Supported syntax

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

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE or REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order and with the same data types), but it may add additional columns to the end of the list. The calculations giving rise to the output columns may be different.

If a schema name is given, such as `CREATE VIEW myschema.myview ...`) then the view is created in the specified schema. Otherwise, it is created in the current schema.

The name of the view must be distinct from the name of any other relation (table, index, view) in the same schema.

Parameters

`CREATE VIEW` supports various parameters to control the behavior of automatically updatable views.

`RECURSIVE`

Creates a recursive view. The syntax: `CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;` is equivalent to `CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...) SELECT column_names FROM view_name;`.

A view column name list must be specified for a recursive view.

`name`

The name of the view to be created, which may be optionally schema-qualified. A column name list must be specified for a recursive view.

`column_name`

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

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

This clause specifies optional parameters for a view; the following parameters are supported.

- `check_option (enum)` — This parameter may be either `local` or `cascaded`, and is equivalent to specifying `WITH [ CASCADED | LOCAL ] CHECK OPTION`.

- `security_barrier (boolean)`—This should be used if the view is intended to provide row-level security. Aurora DSQL does not currently support row-level security, but this option will still force the view's `WHERE` conditions (and any conditions using operators which are marked as `LEAKPROOF`) to be evaluated first.
- `security_invoker (boolean)`—This option causes the underlying base relations to be checked against the privileges of the user of the view rather than the view owner. See the notes below for full details.

All of the above options can be changed on existing views using `ALTER VIEW`.

`query`

A `SELECT` or `VALUES` command which will provide the columns and rows of the view.

`WITH [ CASCADED | LOCAL ] CHECK OPTION`

This option controls the behavior of automatically updatable views. When this option is specified, `INSERT` and `UPDATE` commands on the view will be checked to ensure that new rows satisfy the view-defining condition (that is, the new rows are checked to ensure that they are visible through the view). If they are not, the update will be rejected. If the `CHECK OPTION` is not specified, `INSERT` and `UPDATE` commands on the view are allowed to create rows that are not visible through the view.

`LOCAL`—New rows are only checked against the conditions defined directly in the view itself. Any conditions defined on underlying base views are not checked (unless they also specify the `CHECK OPTION`).

`CASCADED`—New rows are checked against the conditions of the view and all underlying base views. If the `CHECK OPTION` is specified, and neither `LOCAL` nor `CASCADED` are specified, then `CASCADED` is assumed.

> **Note**
>
> The `CHECK OPTION` may not be used with `RECURSIVE` views. The `CHECK OPTION` is only supported on views that are automatically updatable.

Notes

Use the `DROP VIEW` statement to drop views.

The names and data types of the view's columns should be carefully considered. For example, CREATE VIEW vista AS SELECT 'Hello World'; is not recommended because the column name defaults to `?column?;`. Also, the column data type defaults to `text`, which might not be what you wanted.

A better approach is to explicitly specify the column name and data type, such as: `CREATE VIEW vista AS SELECT text 'Hello World' AS hello;`.

By default, access to the underlying base relations referenced in the view is determined by the permissions of the view owner. In some cases, this can be used to provide secure but restricted access to the underlying tables. However, not all views are secure against tampering.

- If the view has the `security_invoker` property set to true, access to the underlying base relations is determined by the permissions of the user executing the query, rather than the view owner. Thus, the user of a security invoker view must have the relevant permissions on the view and its underlying base relations.
- If any of the underlying base relations is a security invoker view, it will be treated as if it had been accessed directly from the original query. Thus, a security invoker view will always check its underlying base relations using the permissions of the current user, even if it is accessed from a view without the `security_invoker` property.
- Functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore, the user of a view must have permissions to call all functions used by the view. Functions in the view are executed with the privileges of the user executing the query or the function owner, depending on whether the functions are defined as `SECURITY INVOKER` or `SECURITY DEFINER`.
- The user creating or replacing a view must have `USAGE` privileges on any schemas referred to in the view query, in order to look up the referenced objects in those schemas.
- When `CREATE OR REPLACE VIEW` is used on an existing view, only the view's defining `SELECT` rule, plus any `WITH ( ... )` parameters and its `CHECK OPTION` are changed. Other view properties, including ownership, permissions, and non-SELECT rules, remain unchanged. You must own the view to replace it (this includes being a member of the owning role).

Updatable views

Simple views are automatically updatable: the system will allow `INSERT`, `UPDATE`, and `DELETE` statements to be used on the view in the same way as on a regular table. A view is automatically updatable if it satisfies all of the following conditions:

- The view must have exactly one entry in its FROM list, which must be a table or another updatable view.
- The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level.
- The view definition must not contain set operations (UNION, INTERSECT, or EXCEPT) at the top level.
- The view's select list must not contain any aggregates, window functions, or set-returning functions.

An automatically updatable view may contain a mix of updatable and non-updatable columns. A column is updatable if it's a simple reference to an updatable column of the underlying base relation. Otherwise, the column is read-only, and an error occurs if an INSERT or UPDATE statement attempts to assign a value to it.

A more complex view that doesn't satisfy all these conditions is read-only by default: the system doesn't allow an insert, update, or delete on the view.

> **Note**
>
> The user performing the insert, update, or delete on the view must have the corresponding insert, update, or delete privilege on the view. By default, the view's owner must have the relevant privileges on the underlying base relations, while the user performing the update doesn't need any permissions on the underlying base relations. However, if the view has security_invoker set to true, the user performing the update, rather than the view owner, must have the relevant privileges on the underlying base relations.

Examples

To create a view consisting of all comedy films.

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

Create a view with LOCAL CHECK OPTION.

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

Create a recursive view.

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

Compatibility

`CREATE OR REPLACE VIEW` is a PostgreSQL language extension. The `WITH ( ... )` clause is an extension as well, as are security barrier views and security invoker views. Aurora DSQL supports these language extensions.

`ALTER VIEW`

The `ALTER VIEW` statement allows changing various properties of an existing view, and Aurora DSQL supports all the PostgreSQL syntax for this command.

Supported syntax

```
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 [, ... ] )
```

Description

`ALTER VIEW` changes various auxiliary properties of a view. (If you want to modify the view's defining query, use `CREATE OR REPLACE VIEW`.) You must own the view to use `ALTER VIEW`.

To change a view's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must be able to SET ROLE to the new owning role, and that role must have CREATE privilege on the view's schema.

Parameters

name

The name (optionally schema-qualified) of an existing view.

column_name

Name of an existing column, or new name for an existing column.

IF EXISTS

Don't throw an error if the view doesn't exist. A notice is issued in this case.

SET/DROP DEFAULT

These forms set or remove the default value for a column. The default value for a view column is substituted into any INSERT or UPDATE command where the target is the view.

new_owner

The user name of the new owner of the view.

new_name

The new name for the view.

new_schema

The new schema for the view.

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

Sets a view option. The following are supported options:

- check_option (enum) - Changes the check option of the view. The value must be local or cascaded.
- security_barrier (boolean) - Changes the security-barrier property of the view.
- security_invoker (boolean) - Changes the security-invoker property of the view.

RESET (view_option_name [, ...])

Resets a view option to its default value.

Examples

Renaming the view `foo` to `bar`:

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

Attaching a default column value to an updatable view:

```
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
```

Compatibility

`ALTER VIEW` is a PostgreSQL extension of the SQL standard that Aurora DSQL supports.

DROP VIEW

The `DROP VIEW` statement removes an existing view. Aurora DSQL supports the full PostgreSQL syntax for this command.

Supported syntax

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

Description

`DROP VIEW` drops an existing view. To execute this command you must be the owner of the view.

Parameters

`IF EXISTS`

Don't throw an error if the view doesn't exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of the view to remove.

CASCADE

Automatically drop objects that depend on the view (such as other views), and in turn all objects that depend on those objects.

RESTRICT

Refuse to drop the view if any objects depend on it. This is the default.

Examples

```
DROP VIEW kinds;
```

Compatibility

This command conforms to the SQL standard, except that the standard only allows one view to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension that Aurora DSQL supports.

SET CONSTRAINTS

`SET CONSTRAINTS` sets the behavior of foreign key constraint checking within the current transaction.

Supported syntax

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint checking within the current transaction. `IMMEDIATE` constraints are checked at the end of each statement. `DEFERRED` constraints aren't checked until the transaction commits. Each constraint has its own `IMMEDIATE` or `DEFERRED` mode.

When you create a constraint, you assign it one of three characteristics: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, or `NOT DEFERRABLE`. The third class is always `IMMEDIATE`, and the `SET CONSTRAINTS` command doesn't change it. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by `SET CONSTRAINTS`.

`SET CONSTRAINTS` with a list of constraint names changes the mode of just those constraints. All named constraints must be deferrable. `SET CONSTRAINTS ALL` changes the mode of all deferrable constraints.

When `SET CONSTRAINTS` changes the mode of a constraint from `DEFERRED` to `IMMEDIATE`, the new mode takes effect retroactively. Aurora DSQL immediately checks any outstanding data modifications that would otherwise have been deferred to the end of the transaction. If any such constraint is violated, the `SET CONSTRAINTS` command fails and doesn't change the constraint mode.

For a description of the deferrability options (`DEFERRABLE`, `NOT DEFERRABLE`, `INITIALLY DEFERRED`, `INITIALLY IMMEDIATE`), see Deferrability on the `CREATE TABLE` page.

> **Foreign key constraints only**
>
> In Aurora DSQL, `SET CONSTRAINTS` applies to foreign key constraints only. Primary key, unique, `NOT NULL`, and `CHECK` constraints are always checked immediately when a row is inserted or modified, even when you run `SET CONSTRAINTS ALL DEFERRED`.

Notes

Aurora DSQL changes the behavior of constraints only within the current transaction. To use `SET CONSTRAINTS`, run it inside an explicit transaction block.

Compatibility

This command complies with the behavior defined in the SQL standard, with the restriction that Aurora DSQL supports deferral for foreign key constraints only.

`CREATE FUNCTION`

`CREATE FUNCTION` defines a new function.

> **Important**
>
> In Amazon Aurora DSQL, `CREATE FUNCTION` only supports SQL-language functions (`LANGUAGE sql`).

Supported syntax

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE sql
    | { IMMUTABLE | STABLE | VOLATILE }
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | RETURN expression
    | BEGIN ATOMIC statement; [ statement; ... ] END
  } ...

where LANGUAGE must be sql
```

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition. To be able to define a function, you must have the `USAGE` privilege on the language.

If a schema name is included, then the function is created in the specified schema. Otherwise, the function is created in the current schema. The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different argument types can share a name (this is called *overloading*).

To replace the current definition of an existing function, use `CREATE OR REPLACE FUNCTION`. You can't change the name or argument types of a function this way (this would actually create a new, distinct function). Also, you can't use `CREATE OR REPLACE FUNCTION` to change the return type of an existing function. To do that, you must drop and recreate the function.

When `CREATE OR REPLACE FUNCTION` is used to replace an existing function, the ownership and permissions of the function don't change. All other function properties are assigned the values specified or implied in the command. You must own the function to replace it (this includes being a member of the owning role).

If you drop and then recreate a function, the new function isn't the same entity as the old; you will have to drop existing rules, views, and so on, that refer to the old function. Use CREATE OR REPLACE FUNCTION to change a function definition without breaking objects that refer to the function.

You become the owner of a function that you created.

To be able to create a function, you must have USAGE privilege on the argument types and the return type.

A SQL function wraps one or more SQL statements into a reusable, callable unit. Use functions to centralize common expressions or calculations, apply consistent logic across multiple queries, and simplify application code by calling a single function instead of repeating the same SQL.

Parameters

name

The name (optionally schema-qualified) of the function to create.

argmode

The mode of an argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. VARIADIC must be the last input parameter; only OUT arguments can follow it. OUT and INOUT arguments can't be combined with the RETURNS TABLE notation.

argname

The name of an argument. In a SQL function, you can refer to an argument by name in the function body. The name of an OUT or INOUT argument is also significant, because it defines the corresponding column name in the result row type.

argtype

The data type of an argument. This can be any Aurora DSQL-supported base type, array type, or domain. For more information, see Supported data types in Aurora DSQL. You can also reference the type of an existing table column by using the %TYPE notation.

default_expr

An expression to be used as the default value if the argument isn't specified. The expression has to be coercible to the argument type of the parameter. Only IN and INOUT arguments can have a default value. Each argument after one with a default value must also have a default value.

rettype

The return data type. This can be any Aurora DSQL-supported base type, array type, or domain. For more information, see Supported data types in Aurora DSQL. To return a set of rows, use `SETOF` *`rettype`* or `RETURNS TABLE`. If the function isn't supposed to return a value, specify `void` as the return type. When there are `OUT` or `INOUT` parameters, the `RETURNS` clause can be omitted, and the function returns a `record` or the corresponding row type.

`RETURNS TABLE ( column_name column_type [, ...] )`

Specifies that the function returns a set of rows with the named columns and their types. This is equivalent to declaring one or more `OUT` parameters and specifying `SETOF` as the return type.

`LANGUAGE sql`

The language used to implement the function. In Aurora DSQL, this must be `sql`. If you omit the `LANGUAGE` clause and provide a `BEGIN ATOMIC` or `RETURN` body, `sql` is assumed. A string-constant body (`AS '`*`definition`*`'`) requires an explicit `LANGUAGE sql` clause; omitting it fails with the error `no language specified`.

`IMMUTABLE` / `STABLE` / `VOLATILE`

These attributes inform the query optimizer about the behavior of the function. You can specify at most one. If you omit all three, the function defaults to `VOLATILE`.

- `IMMUTABLE` indicates that the function can't modify the database and always returns the same result when given the same argument values.
- `STABLE` indicates that the function can't modify the database. Within a single table scan, it consistently returns the same result for the same argument values, but the result might change across SQL statements.
- `VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made.

`CALLED ON NULL INPUT` / `RETURNS NULL ON NULL INPUT` / `STRICT`

`CALLED ON NULL INPUT` (the default) indicates that the function is called normally when some of its arguments are null. `RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns null when any of its arguments are null. If specified, Aurora DSQL doesn't execute the function when any argument is null; instead, the function automatically returns null.

[EXTERNAL] SECURITY INVOKER / [EXTERNAL] SECURITY DEFINER

SECURITY INVOKER (the default) indicates that the function is to be executed with the privileges of the user that calls it. SECURITY DEFINER specifies that the function is to be executed with the privileges of the user that owns it. The key word EXTERNAL is allowed for SQL conformance, but it's optional.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

Aurora DSQL accepts and stores the PARALLEL marking. However, this marking doesn't affect query execution.

COST *execution_cost*

A positive number giving the estimated execution cost for the function. If the function returns a set, this value is the estimated cost per returned row. The query planner uses this estimate when it does not inline the function.

ROWS *result_rows*

A positive number giving the estimated number of rows that the query planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT }

The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the value of the parameter that is current when CREATE FUNCTION is run as the value to be applied when the function is entered.

AS '*definition*'

A string constant defining the function; the meaning depends on the language. For a SQL-language function, it is the SQL statements that make up the function body.

RETURN *expression*

Defines the function body using the SQL-standard syntax for a function that returns a single expression. This is an alternative to the string-constant form.

BEGIN ATOMIC *statement*; [...] END

Defines the function body using the SQL-standard syntax. The body consists of one or more SQL statements. This form is only valid for LANGUAGE sql. The statements in an atomic body

are tracked as dependencies of the function. For example, a table referenced in the body cannot be dropped without also dropping the function, or using `DROP ... CASCADE`.

Notes

Use the `DROP FUNCTION` statement to remove a function. You can also grant and revoke privileges on a function with `GRANT` and `REVOKE` (for example, `REVOKE ALL ON FUNCTION`).

Any function that reads or modifies data in tables is subject to the same transaction and concurrency behavior as the equivalent SQL statements run directly. Functions declared `IMMUTABLE` or `STABLE` must not modify the database.

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. You can't change the argument types or return type of a function this way; instead, drop and recreate the function.

To rename a function, change its owner, or move it to another schema, use `ALTER FUNCTION`. To change any other attributes, use `CREATE OR REPLACE FUNCTION` with the desired changes. For more information, see `ALTER FUNCTION`.

Function overloading is supported: you can define several functions with the same name as long as their input argument types differ. A user-defined function can also call other functions, and can shadow a built-in function of the same name and argument types.

For a function that returns a set (`SETOF` or `RETURNS TABLE`) whose body contains more than one statement, only the result of the final statement is returned. The function still executes the earlier statements, but discards their results. This is the standard behavior for SQL-language functions. Structure the function so that its final statement produces the complete result set.

A set-returning function doesn't guarantee any particular row order. To control row order, include an `ORDER BY` clause in the query.

Considerations

The following `CREATE FUNCTION` capabilities return the error shown.

`LEAKPROOF`

Returns the error `only superuser can define a leakproof function`. `NOT LEAKPROOF` (the default) is accepted.

`TRANSFORM FOR TYPE`

Returns the error `transform for type` *`type`* `language "sql" does not exist`.

`SUPPORT` (planner support function)

Returns the error `function` *`name`*`(internal) does not exist`.

User-defined composite types (`CREATE TYPE ... AS`), including functions that return a composite type

Returns the error `unsupported statement: CompositeType`.

Aurora DSQL doesn't support `CREATE PROCEDURE`, which fails with the error `PROCEDURE is not supported`.

The `WINDOW` attribute is accepted syntactically but isn't useful for SQL-language functions. Avoid using this attribute.

Examples

Create a simple SQL function that adds two integers, using a string-constant body:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'SELECT $1 + $2;'
    LANGUAGE sql
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Create the same function using the SQL-standard `RETURN` body and named arguments:

```
CREATE FUNCTION add(a integer, b integer) RETURNS integer
    LANGUAGE sql
    IMMUTABLE
    RETURN a + b;
```

Create a function that returns a set of rows using `RETURNS TABLE` and a `BEGIN ATOMIC` body:

```
CREATE FUNCTION employees_in_dept(dept_id integer)
    RETURNS TABLE (id integer, name text)
    LANGUAGE sql
```

```
    STABLE
    BEGIN ATOMIC
        SELECT id, name FROM employees WHERE department_id = dept_id;
    END;
```

Create a function that runs with the privileges of its owner, and sets a configuration parameter for the duration of the call:

```
CREATE FUNCTION current_department() RETURNS text
    LANGUAGE sql
    STABLE
    SECURITY DEFINER
    SET search_path = app
    AS 'SELECT current_setting(''app.department'');';
```

Compatibility

A `CREATE FUNCTION` command is defined in the SQL standard. The PostgreSQL implementation can be used in a compatible way but has many extensions. Conversely, the SQL standard specifies a number of optional features that are not implemented in PostgreSQL.

The following are important compatibility issues:

- `OR REPLACE` is a PostgreSQL extension.
- For compatibility with some other database systems, *`argmode`* can be written either before or after *`argname`*. But only the first way is standard-compliant.
- For parameter defaults, the SQL standard specifies only the syntax with the `DEFAULT` key word. The syntax with = is used in T-SQL and Firebird.
- The `SETOF` modifier is a PostgreSQL extension.
- Only SQL is standardized as a language.
- All other attributes except `CALLED ON NULL INPUT` and `RETURNS NULL ON NULL INPUT` are not standardized.
- For the body of `LANGUAGE sql` functions, the SQL standard only specifies the *`sql_body`* form.

Simple `LANGUAGE sql` functions can be written in a way that is both standard-conforming and portable to other implementations. More complex functions using advanced features, optimization attributes, or other languages will necessarily be specific to PostgreSQL in a significant way.

ALTER FUNCTION

ALTER FUNCTION changes the definition of a function.

Supported syntax

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
```

Description

ALTER FUNCTION changes the definition of an existing function.

You must own the function to use ALTER FUNCTION. To change a function's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must be a member of the new owning role, and that role must have CREATE privilege on the function's schema.

RENAME TO

Changes the name of the function.

OWNER TO

Changes the owner of the function to the specified user.

SET SCHEMA

Moves the function into another schema.

Parameters

name

The name (optionally schema-qualified) of an existing function to alter.

argmode

The mode of an argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. The function uses only the input arguments for identification, so list only the IN, INOUT, and VARIADIC arguments.

argname

The name of an argument.

argtype

The data type(s) of the function's arguments.

new_name

The new name for the function.

new_owner

The new owner of the function.

new_schema

The new schema for the function.

Considerations

To change other attributes than those listed above, use `CREATE OR REPLACE FUNCTION`. For more information, see `CREATE FUNCTION`.

To add, change, or remove a comment on a function, use the separate `COMMENT ON FUNCTION` command.

Examples

Rename the function `add` (which takes two `integer` arguments) to `sum_two`:

```
ALTER FUNCTION add(integer, integer) RENAME TO sum_two;
```

Change the owner of the function `add` to the current user:

```
ALTER FUNCTION add(integer, integer) OWNER TO CURRENT_USER;
```

Move the function `add` into the schema `utils`:

```
ALTER FUNCTION add(integer, integer) SET SCHEMA utils;
```

Compatibility

Aurora DSQL supports the `RENAME TO`, `OWNER TO`, and `SET SCHEMA` forms of the PostgreSQL `ALTER FUNCTION` statement. The other forms available in PostgreSQL are not supported.

Migrating from PostgreSQL to Aurora DSQL

Aurora DSQL is designed to be PostgreSQL compatible, supporting core relational features such as ACID transactions, secondary indexes, joins, and standard DML operations. Most existing PostgreSQL applications can migrate to Aurora DSQL with minimal changes.

This section provides practical guidance for migrating your application to Aurora DSQL, including framework compatibility, migration patterns, and architectural considerations.

Framework and ORM compatibility

Aurora DSQL uses the standard PostgreSQL wire protocol, ensuring compatibility with PostgreSQL drivers and frameworks. Most popular ORMs work with Aurora DSQL with minimal or no changes. See the section called “Aurora DSQL Adapters” for reference implementations and available ORM integrations.

Common migration patterns

When migrating from PostgreSQL to Aurora DSQL, some features work differently or have alternative syntax. This section provides guidance on common migration scenarios.

DDL operation alternatives

Aurora DSQL provides modern alternatives to traditional PostgreSQL DDL operations:

Index creation

Use `CREATE INDEX ASYNC` instead of `CREATE INDEX` for non-blocking index creation.

Benefit: Zero-downtime index creation on large tables.

Data removal

Use `DELETE FROM table_name` instead of `TRUNCATE`.

Alternative: For complete table recreation, use `DROP TABLE` followed by `CREATE TABLE`.

System configuration

Aurora DSQL is fully managed, so configuration is handled automatically based on workload patterns. Use the AWS Management Console or API to manage cluster settings.

Benefit: No need for database tuning or parameter management.

Schema design patterns

Adapt these common PostgreSQL patterns for Aurora DSQL compatibility:

Referential integrity patterns

Aurora DSQL supports foreign keys, table relationships, and `JOIN` operations. Use foreign keys to keep data consistent across your tables. To learn more, see Working with foreign key constraints in Aurora DSQL.

Pattern: Add foreign keys when you create your tables. Aurora DSQL then checks that related rows exist. You can also add checks in your app code for custom error handling.

Temporary data handling

Use CTEs, subqueries, or regular tables with cleanup logic instead of temporary tables.

Alternative: Create tables with session-specific names and clean them up in your application.

Understanding architectural differences

Aurora DSQL's distributed, serverless architecture intentionally differs from traditional PostgreSQL in several areas. These differences enable Aurora DSQL's key benefits of simplicity and scale.

Simplified database model

Single database per cluster

Aurora DSQL provides one built-in database named `postgres` per cluster.

Migration tip: If your application uses multiple databases, create separate Aurora DSQL clusters for logical separation, or use schemas within a single cluster.

No temporary tables

For temporary data handling, you SHOULD use common table expressions (CTEs) and subqueries, which provide flexible alternatives for complex queries.

Alternative: Use CTEs with `WITH` clauses for temporary result sets, or regular tables with unique naming for session-specific data.

Automatic storage management

Aurora DSQL eliminates tablespaces and manual storage management. Storage automatically scales and optimizes based on your data patterns.

Benefit: No need to monitor disk space, plan storage allocation, or manage tablespace configurations.

Modern application patterns

Aurora DSQL encourages modern application development patterns that improve maintainability and performance:

Application-level logic instead of database triggers

For trigger-like functionality, implement event-driven logic in your application layer.

Migration strategy: Move trigger logic to application code, use event-driven architectures with AWS services like EventBridge, or implement audit trails using application logging.

SQL functions for data processing

Aurora DSQL supports SQL-based functions but not procedural languages like PL/pgSQL.

Alternative: Use SQL functions for data transformations, or move complex logic to your application layer or AWS Lambda functions.

Optimistic concurrency control instead of pessimistic locking

Aurora DSQL uses optimistic concurrency control (OCC), a lock-free approach that differs from traditional database locking mechanisms. Instead of acquiring locks that block other transactions, Aurora DSQL allows transactions to proceed without blocking and detects conflicts at commit time. This eliminates deadlocks and prevents slow transactions from blocking other operations.

Key difference: When conflicts occur, Aurora DSQL returns a serialization error rather than making transactions wait for locks. This requires applications to implement retry logic, similar to handling lock timeouts in traditional databases, but conflicts are resolved immediately rather than causing blocking waits.

Design pattern: Implement idempotent transaction logic with retry mechanisms. Design schemas to minimize contention by using random primary keys and spreading updates across your key range. For details, see Concurrency control in Aurora DSQL.

Operational simplifications

Aurora DSQL eliminates many traditional database maintenance tasks, reducing operational overhead:

No manual maintenance required

Aurora DSQL automatically manages storage optimization, statistics collection, and performance tuning. Traditional maintenance commands like `VACUUM` are handled by the system.

Benefit: Eliminates the need for database maintenance windows, vacuum scheduling, and system parameter tuning.

Automatic partitioning and scaling

Aurora DSQL automatically partitions and distributes your data based on access patterns. Use UUIDs or application-generated IDs for optimal distribution.

Migration tip: Remove manual partitioning logic and let Aurora DSQL handle data distribution. Use UUIDs or application-generated IDs for optimal distribution. If your application requires sequential identifiers, see Sequences and identity columns.

Agentic migration with AI tools

AI coding agents can accelerate your migration to Aurora DSQL by analyzing schemas, transforming code, and executing DDL migrations with built-in safety checks.

Using Kiro for migration

Coding agents such as Kiro can help you analyze and migrate your PostgreSQL code to Aurora DSQL:

- **Schema analysis:** Upload your existing schema files and ask Kiro to identify potential compatibility issues and suggest alternatives
- **Code transformation:** Provide your application code and ask Kiro to help refactor trigger logic, replace sequences with UUIDs, or modify transaction patterns
- **Migration planning:** Ask Kiro to create a step-by-step migration plan based on your specific application architecture
- **DDL migrations:** Execute schema modifications using the table recreation pattern with built-in safety checks and user verification

Example prompts:

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

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

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

"Drop the legacy_status column from the orders table"

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

DDL migration with table recreation

When using AI agents with the Aurora DSQL MCP server, certain ALTER TABLE operations use a *table recreation pattern* that safely migrates your data. The agent handles the complexity while keeping you informed at each step.

The following operations use the table recreation pattern:

Operation	Approach
`ALTER COLUMN TYPE`	Cast data type during migration
`ALTER COLUMN SET/DROP NOT NULL`	Change constraint in new table definition

Operation	Approach
`ALTER COLUMN SET/DROP DEFAULT`	Define default in new table definition
`ADD/DROP CONSTRAINT`	Include or remove constraint in new table
`MODIFY PRIMARY KEY`	Define new PK with uniqueness validation
Split/Merge columns	Use SPLIT_PART, SUBSTRING, or CONCAT

The following ALTER TABLE operations are supported directly without table recreation:

- `ALTER TABLE ... RENAME COLUMN` – Rename a column
- `ALTER TABLE ... RENAME TO` – Rename a table
- `ALTER TABLE ... ADD COLUMN` – Add a new column
- `ALTER TABLE ... DROP COLUMN` – Drop a column, except a primary key column

Safety features: When executing DDL migrations, AI agents present the migration plan, verify data compatibility, confirm row counts, and request explicit approval before any destructive operations like DROP TABLE.

Batched migrations: For tables exceeding 3,000 rows, the agent automatically batches the migration in increments of 500-1,000 rows to stay within transaction limits.

Aurora DSQL MCP server

The Aurora DSQL Model Context Protocol (MCP) server allows AI assistants to connect directly to your Aurora DSQL cluster and search Aurora DSQL documentation. This enables the AI to:

- Analyze your existing schema and suggest migration changes
- Execute DDL migrations with the table recreation pattern
- Test queries and verify compatibility during migration
- Provide accurate, up-to-date guidance based on the latest Aurora DSQL documentation

To use the Aurora DSQL MCP server with AI assistants, see the setup instructions for the Aurora DSQL MCP server.

Aurora DSQL considerations for PostgreSQL compatibility

Aurora DSQL has feature support differences from self-managed PostgreSQL that enable its distributed architecture, serverless operation, and automatic scaling. Most applications work within these differences without modification.

For general considerations, see Considerations for working with Amazon Aurora DSQL. For quotas and limits, see Cluster quotas and database limits in Amazon Aurora DSQL.

- Aurora DSQL uses a single built-in database named `postgres` per cluster. For logical separation, create separate Aurora DSQL clusters or use schemas within a single cluster.
- The `postgres` database uses UTF-8 character encoding, which provides broad international character support.
- The database uses the C collation only.
- Aurora DSQL uses UTC as the system timezone. Postgres stores all timezone-aware dates and times internally in UTC. You can set the `TimeZone` configuration parameter to convert how it is displayed to the client and serve as the default for client input that the server will use to convert to UTC internally.
- The transaction isolation level is fixed at PostgreSQL `Repeatable Read`.
- Transactions have the following constraints:
  - DDL and DML operations require separate transactions
  - A transaction can include only 1 DDL statement
  - A transaction can modify up to 3,000 rows, regardless of the number of secondary indexes
  - The 3,000-row limit applies to all DML statements (`INSERT`, `UPDATE`, `DELETE`)
- Database connections time out after 1 hour.
- Aurora DSQL manages permissions through schema-level grants. Admin users create schemas using `CREATE SCHEMA` and grant access using `GRANT USAGE ON SCHEMA`. Admin users manage objects in the public schema, while non-admin users create objects in user-created schemas for clear ownership boundaries. For more information, see Authorizing database roles to use SQL in your database.

Need help with migration?

If you encounter features that are critical for your migration but not currently supported in Aurora DSQL, see Providing feedback on Amazon Aurora DSQL for information on how to share feedback with AWS.

Concurrency control in Aurora DSQL

Concurrency allows multiple sessions to access and modify data simultaneously without compromising data integrity and consistency. Aurora DSQL provides PostgreSQL compatibility while implementing a modern, lock-free concurrency control mechanism. It maintains full ACID compliance through snapshot isolation, ensuring data consistency and reliability.

A key advantage of Aurora DSQL is its lock-free architecture, which eliminates common database performance bottlenecks. Aurora DSQL prevents slow transactions from blocking other operations and eliminates the risk of deadlocks. This approach makes Aurora DSQL particularly valuable for high-throughput applications where performance and scalability are critical.

Transaction snapshots

For explicit transactions, Aurora DSQL takes a snapshot when it processes `BEGIN` or `START TRANSACTION`. This differs from PostgreSQL, where a `REPEATABLE READ` transaction takes its snapshot when the first statement that requires one runs.

Concurrency control responses

Aurora DSQL uses optimistic concurrency control (OCC), which works differently from traditional lock-based systems. Instead of using locks, OCC evaluates conflicts at commit time. This process of commit time conflict evaluation is also called adjudication. When Aurora DSQL detects a conflict, it returns a PostgreSQL serialization failure with SQLSTATE code `40001`. The response message includes an OCC code that identifies the type of conflict:

OC000 — Data conflict

Two transactions attempted to modify the same row. The transaction with the earliest commit time succeeds, and the conflicting transaction receives the OC000 response:

```
ERROR: change conflicts with another transaction (OC000) (SQLSTATE 40001)
```

OC001 — Schema conflict

The session's cached schema catalog is out of date. When Aurora DSQL detects that the catalog version has changed since the session loaded its cache, and the transaction can't safely rebase to the current version, the transaction receives the OC001 response:

```
ERROR: schema has been updated by another transaction (OC001) (SQLSTATE 40001)
```

Any operation that modifies the schema catalog can cause an OC001 response, including DDL statements such as `CREATE TABLE` and `ALTER TABLE`, as well as `GRANT` and `REVOKE` statements. For more information, see DDL and distributed transactions in Aurora DSQL.

Design your applications to implement retry logic to handle these responses. The ideal design pattern is idempotent, enabling transaction retry as a first recourse whenever possible. The recommended logic is similar to the abort and retry logic in a standard PostgreSQL lock timeout or deadlock situation. However, OCC requires your applications to exercise this logic more frequently.

Data conflict types

Because of the Aurora DSQL concurrency control mechanism, the `SELECT ... FOR UPDATE` and `SELECT ... FOR KEY SHARE` clauses produce results through optimistic conflict detection at commit time rather than locking. In Aurora DSQL, when one transaction writes a row and another reads it with one of the preceding clauses, a conflict might surface at commit time depending on which columns the transactions use. The following clauses determine how Aurora DSQL detects these conflicts.

Key column definition

Key columns are columns that are members of a unique, non-partial, non-expression index. All other columns are non-key columns.

`SELECT ... FOR UPDATE`

Declares that Aurora DSQL adjudicates the selected rows as if the transaction writes to them. If another transaction runs `UPDATE`, `DELETE`, `SELECT ... FOR UPDATE`, or `SELECT ... FOR KEY SHARE` on the same row and commits first, the transaction that ran `SELECT ... FOR`

UPDATE fails with an OC000 response. This clause conflicts with any concurrent write to the row, and with concurrent FOR UPDATE or FOR KEY SHARE reads.

SELECT ... FOR KEY SHARE

Declares that the transaction depends on the key columns of the selected rows. If another transaction deletes the row, changes its key columns, or runs SELECT ... FOR UPDATE and commits first, the transaction that ran SELECT ... FOR KEY SHARE fails with an OC000 response. A concurrent UPDATE to non-key columns doesn't conflict.

Aurora DSQL doesn't support the NO KEY UPDATE or FOR SHARE clauses. However, DML implicitly uses the NO KEY UPDATE mechanism. The following matrix summarizes when two concurrent transactions that access the same row conflict. An X indicates that the two operations conflict: whichever transaction commits last fails with an OC000 response. A blank cell indicates that both transactions can commit.

Operation	INSERT, DELETE, UPDATE (key columns), or SELECT ... FOR UPDATE	UPDATE (non-key columns only)	SELECT ... FOR KEY SHARE
INSERT, DELETE, UPDATE (key columns), or SELECT ... FOR UPDATE	X	X	X
UPDATE (non-key columns only)	X	X	
SELECT ... FOR KEY SHARE	X		

Guidelines for optimizing transaction performance

To optimize performance, minimize high contention on single keys or small key ranges. To achieve this goal, design your schema to spread updates over your cluster key range by using the following guidelines:

- Choose a random primary key for your tables.

- Avoid patterns that increase contention on single keys. This approach ensures optimal performance even as transaction volume grows.

DDL and distributed transactions in Aurora DSQL

Data definition language (DDL) behaves differently in Aurora DSQL from PostgreSQL. Aurora DSQL features a Multi-AZ distributed and shared-nothing database layer built on top of multi-tenant compute and storage fleets. Because no single primary database node or leader exists, the database catalog is distributed. Thus, Aurora DSQL manages DDL schema changes as distributed transactions.

Specifically, DDL behaves differently in Aurora DSQL as follows:

Concurrency control responses

Because the database catalog is distributed, Aurora DSQL manages DDL schema changes as distributed transactions that update the catalog version. Sessions that have a cached copy of the catalog at an earlier version can receive a concurrency control response with SQLSTATE code `40001` and OCC code `OC001` when they next interact with storage.

For example, consider the following sequence of actions:

1. In session 1, a user adds a column to the table `mytable`. This updates the catalog version.
2. In session 2, a user attempts to insert a row into `mytable`. This session still has the previous catalog version cached.

   Aurora DSQL returns `SQL Error [40001]: ERROR: schema has been updated by another transaction (OC001).`

> **Note**
>
> An OC001 response can also occur when the schema change has already completed before the affected transaction starts. Aurora DSQL query processors discover catalog changes reactively during query execution, so a session that has been idle might still be operating with a stale catalog version. On retry, the session refreshes its catalog cache and the transaction typically succeeds.

DDL and DML in the same transaction

Transactions in Aurora DSQL can contain only one DDL statement and can't have both DDL and DML statements. This restriction means that you can't create a table and insert data into the same table within the same transaction. For example, Aurora DSQL supports the following sequential transactions.

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

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

Aurora DSQL doesn't support the following transaction, which includes both `CREATE` and `INSERT` statements.

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

Asynchronous DDL

In standard PostgreSQL, DDL operations such as `CREATE INDEX` lock the affected table, making it unavailable for reads and writes from other sessions. In Aurora DSQL, these DDL statements run asynchronously using a background manager. Access to the affected table isn't blocked. Thus, DDL on large tables can run without downtime or performance impact. For more information about the asynchronous job manager in Aurora DSQL, see Asynchronous indexes in Aurora DSQL.

Primary keys in Aurora DSQL

In Aurora DSQL, a *primary key* is a feature that physically organizes table data. It's similar to the `CLUSTER` operation in PostgreSQL or a clustered index in other databases. When you define a primary key, Aurora DSQL creates an index that includes all columns in the table. The primary key structure in Aurora DSQL ensures efficient data access and management.

Data structure and storage

When you define a primary key, Aurora DSQL stores table data in primary key order. This index-organized structure allows a primary key lookup to retrieve all column values directly, instead of following a pointer to the data as in a traditional B-tree index. Unlike the CLUSTER operation in PostgreSQL, which reorganizes data only once, Aurora DSQL maintains this order automatically and continuously. This approach improves the performance of queries that rely on primary key access.

Aurora DSQL also uses the primary key to generate a cluster-wide unique key for each row in tables and indexes. This unique key also underpins distributed data management. It enables automatic partitioning of data across multiple nodes, supporting scalable storage and high concurrency. As a result, the primary key structure helps Aurora DSQL scale automatically and manage concurrent workloads efficiently.

Guidelines for choosing a primary key

When choosing and using a primary key in Aurora DSQL, consider the following guidelines:

- Define a primary key when you create a table. You can't change this key or add a new primary key later. The primary key becomes part of the cluster-wide key used for data partitioning and automatic scaling of write throughput. If you don't specify a primary key, Aurora DSQL assigns a synthetic hidden ID.
- For tables with high write volumes, avoid using monotonically increasing integers as primary keys. This can lead to performance issues by directing all new inserts to a single partition. Instead, use primary keys with random distribution to ensure even distribution of writes across storage partitions.
- For tables that change infrequently or are read-only, you can use an ascending key. Examples of ascending keys are timestamps or sequence numbers. A dense key has many closely spaced or duplicate values. You can use an ascending key even if it is dense because write performance is less critical.
- If a full table scan doesn't meet your performance requirements, choose a more efficient access method. In most cases, this means using a primary key that matches your most common join and lookup key in queries.
- The maximum combined size of columns in a primary key is 1 kibibyte. For more information, see Database limits in Aurora DSQL and Supported data types in Aurora DSQL.

- You can include up to 8 columns in a primary key or a secondary index. For more information, see Database limits in Aurora DSQL and Supported data types in Aurora DSQL.

Sequences and identity columns

Sequences and identity columns generate integer values and are useful when compact or human-readable identifiers are needed. These values involve allocation and caching behavior described in the `CREATE SEQUENCE` documentation.

Topics

- Sequence manipulation functions
- Identity columns
- Working with sequences and identity columns

Sequence manipulation functions

This section describes functions for operating on sequence objects, also called sequence generators or just sequences. Sequence objects are special single-row tables created with `CREATE SEQUENCE`. Sequence objects are commonly used to generate unique identifiers for rows of a table. The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

> **Important**
>
> When using sequences, the cache value should be carefully considered. For more information, see the Important callout on the `CREATE SEQUENCE` page.
> For guidance on how best to use sequences based on workload patterns, see Working with sequences and identity columns.

Function	Description
`nextval ( regclass ) # bigint`	Advances the sequence object to its next value and returns that value. This is done atomically: even if multiple sessions run `nextval` concurrently, each will safely receive a distinct sequence value. If the sequence

Function	Description
	object has been created with default parameters, successive `nextval` calls will return increasing values beginning with 1. Other behaviors can be obtained by using appropriate parameters in the `CREATE SEQUENCE` command. This function requires USAGE or UPDATE privilege on the sequence.

`setval ( regclass, bigint [, boolean ] ) # bigint`	Sets the sequence object's current value, and optionally its `is_called` flag. The two-parameter form sets the sequence's `last_value` field to the specified value and sets its `is_called` field to `true`, meaning that the next `nextval` will advance the sequence before returning a value. The value that will be reported by `currval` is also set to the specified value. In the three-parameter form, `is_called` can be set to either `true` or `false`. `true` has the same effect as the two-parameter form. If it's set to `false`, the next `nextval` will return exactly the specified value, and sequence advancement commences with the following `nextval`. Furthermore, the value reported by `currval` isn't changed here. For example:
	`SELECT setval('myseq', 42);  -- Next nextval will return 43`
	`SELECT setval('myseq', 42, true);  -- Same as above`
	`SELECT setval('myseq', 42, false);  -- Next nextval will return 42`
	The result returned by `setval` is just the value of its second argument. This function requires UPDATE privilege on the sequence.

Function	Description
`currval ( regclass ) # bigint`	When called after `nextval` in the same transaction, returns the value most recently returned by `nextval` for this sequence. This function requires `USAGE` or `SELECT` privilege on the sequence.
`lastval () # bigint`	When called after `nextval` in the same transaction, returns the value most recently returned by `nextval`. This function is identical to `currval`, except that instead of taking the sequence name as an argument it refers to whichever sequence `nextval` was most recently applied to. This function requires `USAGE` or `SELECT` privilege on the last used sequence.

Warning

Call `currval` or `lastval` in the same transaction as `nextval`. Across transaction boundaries, their results are nondeterministic. Either function might return an error or a value from another transaction.

If your application requires the exact value assigned by a sequence, use the value returned by `nextval` directly. For values generated during an `INSERT`, use `INSERT ... RETURNING`.

Warning

The value obtained by `nextval` isn't reclaimed for re-use if the calling transaction later aborts. This can result in gaps in the sequence of assigned values. Gaps can also occur without a transaction abort. For example, an `INSERT` with an `ON CONFLICT` clause computes the row to insert, including any required `nextval` calls, before it detects a conflict. When the command follows the `ON CONFLICT` action, Aurora DSQL doesn't reclaim the sequence value.

Likewise, sequence state changes made by `setval` are immediately visible to other transactions, and aren't undone if the calling transaction rolls back.

The sequence to be operated on by a sequence function is specified by a `regclass` argument, which is simply the OID of the sequence in the `pg_class` system catalog. You don't have to look up the OID by hand, however, because the `regclass` data type's input converter will do the work for you. See the PostgreSQL documentation on Object Identifier Types for details.

Identity columns

> **Important**
>
> When using identity columns, the cache value should be carefully considered. For more information, see the Important callout on the `CREATE SEQUENCE` page.
> For guidance on how best to use identity columns based on workload patterns, see Working with sequences and identity columns.

An identity column is a special column that's generated automatically from an implicit sequence. It can be used to generate key values. To create an identity column, use the `GENERATED ... AS IDENTITY` clause in `CREATE TABLE`, for example:

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

or alternatively:

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

See `CREATE TABLE` for more details.

If an `INSERT` command is executed on the table with the identity column and no value is explicitly specified for the identity column, then a value generated by the implicit sequence is inserted. For example, with the preceding definitions and assuming additional appropriate columns, writing:

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

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

would generate values for the `id` column starting at 1 and result in the following table data:

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

Alternatively, the keyword `DEFAULT` can be specified in place of a value to explicitly request the sequence-generated value:

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

Similarly, the keyword `DEFAULT` can be used in `UPDATE` commands.

Thus, in many ways, an identity column behaves like a column with a default value.

The clauses `ALWAYS` and `BY DEFAULT` in the column definition determine how explicitly user-specified values are handled in `INSERT` and `UPDATE` commands. In an `INSERT` command, if `ALWAYS` is selected, a user-specified value is only accepted if the `INSERT` statement specifies `OVERRIDING SYSTEM VALUE`. If `BY DEFAULT` is selected, then the user-specified value takes precedence. Thus, using `BY DEFAULT` results in a behavior more similar to default values, where the default value can be overridden by an explicit value, whereas `ALWAYS` provides some more protection against accidentally inserting an explicit value.

The data type of an identity column must be one of the data types supported by sequences. (See `CREATE SEQUENCE`.) The properties of the associated sequence might be specified when creating an identity column (see `CREATE TABLE`) or changed afterwards (see `ALTER TABLE`).

An identity column is automatically marked as `NOT NULL`. An identity column, however, doesn't guarantee uniqueness. (A sequence normally returns unique values, but a sequence could be reset, or values could be inserted manually into the identity column, as discussed previously.) Uniqueness would need to be enforced using a `PRIMARY KEY` or `UNIQUE` constraint.

Working with sequences and identity columns

This section helps you understand how best to use sequences and identity columns based on workload patterns.

> **Important**
>
> See the Important callout on the `CREATE SEQUENCE` page for more details on allocation and caching behavior.

Choosing identifier types

Amazon Aurora DSQL supports both UUID-based identifiers and integer values generated using sequences or identity columns. These options differ in how values are allocated and how they scale under load.

UUID values can be generated without coordination and are well suited to workloads where identifiers are created frequently or across many sessions. Because Amazon Aurora DSQL is designed for distributed operation, avoiding coordination is often beneficial. For this reason, UUIDs are recommended as the default identifier type, especially for primary keys in workloads where scalability is important and strict ordering of identifiers isn't required.

Sequences and identity columns generate compact integer values that are convenient for human-readable identifiers, reporting, and external interfaces. When numeric identifiers are preferred for usability or integration reasons, consider using a sequence or identity column in combination with UUID-based identifiers. When integer sequence or identity values are required, choosing an appropriate cache size becomes an important part of workload design. See the following section for guidance on choosing a cache size.

Choosing a cache size

Selecting an appropriate cache value is an important part of using sequences and identity columns effectively. The cache setting determines how identifier allocation behaves under load, influencing both system throughput and how closely values reflect allocation order.

A larger cache size of `CACHE >= 65536` is well suited when:

- Identifiers are generated at high frequency
- Many sessions insert concurrently
- The workload can tolerate gaps and visible ordering effects

For example, high-volume event ingestion workloads (such as IoT or telemetry), as well as operational identifiers like job run IDs, support case references, or internal order numbers typically

benefit from larger cache sizes, where identifiers are generated frequently and strict ordering isn't required.

A cache size of 1 is better aligned when:

- Allocation rates are relatively low
- Identifiers are expected to follow allocation order more closely over time
- Minimizing gaps is more important than maximum throughput

Workloads such as assigning account or reference numbers, where identifiers are generated less often and closer ordering is desirable, are better aligned with a cache size of 1.

Working with foreign key constraints in Aurora DSQL

With foreign key constraints in Aurora DSQL, you can push an application's referential integrity logic into the database. Aurora DSQL supports the `NO ACTION`, `RESTRICT`, `CASCADE`, `SET NULL`, and `SET DEFAULT` referential actions. It also supports the `MATCH FULL` and `MATCH SIMPLE` match types, and deferrable foreign key constraints. For the full syntax breakdown, see Foreign key constraints.

How Aurora DSQL maintains referential integrity

Aurora DSQL maintains referential integrity in two steps: snapshot verification as the transaction runs, and conflict resolution at commit time. Together, these steps guarantee that a committed transaction never violates a foreign key constraint.

Snapshot verification. Every transaction in Aurora DSQL runs against a consistent snapshot of the database taken at its start time. When you insert or update a referencing row, Aurora DSQL reads the referenced table at your transaction's start-time snapshot to confirm the referenced key exists. When you delete or update a referenced key, Aurora DSQL reads the referencing table in the transaction snapshot. It confirms that no referencing rows exist (for `RESTRICT`) or that the operation leaves no orphaned rows (for `NO ACTION`). Because this verification reads from the transaction start-time snapshot instead of taking a lock, other transactions can continue to modify the referenced and referencing tables in parallel.

Commit-time resolution. Snapshot verification ensures that the constraint held at the transaction's start time, but not between start and commit. A concurrent transaction can delete the

referenced row or insert a conflicting referencing row after your transaction started. To adjudicate these conflicts, Aurora DSQL implicitly applies the KEY SHARE clause to referenced rows to detect whether any concurrent change invalidated your snapshot. If Aurora DSQL detects a conflict, it fails the transaction with a serialization error. For more information about how the KEY SHARE clause affects concurrent transactions, see Concurrency control in Aurora DSQL.

> ⚠ **Referential integrity checks incur extra reads**
>
> All data manipulation language (DML) operations on referenced or referencing tables incur extra reads to guarantee referential integrity. Before you add a foreign key constraint to a table, benchmark the workload and validate that the performance characteristics meet your expectations.

Foreign key example scenarios

In the following scenario, the `orders` table has a foreign key constraint on its `product_id` column that references the `products` table. This makes `products` the referenced table and `orders` the referencing table.

```
CREATE TABLE products (
    product_id integer PRIMARY KEY,
    name text,
    price numeric
);
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_id integer REFERENCES products,
    quantity integer
);
INSERT INTO products VALUES (1, 'Widget', 9.99);
```

Conflict: concurrent delete and insert

In this scenario, one session deletes a referenced row while another session inserts a referencing row.

```
-- Session A
```

```
BEGIN;
DELETE FROM products WHERE product_id = 1;

-- Session B
BEGIN;
INSERT INTO orders VALUES (100, 1, 5);

-- Session A
COMMIT;  -- succeeds

-- Session B
COMMIT;  -- fails with serialization error
ERROR: change conflicts with another transaction (OC000) (SQLSTATE 40001)
```

Both sessions run at the same time. Aurora DSQL resolves the conflict at commit. You can't end up with an order that points to a deleted product.

No conflict: non-key column update

In this scenario, one session updates a non-key column on the referenced row while another session inserts a referencing row.

```
-- Session A
BEGIN;
UPDATE products SET name = 'Super Widget' WHERE product_id = 1;

-- Session B
BEGIN;
INSERT INTO orders VALUES (101, 1, 3);

-- Session A
COMMIT;  -- succeeds

-- Session B
COMMIT;  -- succeeds
```

Updating `name` (a non-key column) doesn't conflict with the foreign key on `product_id`. The referencing row only cares that the referenced row's key columns stay the same.

Best practices with foreign keys in Aurora DSQL

Follow these recommendations when you use foreign key constraints in Aurora DSQL:

Implement retry logic

Conflicts cause errors instead of waits. Design your workload to retry failed transactions. For more information about concurrency in Aurora DSQL, see Concurrency control in Aurora DSQL.

Minimize key-column churn on heavily-referenced rows

If multiple referencing rows reference the same row and its key columns change often, consider restructuring the schema. Move frequently-changing values to non-key columns so that the referenced column stays stable. Changing non-key columns on the referenced table doesn't conflict with referencing inserts.

Asynchronous indexes in Aurora DSQL

The `CREATE INDEX ASYNC` command creates an index on one or more columns of a specified table. This command is an asynchronous DDL operation that doesn't block other transactions. When you run `CREATE INDEX ASYNC`, Aurora DSQL immediately returns a `job_id`.

You can monitor the status of this asynchronous job using the `sys.jobs` system view. While the index creation job is in progress, you can use these procedures and commands:

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

Blocks the current session until the specified job completes or fails. Returns a Boolean value indicating success or failure.

`DROP INDEX`

Cancels an in-progress index build job.

When the asynchronous index creation completes, Aurora DSQL updates the system catalog to mark the index as active.

> **Note**
>
> Note that concurrent transactions accessing objects in the same namespace during this update might encounter concurrency errors.

When Aurora DSQL finishes an asynchronous index task, it updates the system catalog to show that the index is active. If other transactions reference the objects in the same namespace at this time, you might see a concurrency error.

For the supported syntax and a description of each parameter, see CREATE INDEX.

Usage notes

Consider the following guidelines:

- The CREATE INDEX ASYNC command doesn't introduce locks. It also doesn't affect the base table that Aurora DSQL uses to create the index.
- During schema migration operations, the sys.wait_for_job(job_id)*'your_index_creation_job_id'* procedure is useful. It ensures that subsequent DDL and DML operations target the newly created index.
- Every time Aurora DSQL runs a new asynchronous task, it checks the sys.jobs view and deletes tasks that have a status of completed or failed for more than 30 minutes. Thus, sys.jobs primarily shows in-progress tasks and doesn't contain information about old tasks.
- If Aurora DSQL fails to build an asynchronous index, the index stays INVALID. For unique indexes, DML operations are subject to uniqueness constraints until you drop the index. We recommend that you drop invalid indexes and recreate them.

Creating an index: example

The following example demonstrates how to create a schema, a table, and then an index.

1. Create a table named test.departments.

```
CREATE SCHEMA test;

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

2. Insert a row into the table.

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

3. Create an asynchronous index.

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

The `CREATE INDEX` command returns a job ID, as shown below.

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

The `job_id` indicates that Aurora DSQL has submitted a new job to create the index. You can use the procedure `sys.wait_for_job(job_id)` *`'your_index_creation_job_id'`* to block other work on the session until the job finishes or times out.

Querying the status of index creation: example

Query the `sys.jobs` system view to check the creation status of your index, as shown in the following example.

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

Aurora DSQL returns a response similar to the following.

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

The status column can be one of the following values.

Status	Description
`submitted`	The task is submitted, but Aurora DSQL hasn't started to process it yet.
`processing`	Aurora DSQL is processing the task.
`failed`	The task failed. See the details column for more information. If Aurora DSQL failed to build the index, Aurora DSQL doesn't automatically

Status	Description
	remove the index definition. You must manually remove the index with the `DROP INDEX` command.
`completed`	Aurora DSQL has completed the task successfully.

You can also query the state of the index via the catalog tables `pg_index`and `pg_class`. Specifically, the attributes `indisvalid` and `indisimmediate` can tell you what state your index is in. While Aurora DSQL creates your index, it has an initial status of `INVALID`. The `indisvalid` flag for the index returns `FALSE` or `f`, which indicates that the index isn't valid. If the flag returns `TRUE` or `t`, the index is ready.

```
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)
```

Unique index build failures

If your asynchronous unique index build job shows a failed state with the detail `Found duplicate key while validating index for UCVs`, this indicates that a unique index could not be built due to uniqueness constraint violations.

To resolve unique index build failures

1. Remove any rows in your primary table that have duplicate entries for the keys specified in your unique secondary index.
2. Drop the failed index.

3. Issue a new create index command.

Detecting uniqueness violations in primary tables

The following SQL query helps you identify duplicate values in a specified column of your table. This is particularly useful when you need to enforce uniqueness on a column that isn't currently set as a primary key or doesn't have a unique constraint, such as email addresses in a user table.

The examples below demonstrate how to create a sample users table, populate it with test data containing known duplicates, and then run the detection query.

Define table schema

```
-- 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
);
```

Insert sample data that includes sets of duplicate email addresses

```
-- 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');
```

Run duplicate detection query

```
-- 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;
```

View all records with duplicate email addresses

```
 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)
```

If we were to try the index creation statement now, it would fail:

```
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)
```

System tables and commands in Aurora DSQL

See the following sections to learn about the supported system tables and catalogs in Aurora DSQL as well as useful queries for fetching information about the system, such as the version.

System tables

Aurora DSQL is compatible with PostgreSQL, so many system catalog tables and views from PostgreSQL also exist in Aurora DSQL.

Important PostgreSQL catalog tables and views

The following table describes the most common tables and views you might use in Aurora DSQL.

Name	Description
`pg_namespace`	Information on all schemas
`pg_tables`	Information on the all tables
`pg_attribute`	Information on all attributes
`pg_views`	Information on (pre-)defined views
`pg_class`	Describes all tables, column, indices, and similar objects
`pg_stats`	A view on the planner statistics
`pg_user`	Information on users
`pg_roles`	Information on users and groups
`pg_indexes`	Lists all indexes

Name	Description
pg_constraint	Lists constraints on tables

Supported and unsupported catalog tables

The following table indicates which tables are supported and unsupported in Aurora DSQL.

Name	Applicable to Aurora DSQL
pg_aggregate	No
pg_am	Yes
pg_amop	No
pg_amproc	No
pg_attrdef	Yes
pg_attribute	Yes
pg_authid	No (use pg_roles)
pg_auth_members	Yes
pg_cast	Yes
pg_class	Yes
pg_collation	Yes
pg_constraint	Yes
pg_conversion	No
pg_database	No
pg_db_role_setting	Yes

Name	Applicable to Aurora DSQL
pg_default_acl	Yes
pg_depend	Yes
pg_description	Yes
pg_enum	No
pg_event_trigger	No
pg_extension	No
pg_foreign_data_wrapper	No
pg_foreign_server	No
pg_foreign_table	No
pg_index	Yes
pg_inherits	Yes
pg_init_privs	No
pg_language	No
pg_largeobject	No
pg_largeobject_metadata	Yes
pg_namespace	Yes
pg_opclass	No
pg_operator	Yes
pg_opfamily	No
pg_parameter_acl	Yes

Name	Applicable to Aurora DSQL
pg_partitioned_table	No
pg_policy	No
pg_proc	No
pg_publication	No
pg_publication_namespace	No
pg_publication_rel	No
pg_range	Yes
pg_replication_origin	No
pg_rewrite	No
pg_seclabel	No
pg_sequence	No
pg_shdepend	Yes
pg_shdescription	Yes
pg_shseclabel	No
pg_statistic	Yes
pg_statistic_ext	No
pg_statistic_ext_data	No
pg_subscription	No
pg_subscription_rel	No
pg_tablespace	No

Name	Applicable to Aurora DSQL
pg_transform	No
pg_trigger	No
pg_ts_config	Yes
pg_ts_config_map	Yes
pg_ts_dict	Yes
pg_ts_parser	Yes
pg_ts_template	Yes
pg_type	Yes
pg_user_mapping	No

Supported and unsupported system views

The following table indicates which views are supported and unsupported in Aurora DSQL.

Name	Applicable to Aurora DSQL
pg_available_extensions	No
pg_available_extension_versions	No
pg_backend_memory_contexts	Yes
pg_config	No
pg_cursors	No
pg_file_settings	No
pg_group	Yes

Name	Applicable to Aurora DSQL
pg_hba_file_rules	No
pg_ident_file_mappings	No
pg_indexes	Yes
pg_locks	No
pg_matviews	No
pg_policies	No
pg_prepared_statements	No
pg_prepared_xacts	No
pg_publication_tables	No
pg_replication_origin_status	No
pg_replication_slots	No
pg_roles	Yes
pg_rules	No
pg_seclabels	No
pg_sequences	No
pg_settings	Yes
pg_shadow	Yes
pg_shmem_allocations	Yes
pg_stats	Yes
pg_stats_ext	No

Name	Applicable to Aurora DSQL
pg_stats_ext_exprs	No
pg_tables	Yes
pg_timezone_abbrevs	Yes
pg_timezone_names	Yes
pg_user	Yes
pg_user_mappings	No
pg_views	Yes
pg_stat_activity	No
pg_stat_replication	No
pg_stat_replication_slots	No
pg_stat_wal_receiver	No
pg_stat_recovery_prefetch	No
pg_stat_subscription	No
pg_stat_subscription_stats	No
pg_stat_ssl	Yes
pg_stat_gssapi	No
pg_stat_archiver	No
pg_stat_io	No
pg_stat_bgwriter	No
pg_stat_wal	No

Name	Applicable to Aurora DSQL
pg_stat_database	No
pg_stat_database_conflicts	No
pg_stat_all_tables	No
pg_stat_all_indexes	No
pg_statio_all_tables	No
pg_statio_all_indexes	No
pg_statio_all_sequences	No
pg_stat_slru	No
pg_statio_user_tables	No
pg_statio_user_sequences	No
pg_stat_user_functions	No
pg_stat_user_indexes	No
pg_stat_progress_analyze	No
pg_stat_progress_basebackup	No
pg_stat_progress_cluster	No
pg_stat_progress_create_index	No
pg_stat_progress_vacuum	No
pg_stat_sys_indexes	No
pg_stat_sys_tables	No
pg_stat_xact_all_tables	No

Name	Applicable to Aurora DSQL
pg_stat_xact_sys_tables	No
pg_stat_xact_user_functions	No
pg_stat_xact_user_tables	No
pg_statio_sys_indexes	No
pg_statio_sys_sequences	No
pg_statio_sys_tables	No
pg_statio_user_indexes	No

The sys.jobs view

`sys.jobs` provides status information about asynchronous jobs. For example, after you create an asynchronous index, Aurora DSQL returns a `job_uuid`. You can use this `job_uuid` with `sys.jobs` to look up the status of the job.

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

Aurora DSQL returns a response similar to the following.

```
           job_id           |  status   | details |  job_type   | class_id | object_id
 |    object_name    |       start_time       |      update_time
----------------------------+-----------+---------+-------------+----------+-----------
+------------------+-----------------------+-----------------------
 wqhu6ewifze5xitg3umt24h5ua | completed |         | INDEX_BUILD |     1259 |     26433
 | public.nt2_c1_idx | 2025-09-25 22:07:31+00 | 2025-09-25 22:07:46+00
 kkngzf33dndl3daacxehpx5eba | completed |         | ANALYZE     |     1259 |     26419
 | public.nt         | 2025-09-25 21:57:05+00 | 2025-09-25 21:57:27+00
 fyopxjb6ovdn7po6lrkj63cyea | completed |         | DROP        |     1259 |     26422
 |                   | 2025-09-25 22:05:57+00 | 2025-09-25 22:06:03+00
```

The following table describes the columns in the `sys.jobs` view.

sys.jobs view columns

Column	Type	Description
job_id	text	A base-32 UUID representing the job.
status	text	The current status of the job. Possible values are submitted , processing , completed , and failed. For more information, see sys.jobs status values.
details	text	Any relevant details about the job. If the job fails, a detailed reason is provided.
job_type	text	The type of asynchronous job. Possible values are: INDEX_BUILD – an asynchronous index build. ANALYZE – a system-submitted auto-analyze job. DROP – removes physical data after a DROP TABLE or DROP INDEX operation.
class_id	oid	The OID of the catalog table which contains the object.
object_id	oid	The OID of the object.
object_name	text	The fully qualified name of the object. DROP jobs cannot reference already dropped objects. If a referenced object has already been dropped, the object_name may be NULL.
start_time	timestamp with time zone	The timestamp at which the job was submitted.
update_time	timestamp with time zone	The timestamp at which the job row was last updated.

sys.jobs status values

Status	Description
`submitted`	The task is submitted, but Aurora DSQL hasn't started to process it yet.
`processing`	Aurora DSQL is processing the task.
`failed`	The task failed. See the `details` column for more information.
`completed`	Aurora DSQL has completed the task successfully.

The sys.iam_pg_role_mappings view

The view `sys.iam_pg_role_mappings` provides information about the permissions granted to IAM users. For example, if `DQSLDBConnect` is an IAM role that gives Aurora DSQL access to non-admins and a user named `testuser` is granted the `DQSLDBConnect` role and corresponding permissions, you can query the `sys.iam_pg_role_mappings` view to see which users are granted which permissions.

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

Useful system metadata queries

Use these queries to get table statistics and system metadata without performing expensive operations like full table scans.

Get estimated row count for a table

To get the approximate count of rows in a table without performing a full table scan, use the following query:

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

The command returns output similar to the following:

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

This approach is more efficient than `SELECT COUNT(*)` for large tables in Aurora DSQL.

Get current Aurora DSQL major version

To get the current major version of the Aurora DSQL cluster, use the following query:

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

The command returns output similar to the following:

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

This returns the major version the SQL connection is on in Aurora DSQL.

Get current PostgreSQL version

To get the current PostgreSQL version of the Aurora DSQL cluster, use the following query:

```
SHOW server_version;
```

The command returns output similar to the following:

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

This returns the PostgreSQL version the SQL connection is on in Aurora DSQL.

The ANALYZE command

The `ANALYZE` command collects statistics about the contents of tables in the database and stores the results in the `pg_stats` system view. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

In Aurora DSQL, you can't run the `ANALYZE` command within an explicit transaction. `ANALYZE` isn't subject to the database transaction timeout limit.

To reduce the need for manual intervention and keep statistics consistently up to date, Aurora DSQL automatically runs `ANALYZE` as a background process. This background job is triggered

automatically based on the observed rate of change in the table. It is linked to the number of rows (tuples) that have been inserted, updated, or deleted since the last analyze.

ANALYZE runs asynchronously in the background and its activity can be monitored in the system view sys.jobs with the following query:

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

Key considerations

> **Note**
>
> ANALYZE jobs are billed like other asynchronous jobs in Aurora DSQL. When you modify a table, this may indirectly trigger an automatic background statistics collection job, which can result in metering charges due to the associated system-level activity.

Background ANALYZEjobs, triggered automatically, collect the same types of statistics as a manual ANALYZE and apply them by default to user tables. System and catalog tables are excluded from this automated process.

Working with Aurora DSQL EXPLAIN plans

Aurora DSQL uses a similar EXPLAIN plan structure to PostgreSQL, but with key additions that reflect its distributed architecture and execution model.

In this documentation, we'll provide an overview of Aurora DSQL EXPLAIN plans, highlighting the similarities and differences compared to PostgreSQL. We'll cover the various types of scan operations available in Aurora DSQL and help you understand the cost of running your queries.

PostgreSQL VS Aurora DSQL EXPLAIN plans

Aurora DSQL is built on top of the PostgreSQL database and shares most plan structures with PostgreSQL, but has key architectural differences that affect query execution and optimization:

Feature	PostgreSQL	Aurora DSQL
Data Storage	Heap Storage	No heap, all rows are indexed by a unique identifier

Feature	PostgreSQL	Aurora DSQL
Primary Key	Primary key index is separate from table data	Primary key index is the table with all extra columns as INCLUDE columns
Secondary Indexes	Standard secondary indexes	Work the same as PostgreSQL, with ability to include non-key columns
Filtering Capabilities	Index Condition, Heap Filter	Index Condition, Storage Filter, Query Processor Filter
Scan Types	Sequential Scan, Index Scan, Index Only Scan	Full Scan, Index Only Scan, Index Scan
Query Execution	Local to the Database	Distributed (compute and storage are separate)

Aurora DSQL stores table data directly in primary key order rather than in a separate heap. Each row is identified by a unique key, typically the primary key, which allows the database to optimize lookups more efficiently. The architectural difference explains why Aurora DSQL often uses Index Only Scans in cases where PostgreSQL might choose a sequential scan.

Another key distinction is that Aurora DSQL separates compute from storage, enabling filters to be applied earlier in the execution path to reduce data movement and improve performance.

For more using EXPLAIN plans with PostgreSQL, see the PostgreSQL EXPLAIN documentation.

Key elements in Aurora DSQL EXPLAIN plans

Aurora DSQL EXPLAIN plans provide detailed information about how queries are executed, including where filtering occurs and which columns are retrieved from storage. Understanding this output helps you optimize query performance.

Index Cond

> Conditions used to navigate the index. Most efficient filtering that reduces data scanned. In Aurora DSQL, index conditions can be applied at multiple layers of the execution plan.

Projections

Columns retrieved from storage. Fewer projections mean better performance.

Storage Filter

Conditions applied at storage level. More efficient than query processor filters.

Query Processor Filter

Conditions applied at the query processor level. Requires transferring all data before filtering, which results in higher data movement and processing overhead.

Filters in Aurora DSQL

Aurora DSQL separates compute from storage, which means that the point where filters are applied during query execution has a significant impact on performance. Filters applied before large volumes of data are transferred reduce latency and improve efficiency. The earlier a filter is applied, the less data needs to be processed, moved, and scanned, resulting in faster queries.

Aurora DSQL can apply filters at multiple stages in the query path. Understanding these stages is key to interpreting query plans and optimizing performance.

Level	Filter Type	Description
1	Index Condition	Applied while scanning the index. Limits how much data is read from storage and reduces the data sent to the compute layer.
2	Storage Filter	Applied after data is read from storage but before it's sent to compute. An example here is a filter on an include column of an index. Reduces data transfer but not the amount read.
3	Query Processor Filter	Applied after data reaches the compute layer. All data must be transferred first, which increases latency and cost. Currently, Aurora DSQL cannot perform all filtering and projection operations on storage, so some queries might be forced to fall back to this type of filtering.

Reading Aurora DSQL EXPLAIN plans

Understanding how to read EXPLAIN plans is key to optimizing query performance. In this section, we'll walk through real examples of Aurora DSQL query plans, show how different scan types behave, explain where filters are applied, and highlight opportunities for optimization.

Sample tables used in these examples

The examples below reference two tables: `transaction` and `account`.

The `transaction` table does not have a primary key, which causes Aurora DSQL to perform full table scans when querying it.

The `account` table has an index on `customer_id`. This index includes `balance` and `status` as covering columns, which allows certain queries to be satisfied directly from the index without reading from the base table. However, the index does not include `created_at`, so queries that reference this column require additional table access.

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

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

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

Full Scan example

Aurora DSQL has both Sequential Scans, which is functionally identical to PostgreSQL, as well as Full Scans. The only difference between these two are that Full Scans can utilize extra filtering on storage. Due to this, it is almost always selected above Sequential Scans. Due to the similarity, we will only cover examples of the more interesting Full Scans.

Full Scans will mostly be used on tables with no primary key. Because Aurora DSQL primary keys are by default full covering indexes, Aurora DSQL will most likely use Index Only Scans on the primary key in many situations where PostgreSQL would use a Sequential Scan. As with most other databases, a table with no indexes on it will scale badly.

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

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
 Full Scan (btree-table) on transaction  (cost=125100.05..177933.38 rows=33333
 width=16)
   Filter: (description ~~ '%external%'::text)
   -> Storage Scan on transaction (cost=12510.05..17793.38 rows=66666 width=16)
        Projections: account_id, description
        Filters: (transaction_date > '2025-01-01 00:00:00'::timestamp without time
 zone)
        -> B-Tree Scan on transaction (cost=12510.05..17793.38 rows=100000 width=30)
```

This plan shows two filters applied at different stages. The `transaction_date > '2025-01-01'` condition is applied at the storage layer, reducing how much data is returned. The `description LIKE '%external%'` condition is applied later in the query processor, after data is transferred, making it less efficient. Pushing more selective filters into the storage or index layers generally improves performance.

Index Only Scan example

Index Only Scans are the most optimal scan types in Aurora DSQL as they result in the fewest round trips to the storage layer and can do the most filtering. But just because you see Index Only Scan does not mean that you have the best plan. Because of all the different levels of filtering that can happen, it is essential to still pay attention to the different places filtering can happen.

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

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using idx1 on account  (cost=725.05..1025.08 rows=8 width=18)
```

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

In this plan, the index condition, `customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'`), is evaluated first during the index scan, which is the most efficient stage because it limits how much data is read from storage. The storage filter, `status = 'pending'`, is applied after data is read but before it's sent to the compute layer, reducing the amount of data transferred. Finally, the query processor filter, `balance > 100`, runs last, after the data has been moved, making it the least efficient. Of these, the index condition provides the greatest performance because it directly controls how much data is scanned.

Index Scan example

Index Scans are similar to Index Only Scans, except they have the extra step of having to call into the base table. Because Aurora DSQL can specify storage filters, it is able to do so on both the index call as well as the lookup call.

To make this clear, Aurora DSQL presents the plan as two nodes. This way, you can clearly see how much adding an include column will help in terms of rows returned from storage.

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

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx1 on account  (cost=826.05..1142.56 rows=3 width=20)
   Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
   -> Storage Scan on idx1  (cost=826.05..1142.56 rows=8 width=20 loops=1)
       Filters: ((balance > '100'::numeric) AND ((status)::text = 'pending'::text))
       -> B-Tree Scan on idx1  (cost=826.05..1142.56 rows=5000 width=20 loops=1)
           Index Cond: (customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'::uuid)
   -> Storage Lookup on account  (cost=826.05..1142.56 rows=3 width=20 loops=1)
       Projections: balance
```

```
        Filters: (created_at > '2025-01-01 00:00:00'::timestamp without time zone)
        -> B-Tree Lookup on account  (cost=826.05..1142.56 rows=8 width=20 loops=1)
```

This plan shows how filtering happens across multiple stages:

- The index condition on `customer_id` filters data early.
- The storage filters on `status` and `balance` further narrow results before they're sent to compute.
- The lookup filter on `created_at` is evaluated when fetching additional columns from the base table.

Adding frequently used columns as `INCLUDE` fields can often eliminate this lookup and improve performance.

Best Practices

- **Align filters with indexed columns** to push filtering earlier.
- **Use INCLUDE columns** to allow Index-Only Scans and avoid lookups.
- **Validate row estimates** when investigating performance issues. Aurora DSQL manages statistics automatically by running `ANALYZE` in the background based on data change rates. If estimates appear inaccurate, you can run `ANALYZE` manually to refresh statistics immediately.
- **Avoid unindexed queries** on large tables to prevent expensive Full Scans.

Batched nested-loop joins in Aurora DSQL EXPLAIN plans

When a query joins a small outer input to an inner input that Aurora DSQL can scan efficiently from storage, Aurora DSQL can choose a `Nested Loop (Batched Join)` plan. This join type reduces round trips between the compute and storage layers by grouping multiple outer rows together before probing the inner side.

A standard nested loop processes one outer row at a time and runs the inner scan again for every row. A batched nested-loop join collects a batch of outer rows, builds the inner scan work for the entire batch, and then joins the returned inner rows back to the matching outer rows.

How batched nested-loop joins work

1. Aurora DSQL reads a batch of rows from the outer side of the join.

2. If the inner side is an `Index Scan` or `Index Only Scan` with an `Index Cond` parameterized on the outer side, Aurora DSQL binds the condition for each outer row and combines the resulting index scan keys into one batched inner scan. Other join predicates don't drive the inner scan.
3. An `Index Scan` or `Index Only Scan` without an outer-parameterized `Index Cond` behaves like a `Full Scan` or `Sequential Scan`. Aurora DSQL has no parameterized index scan keys to combine and runs one inner scan per outer batch instead of one inner scan per outer row.
4. Aurora DSQL streams rows from the inner side as they arrive for the batch. It doesn't materialize the entire inner result for the batch before joining.
5. When the inner `Index Scan` or `Index Only Scan` has an outer-parameterized `Index Cond`, Aurora DSQL uses the `Recheck Cond` to match each returned inner row back to the applicable outer rows in the batch. Aurora DSQL then applies any remaining join predicates. Without an outer-parameterized `Index Cond`, Aurora DSQL applies the join predicates directly as it matches each streamed inner row against the outer batch.
6. For left and anti joins, Aurora DSQL tracks which outer rows matched and emits unmatched rows after the inner scan for the batch finishes.

This approach is most useful when the inner side is an index scan with a parameterized `Index Cond`, because the combined scan keys provide targeted storage access for the entire outer batch.

When Aurora DSQL uses this join

Aurora DSQL considers batched nested-loop joins when the inner side of the join is a physical scan node, meaning an `Index Scan`, `Index Only Scan`, `Full Scan`, or `Sequential Scan`, and batching is expected to cost less than running the inner scan for each outer row. In practice, the greatest benefit typically comes from a smaller outer input and an inner index scan with an `Index Cond` parameterized on the outer side.

If another plan shape is cheaper, such as a hash join or merge join, Aurora DSQL chooses that plan instead. To compare plan shapes during tuning, you can disable batched nested-loop joins for the current session:

```
SET dsql.enable_batched_nestloop = off;
```

How to read batched nested-loop join output

The following query uses the sample `transaction` and `account` tables from Reading Aurora DSQL EXPLAIN plans:

```
EXPLAIN
SELECT t.account_id, a.balance
FROM transaction t
LEFT JOIN account a
  ON t.account_id = a.customer_id
 AND a.balance > CASE
                   WHEN t.description LIKE 'fee%' THEN 0
                   ELSE 100
                 END
WHERE t.transaction_date >= '2025-01-01'
  AND (a.status = 'active' OR a.customer_id IS NULL)
ORDER BY t.account_id;
```

An EXPLAIN plan can include output similar to the following:

```
Sort
  Sort Key: t.account_id
  -> Nested Loop (Batched Join)
       Filter: (((status)::text = 'active'::text) OR (customer_id IS NULL))
       Join Type: Left
       Recheck Cond: (customer_id = t.account_id)
       Join Filter: (balance > CASE WHEN (t.description ~~ 'fee%'::text) THEN
 '0'::numeric ELSE '100'::numeric END)
       -> Full Scan (btree-table) on transaction t
            -> Storage Scan on transaction t
                 Filters: (transaction_date >= '2025-01-01 00:00:00'::timestamp without
 time zone)
                 -> B-Tree Scan on transaction t
       -> Index Only Scan using idx1 on account a
            Index Cond: (customer_id = t.account_id)
```

`Nested Loop (Batched Join)`

Shows that Aurora DSQL is batching outer rows before probing the inner side of the join.

`Index Cond`

Shows the predicate used to probe the inner side for the current batch. When the inner side is an `Index Scan` or `Index Only Scan`, this predicate often references columns from the outer side of the join.

`Filter`

Shows a condition applied to the result of the join. In this example, the `WHERE (a.status = 'active' OR a.customer_id IS NULL)` condition appears here. Under standard SQL left join semantics, this condition can't be pushed into the inner scan because doing so would change the query results. Aurora DSQL displays it as a filter on the join node. By contrast, the condition on `t.transaction_date` references only the outer table, so it appears under the outer physical scan.

`Join Type`

Shows the join semantics for the batched join, such as `Left`, `Semi`, or `Anti`.

`Recheck Cond`

Appears only when the inner side is an `Index Scan` or `Index Only Scan` with at least one `Index Cond` parameterized on the outer side of the join. For each returned inner row, Aurora DSQL runs the recheck against every outer row in the batch to determine which outer rows produced that probe and should join with that inner row.

`Join Filter`

Shows join predicates that Aurora DSQL evaluates after it has matched an inner row back to candidate outer rows in the batch. These predicates affect which row pairs join, but they don't drive the storage probe the way `Index Cond` does.

A preceding sort node

Can appear when the query needs ordered output. Batched nested-loop joins don't preserve the same output order guarantees as a standard nested loop, so Aurora DSQL can add an explicit sort before returning the final result set.

Understanding DPUs in EXPLAIN ANALYZE

Aurora DSQL provides **statement-level** Distributed Processing Unit (DPU) information in `EXPLAIN ANALYZE VERBOSE` plan output, giving you deeper visibility into query cost during development.

This section explains what DPUs are and how to interpret them in the `EXPLAIN ANALYZE VERBOSE` output.

What is a DPU?

A Distributed Processing Unit (DPU) is the normalized measure of work done by Aurora DSQL. It is composed of:

- **ComputeDPU** – Time spent executing SQL queries
- **ReadDPU** – Resources used to read data from storage
- **WriteDPU** - Resources used to write data to storage
- **MultiRegionWriteDPU** – Resources used to replicate writes to peered clusters in multi-Region configurations.

DPU usage in EXPLAIN ANALYZE VERBOSE

Aurora DSQL extends `EXPLAIN ANALYZE VERBOSE` to include a statement-level DPU usage estimate to the end of the output. This provides immediate visibility into query cost, helping you identify workload cost drivers, tune query performance, and better forecast resource usage.

The following examples show how to interpret the statement-level DPU estimates included in EXPLAIN ANALYZE VERBOSE output.

Example 1: SELECT Query

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

```
QUERY PLAN
------------------------------------------------------
Index Only Scan using test_table_pkey on public.test_table  (cost=125100.05..171100.05
 rows=1000000 width=36) (actual time=2.973..4.482 rows=120 loops=1)
  Output: id, context
  -> Storage Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000 width=36)
 (actual rows=120 loops=1)
      Projections: id, context
      -> B-Tree Scan on test_table_pkey (cost=125100.05..171100.05 rows=1000000
 width=36) (actual rows=120 loops=1)
Query Identifier: qymgw1m77maoe
Planning Time: 11.415 ms
```

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

In this example, the SELECT statement performs an index-only scan, so most of the cost comes from Read DPU (0.04312), representing the data retrieved from storage and Compute DPU (0.01607), which reflects the compute resources used to process and return the results. There is no Write DPU since the query doesn't modify data. The total DPU (0.05919) is the sum of Compute + Read + Write.

Example 2: INSERT Query

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

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

This statement primarily performs writes, so most of the cost is associated with Write DPU. The Compute DPU (0.01550) represents the work done to process and insert the values. The Read DPU (0.00307) reflects minor system reads (for catalog lookups or index checks).

Notice the Transaction minimums shown next to Read and Write DPUs. These indicate the baseline per-transaction costs that apply *only when the operation includes reads or writes*. They do not mean that every transaction automatically incurs a 0.00375 Read DPU or 0.05 Write DPU charge. Instead,

these minimums are applied at the transaction level during cost aggregation and only if reads or writes occur within that transaction. Because of this difference in scope, statement-level estimates in `EXPLAIN ANALYZE VERBOSE` may not exactly match the transaction-level metrics reported in CloudWatch or billing data.

Using DPU Information for Optimization

Per-statement DPU estimates give you a powerful way to optimize queries beyond just execution time. Common use cases include:

- **Cost Awareness:** Understand how expensive a query is relative to others.
- **Schema Optimization:** Compare the impact of indexes or schema changes on both performance and resource efficiency.
- **Budget Planning:** Estimate workload cost based on observed DPU usage.
- **Query Comparison:** Evaluate alternative query approaches by their relative DPU consumption.

Interpreting DPU Information

Keep the following best practices in mind when using DPU data from `EXPLAIN ANALYZE VERBOSE`:

- **Use it directionally:** Treat the reported DPU as a way to understand the *relative* cost of a query rather than an exact match with CloudWatch metrics or billing data. Differences are expected because `EXPLAIN ANALYZE VERBOSE` reports statement-level cost, while CloudWatch aggregates transaction-level activity. CloudWatch also includes background operations (such as ANALYZE or compactions) and transaction overhead (`BEGIN`/`COMMIT`) that `EXPLAIN ANALYZE VERBOSE` intentionally excludes.
- **DPU variability across runs is normal** in distributed systems and does not indicate errors. Factors such as caching, execution plan changes, concurrency, or shifts in data distribution can all cause the same query to consume different resources from one run to the next.
- **Batch small operations:** If your workload issues many small statements, consider batching them into larger operations (not to exceed 10MB). This reduces rounding overhead and produces more meaningful cost estimates.
- **Use for tuning, not billing:** DPU data in `EXPLAIN ANALYZE VERBOSE` is designed for cost awareness, query tuning, and optimization. It is not a billing-grade metric. Always rely on CloudWatch metrics or monthly billing reports for authoritative cost and usage data.

Managing Aurora DSQL clusters

Aurora DSQL provides several configuration options to help you establish the right database infrastructure for your needs. To set up your Aurora DSQL cluster infrastructure, review the following sections.

Topics

- Configuring single-Region clusters
- Configuring multi-Region clusters
- Configuring Aurora DSQL clusters using AWS CloudFormation
- Aurora DSQL cluster lifecycle

The features and functionality discussed in this guide ensure that your Aurora DSQL environment is more resilient, responsive, and capable of supporting your applications as they grow and evolve.

Configuring single-Region clusters

Configure and manage clusters for an AWS Region using either the AWS CLI or your preferred programming language including Python, C++, JavaScript, Java, Rust, Ruby, .NET, and Golang. The AWS CLI provides quick access through shell commands, while AWS Software Development Kits (SDKs) enable programmatic control through native language support.

Topics

- Using AWS SDKs
- Using AWS CLI

Using AWS SDKs

The AWS SDKs provide programmatic access to Aurora DSQL in your preferred programming language. The following sections show how to perform common cluster operations using different programming languages.

Create cluster

The following examples show how to create a single-Region cluster using different programming languages.

Python

To create a cluster in a single AWS Region, use the following example.

```
import boto3


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

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

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


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


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

C++

The following example lets you create a cluster in a single AWS Region.

```
#include <aws/core/Aws.h>
```

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

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

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

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

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

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

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

    return cluster;
}
```

```
int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define region for the single-region setup
            Aws::String region = "us-east-1";

            auto cluster = CreateCluster(region);

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

JavaScript

To create a cluster in a single AWS Region, use the following example.

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

async function createCluster(region) {

    const client = new DSQLClient({ region });

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

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

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

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

    await createCluster(region);
}

main();
```

Java

Use the following example to create a cluster in a single AWS Region.

```
package org.example;

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

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

public class CreateCluster {
```

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

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

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

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

Rust

To create a cluster in a single AWS Region, use the following example.

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

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

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

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

    Client::from_conf(config)
}

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

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

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

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

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

    cluster_output
}
```

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let region = "us-east-1";

    let cluster = create_cluster(region).await;

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

    Ok(())
}
```

Ruby

To create a cluster in a single AWS Region, use the following example.

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

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

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

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

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

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

  cluster = create_cluster(region)

  puts "Created single region cluster:"
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

.NET

To create a cluster in a single AWS Region, use the following example.

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

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

```
        /// <summary>
        /// Create a cluster with deletion protection enabled and a name tag.
        /// </summary>
        public static async Task<CreateClusterResponse> Create(RegionEndpoint
 region)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp single region cluster" }
                };

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

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

                return cluster;
            }
        }

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

            var cluster = await Create(region);

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

Golang

To create a cluster in a single AWS Region, use the following example.

```
package main
```

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

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

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

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

 client := dsql.NewFromConfig(cfg)

 deleteProtect := true

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

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

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

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

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

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

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

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

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

}
```

Get cluster

The following examples show how to get information about a single-Region cluster using different programming languages.

Python

To get information about a single-Region cluster, use the following example.

```
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++

Use the following example to get information about a single-Region cluster.

```
#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

To get information about a single-Region cluster, use the following example.

```
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

The following example lets you get information about a single-Region cluster.

```
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

The following example lets you get information about a single-Region cluster.

```
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

The following example lets you get information about a single-Region cluster.

```
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

The following example lets you get information about a single-Region cluster.

```
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

The following example lets you get information about a single-Region cluster.

```
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)
 }
}
```

Update cluster

The following examples show how to update a single-Region cluster using different programming languages.

Python

To update a single-Region cluster, use the following example.

```
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++

Use the following example to update a single-Region cluster.

```
#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

To update a single-Region cluster, use the following example.

```
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

Use the following example to update a single-Region cluster.

```
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

Use the following example to update a single-Region cluster.

```
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

Use the following example to update a single-Region cluster.

```
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

Use the following example to update a single-Region cluster.

```
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

Use the following example to update a single-Region cluster.

```
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)
 }
}
```

Delete cluster

The following examples show how to delete a single-Region cluster using different programming languages.

Python

To delete a cluster in a single AWS Region, use the following example.

```
import boto3


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

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


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


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

C++

To delete a cluster in a single AWS Region, use the following example.

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

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

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

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

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

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

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

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

```
            Aws::String clusterId = "<your cluster id>";

            DeleteCluster(region, clusterId);

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

JavaScript

To delete a cluster in a single AWS Region, use the following example.

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

async function deleteCluster(region, clusterId) {

  const client = new DSQLClient({ region });

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

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

    await waitUntilClusterNotExists(
      {
        client: client,
        maxWaitTime: 300 // Wait for 5 minutes
      },
      {
        identifier: response.identifier
      }
    );
    console.log(`Cluster Id ${response.identifier} is now deleted`);
    return;
```

```
  } catch (error) {
    if (error.name === "ResourceNotFoundException") {
      console.log("Cluster ID not found or already deleted");
    } else {
      console.error("Unable to delete cluster: ", error.message);
    }
    throw error;
  }
}

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

  await deleteCluster(region, clusterId);
}

main();
```

Java

To delete a cluster in a single AWS Region, use the following example.

```
package org.example;

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

import java.time.Duration;

public class DeleteCluster {

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


        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
```

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

            // The DSQL SDK offers a built-in waiter to poll for deletion.
            System.out.println("Waiting for cluster to finish deletion");
            client.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId),
                    config -> config.backoffStrategyV2(

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

Rust

To delete a cluster in a single AWS Region, use the following example.

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

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

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

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

    Client::from_conf(config)
}

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

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

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

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

    Ok(())
}
```

Ruby

To delete a cluster in a single AWS Region, use the following example.

```
require "aws-sdk-dsql"

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

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

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

main if $PROGRAM_NAME == __FILE__
```

.NET

To delete a cluster in a single AWS Region, use the following example.

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

namespace DSQLExamples.examples
{
    public class DeleteSingleRegionCluster
    {
        /// <summary>
```

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

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

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

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

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

Golang

To delete a cluster in a single AWS Region, use the following example.

```
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)
 }

}
```

For more code samples and examples, visit the Aurora DSQL Samples GitHub repository.

Using AWS CLI

The AWS CLI provides a command-line interface for managing your Aurora DSQL clusters. The following examples demonstrate common cluster management operations.

Create cluster

Create a cluster using the **create-cluster** command.

> **Note**
>
> Cluster creation is an asynchronous operation. Call the **GetCluster** API until the status changes to `ACTIVE`. You can connect to your cluster after it becomes active.

Example Command

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

> **Note**
>
> To disable deletion protection during creation, include the `--no-deletion-protection-enabled` flag.

Example Response

```
{
    "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"
    }
}
```

Describing a cluster

Get information about a cluster using the **get-cluster** command.

Example Command

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

Example Response

```
{
    "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"
    }
}
```

Updating a cluster

Update an existing cluster using the **update-cluster** command.

> **Note**
>
> Updates are asynchronous operations. Call the **GetCluster** API until the status changes to ACTIVE to see your changes.

Example Command

```
aws dsql update-cluster \
  --region us-east-1 \
```

```
  --no-deletion-protection-enabled \
  --identifier your_cluster_id
```

Example Response

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

Deleting a cluster

Delete an existing cluster using the **delete-cluster** command.

> **Note**
>
> You can only delete clusters that have deletion protection disabled. By default, deletion protection is enabled when you create new clusters.

Example Command

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

Example Response

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

Listing clusters

List your clusters using the **list-clusters** command.

Example Command

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

Example Response

```
{
    "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"
        }
    ]
}
```

Configuring multi-Region clusters

Configure and manage clusters across multiple AWS Regions using either the AWS CLI or your preferred programming language including Python, C++, JavaScript, Java, Rust, Ruby, .NET, and Golang. The AWS CLI provides quick access through shell commands, while AWS SDKs enable programmatic control through native language support.

Topics

- Using AWS SDKs
- Using AWS CLI

Using AWS SDKs

The AWS SDKs provide programmatic access to Aurora DSQL in your preferred programming language. The following sections show how to perform common cluster operations using different programming languages.

Create cluster

The following examples show how to create a multi-Region cluster using different programming languages.

Python

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
import boto3


def create_multi_region_clusters(region_1, region_2, witness_region):
    try:
        client_1 = boto3.client("dsql", region_name=region_1)
        client_2 = boto3.client("dsql", region_name=region_2)

        # We can only set the witness region for the first cluster
        cluster_1 = client_1.create_cluster(
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region},
            tags={"Name": "Python multi region cluster"}
        )
        print(f"Created {cluster_1["arn"]}")

        # For the second cluster we can set witness region and designate cluster_1
 as a peer
        cluster_2 = client_2.create_cluster(
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_1["arn"]]},
            tags={"Name": "Python multi region cluster"}
        )

        print(f"Created {cluster_2["arn"]}")
        # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
        client_1.update_cluster(
            identifier=cluster_1["identifier"],
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_2["arn"]]}
        )
        print(f"Added {cluster_2["arn"]} as a peer of {cluster_1["arn"]}")
```

```
        # Now that multiRegionProperties is fully defined for both clusters
        # they'll begin the transition to ACTIVE
        print(f"Waiting for {cluster_1["arn"]} to become ACTIVE")
        client_1.get_waiter("cluster_active").wait(
            identifier=cluster_1["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        print(f"Waiting for {cluster_2["arn"]} to become ACTIVE")
        client_2.get_waiter("cluster_active").wait(
            identifier=cluster_2["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        return (cluster_1, cluster_2)

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


def main():
    region_1 = "us-east-1"
    region_2 = "us-east-2"
    witness_region = "us-west-2"
    (cluster_1, cluster_2) = create_multi_region_clusters(region_1, region_2,
 witness_region)
    print("Created multi region clusters:")
    print("Cluster id: " + cluster_1['arn'])
    print("Cluster id: " + cluster_2['arn'])


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

C++

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

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

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

/**
 * Creates multi-region clusters in Amazon Aurora DSQL
 */
std::pair<CreateClusterResult, CreateClusterResult> CreateMultiRegionClusters(
    const Aws::String& region1,
    const Aws::String& region2,
    const Aws::String& witnessRegion) {

    // Create clients for each region
    DSQL::DSQLClientConfiguration clientConfig1;
    clientConfig1.region = region1;
    DSQL::DSQLClient client1(clientConfig1);

    DSQL::DSQLClientConfiguration clientConfig2;
    clientConfig2.region = region2;
    DSQL::DSQLClient client2(clientConfig2);

    std::cout << "Creating cluster in " << region1 << std::endl;

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region
    MultiRegionProperties multiRegionProps1;
```

```
    multiRegionProps1.SetWitnessRegion(witnessRegion);
    createClusterRequest1.SetMultiRegionProperties(multiRegionProps1);

    // Add tags
    Aws::Map<Aws::String, Aws::String> tags;
    tags["Name"] = "cpp multi region cluster 1";
    createClusterRequest1.SetTags(tags);
    createClusterRequest1.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome1 = client1.CreateCluster(createClusterRequest1);
    if (!createOutcome1.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region1 << ": "
                  << createOutcome1.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }

    auto cluster1 = createOutcome1.GetResult();
    std::cout << "Created " << cluster1.GetArn() << std::endl;

    // Create second cluster
    std::cout << "Creating cluster in " << region2 << std::endl;

    CreateClusterRequest createClusterRequest2;
    createClusterRequest2.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region and cluster1 as peer
    MultiRegionProperties multiRegionProps2;
    multiRegionProps2.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> clusters;
    clusters.push_back(cluster1.GetArn());
    multiRegionProps2.SetClusters(clusters);

    tags["Name"] = "cpp multi region cluster 2";
    createClusterRequest2.SetMultiRegionProperties(multiRegionProps2);
    createClusterRequest2.SetTags(tags);
    createClusterRequest2.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome2 = client2.CreateCluster(createClusterRequest2);
    if (!createOutcome2.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region2 << ": "
                  << createOutcome2.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }
```

```
    auto cluster2 = createOutcome2.GetResult();
    std::cout << "Created " << cluster2.GetArn() << std::endl;

    // Now that we know the cluster2 arn we can set it as a peer of cluster1
    UpdateClusterRequest updateClusterRequest;
    updateClusterRequest.SetIdentifier(cluster1.GetIdentifier());

    MultiRegionProperties updatedProps;
    updatedProps.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> updatedClusters;
    updatedClusters.push_back(cluster2.GetArn());
    updatedProps.SetClusters(updatedClusters);

    updateClusterRequest.SetMultiRegionProperties(updatedProps);
    updateClusterRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto updateOutcome = client1.UpdateCluster(updateClusterRequest);
    if (!updateOutcome.IsSuccess()) {
        std::cerr << "Failed to update cluster in " << region1 << ": "
                  << updateOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to update multi-region clusters");
    }

    std::cout << "Added " << cluster2.GetArn() << " as a peer of " <<
 cluster1.GetArn() << std::endl;

    return std::make_pair(cluster1, cluster2);
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define regions for the multi-region setup
            Aws::String region1 = "us-east-1";
            Aws::String region2 = "us-east-2";
            Aws::String witnessRegion = "us-west-2";

            auto [cluster1, cluster2] = CreateMultiRegionClusters(region1, region2,
 witnessRegion);
```

```
            std::cout << "Created multi region clusters:" << std::endl;
            std::cout << "Cluster 1 ARN: " << cluster1.GetArn() << std::endl;
            std::cout << "Cluster 2 ARN: " << cluster2.GetArn() << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

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

async function createMultiRegionCluster(region1, region2, witnessRegion) {

    const client1 = new DSQLClient({ region: region1 });
    const client2 = new DSQLClient({ region: region2 });

    try {
        // We can only set the witness region for the first cluster
        console.log(`Creating cluster in ${region1}`);
        const createClusterCommand1 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 1"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion
            }
        });
        const response1 = await client1.send(createClusterCommand1);
        console.log(`Created ${response1.arn}`);

        // For the second cluster we can set witness region and designate the first
 cluster as a peer
        console.log(`Creating cluster in ${region2}`);
```

```
        const createClusterCommand2 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 2"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response1.arn]
            }
        });
        const response2 = await client2.send(createClusterCommand2);
        console.log(`Created ${response2.arn}`);

        // Now that we know the second cluster arn we can set it as a peer of the
first cluster
        const updateClusterCommand = new UpdateClusterCommand({
            identifier: response1.identifier,
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response2.arn]
            }
        });
        await client1.send(updateClusterCommand);
        console.log(`Added ${response2.arn} as a peer of ${response1.arn}`);

        // Now that multiRegionProperties is fully defined for both clusters they'll
begin the transition to ACTIVE
        console.log(`Waiting for cluster ${response1.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client1,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response1.identifier
            }
        );
        console.log(`Cluster 1 is now active`);

        console.log(`Waiting for cluster ${response2.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client2,
                maxWaitTime: 300 // Wait for 5 minutes
```

```
            },
            {
                identifier: response2.identifier
            }
        );
        console.log(`Cluster 2 is now active`);
        console.log("The multi region clusters are now active");
        return;
    } catch (error) {
        console.error("Failed to create cluster: ", error.message);
        throw error;
    }
}

async function main() {
    const region1 = "us-east-1";
    const region2 = "us-east-2";
    const witnessRegion = "us-west-2";

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();
```

Java

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
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

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
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

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

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

def create_multi_region_clusters(region_1, region_2, witness_region)
  client_1 = Aws::DSQL::Client.new(region: region_1)
  client_2 = Aws::DSQL::Client.new(region: region_2)

  # We can only set the witness region for the first cluster
  puts "Creating cluster in #{region_1}"
  cluster_1 = client_1.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region
    },
    tags: {
      Name: "ruby multi region cluster"
    }
  )
  puts "Created #{cluster_1.arn}"

  # For the second cluster we can set witness region and designate cluster_1 as a
 peer
  puts "Creating cluster in #{region_2}"
  cluster_2 = client_2.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_1.arn ]
    },
    tags: {
      Name: "ruby multi region cluster"
    }
```

```
  )
  puts "Created #{cluster_2.arn}"

  # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
  client_1.update_cluster(
    identifier: cluster_1.identifier,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_2.arn ]
    }
  )
  puts "Added #{cluster_2.arn} as a peer of #{cluster_1.arn}"

  # Now that multi_region_properties is fully defined for both clusters
  # they'll begin the transition to ACTIVE
  puts "Waiting for #{cluster_1.arn} to become ACTIVE"
  cluster_1 = client_1.wait_until(:cluster_active, identifier: cluster_1.identifier)
 do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end

  puts "Waiting for #{cluster_2.arn} to become ACTIVE"
  cluster_2 = client_2.wait_until(:cluster_active, identifier: cluster_2.identifier)
 do |w|
    w.max_attempts = 30
    w.delay = 10
  end

  [ cluster_1, cluster_2 ]
rescue Aws::Errors::ServiceError => e
  abort "Failed to create multi-region clusters: #{e.message}"
end

def main
  region_1 = "us-east-1"
  region_2 = "us-east-2"
  witness_region = "us-west-2"

  cluster_1, cluster_2 = create_multi_region_clusters(region_1, region_2,
 witness_region)

  puts "Created multi region clusters:"
```

```
  pp cluster_1
  pp cluster_2
end

main if $PROGRAM_NAME == __FILE__
```

Golang

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
package main

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

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

func CreateMultiRegionClusters(ctx context.Context, witness, region1, region2
 string) error {

 cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion(region1))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Create a DSQL region 1 client
 client := dsql.NewFromConfig(cfg)

 cfg2, err := config.LoadDefaultConfig(ctx, config.WithRegion(region2))
 if err != nil {
  log.Fatalf("Failed to load AWS configuration: %v", err)
 }

 // Create a DSQL region 2 client
```

```
client2 := dsql.NewFromConfig(cfg2, func(o *dsql.Options) {
 o.Region = region2
})

// Create cluster
deleteProtect := true

// We can only set the witness region for the first cluster
input := &dsql.CreateClusterInput{
 DeletionProtectionEnabled: &deleteProtect,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String(witness),
 },
 Tags: map[string]string{
  "Name": "go multi-region cluster",
 },
}

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

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

// create second cluster
cluster2Arns := []string{*clusterProperties.Arn}

// For the second cluster we can set witness region and designate the first cluster
as a peer
input2 := &dsql.CreateClusterInput{
 DeletionProtectionEnabled: &deleteProtect,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String("us-west-2"),
  Clusters:      cluster2Arns,
 },
 Tags: map[string]string{
  "Name": "go multi-region cluster",
 },
}

clusterProperties2, err := client2.CreateCluster(context.Background(), input2)

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

```
}

// link initial cluster to second cluster
cluster1Arns := []string{*clusterProperties2.Arn}

// Now that we know the second cluster arn we can set it as a peer of the first
cluster
input3 := dsql.UpdateClusterInput{
 Identifier: clusterProperties.Identifier,
 MultiRegionProperties: &dtypes.MultiRegionProperties{
  WitnessRegion: aws.String("us-west-2"),
  Clusters:      cluster1Arns,
 }}

_, err = client.UpdateCluster(context.Background(), &input3)

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

// Create the waiter with our custom options for first cluster
waiter := dsql.NewClusterActiveWaiter(client, func(o
*dsql.ClusterActiveWaiterOptions) {
 o.MaxDelay = 30 * time.Second // Creating a multi-region cluster can take a few
minutes
 o.MinDelay = 10 * time.Second
 o.LogWaitAttempts = true
})

// Now that multiRegionProperties is fully defined for both clusters
// they'll begin the transition to ACTIVE

// Create the input for the clusterProperties to monitor for first cluster
getInput := &dsql.GetClusterInput{
 Identifier: clusterProperties.Identifier,
}

// Wait for the first cluster to become active
fmt.Printf("Waiting for first cluster %s to become active...\n",
*clusterProperties.Identifier)
err = waiter.Wait(ctx, getInput, 5*time.Minute)
if err != nil {
 return fmt.Errorf("error waiting for first cluster to become active: %w", err)
```

```
 }

 // Create the waiter with our custom options
 waiter2 := dsql.NewClusterActiveWaiter(client2, func(o
 *dsql.ClusterActiveWaiterOptions) {
  o.MaxDelay = 30 * time.Second // Creating a multi-region cluster can take a few
 minutes
  o.MinDelay = 10 * time.Second
  o.LogWaitAttempts = true
 })

 // Create the input for the clusterProperties to monitor for second
 getInput2 := &dsql.GetClusterInput{
  Identifier: clusterProperties2.Identifier,
 }

 // Wait for the second cluster to become active
 fmt.Printf("Waiting for second cluster %s to become active...\n",
 *clusterProperties2.Identifier)
 err = waiter2.Wait(ctx, getInput2, 5*time.Minute)
 if err != nil {
  return fmt.Errorf("error waiting for second cluster to become active: %w", err)
 }

 fmt.Printf("Cluster %s is now active\n", *clusterProperties.Identifier)
 fmt.Printf("Cluster %s is now active\n", *clusterProperties2.Identifier)
 return nil
}

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

 err := CreateMultiRegionClusters(ctx, "us-west-2", "us-east-1", "us-east-2")
 if err != nil {
  fmt.Printf("failed to create multi-region clusters: %v", err)
  panic(err)
 }

}
```

.NET

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
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}");
        }
    }
}
```

Get cluster

The following examples show how to get information about a multi-Region cluster using different programming languages.

Python

To get information about a multi-Region cluster, use the following example.

```
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++

Use the following example to get information about a multi-Region cluster.

```
#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

To get information about a multi-Region cluster, use the following example.

```
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

The following example lets you get information about a multi-Region cluster.

```
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

The following example lets you get information about a multi-Region cluster.

```
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

The following example lets you get information about a multi-Region cluster.

```
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

The following example lets you get information about a multi-Region cluster.

```
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

The following example lets you get information about a multi-Region cluster.

```
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)
 }
}
```

Update cluster

The following examples show how to update a multi-Region cluster using different programming languages.

Python

To update a multi-Region cluster, use the following example.

```
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++

Use the following example to update a multi-Region cluster.

```
#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

To update a multi-Region cluster, use the following example.

```
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

Use the following example to update a multi-Region cluster.

```
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

Use the following example to update a multi-Region cluster.

```
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

Use the following example to update a multi-Region cluster.

```
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

Use the following example to update a multi-Region cluster.

```
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

Use the following example to update a multi-Region cluster.

```
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)
 }
}
```

Delete cluster

The following examples show how to delete a multi-Region cluster using different programming languages.

Python

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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++

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
#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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
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)
 }
}
```

For more code samples and examples, visit the Aurora DSQL Samples GitHub repository.

Using AWS CLI

The AWS CLI provides a command-line interface for managing your multi-Region Aurora DSQL clusters. The following examples demonstrate how to create, configure, and delete multi-Region clusters.

Connect to your multi-Region cluster

Multi-Region peered clusters provide two regional endpoints, one in each peered cluster AWS Region. Both endpoints present a single logical database that supports concurrent read and write operations with strong data consistency. In addition to peered clusters, a multi-Region cluster also has a witness Region that stores a limited window of encrypted transaction logs, which is used

to improve multi-Region durability and availability. Multi-Region witness Regions do not have endpoints.

Create multi-Region clusters

To create multi-Region clusters, you first create a cluster with a witness Region. Then you peer this cluster with a second cluster that shares the same witness Region as your first cluster. The following example shows how to create clusters in US East (N. Virginia) and US East (Ohio) with US West (Oregon) as the witness Region.

Step 1: Create cluster one in US East (N. Virginia)

To create a cluster in the US East (N. Virginia) AWS Region with multi-Region properties, use the command below.

```
aws dsql create-cluster \
--region us-east-1 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example Response:

```
{
    "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**
>
> When the API operation succeeds, the cluster enters the `PENDING_SETUP` state. Cluster creation remains in the `PENDING_SETUP` until you update the cluster with the ARN of its peer cluster.

Step 2: Create cluster two in US East (Ohio)

To create a cluster in the US East (Ohio) AWS Region with multi-Region properties, use the command below.

```
aws dsql create-cluster \
--region us-east-2 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example Response:

```
{
    "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"
        ]
    }
}
```

When the API operation succeeds, the cluster transitions to `PENDING_SETUP` state. The cluster creation remains in the `PENDING_SETUP` state until you update it with the ARN of another cluster for peering.

Step 3: Peer cluster in US East (N. Virginia) with US East (Ohio)

To peer your US East (N. Virginia) cluster with your US East (Ohio) cluster, use the `update-cluster` command. Specify your US East (N. Virginia) cluster name and a JSON string with the ARN of the US East (Ohio) cluster.

```
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 Response

```
{
    "identifier": "foo0bar1baz2quux3quuxquux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

Step 4: Peer cluster in US East (Ohio) with US East (N. Virginia)

To peer your US East (Ohio) cluster with your US East (N. Virginia) cluster, use the `update-cluster` command. Specify your US East (Ohio) cluster name and a JSON string with the ARN of the US East (N. Virginia) cluster.

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 Response

```
{
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:51:16.145000-07:00"
}
```

Note

After successful peering, both clusters transition from "PENDING_SETUP" to "CREATING" and finally to "ACTIVE" status when ready for use.

View multi-Region cluster properties

When you describe a cluster, you can view multi-Region properties for clusters in different AWS Regions.

Example

```
aws dsql get-cluster \
--region us-east-1 \
--identifier 'foo0bar1baz2quux3quuxquux4'
```

Example Response

```
{
    "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"
       ]
    }
}
```

Peer clusters during creation

You can reduce the number of steps by including peering information during cluster creation. After creating your first cluster in US East (N. Virginia) (Step 1), you can create your second cluster in US East (Ohio) while simultaneously initiating the peering process by including the ARN of the first cluster.

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"]}'
```

This combines Steps 2 and 4, but you still need to complete Step 3 (updating the first cluster with the ARN of the second cluster) to establish the peering relationship. After all steps are completed, both clusters will transition through the same states as in the standard process: from **PENDING_SETUP** to **CREATING**, and finally to **ACTIVE** when ready for use.

Delete multi-Region clusters

To delete a multi-Region cluster, you need to complete two steps.

1. Turn off deletion protection for each cluster.
2. Delete each peered cluster separately in their respective AWS Region

Update and delete cluster in US East (N. Virginia)

1. Turn off deletion protection using the `update-cluster` command.

   ```
   aws dsql update-cluster \
     --region us-east-1 \
     --identifier 'foo0bar1baz2quux3quuxquux4' \
     --no-deletion-protection-enabled
   ```

2. Delete the cluster using the `delete-cluster` command.

   ```
   aws dsql delete-cluster \
     --region us-east-1 \
     --identifier 'foo0bar1baz2quux3quuxquux4'
   ```

   The command returns the following response.

   ```
   {
       "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**
>
> The cluster transitions to `PENDING_DELETE` status. The deletion isn't complete until you delete the peered cluster in US East (Ohio).

Update and delete cluster in US East (Ohio)

1. Turn off deletion protection using the `update-cluster` command.

```
aws dsql update-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quux4quuux' \
--no-deletion-protection-enabled
```

2. Delete the cluster using the `delete-cluster` command.

```
aws dsql delete-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5'
```

The command returns the following response:

```
{
    "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**
>
> The cluster transitions to `PENDING_DELETE` status. After a few seconds, the system automatically transitions both peered clusters to `DELETING` status after validation.

Configuring Aurora DSQL clusters using AWS CloudFormation

You can use the same CloudFormation resource `AWS::DSQL::Cluster` to deploy and manage single-Region and multi-Region Aurora DSQL clusters.

See the Amazon Aurora DSQL resource type reference for more on how to create, modify, and manage clusters using the `AWS::DSQL::Cluster` resource.

Creating the Initial Cluster Configuration

First, create an AWS CloudFormation template to define your multi-Region cluster:

```
---
Resources:
  MRCluster:
    Type: AWS::DSQL::Cluster
    Properties:
      DeletionProtectionEnabled: true
      MultiRegionProperties:
        WitnessRegion: us-west-2
```

Create stacks in both Regions using the following AWS CLI commands:

```
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
```

Finding Cluster Identifiers

Retrieve the physical resource IDs for your clusters:

```
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"
]
```

Updating the Cluster Configuration

Update your AWS CloudFormation template to include both cluster ARNs:

```
---
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
```

Apply the updated configuration to both Regions:

```
aws cloudformation update-stack --region us-east-2 \
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

```
aws cloudformation update-stack --region us-east-1 \
    --stack-name MRCluster \
    --template-body file://mr-cluster.yaml
```

Aurora DSQL cluster lifecycle

Understanding the Aurora DSQL cluster lifecycle helps you manage your clusters effectively. This section covers cluster status definitions and the scale to zero feature that optimizes costs.

Defining Aurora DSQL cluster status

The Aurora DSQL cluster status provides critical information about cluster health and connectivity. You can view the status of clusters and cluster instances by using the AWS Management Console, AWS CLI, or Aurora DSQL API.

The following table describes each possible status for a Aurora DSQL cluster and what each status means.

Status	Description
Creating	Aurora DSQL is attempting to create or configure resources for the cluster. Any connection attempts will fail while a cluster is in this state.
Active	The cluster is operational and ready to use.
Idle	A cluster becomes idle when it's idle long enough for Aurora DSQL to scale down running resources to reduce capacity and costs. When you connect to an idle cluster, Aurora DSQL transitions the cluster back to the **Active** state.
Inactive	An Idle cluster becomes Inactive when there's been no activity on the cluster for a prolonged period. In this suspended state, running resources are scaled to zero while your data is preserved. When you attempt to connect to an inactive cluster, Aurora DSQL automatically transitions the cluster back to the **Active** state. The time to restore depends on the cluster size.
Updating	A cluster transitions to the **Updating** status when you make changes to the cluster configuration.
Deleting	A cluster transitions to the **Deleting** status when you submit a request to delete it.
Deleted	The cluster has been successfully deleted.
Failed	Aurora DSQL couldn't create the cluster because it encountered an error.
Pending Setup	For multi-Region clusters only. A multi-Region cluster enters the **Pending Setup** status when you create a multi-Region cluster in your first Region with

Status	Description
	a witness Region. Cluster creation pauses until you create another cluster in a secondary Region and peer the two clusters together.
Pending Delete	For multi-Region clusters only. A multi-Region cluster enters the **Pending Delete** status when you delete a cluster from it. The cluster moves to the **Deleting** state after you delete the last peer cluster.

Working with Idle and Inactive clusters

When Aurora DSQL detects no connection activity on a cluster for some period of time, it transitions the cluster to the **Idle** state, reducing running resources to minimize capacity and costs. If connection activity remains absent for a prolonged period, the **Idle** cluster automatically transitions to the **Inactive** state, where running resources are scaled to zero while your data is preserved.

To return a cluster to the **Active** state, connect to it. When you connect, Aurora DSQL moves the cluster back to **Active**.

> **Note**
>
> The first connection attempt to an **Idle** or **Inactive** cluster will be slower than usual.

You can still view a cluster or change its settings in the console, AWS CLI, or API. The cluster stays **Idle** or **Inactive** during these actions. Changes take effect right away. Some tasks, such as backups, need an **Active** cluster. For more information, see the section called “Operations requiring Active cluster state”.

Operations requiring Active cluster state

Some operations require your cluster to be in an Active state. To perform these operations on an **Idle** or **Inactive** cluster, you need to transition your cluster back to Active by connecting to your cluster.

Backup operations

Taking a backup requires an Active cluster state. If your cluster is **Idle** or **Inactive**, backups fail with the following error:

```
"Error": {
    "Code": "FailedPrecondition",
    "Message": "Cluster 'cluster-id' is in state 'IDLE' and can't be backed up.
    In order to take a backup of your cluster, it must be in Active state. Please
    connect to your cluster to transition it to Active to perform the backup."
}
```

To proceed with a backup:

1. Connect to the cluster using your preferred database client or the Aurora DSQL console to wake it up.
2. Wait for automatic transition to **Active** state.
3. Initiate the backup after the cluster is fully operational.

> **Note**
>
> Existing backups taken before the cluster was Idle or Inactive remain valid and unaffected. New backup attempts on the cluster will fail until the cluster is connected to for auto wake up.

Viewing your Aurora DSQL cluster status

To view the status of your cluster, use the AWS Management Console, AWS CLI, or Aurora DSQL API.

Console

Follow these steps to view cluster status in the AWS Management Console:

To view cluster status in the console

1. Open the Aurora DSQL console at https://console.aws.amazon.com/dsql.
2. Choose **Clusters** in the navigation pane.
3. View the status for each cluster in the dashboard.

AWS CLI

Use the following AWS CLI command to check the status of a single cluster.

```
aws dsql get-cluster --identifier cluster-id --query status --output text
```

Run the following command to list the status of all clusters.

```
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
```

This sample output shows two active clusters and one cluster in the process of being deleted.

```
aaabbb2bkx555xa7p42qd5cdef    ACTIVE
abcde123efghi77t35abcdefgh    ACTIVE
12abc6lqasc5bbbbbbbbbbbbbb    DELETING
```

Programming with Aurora DSQL

Aurora DSQL provides you with the following tools to manage your Aurora DSQL resources programmatically.

AWS Command Line Interface (AWS CLI)

You can create and manage your resources by using the AWS CLI in a command-line shell. The AWS CLI provides direct access to the APIs for AWS services, such as Aurora DSQL. For syntax and examples for the commands for Aurora DSQL, see dsql in the *AWS CLI Command Reference*.

AWS software development kits (SDKs)

AWS provides SDKs for many popular technologies and programming languages. They make it easier for you to call AWS services from within your applications in that language or technology. For more information about these SDKs, see Tools for developing and managing applications on AWS.

Aurora DSQL API

This API is another programming interface for Aurora DSQL. When using this API, you must format every HTTPS request correctly and add a valid digital signature to every request. For more information, see *API reference*.

CloudFormation

The AWS::DSQL::Cluster is an CloudFormation resource that enables you to create and manage Aurora DSQL clusters as part of your infrastructure as code. CloudFormation helps you define your entire AWS environment in code, making it easier to provision, update, and replicate your infrastructure in a consistent and reliable way.

When you use the AWS::DSQL::Cluster resource in your CloudFormation templates, you can declaratively provision Aurora DSQL clusters alongside your other cloud resources. This helps ensure that your data infrastructure deploys and manages alongside the rest of your application stack.

Connectors for Aurora DSQL

Aurora DSQL provides specialized connectors that extend existing database drivers to enable seamless IAM authentication and integration with AWS services. These connectors are designed to

work with popular programming languages and frameworks while maintaining compatibility with existing PostgreSQL workflows.

Additional connectors are planned for future releases. For the latest information on connector availability, see the Aurora DSQL samples repository.

Aurora DSQL Connector for Java JDBC

The Aurora DSQL Connector for JDBC is designed as an authentication plugin that extends the functionality of the PostgreSQL JDBC driver to enable applications to authenticate with Aurora DSQL using IAM credentials. The connector does not connect directly to the database, but provides seamless IAM authentication on top of the underlying PostgreSQL JDBC driver.

The Aurora DSQL Connector for JDBC is built to work with the PostgreSQL JDBC Driver and provides seamless integration with Aurora DSQL's IAM authentication requirements.

In conjunction with the PostgreSQL JDBC Driver, the Aurora DSQL Connector for JDBC enables IAM-based authentication for Aurora DSQL. It introduces deep integration with AWS authentication services such as AWS Identity and Access Management (IAM).

About the connector

Aurora DSQL is a distributed SQL database service that provides high availability and scalability for PostgreSQL-compatible applications. Aurora DSQL requires IAM-based authentication with time-limited tokens that existing JDBC drivers do not natively support.

The main idea behind the Aurora DSQL Connector for JDBC is to add an authentication layer on top of the PostgreSQL JDBC driver that handles IAM token generation, allowing users to connect to Aurora DSQL without changing their existing JDBC workflows.

What is Aurora DSQL Authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: Authentication tokens are generated using AWS credentials and have configurable lifetimes

The Aurora DSQL Connector for JDBC is designed to understand these requirements and automatically generate IAM authentication tokens when establishing connections.

Benefits of the Aurora DSQL Connector for JDBC

Although Aurora DSQL provides a PostgreSQL-compatible interface, existing PostgreSQL drivers do not currently support Aurora DSQL's IAM authentication requirements. The Aurora DSQL Connector for JDBC allows customers to continue using their existing PostgreSQL workflows while enabling IAM authentication through:

- **Automatic Token Generation**: IAM tokens are generated automatically using AWS credentials
- **Seamless Integration**: Works with existing JDBC connection patterns
- **AWS Credentials Support**: Supports various AWS credential providers (default, profile-based, etc.)

Using the Aurora DSQL Connector for JDBC with connection pooling

The Aurora DSQL Connector for JDBC works with connection pooling libraries such as HikariCP. The connector handles IAM token generation during connection establishment, allowing connection pools to operate normally.

Key features

Automatic Token Generation

: IAM tokens are generated automatically using AWS credentials.

Seamless Integration

: Works with existing JDBC connection patterns without requiring workflow changes.

AWS Credentials Support

: Supports various AWS credential providers (default, profile-based, etc.).

Connection Pooling Compatibility

: Works seamlessly with connection pooling libraries like HikariCP.

Prerequisites

Before you begin, make sure that you have completed the following prerequisites:

- Created a cluster in Aurora DSQL.
- Installed the Java Development Kit (JDK). Make sure that you have version 17 or higher.

- Set up appropriate IAM permissions to allow your application to connect to Aurora DSQL.
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles).

Using the Aurora DSQL Connector for JDBC

To use the Aurora DSQL Connector for JDBC in your Java application, follow these steps:

1. Add the following dependencies to your Maven project:

```
<dependencies>
    <!-- Aurora DSQL Connector for JDBC -->
    <dependency>
        <groupId>software.amazon.dsql</groupId>
        <artifactId>aurora-dsql-jdbc-connector</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

   For Gradle projects, add this dependency:

```
implementation("software.amazon.dsql:aurora-dsql-jdbc-connector:1.0.0")
```

2. Create a basic connection to your Aurora DSQL cluster using the AWS DSQL PostgreSQL connector format:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DsqlJdbcConnectorExample {
    public static void main(String[] args) {
        // Using AWS DSQL PostgreSQL Connector prefix
        String jdbcUrl = "jdbc:aws-dsql:postgresql://your-cluster.dsql.us-
east-1.on.aws/postgres?user=admin";

        try (Connection connection = DriverManager.getConnection(jdbcUrl)) {
            // Use the connection
            try (Statement statement = connection.createStatement()) {
                // Create a table
```

```
                statement.execute("CREATE TABLE IF NOT EXISTS test_table (id UUID
 PRIMARY KEY DEFAULT gen_random_uuid(), name VARCHAR(100))");

                // Insert data
                statement.execute("INSERT INTO test_table (name) VALUES ('Test
 Name')");

                // Query data
                try (ResultSet resultSet = statement.executeQuery("SELECT * FROM
 test_table")) {
                    while (resultSet.next()) {
                        System.out.println("ID: " + resultSet.getString("id") + ",
 Name: " + resultSet.getString("name"));
                    }
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Configuration properties

The Aurora DSQL Connector for JDBC supports the following connection properties:

user

Determines the user for the connection and the token generation method used. Example: `admin`

token-duration-secs

Duration in seconds for token validity. For more information on token limits, see Generating an authentication token in Amazon Aurora DSQL.

profile

Used for instantiating a ProfileCredentialsProvider for token generation with the provided profile name.

region

AWS region for Aurora DSQL connections. It is optional. When provided, it will override the region extracted from the URL.

database

The database name to connect to. Default is `postgres`.

Logging

Enable logging for troubleshooting any issue you might experience while using the Aurora DSQL JDBC connector.

The connector uses the built-in logging system (java.util.logging) of Java. You can configure logging levels by creating a `logging.properties` file:

```
# Set root logger level to INFO for clean output
.level = INFO

# Show Aurora DSQL Connector for JDBC FINE logs for detailed debugging
software.amazon.dsql.level = FINE

# Console handler configuration
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Detailed formatter pattern with timestamp and logger name
java.util.logging.SimpleFormatter.format = %1$tH:%1$tM:%1$tS.%1$tL [%4$s] %3$s - %5$s%n
```

Examples

For more comprehensive examples and use cases, refer to the Aurora DSQL Connector for JDBC repository

Aurora DSQL Connector for Python

The Aurora DSQL Connector for Python integrates IAM Authentication for connecting Python applications to Amazon Aurora DSQL clusters. Internally, it utilizes psycopg, psycopg2, and asyncpg client libraries.

The Aurora DSQL Connector for Python is designed as an authentication plugin that extends the functionality of the psycopg, psycopg2, and asyncpg client libraries to enable applications to authenticate with Amazon Aurora DSQL using IAM credentials. The connector does not connect directly to the database but provides seamless IAM authentication on top of the underlying client libraries.

About the Connector

Amazon Aurora DSQL is a distributed SQL database service that provides high availability and scalability for PostgreSQL-compatible applications. Aurora DSQL requires IAM-based authentication with time-limited tokens that existing Python libraries do not natively support.

The idea behind the Aurora DSQL Connector for Python is to add an authentication layer on top of the psycopg, psycopg2, and asyncpg client libraries that handles IAM token generation, allowing users to connect to Aurora DSQL without changing their existing workflows.

What is Aurora DSQL Authentication?

In Aurora DSQL, authentication involves:

- **IAM Authentication:** All connections use IAM-based authentication with time-limited tokens
- **Token Generation:** Authentication tokens are generated using AWS credentials and have configurable lifetimes

The Aurora DSQL Connector for Python is designed to understand these requirements and automatically generate IAM authentication tokens when establishing connections.

Features

- **Automatic IAM Authentication** - IAM tokens are generated automatically using AWS credentials
- **Built on psycopg, psycopg2, and asyncpg** - Leverages the psycopg, psycopg2, and asyncpg client libraries
- **Seamless Integration** - Works with existing psycopg, psycopg2, and asyncpg connection patterns without requiring workflow changes
- **Region Auto-Discovery** - Extracts AWS region from DSQL cluster hostname
- **AWS Credentials Support** - Supports various AWS credential providers (default, profile-based, custom)
- **Connection Pooling Compatibility** - Works with psycopg, psycopg2, and asyncpg built-in connection pooling

Quick start guide

Requirements

- Python 3.10 or higher
- Access to an Aurora DSQL cluster
- Set up appropriate IAM permissions to allow your application to connect to Aurora DSQL.
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

```
pip install aurora-dsql-python-connector
```

Install psycopg or psycopg2 or asyncpg separately

The Aurora DSQL Connector for Python installer does not install the underlying libraries. They need to be installed separately, e.g.:

```
# Install psycopg and psycopg pool
pip install "psycopg[binary,pool]"
```

```
# Install psycopg2
pip install psycopg2-binary
```

```
# Install asyncpg
pip install asyncpg
```

Note:

Only the library that is needed must be installed. Therefore, if the client is going to use psycopg, then only psycopg needs to be installed. If the client is going to use psycopg2, then only psycopg2 needs to be installed. If the client is going to use asyncpg, then only asyncpg needs to be installed.

If the client needs more than one, then all the needed libraries need to be installed.

Basic Usage

psycopg

```
    import aurora_dsql_psycopg as dsql
```

```
config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
}

conn = await dsql.connect(**config)
result = await conn.fetchrow("SELECT 1")
await conn.close()
```

```
    print(result)
```

Using just host

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

Using just cluster ID

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("your-cluster")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("your-cluster")
```

asyncpg

```
    import asyncio
```

```
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("your-cluster")
```

Note:

In the "using just cluster ID" scenario, the region that was set previously on the machine is used, e.g.:

```
aws configure set region us-east-1
```

If the region has not been set, or the given cluster ID is in a different region, the connection will fail. To make it work, provide region as a parameter as in the example below:

psycopg

```
    import aurora_dsql_psycopg as dsql

    config = {
            "region": "us-east-1",
    }

    conn = dsql.connect("your-cluster", **config)
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    config = {
            "region": "us-east-1",
    }

    conn = dsql.connect("your-cluster", **config)
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    config = {
            "region": "us-east-1",
```

```
    }

    conn = await dsql.connect("your-cluster", **config)
```

Connection String

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/
postgres?user=admin&token_duration_secs=15")
```

Advanced Configuration

psycopg

```
    import aurora_dsql_psycopg as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
        "profile": "default",
        "token_duration_secs": "15",
    }
```

```
conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

psycopg2

```
import aurora_dsql_psycopg2 as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
    "profile": "default",
    "token_duration_secs": "15",
}

conn = dsql.connect(**config)
with conn.cursor() as cur:
    cur.execute("SELECT 1")
    result = cur.fetchone()
    print(result)
```

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin",
    "profile": "default",
    "token_duration_secs": "15",
}

conn = await dsql.connect(**config)
result = await conn.fetchrow("SELECT 1")
await conn.close()
print(result)
```

Configuration Options

Option	Type	Requi	Description
`host`	`string`	Yes	DSQL cluster hostname or cluster ID
`user`	`string`	No	DSQL username. Default: admin
`dbname`	`string`	No	Database name. Default: postgres
`region`	`string`	No	AWS region (auto-detected from hostname if not provided)
`port`	`int`	No	Default to 5432
`custom_credentials_provider`	`CredentialProvider`	No	Custom AWS credentials provider
`profile`	`string`	No	The IAM profile name. Default: default.
`token_duration_secs`	`int`	No	Token expiration time in seconds

All standard connection options of the underlying psycopg, psycopg2, and asyncpg libraries are also supported, with the exception of asyncpg parameters **krbsrvname** and **gsslib** which are not supported by DSQL.

Using the Aurora DSQL connector for Python with connection pooling

The Aurora DSQL Connector for Python works with psycopg, psycopg2, and asyncpg built-in connection pooling. The connector handles IAM token generation during connection establishment, allowing connection pools to operate normally.

psycopg

For psycopg, the connector implements a connection class named DSQLConnection that can be passed directly to the psycopg_pool.ConnectionPool constructor. For asynchronous operations, there is also an async version of the class named DSQLAsyncConnection.

```
from psycopg_pool import ConnectionPool as PsycopgPool

...
pool = PsycopgPool(
    "",
    connection_class=dsql.DSQLConnection,
    kwargs=conn_params,
    min_size=2,
    max_size=8,
    max_lifetime=3300
)
```

Note: Connection max_lifetime Configuration

The max_lifetime parameter should be set to less than 3600 seconds (one hour), as this is the maximum connection duration allowed by Aurora DSQL database. Setting a lower max_lifetime allows the connection pool to proactively manage connection recycling, which is more efficient than handling connection timeout errors from the database.

psycopg2

For psycopg2, the connector provides a class named AuroraDSQLThreadedConnectionPool that inherits from psycopg2.pool.ThreadedConnectionPool. The AuroraDSQLThreadedConnectionPool class only overrides the internal _connect method. The rest of the implementation is provided by psycopg2.pool.ThreadedConnectionPool unchanged.

```
import aurora_dsql_psycopg2 as dsql

pool = dsql.AuroraDSQLThreadedConnectionPool(
        minconn=2,
        maxconn=8,
        **conn_params,
)
```

asyncpg

For asyncpg, the connector provides a create_pool function that returns an instance of asyncpg.Pool.

```
import asyncio
import os
```

```
import aurora_dsql_asyncpg as dsql

pool_params = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'user': "admin",
    "min_size": 2,
    "max_size": 5,
}

pool = await dsql.create_pool(**pool_params)
```

Authentication

The connector automatically handles DSQL authentication by generating tokens using the DSQL client token generator. If the AWS region is not provided, it will be automatically parsed from the hostname provided.

For more information on authentication in Aurora DSQL, see the user guide.

Admin vs Regular Users

- Users named `"admin"` automatically use admin authentication tokens
- All other users use non-admin authentication tokens
- Tokens are generated dynamically for each connection

Examples

For full example code, refer to the examples as indicated in the sections below. For instructions how to run the examples please refer to the examples READMDE files.

psycopg

Examples README

Description	Examples
Using the Aurora DSQL Connector for Python for basic connections	Basic Connection Example

Description	Examples
Using the Aurora DSQL Connector for Python for basic asynchronous connections	Basic Asynchronous Connection Example
Using the Aurora DSQL Connector for Python with connection pool	Basic Connection Example With Connection Pool
	Concurrent Connections Example With Connection Pool
Using the Aurora DSQL Connector for Python with asynchronous connection pool	Basic Connection Example With Asynchronous Connection Pool

psycopg2

Examples README

Description	Examples
Using the Aurora DSQL Connector for Python for basic connections	Basic Connection Example
Using the Aurora DSQL Connector for Python with connection pool	Basic Connection Example With Connection Pool
	Concurrent Connections Example With Connection Pool

asyncpg

Examples README

Description	Examples
Using the Aurora DSQL Connector for Python for basic connections	Basic Connection Example
Using the Aurora DSQL Connector for Python with connection pool	Basic Connection Example With Connection Pool
	Concurrent Connections Example With Connection Pool

Aurora DSQL Connector for Go pgx

The Aurora DSQL Connector for Go wraps pgx with automatic IAM authentication. The connector handles token generation, SSL configuration, and connection management so you can focus on your application logic.

About the connector

Aurora DSQL requires IAM-based authentication with time-limited tokens that existing Go PostgreSQL drivers do not natively support. The Aurora DSQL Connector for Go adds an authentication layer on top of the pgx driver that handles IAM token generation, allowing you to connect to Aurora DSQL without changing your existing pgx workflows.

What is Aurora DSQL Authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: The connector generates authentication tokens using AWS credentials, and these tokens have configurable lifetimes

The Aurora DSQL Connector for Go is designed to understand these requirements and automatically generate IAM authentication tokens when establishing connections.

Benefits of the Aurora DSQL Connector for Go

The Aurora DSQL Connector for Go allows you to continue using your existing pgx workflows while enabling IAM authentication through:

- **Automatic Token Generation**: The connector generates IAM tokens automatically for each connection
- **Connection Pooling**: Built-in support for `pgxpool` with automatic token generation per connection
- **Flexible Configuration**: Support for full endpoints or cluster IDs with region auto-detection
- **AWS Credentials Support**: Supports AWS profiles and custom credentials providers

Key features

Automatic Token Management

The connector generates IAM tokens automatically for each new connection using pre-resolved credentials.

Connection Pooling

Connection pooling via `pgxpool` with automatic token generation per connection.

Flexible Host Configuration

Supports both full cluster endpoints and cluster IDs with automatic region detection.

SSL Security

SSL is always enabled with verify-full mode and direct TLS negotiation.

Prerequisites

- Go 1.24 or later
- AWS credentials configured
- An Aurora DSQL cluster

The connector uses the AWS SDK for Go v2 default credential chain, which resolves credentials in the following order:

1. Environment variables (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY)
2. Shared credentials file (~/.aws/credentials)
3. Shared config file (~/.aws/config)
4. IAM role for Amazon EC2/ECS/Lambda

Installation

Install the connector using Go modules:

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql
```

Quick start

The following example shows how to create a connection pool and execute a query:

```
package main

import (
    "context"
    "log"

    "github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql"
)

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

    // Create a connection pool
    pool, err := dsql.NewPool(ctx, dsql.Config{
        Host: "your-cluster.dsql.us-east-1.on.aws",
    })
    if err != nil {
        log.Fatal(err)
    }
    defer pool.Close()

    // Execute a query
    var greeting string
    err = pool.QueryRow(ctx, "SELECT 'Hello, DSQL!'").Scan(&greeting)
    if err != nil {
```

```
        log.Fatal(err)
    }
    log.Println(greeting)
}
```

Configuration options

The connector supports the following configuration options:

Field	Type	Default	Description
Host	string	(required)	Cluster endpoint or cluster ID
Region	string	(auto-detected)	AWS region; required if Host is a cluster ID
User	string	"admin"	Database user
Database	string	"postgres"	Database name
Port	int	5432	Database port
Profile	string	""	AWS profile name for credentials
TokenDurationSecs	int	900 (15 min)	Token validity duration in seconds (max allowed: 1 week, default: 15 min)
MaxConns	int32	0	Maximum pool connections (0 = pgxpool default)
MinConns	int32	0	Minimum pool connections (0 = pgxpool default)
MaxConnLifetime	time.Duration	55 minutes	Maximum connection lifetime

Connection string format

The connector supports PostgreSQL and DSQL connection string formats:

```
postgres://[user@]host[:port]/[database][?param=value&...]
dsql://[user@]host[:port]/[database][?param=value&...]
```

Supported query parameters:

- `region` - AWS region
- `profile` - AWS profile name
- `tokenDurationSecs` - Token validity duration in seconds

Examples:

```
// Full endpoint (region auto-detected)
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres")

// Using dsql:// scheme (also supported)
pool, _ := dsql.NewPool(ctx, "dsql://admin@cluster.dsql.us-east-1.on.aws/postgres")

// With explicit region
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/mydb?
region=us-east-1")

// With AWS profile
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?
profile=dev")
```

Advanced usage

Host configuration

The connector supports two host formats:

Full endpoint (region auto-detected):

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
```

Cluster ID (region required):

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
```

```
    Host:   "your-cluster-id",
    Region: "us-east-1",
})
```

Pool configuration tuning

Configure the connection pool for your workload:

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:              "your-cluster.dsql.us-east-1.on.aws",
    MaxConns:          20,
    MinConns:          5,
    MaxConnLifetime:   time.Hour,
    MaxConnIdleTime:   30 * time.Minute,
    HealthCheckPeriod: time.Minute,
})
```

Single connection usage

For simple scripts or when connection pooling is not needed:

```
conn, err := dsql.Connect(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
if err != nil {
    log.Fatal(err)
}
defer conn.Close(ctx)

// Use the connection
rows, err := conn.Query(ctx, "SELECT * FROM users")
```

Using AWS profiles

Specify an AWS profile for credentials:

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:    "your-cluster.dsql.us-east-1.on.aws",
    Profile: "production",
})
```

OCC retry

Aurora DSQL uses optimistic concurrency control (OCC). When two transactions modify the same data, the first to commit wins and the second receives an OCC error.

The `occretry` package provides helpers for automatic retry with exponential backoff and jitter. Install it with:

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/occretry
```

Use `WithRetry` for transactional writes:

```
err := occretry.WithRetry(ctx, pool, occretry.DefaultConfig(), func(tx pgx.Tx) error {
    _, err := tx.Exec(ctx, "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
 100, fromID)
    if err != nil {
        return err
    }
    _, err = tx.Exec(ctx, "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
 100, toID)
    return err
})
```

For DDL or single statements, use `ExecWithRetry`:

```
err := occretry.ExecWithRetry(ctx, pool, occretry.DefaultConfig(),
    "CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY, name TEXT)")
```

> **Important**
>
> `WithRetry` manages `BEGIN/COMMIT/ROLLBACK` internally. Your callback receives a transaction and should contain only database operations and be safe to retry.

Examples

For more comprehensive examples and use cases, refer to the Aurora DSQL Connector for Go examples.

Example	Description
example_preferred	Recommended: Connection pool with concurrent queries
transaction	Transaction handling with BEGIN/COMMIT/ROLLBACK
occ_retry	Handling OCC conflicts with exponential backoff
connection_string	Using connection strings for configuration

Aurora DSQL Connectors for Node.js

The Aurora DSQL Connector for node-postgres and the Aurora DSQL Connector for Postgres.js are authentication plugins that extend the functionality of the node-postgres and Postgres.js clients to enable applications to authenticate with Aurora DSQL using IAM credentials.

Aurora DSQL Connector for node-postgres

The Aurora DSQL Connector for node-postgres is a Node.js connector built on node-postgres that integrates IAM Authentication for connecting JavaScript/TypeScript applications to Amazon Aurora DSQL clusters.

The Aurora DSQL Connector is designed as an authentication plugin that extends the functionality of the node-postgres' Client and Pool to enable applications to authenticate with Amazon Aurora DSQL using IAM credentials.

About the Connector

Amazon Aurora DSQL is a cloud-native distributed database with PostgreSQL compatibility. While it requires IAM authentication and time-bound tokens, traditional Node.js database drivers lack this built-in support.

The Aurora DSQL Connector for node-postgres bridges this gap by implementing an authentication middleware that works seamlessly with node-postgres. This approach allows developers to maintain their existing node-postgres code while gaining secure IAM-based access to Aurora DSQL clusters through automated token management.

What is Aurora DSQL Authentication?

In Aurora DSQL, authentication involves:

- **IAM Authentication:** All connections use IAM-based authentication with time-limited tokens
- **Token Generation:** Authentication tokens are generated using AWS credentials and have configurable lifetimes

The Aurora DSQL Connector for node-postgres is designed to understand these requirements and automatically generate IAM authentication tokens when establishing connections.

Features

- **Automatic IAM Authentication** - Handles DSQL token generation and refresh
- **Built on node-postgres** - Leverages the popular PostgreSQL client for Node.js
- **Seamless Integration** - Works with existing node-postgres connection patterns
- **Region Auto-Discovery** - Extracts AWS region from DSQL cluster hostname
- **Full TypeScript Support** - Provides full type safety
- **AWS Credentials Support** - Supports various AWS credential providers (default, profile-based, custom)
- **Connection Pooling Compatibility** - Works seamlessly with built-in connection pooling

Example Application

There is an included sample application in example that shows how to use Aurora DSQL Connector for node-postgres. To run the included example please refer to the example README.

Quick start guide

Requirements

- Node.js 20+
- Access to an Aurora DSQL cluster
- Set up appropriate IAM permissions to allow your application to connect to Aurora DSQL.
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

```
npm install @aws/aurora-dsql-node-postgres-connector
```

Peer Dependencies

```
npm install @aws-sdk/credential-providers @aws-sdk/dsql-signer pg tsx
```

```
npm install --save-dev @types/pg
```

Usage

Client Connection

```
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
});
await client.connect();
const result = await client.query("SELECT NOW()");
await client.end();
```

Pool Connection

```
import { AuroraDSQLPool } from "@aws/aurora-dsql-node-postgres-connector";

const pool = new AuroraDSQLPool({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
  max: 3,
  idleTimeoutMillis: 60000,
});

const result = await pool.query("SELECT NOW()");
```

Advanced Usage

```
import { fromNodeProviderChain } from "@aws-sdk/credential-providers";
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "example.dsql.us-east-1.on.aws",
  user: "admin",
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
});

await client.connect();
const result = await client.query("SELECT NOW()");
```

```
await client.end();
```

Configuration Options

Option	Type	Req	Description
`host`	`string`	Yes	DSQL cluster hostname
`username`	`string`	Yes	DSQL username
`database`	`string`	No	Database name
`region`	`string`	No	AWS region (auto-detected from hostname if not provided)
`port`	`number`	No	Default to 5432
`customCredentialsProvider`	`AwsCredentialIdentity / AwsCredentialIdentityProvider`	No	Custom AWS credentials provider
`profile`	`string`	No	The IAM profile name. Default to "default"
`tokenDurationSecs`	`number`	No	Token expiration time in seconds

All other parameters from Client / Pool are supported.

Authentication

The connector automatically handles DSQL authentication by generating tokens using the DSQL client token generator. If the AWS region is not provided, it will be automatically parsed from the hostname provided.

For more information on authentication in Aurora DSQL, see the user guide.

Admin vs Regular Users

- Users named "admin" automatically use admin authentication tokens

- All other users use regular authentication tokens
- Tokens are generated dynamically for each connection

Aurora DSQL Connector for Postgres.js

The Aurora DSQL Connector for Postgres.js is a Node.js connector built on Postgres.js that integrates IAM Authentication for connecting JavaScript applications to Amazon Aurora DSQL clusters.

The Aurora DSQL Connector for Postgres.js is designed as an authentication plugin that extends the functionality of the Postgres.js client to enable applications to authenticate with Amazon Aurora DSQL using IAM credentials. The connector does not connect directly to the database, but provides seamless IAM authentication on top of the underlying Postgres.js driver.

About the Connector

Amazon Aurora DSQL is a distributed SQL database service that provides high availability and scalability for PostgreSQL-compatible applications. Aurora DSQL requires IAM-based authentication with time-limited tokens that existing Node.js drivers do not natively support.

The idea behind the Aurora DSQL Connector for Postgres.js is to add an authentication layer on top of the Postgres.js client that handles IAM token generation, allowing users to connect to Aurora DSQL without changing their existing Postgres.js workflows.

The Aurora DSQL Connector for Postgres.js works with most versions of Postgres.js. Users provide their own version by installing Postgres.js directly.

What is Aurora DSQL Authentication?

In Aurora DSQL, authentication involves:

- **IAM Authentication:** All connections use IAM-based authentication with time-limited tokens
- **Token Generation:** Authentication tokens are generated using AWS credentials and have configurable lifetimes

The Aurora DSQL Connector for Postgres.js is designed to understand these requirements and automatically generate IAM authentication tokens when establishing connections.

Features

- **Automatic IAM Authentication** - Handles DSQL token generation and refresh

- **Built on Postgres.js** - Leverages the fast PostgreSQL client for Node.js
- **Seamless Integration** - Works with existing Postgres.js connection patterns
- **Region Auto-Discovery** - Extracts AWS region from DSQL cluster hostname
- **Full TypeScript Support** - Provides full type safety
- **AWS Credentials Support** - Supports various AWS credential providers (default, profile-based, custom)
- **Connection Pooling Compatibility** - Works seamlessly with Postgres.js' built-in connection pooling

Quick start guide

Requirements

- Node.js 20+
- Access to an Aurora DSQL cluster
- Set up appropriate IAM permissions to allow your application to connect to Aurora DSQL.
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

```
npm install @aws/aurora-dsql-postgresjs-connector
# Postgres.js is a peer-dependency, so users must install it themselves
npm install postgres
```

Basic Usage

```
import { auroraDSQLPostgres } from '@aws/aurora-dsql-postgresjs-connector';

const sql = auroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  username: 'admin',

});

// Execute queries
const result = await sql`SELECT current_timestamp`;
console.log(result);

// Clean up
```

```
await sql.end();
```

Using cluster ID instead of host

```
const sql = auroraDSQLPostgres({
  host: 'your-cluster-id',
  region: 'us-east-1',
  username: 'admin',

});
```

Connection String

```
const sql = AuroraDSQLPostgres(
  'postgres://admin@your-cluster.dsql.us-east-1.on.aws'
);

const result = await sql`SELECT current_timestamp`;
```

Advanced Configuration

```
import { fromNodeProviderChain } from '@aws-sdk/credential-providers';

const sql = AuroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  database: 'postgres',
  username: 'admin',
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
  tokenDurationSecs: 3600,                            // Token expiration (seconds)

  // Standard Postgres.js options
  max: 20,                              // Connection pool size
  ssl: { rejectUnauthorized: false }    // SSL configuration
});
```

Configuration Options

Option	Type	Requi	Description
host	string	Yes	DSQL cluster hostname or cluster ID

Option	Type	Requi	Description
`database`	`string?`	No	Database name
`username`	`string?`	No	Database username (uses admin if not provided)
`region`	`string?`	No	AWS region (auto-detected from hostname if not provided)
`customCredentialsProvider`	`AwsCredentialIdentityProvider?`	No	Custom AWS credentials provider
`tokenDurationSecs`	`number?`	No	Token expiration time in seconds

All standard Postgres.js options are also supported.

Authentication

The connector automatically handles DSQL authentication by generating tokens using the DSQL client token generator. If the AWS region is not provided, it will be automatically parsed from the hostname provided.

For more information on authentication in Aurora DSQL, see the user guide.

Admin vs Regular Users

- Users named "admin" automatically use admin authentication tokens
- All other users use regular authentication tokens
- Tokens are generated dynamically for each connection

Sample usage

JavaScript examples using the Aurora DSQL Connector for Postgres.js are available on GitHub. For instructions on how to run the examples, refer to the examples directory.

Description	Example
Connection pooling with concurrent queries, including table creation, inserts, and reads across multiple workers	Connection pool example (preferred)
CRUD operations (create table, insert, select, delete) without connection pooling	Example without connection pool

Aurora DSQL Connector for Ruby pg

The Aurora DSQL Connector for Ruby is a Ruby connector built on pg that integrates IAM authentication for connecting Ruby applications to Amazon Aurora DSQL clusters.

The connector handles token generation, SSL configuration, and connection pooling so you can focus on your application logic.

About the connector

Amazon Aurora DSQL requires IAM authentication with time-limited tokens that existing Ruby PostgreSQL drivers do not natively support. The Aurora DSQL Connector for Ruby adds an authentication layer on top of the pg gem that handles IAM token generation, allowing you to connect to Aurora DSQL without changing your existing pg workflows.

What is Aurora DSQL authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: The connector generates authentication tokens using AWS credentials, and these tokens have configurable lifetimes

The Aurora DSQL Connector for Ruby understands these requirements and automatically generates IAM authentication tokens when establishing connections.

Features

- **Automatic IAM authentication** - Handles Aurora DSQL token generation and refresh
- **Built on pg** - Wraps the popular PostgreSQL gem for Ruby
- **Seamless integration** - Works with existing pg gem workflows
- **Connection pooling** - Built-in support via the `connection_pool` gem with max_lifetime enforcement
- **Region auto-detection** - Extracts AWS region from Aurora DSQL cluster hostname
- **AWS credentials support** - Supports AWS profiles and custom credentials providers
- **OCC retry** - Opt-in optimistic concurrency control retry with exponential backoff

Example application

For a complete example, see the example application on GitHub.

Quick start guide

Requirements

- Ruby 3.1 or later
- Access to an Aurora DSQL cluster
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

Add to your Gemfile:

```
gem "aurora-dsql-ruby-pg"
```

Or install directly:

```
gem install aurora-dsql-ruby-pg
```

Usage

Pool connection

```
require "aurora_dsql_pg"

# Create a connection pool with OCC retry enabled
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

# Read
pool.with do |conn|
  result = conn.exec("SELECT 'Hello, DSQL!'")
  puts result[0]["?column?"]
end

# Write — you must wrap writes in a transaction
pool.with do |conn|
  conn.transaction do
    conn.exec_params("INSERT INTO users (id, name) VALUES (gen_random_uuid(), $1)",
 ["Alice"])
  end
end

pool.shutdown
```

Single connection

For simple scripts or when connection pooling is not needed:

```
conn = AuroraDsql::Pg.connect(host: "your-cluster.dsql.us-east-1.on.aws")
conn.exec("SELECT 1")
conn.close
```

Advanced usage

Host configuration

The connector supports both full cluster endpoints (region auto-detected) and cluster IDs (region required):

```
# Full endpoint (region auto-detected)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws"
)

# Cluster ID (region required)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster-id",
  region: "us-east-1"
)
```

AWS profiles

Specify an AWS profile for credentials:

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  profile: "production"
)
```

Connection string format

The connector supports PostgreSQL connection string formats:

```
postgres://[user@]host[:port]/[database][?param=value&...]
postgresql://[user@]host[:port]/[database][?param=value&...]
```

Supported query parameters: `region`, `profile`, `tokenDurationSecs`.

```
# Full endpoint with profile
pool = AuroraDsql::Pg.create_pool(
  "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?profile=dev"
)
```

OCC retry

Aurora DSQL uses optimistic concurrency control (OCC). When two transactions modify the same data, the first to commit wins and the second receives an OCC error.

OCC retry is opt-in. Set `occ_max_retries` when creating the pool to enable automatic retry with exponential backoff and jitter on `pool.with`:

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

pool.with do |conn|
  conn.transaction do
    conn.exec_params("UPDATE accounts SET balance = balance - $1 WHERE id = $2", [100,
 from_id])
    conn.exec_params("UPDATE accounts SET balance = balance + $1 WHERE id = $2", [100,
 to_id])
  end
end
```

Warning

`pool.with` does NOT automatically wrap your block in a transaction. You must call `conn.transaction` yourself for write operations. On OCC conflict the connector re-executes the entire block, so it should contain only database operations and be safe to retry.

To skip retry on individual calls, pass `retry_occ: false`:

```
pool.with(retry_occ: false) do |conn|
  conn.exec("SELECT 1")
end
```

Configuration options

Field	Type	Default	Description
host	String	(required)	Cluster endpoint or cluster ID
region	String	(auto-detected)	AWS region; required if host is a cluster ID
user	String	"admin"	Database user

Field	Type	Default	Description
database	String	"postgres"	Database name
port	Integer	5432	Database port
profile	String	nil	AWS profile name for credentials
token_duration	Integer	900 (15 min)	Token validity duration in seconds (max allowed: 1 week, default: 15 min)
credentials_provider	Aws::Credentials	nil	Custom credentials provider
max_lifetime	Integer	3300 (55 min)	Maximum connection lifetime in seconds
application_name	String	nil	ORM prefix for application_name
logger	Logger	nil	Logger for OCC retry warnings
occ_max_retries	Integer	nil (disabled)	Max OCC retries on `pool.with` ; enables retry when set

`create_pool` also accepts a `pool:` keyword with a hash of options that you pass directly to `ConnectionPool.new`. If you omit `pool:`, the connector defaults to `{size: 5, timeout: 5}`. Keys you provide override only those specific defaults.

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  pool: { size: 10, timeout: 10 }
)
```

Authentication

The connector automatically handles Aurora DSQL authentication by generating tokens using AWS credentials. If you do not provide the AWS region, the connector parses it from the hostname.

For more information on authentication in Aurora DSQL, see Authentication and authorization for Aurora DSQL.

Admin vs regular users

- Users named "admin" automatically use admin authentication tokens
- All other users use regular authentication tokens
- The connector generates tokens dynamically for each connection

Aurora DSQL Connector for PHP PDO_PGSQL

The Aurora DSQL Connector for PHP is a PHP connector built on PDO_PGSQL that integrates IAM authentication for connecting PHP applications to Amazon Aurora DSQL clusters.

The connector handles token generation, SSL configuration, and connection management so you can focus on your application logic.

About the connector

The Aurora DSQL Connector for PHP adds an authentication layer on top of PDO_PGSQL that handles IAM token generation, allowing you to connect to Aurora DSQL using your existing PDO workflows. Amazon Aurora DSQL requires IAM authentication with time-limited tokens, and the connector automatically manages token generation for you.

What is Aurora DSQL authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: The connector generates authentication tokens using AWS credentials, and these tokens have configurable lifetimes

The Aurora DSQL Connector for PHP understands these requirements and automatically generates IAM authentication tokens when establishing connections.

Features

- **Automatic IAM authentication** - Handles Aurora DSQL token generation and refresh

- **Built on PDO_PGSQL** - Wraps the standard PostgreSQL extension for PHP
- **Seamless integration** - Works with existing PDO workflows
- **SSL enforcement** - Always uses SSL with `verify-full` mode and direct TLS negotiation
- **Region auto-detection** - Extracts AWS region from Aurora DSQL cluster hostname
- **AWS credentials support** - Supports AWS profiles and custom credentials providers
- **OCC retry** - Opt-in optimistic concurrency control retry with exponential backoff and jitter
- **PSR-3 logging** - Compatible logging for retry diagnostics

Example application

For a complete example, see the example application on GitHub.

Quick start guide

Requirements

- PHP 8.2 or later
- `ext-pdo_pgsql` extension
- Access to an Aurora DSQL cluster
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

Add the package to your project:

```
composer require awslabs/aurora-dsql-pdo-pgsql
```

Usage

Configuration-based connection

```
<?php

require_once 'vendor/autoload.php';

use Aws\AuroraDsql\PdoPgsql\AuroraDsql;
```

```
use Aws\AuroraDsql\PdoPgsql\DsqlConfig;

$config = new DsqlConfig(
    host: 'your-cluster.dsql.us-east-1.on.aws',
    occMaxRetries: 3
);
$pdo = AuroraDsql::connect($config);

// Read
$stmt = $pdo->query('SELECT 1 AS result');
$row = $stmt->fetch(PDO::FETCH_ASSOC);
echo "Connected: {$row['result']}\n";

// Transactional write with automatic OCC retry
$id = $pdo->transaction(function (PDO $conn): string {
    $stmt = $conn->prepare('INSERT INTO users (name) VALUES (?) RETURNING id');
    $stmt->execute(['Alice']);
    return $stmt->fetchColumn();
});
```

Connection string format

For simple scripts or when you prefer connection string syntax, the connector supports `postgres://` and `postgresql://` connection strings with Aurora DSQL-specific query parameters:

```
$pdo = AuroraDsql::connectFromDsn(
    'postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres?region=us-east-1'
);
```

Supported query parameters: `region`, `profile`, `tokenDurationSecs`, `ormPrefix`.

OCC retry

Aurora DSQL uses optimistic concurrency control (OCC). When two transactions modify the same data, the first to commit wins and the second receives an OCC error.

OCC retry is opt-in. Set `occMaxRetries` in the config to enable automatic retry with exponential backoff and jitter:

```
$config = new DsqlConfig(
    host: 'your-cluster.dsql.us-east-1.on.aws',
```

```
    occMaxRetries: 3
);
$pdo = AuroraDsql::connect($config);

// Single statements are automatically retried via exec()
$pdo->exec("CREATE INDEX ASYNC ON users (email)");

// Multi-statement transactions are retried via transaction()
$pdo->transaction(function (PDO $conn) {
    $conn->exec("UPDATE accounts SET balance = balance - 100 WHERE id = 1");
    $conn->exec("UPDATE accounts SET balance = balance + 100 WHERE id = 2");
});
```

Important

`transaction()` manages `beginTransaction()`/`commit()`/`rollBack()` internally. Your callback should contain only database operations and be safe to retry.

Configuration options

The connector also accepts `postgres://` and `postgresql://` connection strings with query parameters for configuration. Supported query parameters: `region`, `profile`, `tokenDurationSecs`, and `ormPrefix`.

Field	Type	Default	Description
`host`	`string`	(required)	Cluster endpoint or 26-char cluster ID
`region`	`?string`	`null` (auto-detected)	AWS region; required if `host` is a cluster ID
`user`	`string`	`"admin"`	Database user
`database`	`string`	`"postgres"`	Database name
`port`	`int`	`5432`	Database port
`profile`	`?string`	`null`	AWS profile name for credentials

Field	Type	Default	Description
`credentialsProvider`	`?\Closure`	`null`	Custom AWS credentials provider
`tokenDurationSecs`	`int`	`900` (15 min)	Token validity duration in seconds
`occMaxRetries`	`?int`	`null` (disabled)	Default max OCC retries for `exec()` and `transaction()`
`ormPrefix`	`?string`	`null`	ORM prefix prepended to `application_name`
`logger`	`?LoggerInterface`	`null`	PSR-3 logger for retry warnings and diagnostics

Authentication

The connector automatically handles Aurora DSQL authentication by generating tokens using AWS credentials. If you don't provide the AWS region, the connector parses it from the hostname.

For more information on authentication in Aurora DSQL, see Authentication and authorization for Aurora DSQL.

Admin vs regular users

- Users named "admin" automatically use admin authentication tokens
- All other users use regular authentication tokens
- The connector generates tokens dynamically for each connection

Aurora DSQL Connector for .NET Npgsql

The Aurora DSQL Connector for .NET is a .NET connector built on Npgsql that integrates IAM authentication for connecting .NET applications to Amazon Aurora DSQL clusters.

The connector handles token generation, SSL configuration, and connection pooling so you can focus on your application logic.

About the connector

Amazon Aurora DSQL requires IAM authentication with time-limited tokens that existing .NET PostgreSQL drivers do not natively support. The Aurora DSQL Connector for .NET adds an authentication layer on top of Npgsql that handles IAM token generation, allowing you to connect to Aurora DSQL without changing your existing Npgsql workflows.

What is Aurora DSQL authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: The connector generates authentication tokens using AWS credentials, and these tokens have configurable lifetimes

The Aurora DSQL Connector for .NET understands these requirements and automatically generates IAM authentication tokens when establishing connections.

Features

- **Automatic IAM authentication** - Handles Aurora DSQL token generation and refresh
- **Built on Npgsql** - Wraps the popular PostgreSQL driver for .NET
- **Seamless integration** - Works with existing Npgsql workflows
- **Connection pooling** - Built-in support via `NpgsqlDataSource` with max lifetime enforcement
- **Region auto-detection** - Extracts AWS region from Aurora DSQL cluster hostname
- **AWS credentials support** - Supports AWS profiles and custom credentials providers
- **OCC retry** - Opt-in optimistic concurrency control retry with exponential backoff
- **SSL enforcement** - Always uses SSL with `verify-full` mode and direct TLS negotiation

Example application

For a complete example, see the example application on GitHub.

Quick start guide

Requirements

- .NET 8.0 or later

- Access to an Aurora DSQL cluster
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

Add the package to your project:

```
dotnet add package Amazon.AuroraDsql.Npgsql
```

Usage

Pool connection

```
using Amazon.AuroraDsql.Npgsql;

// Create a connection pool
await using var ds = await AuroraDsql.CreateDataSourceAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws",
    OccMaxRetries = 3
});

// Read
await using (var conn = await ds.OpenConnectionAsync())
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT 'Hello, DSQL!'";
    var greeting = await cmd.ExecuteScalarAsync();
    Console.WriteLine(greeting);
}

// Transactional write with OCC retry
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "INSERT INTO users (id, name) VALUES (gen_random_uuid(), @name)";
    cmd.Parameters.AddWithValue("name", "Alice");
    await cmd.ExecuteNonQueryAsync();
});
```

Single connection

For simple scripts or when you do not need connection pooling:

```
await using var conn = await AuroraDsql.ConnectAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws"
});

await using var cmd = conn.CreateCommand("SELECT 1");
await cmd.ExecuteScalarAsync();
```

OCC retry

Aurora DSQL uses optimistic concurrency control (OCC). When two transactions modify the same data, the first to commit wins and the second receives an OCC error.

OCC retry is opt-in. Set `OccMaxRetries` in the config to enable automatic retry with exponential backoff and jitter. Use `WithTransactionRetryAsync` for transactional writes:

```
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();

    cmd.CommandText = "UPDATE accounts SET balance = balance - 100 WHERE id = @from";
    cmd.Parameters.AddWithValue("from", fromId);
    await cmd.ExecuteNonQueryAsync();

    cmd.CommandText = "UPDATE accounts SET balance = balance + 100 WHERE id = @to";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("to", toId);
    await cmd.ExecuteNonQueryAsync();
});
```

For DDL or single statements, use `ExecWithRetryAsync`:

```
await ds.ExecWithRetryAsync("CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY,
 name TEXT)");
```

> **Important**
>
> `WithTransactionRetryAsync` manages `BEGIN/COMMIT/ROLLBACK` internally and opens a fresh connection for each attempt. Your callback should contain only database operations and be safe to retry.

Configuration options

The connector also accepts `postgres://` and `postgresql://` connection strings with `region` and `profile` query parameters.

Field	Type	Default	Description
`Host`	`string`	(required)	Cluster endpoint or 26-char cluster ID
`Region`	`string?`	(auto-detected)	AWS region; required if `Host` is a cluster ID
`User`	`string`	`"admin"`	Database user
`Database`	`string`	`"postgres"`	Database name
`Port`	`int`	`5432`	Database port
`Profile`	`string?`	`null`	AWS profile name for credentials
`CustomCredentialsProvider`	`AWSCredentials?`	`null`	Custom AWS credentials provider
`TokenDurationSecs`	`int?`	`null` (SDK default, 900s)	Token validity duration in seconds
`OccMaxRetries`	`int?`	`null` (disabled)	Default max OCC retries for retry methods on the data source

Field	Type	Default	Description
`OrmPrefix`	`string?`	`null`	ORM prefix prepended to `application_name`
`LoggerFactory`	`ILoggerFactory?`	`null`	Logger factory for retry warnings and diagnostics
`ConfigureConnectionString`	`Action<NpgsqlConnectionStringBuilder>?`	`null`	Callback to override pool settings or set additional Npgsql connection string properties. SSL and `Enlist` are security invariants and cannot be overridden.

Authentication

The connector automatically handles Aurora DSQL authentication by generating tokens using AWS credentials. If you do not provide the AWS region, the connector parses it from the hostname.

For more information on authentication in Aurora DSQL, see Authentication and authorization for Aurora DSQL.

Admin vs regular users

- Users named "admin" automatically use admin authentication tokens
- All other users use regular authentication tokens
- The connector generates tokens dynamically for each connection

Aurora DSQL Connector for Rust SQLx

The Aurora DSQL Connector for Rust is a Rust connector built on SQLx that integrates IAM authentication for connecting Rust applications to Amazon Aurora DSQL clusters.

The connector handles token generation, SSL configuration, and connection management so you can focus on your application logic.

About the connector

The Aurora DSQL Connector for Rust adds an authentication layer on top of SQLx that handles IAM token generation, allowing you to connect to Aurora DSQL without changing your existing SQLx workflows.

What is Aurora DSQL authentication?

In Aurora DSQL, **authentication** involves:

- **IAM Authentication**: All connections use IAM-based authentication with time-limited tokens
- **Token Generation**: The connector generates authentication tokens using AWS credentials, and these tokens have configurable lifetimes

The Aurora DSQL Connector for Rust understands these requirements and automatically generates IAM authentication tokens when establishing connections.

Features

- **Automatic IAM authentication** - Handles Aurora DSQL token generation and refresh
- **Built on SQLx** - Wraps the popular async PostgreSQL driver for Rust
- **Seamless integration** - Works with existing SQLx workflows
- **Connection pooling** - Opt-in pool support with background token refresh via the `pool` feature
- **Region auto-detection** - Extracts AWS region from Aurora DSQL cluster hostname
- **AWS credentials support** - Supports AWS profiles and the default credential chain
- **OCC retry** - Opt-in optimistic concurrency control retry with exponential backoff and jitter

Example application

For a complete example, see the example application on GitHub.

Quick start guide

Requirements

- Rust 1.80 or later
- Getting started with Aurora DSQL
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Installation

Add to your `Cargo.toml`:

```
[dependencies]
aurora-dsql-sqlx-connector = "0.1.2"
```

For most applications, enable both the `pool` and `occ` features:

```
[dependencies]
aurora-dsql-sqlx-connector = { version = "0.1.2", features = ["pool", "occ"] }
```

Feature flags

Feature	Default	Description
`pool`	No	SQLx pool helper with background token refresh
`occ`	No	OCC retry helpers (`retry_on_occ` , `is_occ_error`)

Usage

Pool connection

```
use sqlx::Row;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let pool = aurora_dsql_sqlx_connector::pool::connect(
        "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
    ).await?;

    // Read
    let row = sqlx::query("SELECT 'Hello, DSQL!' as greeting")
        .fetch_one(&pool)
        .await?;
    let greeting: &str = row.get("greeting");
    println!("{}", greeting);
```

```
    // Write — you must wrap writes in a transaction
    let mut tx = pool.begin().await?;
    sqlx::query("INSERT INTO users (id, name) VALUES (gen_random_uuid(), $1)")
        .bind("Alice")
        .execute(&mut *tx)
        .await?;
    tx.commit().await?;

    pool.close().await;
    Ok(())
}
```

Single connection

For simple scripts or when you don't need connection pooling:

```
use sqlx::Row;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let mut conn = aurora_dsql_sqlx_connector::connection::connect(
        "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
    ).await?;

    let row = sqlx::query("SELECT 1 as value")
        .fetch_one(&mut conn)
        .await?;
    let value: i32 = row.get("value");
    println!("Result: {}", value);

    Ok(())
}
```

Each call to `connection::connect()` generates a fresh IAM token. For operations longer than the token duration, create a new connection.

Advanced usage

Host configuration

The connector supports both full cluster endpoints (region auto-detected) and cluster IDs (region required):

```
// Full endpoint (region auto-detected)
let opts = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
)?;

// Cluster ID (region required)
let opts = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster-id/postgres?region=us-east-1"
)?;
```

AWS profiles

Specify an AWS profile for credentials:

```
let pool = aurora_dsql_sqlx_connector::pool::connect(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres?profile=production"
).await?;
```

Connection string format

The connector supports PostgreSQL connection string formats:

```
postgres://[user@]host[:port]/[database][?param=value&...]
postgresql://[user@]host[:port]/[database][?param=value&...]
```

Supported query parameters: `region`, `profile`, `tokenDurationSecs`, `ormPrefix`.

Pool configuration

For custom pool settings, pass `PgPoolOptions` to `connect_with()`:

```
use aurora_dsql_sqlx_connector::DsqlConnectOptions;
use sqlx::postgres::PgPoolOptions;

let config = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
)?;

let pool = aurora_dsql_sqlx_connector::pool::connect_with(
    &config,
    PgPoolOptions::new().max_connections(20),
).await?;
```

Programmatic configuration

Use `DsqlConnectOptionsBuilder` for programmatic configuration:

```
use aurora_dsql_sqlx_connector::{DsqlConnectOptionsBuilder, Region};
use sqlx::postgres::PgConnectOptions;

let pg = PgConnectOptions::new()
    .host("your-cluster.dsql.us-east-1.on.aws")
    .username("admin")
    .database("postgres");

let opts = DsqlConnectOptionsBuilder::default()
    .pg_connect_options(pg)
    .region(Some(Region::new("us-east-1")))
    .build()?;

let mut conn = aurora_dsql_sqlx_connector::connection::connect_with(&opts).await?;
```

OCC retry

Aurora DSQL uses optimistic concurrency control (OCC). When two transactions modify the same data, the first to commit wins and the second receives an OCC error.

OCC retry is opt-in. Enable the `occ` feature and use `retry_on_occ` to enable automatic retry with exponential backoff and jitter:

```
use aurora_dsql_sqlx_connector::{retry_on_occ, OCCRetryConfig};

let config = OCCRetryConfig::default(); // max_attempts: 3, exponential backoff

retry_on_occ(&config, || async {
    let mut tx = pool.begin().await?;

    sqlx::query("UPDATE accounts SET balance = balance - 100 WHERE id = $1")
        .bind(account_id)
        .execute(&mut *tx)
        .await?;

    tx.commit().await?;
    Ok(())
}).await?;
```

> **Warning**
>
> `retry_on_occ` re-executes the entire closure on OCC conflict, so the closure should contain only database operations and be safe to retry.

Configuration options

Field	Type	Default	Description
`host`	`String`	(required)	Cluster endpoint or cluster ID
`region`	`Option<Region>`	(auto-detected)	AWS region; required if host is a cluster ID
`user`	`String`	`"admin"`	Database user
`database`	`String`	`"postgres"`	Database name
`port`	`u16`	`5432`	Database port
`profile`	`Option<String>`	None	AWS profile name for credentials
`tokenDurationSecs`	`u64`	`900` (15 min)	Token validity duration in seconds
`ormPrefix`	`Option<String>`	None	ORM prefix for application_name (for example, `"diesel"` produces `"diesel:aurora-dsql-rust-sqlx/{version}"`)

Authentication

The connector automatically handles Aurora DSQL authentication by generating tokens using AWS credentials. If you don't provide the AWS region, the connector parses it from the hostname.

For more information on authentication in Aurora DSQL, see Authentication and authorization for Aurora DSQL.

Token generation

- **Connection pools**: A background task refreshes the token at 80% of the token duration. Call `pool.close().await` to stop the refresh task and release pool resources.
- **Single connections**: The connector generates a fresh token at connection time.
- **Token generation** is a local SigV4 presigning operation with negligible cost.

Admin vs regular users

- Users named "admin" automatically use admin authentication tokens
- All other users use regular authentication tokens
- The connector generates tokens dynamically for each connection

Adapters and dialects for Aurora DSQL

Adapters and dialects for Amazon Aurora DSQL provide IAM authentication and improved Aurora DSQL compatibility for specific object-relational mapping (ORM) frameworks. Use an adapter or dialect when you work with a supported ORM framework.

For a list of all adapters and dialects, including Drizzle ORM, see Aurora DSQL adapters and dialects.

For more information about adapter and dialect availability, see the Aurora DSQL ORMs repository on the GitHub website.

Aurora DSQL dialect for Hibernate

The Amazon Aurora DSQL dialect for Hibernate integrates Hibernate object-relational mapping (ORM) with Aurora DSQL. With the dialect, your Java applications can use the ORM capabilities of Hibernate while taking advantage of the distributed architecture and high availability of Aurora DSQL. For Java Database Connectivity (JDBC), the dialect uses the PostgreSQL JDBC Driver internally, and the Aurora DSQL JDBC Connector for IAM authentication. For more information, see the Aurora DSQL Dialect for Hibernate and the Aurora DSQL JDBC Connector on the GitHub website, and the PostgreSQL JDBC Driver website.

The Aurora DSQL Dialect for Hibernate extends and overrides Hibernate's SQL generation to match Aurora DSQL's semantics, including asynchronous index creation, optimistic concurrency control

(OCC), universally unique identifier (UUID) primary keys, and OCC-based foreign key enforcement. The dialect layers above the JDBC driver and generates correct SQL for Aurora DSQL.

About the dialect

Amazon Aurora DSQL is a serverless, distributed relational database service optimized for transactional workloads. Aurora DSQL has specific requirements, including IAM-based authentication, asynchronous index creation, optimistic concurrency control, and OCC-based foreign key enforcement, where foreign key conflicts raise a serialization error that your application can retry.

With the Aurora DSQL Dialect for Hibernate, you can use the standard Hibernate APIs without manually accounting for the behavioral differences of Aurora DSQL. The dialect addresses these requirements by customizing the Hibernate SQL generation layer.

What is Aurora DSQL authentication?

In Aurora DSQL, authentication involves:

- **IAM authentication:** All connections use IAM-based authentication with time-limited tokens.
- **Token generation:** The connector generates authentication tokens from your AWS credentials. Tokens have configurable lifetimes. Aurora DSQL enforces a maximum connection duration of 3,600 seconds (one hour) regardless of token lifetime; recycle connections before they reach this limit.

The Aurora DSQL JDBC Connector generates tokens automatically, so you can authenticate to Aurora DSQL with the Hibernate dialect without changing your existing Hibernate workflows.

Features

The dialect provides the following features:

- **Aurora DSQL-compatible SQL generation** – Generates SQL that conforms to Aurora DSQL semantics.
- **Async index creation** – Creates all indexes with `CREATE INDEX ASYNC` and `CREATE UNIQUE INDEX ASYNC`.
- **Foreign key support** – DDL export includes `FOREIGN KEY` constraints with OCC-based enforcement. Foreign key (FK) conflicts raise a serialization error that your application can retry.

For the foreign key options that Aurora DSQL supports, see Working with foreign key constraints in Aurora DSQL.

- **Optimistic concurrency control** – Supports the `NONE` (OCC) and `PESSIMISTIC_WRITE` lock modes.
- **UUID primary keys** – Registers a UUID JDBC type for correct binding. Use `@ColumnDefault("gen_random_uuid()")` to give the ID column a server-side default.
- **Sequence and identity support** – Generates correct DDL with the mandatory `CACHE 65536` parameter.
- **Connection pooling** – Works with HikariCP. To keep connections in the pool after errors that don't affect the connection, add your own `DsqlExceptionOverride` class, as shown in Keeping healthy connections with `DsqlExceptionOverride`.

Quick start guide

Requirements

Before you use the dialect, make sure that you have the following:

- Java 17 or higher
- Hibernate ORM version 7.2 or higher (Spring Boot 4.x for Spring Boot projects)
- Access to an Amazon Aurora DSQL cluster
- Set up appropriate IAM permissions to allow your application to connect to Aurora DSQL.
- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)
- PostgreSQL JDBC driver version 42.x or higher

Installation

Add the `aurora-dsql-hibernate-dialect` dependency to your project.

Maven

Add the following dependency to your `pom.xml` file:

```
<dependency>
    <groupId>software.amazon.dsql</groupId>
    <artifactId>aurora-dsql-hibernate-dialect</artifactId>
    <version>2.2.0</version>
```

```
</dependency>
```

Gradle

Add the following dependency to your `build.gradle` file:

```
implementation("software.amazon.dsql:aurora-dsql-hibernate-dialect:2.2.0")
```

Install the Aurora DSQL JDBC Connector separately

Add the JDBC connector for IAM authentication as a separate dependency:

Maven

```
<dependency>
    <groupId>software.amazon.dsql</groupId>
    <artifactId>aurora-dsql-jdbc-connector</artifactId>
    <version>1.5.0</version>
</dependency>
```

Gradle

```
implementation("software.amazon.dsql:aurora-dsql-jdbc-connector:1.5.0")
```

> **Note**
>
> The Aurora DSQL JDBC Connector handles IAM token generation automatically. AWS recommends it for connecting to Aurora DSQL from Java applications, including those using Hibernate.

Configuring the dialect

Explicitly specify `AuroraDSQLDialect`. Otherwise, Hibernate selects `PostgreSQLDialect` from the database metadata. You can configure the dialect in several ways:

`hibernate.properties`

Add the following line to your `hibernate.properties` file:

```
hibernate.dialect=software.amazon.dsql.hibernate.dialect.AuroraDSQLDialect
```

`persistence.xml`

Add the following property to your `persistence.xml` file:

```
<property name="hibernate.dialect"
 value="software.amazon.dsql.hibernate.dialect.AuroraDSQLDialect"/>
```

Spring Boot `application.properties`

Add the following line to your `application.properties` file:

```
spring.jpa.properties.hibernate.dialect=software.amazon.dsql.hibernate.dialect.AuroraDSQLDialec
```

Programmatic configuration

Set the dialect when you build the service registry:

```
ServiceRegistry registry = new StandardServiceRegistryBuilder()
    .applySetting("hibernate.dialect",
 "software.amazon.dsql.hibernate.dialect.AuroraDSQLDialect")
    .build();
```

Database connection

Configure your database connection for Aurora DSQL with the Aurora DSQL JDBC Connector and Secure Sockets Layer (SSL) enabled.

> **Note**
>
> The Aurora DSQL JDBC Connector turns on `sslmode=verify-full` and direct SSL negotiation (pgJDBC 42.7.4 or later) by default, so the connection uniform resource locator (URL) needs no SSL parameters.

The following properties configure the database connection. In this example, replace *`cluster-id`* and *`region`* with the values for your cluster.

```
hibernate.connection.url=jdbc:aws-dsql:postgresql://cluster-id.dsql.region.on.aws/
postgres
hibernate.connection.username=admin
hibernate.connection.driver_class=software.amazon.dsql.jdbc.DSQLConnector
```

Basic usage

The following example creates a Hibernate `SessionFactory` connected to Aurora DSQL and performs a basic read operation.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

Configuration config = new Configuration()
    .setProperty("hibernate.dialect",
 "software.amazon.dsql.hibernate.dialect.AuroraDSQLDialect")
    .setProperty("hibernate.connection.url",
        "jdbc:aws-dsql:postgresql://cluster-id.dsql.region.on.aws/postgres")
    .setProperty("hibernate.connection.username", "admin")
    .setProperty("hibernate.connection.driver_class",
 "software.amazon.dsql.jdbc.DSQLConnector")
    .addAnnotatedClass(MyEntity.class);

SessionFactory sessionFactory = config.buildSessionFactory();

try (Session session = sessionFactory.openSession()) {
    Transaction tx = session.beginTransaction();
    try {
        MyEntity result = session.get(MyEntity.class, entityId);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();  // roll back so the session can be reused or closed cleanly
        throw e;
    }
}
```

Configuration options

Option	Type	Required	Description
`hibernate.dialect`	string	Yes	Must be `software.amazon.dsql.hibernate.dialect.AuroraDSQLDialect`

Option	Type	Required	Description
`hibernate.connection.url`	string	Yes	JDBC URL for the Aurora DSQL cluster endpoint
`hibernate.connection.username`	string	Yes	Aurora DSQL username
`hibernate.connection.driver_class`	string	Yes	Must be `software.amazon.dsql.jdbc.DSQLConnector`
`hibernate.dialect.aurora_dsql.sequence_cache_size`	int	No	Cache size for sequences and identity columns. Default: 65536

The dialect also supports all standard Hibernate connection and Jakarta Persistence API (JPA) configuration options.

Using the Aurora DSQL dialect for Hibernate with connection pooling

The dialect works with any JDBC connection pool. The recommended pool for production use is HikariCP, which is the default pool in Spring Boot.

Configure HikariCP with the `DsqlExceptionOverride` class so that the pool keeps healthy connections after Aurora DSQL rejects an unsupported statement, instead of evicting them.

```
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.boot.jdbc.autoconfigure.DataSourceProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
```

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@Configuration(proxyBeanMethods = false)
public class DsqlDataSourceConfig {

    @Bean
    @Primary
    public HikariDataSource dataSource(DataSourceProperties properties) {
        HikariDataSource hds = properties.initializeDataSourceBuilder()
            .type(HikariDataSource.class)
            .build();
        hds.setMaxLifetime(3300000); // 55 minutes
        hds.setExceptionOverrideClassName(DsqlExceptionOverride.class.getName());
        return hds;
    }
}
```

Connection `maxLifetime` configuration

Set the `maxLifetime` parameter to less than 3,600,000 milliseconds (one hour), which is the maximum connection duration that Aurora DSQL allows. Setting a lower `maxLifetime` allows the connection pool to proactively recycle connections, which is more efficient than handling connection timeout errors from the database.

Keeping healthy connections with `DsqlExceptionOverride`

Some Aurora DSQL errors leave the connection healthy, such as OCC conflicts (SQLSTATE `40001`) and unsupported statements (SQLSTATE `0Axxx`). By default, HikariCP evicts connections on `0A000`. Implement `SQLExceptionOverride` and register it with HikariCP so that the pool keeps these connections:

```
import com.zaxxer.hikari.SQLExceptionOverride;
import java.sql.SQLException;

public class DsqlExceptionOverride implements SQLExceptionOverride {

    @java.lang.Override
    public Override adjudicate(SQLException sqlException) {
        String sqlState = sqlException.getSQLState();
        if (sqlState == null) return Override.CONTINUE_EVICT;
```

```
        // SQLSTATE 40001 = OCC conflict (OC000/OC001) — application retries the
 transaction; connection is healthy
        // SQLSTATE 0Axxx = feature_not_supported — Aurora DSQL rejected an unsupported
 statement (for example, SAVEPOINT),
        //   but the connection itself is healthy; evicting it would force an
 unnecessary reconnect
        if (sqlState.equals("40001") || sqlState.matches("0A\\d{3}")) {
            return Override.DO_NOT_EVICT;
        }
        return Override.CONTINUE_EVICT;
    }
}
```

Best practices

Primary key generation

UUID primary keys are the recommended choice in Aurora DSQL because they avoid sequence contention in a distributed environment. With `@GeneratedValue`, Hibernate generates the UUID in the application before the insert. The `@ColumnDefault` annotation adds a `gen_random_uuid()` column default for rows that you insert outside Hibernate. The following Java code example adds a UUID primary key to a base entity class:

```
import jakarta.persistence.*;
import java.util.UUID;
import org.hibernate.annotations.ColumnDefault;

@MappedSuperclass
public class BaseEntity {

    @Id
    @GeneratedValue
    @ColumnDefault("gen_random_uuid()")
    @Column(name = "id", updatable = false, nullable = false)
    private UUID id;
}
```

The dialect also supports sequence and identity-based integer primary keys. The dialect uses a default `CACHE` parameter of `65536` in sequence and identity column definitions. You can override this with the configuration property `hibernate.dialect.aurora_dsql.sequence_cache_size`.

```
@Entity
public class OrderEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @SequenceGenerator(name = "order_seq", sequenceName = "order_seq",
                       initialValue = 1000, allocationSize = 100)
    private Integer id;
}
```

Auto-DDL

For production, use explicit DDL scripts instead of automatic schema generation (`hbm2ddl.auto=create` or `update`) to maintain full control over table definitions, index creation, and schema migrations.

Concurrency control and locking

Aurora DSQL uses optimistic concurrency control (OCC). Use only two Hibernate lock modes:

- **`LockModeType.NONE`** – Uses OCC instead of an explicit database lock. Aurora DSQL checks only the rows that the transaction writes. If a concurrent transaction modifies one of those rows, the transaction that commits second fails with an OCC error (`OC000`, SQLSTATE `40001`).
- **`LockModeType.PESSIMISTIC_WRITE`** – Adds `SELECT ... FOR UPDATE`, which adds the selected rows to the commit-time conflict check instead of locking them. If another transaction changes those rows, the transaction that commits second fails with SQLSTATE `40001`.

Use only `NONE` and `PESSIMISTIC_WRITE`, the two lock modes that the dialect overrides for Aurora DSQL compatibility. Hibernate core handles other lock modes and can generate clauses such as `SELECT ... FOR SHARE` and `SKIP LOCKED`. For the locking clauses that Aurora DSQL supports, see SELECT command.

Handling OCC errors with transaction retry

Because OCC errors are retryable, you can retry the transaction. The following Java code example implements a retry aspect around your transactional methods:

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.dao.CannotAcquireLockException;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import java.util.concurrent.ThreadLocalRandom;

@Aspect
@Component
@Order(Ordered.HIGHEST_PRECEDENCE) // run outside the transaction so commit-time OCC
 errors are retried
public class TransactionRetryAspect {

    private static final int MAX_RETRIES = 3;

    @Around("@annotation(org.springframework.transaction.annotation.Transactional)")
    public Object retryOnOccConflict(ProceedingJoinPoint joinPoint) throws Throwable {
        int attempt = 0;
        while (true) {
            try {
                return joinPoint.proceed();
            } catch (ObjectOptimisticLockingFailureException |
 CannotAcquireLockException e) {
                if (++attempt >= MAX_RETRIES) throw e;
                Thread.sleep(ThreadLocalRandom.current().nextLong(100L << attempt)); //
 exponential backoff with jitter
            }
        }
    }
}
```

Dialect features

The following table describes how the dialect handles keys and constraints.

Feature	Behavior
Foreign keys	Supported — DDL export includes `FOREIGN KEY` constrain ts. Aurora DSQL supports `NO ACTION`, `RESTRICT`, `CASCADE`,

Feature	Behavior
	SET NULL, SET DEFAULT, MATCH SIMPLE, MATCH FULL, and deferrable foreign keys, with SET CONSTRAINTS available for foreign keys only. Enforcement is OCC-based: FK conflicts raise a serialization error (OC000 / SQLSTATE 40001), which you can retry with TransactionRetryAspect . FK constrain t violations (for example, referencing a non-existent row) throw DataIntegrityViolationException and aren't retryable. When you add constraints to existing tables with ALTER TABLE ... ADD CONSTRAINT ... NOT VALID , existing rows remain unvalidated. To validate them, run ALTER TABLE ASYNC ... VALIDATE CONSTRAINT , capture its returned job_id, call sys.wait_for_job(job_id) , and check that the returned succeeded value is true.
Check constraints	supportsColumnCheck() returns true in v2.1.0+. The dialect emits CHECK constraints defined in entity annotations in DDL.
Sequences	Correct DDL including mandatory CACHE parameter (default: 65536).
Identity columns	Uses bigint generated by default as identity (START WITH 1 INCREMENT BY 1 CACHE <size>).
UUID	Custom JDBC type registration ensures UUID columns bind correctly via the PostgreSQL JDBC driver.

The following table describes how the dialect handles SQL generation, locking, and data types.

Feature	Behavior
Index creation	Uses CREATE INDEX ASYNC and CREATE UNIQUE INDEX ASYNC.
Locking	Recommended modes: NONE (OCC) and PESSIMIST IC_WRITE (SELECT ... FOR UPDATE). Hibernate core

Feature	Behavior
	handles other modes. The dialect always generates `FOR UPDATE`, regardless of the lock timeout.
Truncate	Uses `DELETE FROM <table>` in place of `TRUNCATE`.
Temporary tables	`supportsTemporaryTables()` returns `false`; bulk mutations use the common table expression (CTE) strategies `CteMutationStrategy` and `CteInsertStrategy`, which create no physical tables.
Data types	`float` (6 digits), `double` (15 digits), `numeric` (precision 18). `varchar` max length: 65,535.

Authentication

The Aurora DSQL JDBC Connector automatically handles Aurora DSQL authentication by generating IAM tokens for each new connection. If you don't provide the AWS Region in the connection URL, the connector parses it from the cluster hostname.

For more information about authentication in Aurora DSQL, see Authentication and authorization.

Admin vs regular users

- Users named `"admin"` automatically use admin authentication tokens.
- All other users use non-admin authentication tokens.
- The JDBC connector generates tokens dynamically for each new connection.

Examples

For full example code, see the Hibernate pet clinic example application on the GitHub website. For instructions on how to run the example, see the Hibernate pet clinic example README on the GitHub website.

The following table lists the example files and what each one demonstrates.

Description	Example
Connecting to Aurora DSQL with Hibernate and HikariCP	DsqlDataSourceConfig.java
Keeping healthy connections after Aurora DSQL errors in HikariCP	DsqlExceptionOverride.java
Retrying transactions on OCC conflicts	TransactionRetryAspect.java
UUID primary key entity definition	BaseEntity.java
OneToMany relationship with OCC-based foreign key enforcement	Owner.java
ManyToMany relationship with OCC-based foreign key enforcement	Vet.java

Aurora DSQL adapter for Django

The Amazon Aurora DSQL adapter for Django (`aurora-dsql-django`) integrates Django object-relational mapping (ORM) with Aurora DSQL. It handles IAM authentication, Secure Sockets Layer (SSL) configuration, DDL generation differences (async indexes, identity/sequence columns), and connection management, so you can use the full Django ORM against a distributed Aurora DSQL cluster with minimal configuration. For more information, see the Aurora DSQL adapter for Django and the Django pet clinic example application on the GitHub website.

About the adapter

Amazon Aurora DSQL is a serverless, distributed relational database service optimized for transactional workloads. Aurora DSQL has specific requirements, including IAM-based authentication and asynchronous index creation. Aurora DSQL supports foreign key constraints enforced through optimistic concurrency control (OCC): foreign key (FK) conflicts raise a

serialization error that your application can retry. As of v0.7.0, the Django adapter enables `supports_foreign_keys = True` and emits `REFERENCES` constraints in migrations and DDL.

With the Aurora DSQL adapter for Django, you can use the full Django ORM against Aurora DSQL without manually accounting for its behavioral differences. The adapter addresses these requirements by customizing the Django database backend.

What is Aurora DSQL authentication?

In Aurora DSQL, authentication involves:

- **IAM authentication:** All connections use IAM-based authentication with time-limited tokens.
- **Token generation:** The connector generates authentication tokens from your AWS credentials. Tokens have configurable lifetimes.

The Aurora DSQL adapter for Django automatically generates IAM authentication tokens when establishing connections, using the `aurora-dsql-python-connector`.

Features

The adapter provides the following features:

- **Automatic IAM authentication** – Generates IAM tokens automatically for each connection from your AWS credentials.
- **SSL** – Configure SSL through `OPTIONS`. The adapter supports the `require` and `verify-full` modes.
- **UUID primary keys** – `AutoField` and `BigAutoField` map to universally unique identifier (UUID) columns with a `gen_random_uuid()` default.
- **Sequence and identity keys** – Opt-in integer primary keys use `GENERATED BY DEFAULT AS IDENTITY`.
- **Async index creation** – Creates all indexes with `CREATE INDEX ASYNC`.
- **Foreign key support** – `supports_foreign_keys = True`, so migrations emit `REFERENCES` constraints with OCC-based enforcement. For the foreign key options that Aurora DSQL supports, see Working with foreign key constraints in Aurora DSQL.
- **OCC error handling** – Surfaces `OC000` (data conflict) and `OC001` (schema conflict) so that your application can retry.
- **Django compatibility** – Works with Django 4.2 long-term support (LTS), 5.0, 5.1, and 5.2 LTS.

Prerequisites

Before you use the adapter, make sure that you have the following:

- Python 3.10 or higher
- Django 4.2 or higher
- `psycopg` (v3) or `psycopg2`
- AWS credentials with `dsql:DbConnectAdmin` permission (admin) or `dsql:DbConnect` (non-admin)

Installation

Use Python 3.10 or newer. Install the adapter, the Python connector, and the recommended psycopg v3 driver in the same environment:

```
PROMPT> python3 -m pip install aurora-dsql-django
PROMPT> python3 -m pip install "aurora-dsql-python-connector[psycopg]>=0.2.7"
 "psycopg[binary]"
```

If you use psycopg2 instead, install the connector and psycopg2 driver without the psycopg v3 extra:

```
PROMPT> python3 -m pip install aurora-dsql-django
PROMPT> python3 -m pip install "aurora-dsql-python-connector>=0.2.7" psycopg2-binary
```

Configuration

`settings.py` — minimal

In the following example, replace *`cluster-id`* and *`region`* with the values for your cluster, and `admin` with your database user.

```
DATABASES = {
    'default': {
        'ENGINE': 'aurora_dsql_django',
        'HOST':   'cluster-id.dsql.region.on.aws',
        'USER':   'admin',
        'NAME':   'postgres',
```

```
        'OPTIONS': {
            'sslmode': 'verify-full',
        }
    }
}
```

With `verify-full`, the adapter defaults to `sslrootcert="system"`, which requires libpq 16 or later. On Linux, the `psycopg[binary]` and `psycopg2-binary` wheels also can't locate the system trust store, so set the `SSL_CERT_FILE` environment variable to your certificate authority (CA) bundle. Alternatively, pass the bundle path as `sslrootcert`, as shown in the next example. This works with any libpq version.

`settings.py` — with a custom certificate bundle

```
DATABASES = {
    'default': {
        'ENGINE': 'aurora_dsql_django',
        'HOST':   'cluster-id.dsql.region.on.aws',
        'USER':   'admin',
        'NAME':   'postgres',
        'OPTIONS': {
            'sslmode':    'verify-full',
            'sslrootcert': '/path/to/cert.pem',  # omit to use the system trust store
        }
    }
}
```

`settings.py` — optional parameters

The following table describes the keys that you can set inside the `OPTIONS` dictionary.

Key	Default	Description
`sslmode`	libpq default (`prefer`)	SSL verification level. Aurora DSQL requires TLS. Use `verify-full` for production. `require` encrypts the connection without verifying the certificate, so use it only outside production.
`sslrootcert`	system CA	Path to CA certificate for `verify-full`

Key	Default	Description
profile	default credential chain	AWS profile name for credential resolution
token_duration_secs	60	IAM token lifetime in seconds (connector default)

The following table describes the top-level keys of the database dictionary. Set these keys alongside ENGINE and HOST, not inside OPTIONS.

Key	Default	Description
USE_SEQUENCE_AUTOFIELDS	False	If True, AutoField /BigAutoField use IDENTITY columns instead of UUID
SEQUENCE_CACHE_SIZE	65536	Cache size used when USE_SEQUENCE_AUTOFIELDS is True

Primary key strategies

UUID (default — recommended)

```
import uuid
from django.db import models

class Owner(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name = models.CharField(max_length=30)
    city = models.CharField(max_length=80)
```

uuid.uuid4 generates the UUID client-side before the INSERT. When you keep the default AutoField or BigAutoField (no explicit id field), the adapter uses server-generated UUIDs (gen_random_uuid()).

Identity / sequence keys (opt-in)

Enable in settings.py:

```
DATABASES = {
    'default': {
        'ENGINE': 'aurora_dsql_django',
        # ...
        'USE_SEQUENCE_AUTOFIELDS': True,
        'SEQUENCE_CACHE_SIZE': 65536,
    }
}
```

With this flag set, `AutoField` and `BigAutoField` generate `BIGINT GENERATED BY DEFAULT AS IDENTITY (CACHE 65536)` columns instead of UUID.

`SequenceAutoField` (per-model explicit sequences)

```
from aurora_dsql_django import SequenceAutoField

class Owner(models.Model):
    id = SequenceAutoField(primary_key=True)
    name = models.CharField(max_length=30)
```

Defining models

Foreign key fields

Aurora DSQL supports foreign key constraints with OCC-based enforcement — conflicts raise a serialization error (`OC000` / `SQLSTATE 40001`) that your application can retry — and supports `NO ACTION`, `RESTRICT`, `CASCADE`, `SET NULL`, `SET DEFAULT`, `MATCH SIMPLE`, `MATCH FULL`, and deferrable foreign keys.

Django generates `REFERENCES` constraints in migrations. Define FK fields normally — `db_constraint=True` (the default):

```
class Pet(models.Model):
    id    = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name  = models.CharField(max_length=30)
    owner = models.ForeignKey(Owner, on_delete=models.CASCADE, null=True)
```

When adding FK constraints to existing tables, the adapter automatically issues `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` followed by `ALTER TABLE ASYNC ... VALIDATE CONSTRAINT` — you don't need to run these manually.

For CHECK constraints added via `ALTER TABLE`, the adapter issues the `NOT VALID` and `VALIDATE CONSTRAINT` calls, but the migration doesn't wait for the async validation job. Check the corresponding job in `sys.jobs` promptly and confirm that it succeeded. The constraint becomes valid only after the job succeeds.

> **Important**
>
> To add an FK column to an existing table, add the column with `null=True`, backfill all rows, and keep the column nullable. Adding the column as non-nullable raises `NotSupportedError`. The following example shows the supported steps:
>
> ```
> # Step 1: add as nullable — this succeeds
> owner = models.ForeignKey(Owner, on_delete=models.CASCADE, null=True)
> # Step 2: backfill owner_id for existing rows; keep the column nullable
> ```

Many-to-many relationships — use a custom intermediate table

Define the intermediate table explicitly rather than relying on Django's auto-generated one. An explicit through-model gives you full control over the primary key type, ensuring it's compatible with Aurora DSQL:

```
class VetSpecialties(models.Model):
    id        = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    vet       = models.ForeignKey('Vet', on_delete=models.CASCADE)
    specialty = models.ForeignKey('Specialty', on_delete=models.CASCADE)

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=['vet', 'specialty'],
 name='unique_vet_specialty'),
        ]

class Vet(models.Model):
    id          = models.UUIDField(primary_key=True, default=uuid.uuid4,
 editable=False)
    name        = models.CharField(max_length=30)
    specialties = models.ManyToManyField('Specialty', through='VetSpecialties')
```

One-to-one relationships

```
class Vet(models.Model):
    id    = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name  = models.CharField(max_length=30)
    owner = models.OneToOneField(
        Owner,
        on_delete=models.SET_DEFAULT,
        null=True, blank=True, default=None
    )
```

Migrations

```
PROMPT> python3 manage.py makemigrations pet_clinic
PROMPT> python3 manage.py migrate pet_clinic 0001
```

> **Important**
>
> Remove the default `django.contrib.admin` and `django.contrib.auth` apps from `INSTALLED_APPS` if you aren't using them — they depend on integer auto-increment primary keys (PKs) and may fail with Aurora DSQL's UUID default.

OCC retry pattern

Aurora DSQL uses optimistic concurrency control (OCC). Conflicting concurrent transactions raise a `django.db.Error` with one of two error codes — `OC000` (data conflict: two transactions modified the same row) or `OC001` (schema conflict: cached schema catalog is out of date). Wrap write views in a retry decorator:

```
import random
import time
from django.db import Error

OCC_SQLSTATE = "40001"

def _is_occ_error(exc):
    # Django wraps the driver exception; the original lives in __cause__
    cause = getattr(exc, "__cause__", None) or getattr(exc, "__context__", None)
    return (
```

```
        getattr(cause, "sqlstate", None) == OCC_SQLSTATE   # psycopg (v3)
        or getattr(cause, "pgcode", None) == OCC_SQLSTATE  # psycopg2
    )

def with_retries(retries=3, initial_wait=1, delay_factor=2):
    def decorator(view):
        def wrapper(*args, **kwargs):
            delay = initial_wait
            for attempt in range(retries):
                try:
                    return view(*args, **kwargs)
                except Error as e:
                    if not _is_occ_error(e):
                        raise  # non-retryable — propagate immediately
                    if attempt == retries - 1:
                        raise  # exhausted retries
                    print(f"OCC conflict, retrying (attempt {attempt+1}/{retries})")
                    time.sleep(random.uniform(0, delay))  # exponential backoff with
 jitter
                    delay *= delay_factor
        return wrapper
    return decorator
```

Apply to view methods:

```
from django.db.transaction import atomic

@with_retries()
@atomic
def post(self, request, *args, **kwargs):
    # ... write logic ...
```

`settings.py` checklist for Aurora DSQL

The following example shows the `settings.py` values to review for Aurora DSQL:

```
INSTALLED_APPS = [
    'your_app',           # add your app
    # remove 'django.contrib.admin', 'django.contrib.auth', and so on
    # unless you have reconfigured them for UUID PKs
]
```

```
DATABASES = {
    'default': {
        'ENGINE': 'aurora_dsql_django',
        'HOST':   'cluster-id.dsql.region.on.aws',
        'USER':   'admin',
        'NAME':   'postgres',
        'OPTIONS': {'sslmode': 'verify-full'},
    }
}
```

If you turn on persistent connections, set `CONN_MAX_AGE` below 3600 seconds (for example, `3300`). Aurora DSQL closes connections after one hour.

Adapter features reference

The following table lists adapter support for indexes and constraints.

Capability	Supported	Notes
`CREATE INDEX ASYNC`	Yes	All index creation uses async syntax
`CREATE UNIQUE INDEX ASYNC`	Yes	The adapter creates unique constraints as async unique indexes
Foreign key enforcement	Yes	`supports_foreign_keys = True` ; the adapter emits `REFERENCES` constraints in migrations. Aurora DSQL enforces FKs via OCC: conflicts raise `OC000` (`SQLSTATE 40001`), which your application can retry
Deferrable unique constraints	No	Aurora DSQL supports deferral for foreign keys; unique constraints use immediate enforcement. `supports_deferrable_unique_constraints = False` reflects this platform behavior.
Check constraints	Yes	`supports_table_check_constraints` is `True`, so the adapter emits CHECK constraints in model `Meta.constraints` . For

Capability	Supported	Notes
		CHECK constraints added to existing tables, inspect the async validation job, because the migration doesn't wait for it
`SET CONSTRAINTS`	Yes	Supported for foreign keys
Index `opclasses`	No	The adapter creates standard indexes.

The following table lists adapter support for primary keys.

Capability	Supported	Notes
UUID auto PKs	Yes (default)	`AutoField` / `BigAutoField` → `uuid` + `gen_random_uuid()`
Identity/sequence PKs	Yes (opt-in)	`USE_SEQUENCE_AUTOFIELDS = True`
`SequenceAutoField`	Yes	Custom field for explicit sequence-backed integer PKs

The following table lists adapter support for transactions and errors.

Capability	Supported	Notes
`SAVEPOINT`	No	The adapter sets the `uses_savepoints` and `can_release_savepoints` feature flags to `False`
OCC error codes	`OC000` (data conflict), `OC001` (schema conflict)	Catch `django.db.Error` and retry

The following table lists adapter support for Django versions.

Capability	Supported	Notes
Django 4.2 LTS	Yes	Not applicable
Django 5.x	Yes	5.0, 5.1, 5.2 LTS

Aurora DSQL dialect for SQLAlchemy

The Amazon Aurora DSQL dialect for SQLAlchemy (`aurora-dsql-sqlalchemy`) integrates SQLAlchemy object-relational mapping (ORM) with Aurora DSQL. It handles IAM authentication, Secure Sockets Layer (SSL) configuration, DDL generation differences (async indexes, identity columns), and connection management, so you can use the full SQLAlchemy API against a distributed Aurora DSQL cluster with minimal configuration. For more information, see the Aurora DSQL dialect for SQLAlchemy and the SQLAlchemy pet clinic example application on the GitHub website.

This topic contains the following sections:

- Prerequisites
- Installation
- Quick start
- Configuration
- Basic usage
- Primary key generation
- Index options
- OCC retry pattern
- SSL configuration
- Connection pooling
- Best practices
- Dialect features reference

About the dialect

Amazon Aurora DSQL is a serverless, distributed relational database service optimized for transactional workloads. Aurora DSQL has specific requirements, including IAM-based authentication, asynchronous index creation, DDL transaction restrictions, and foreign key enforcement through optimistic concurrency control (OCC), where conflicts raise a serialization error that your application can retry.

With the Aurora DSQL dialect for SQLAlchemy, you can use the full SQLAlchemy ORM API against Aurora DSQL without manually accounting for its behavioral differences. The dialect handles these differences automatically.

What is Aurora DSQL authentication?

In Aurora DSQL, authentication involves:

- **IAM authentication:** All connections use IAM-based authentication with time-limited tokens.
- **Token generation:** The connector generates authentication tokens from your AWS credentials. Tokens have configurable lifetimes.

The Aurora DSQL dialect for SQLAlchemy works with the `aurora-dsql-python-connector`, which automatically generates IAM authentication tokens when establishing connections.

Features

The dialect provides the following features:

- **Automatic IAM authentication** – Generates IAM tokens automatically from your AWS credentials.
- **SSL** – Defaults to `verify-full` mode with the system certificate authority (CA) trust store.
- **Foreign key support** – The dialect supports `FOREIGN KEY` constraints. Conflicts raise a serialization error (`OC000` / `SQLSTATE 40001`), so your application needs retry logic. For the foreign key options that Aurora DSQL supports, see Working with foreign key constraints in Aurora DSQL.
- **Async index creation** – Creates all indexes with `CREATE INDEX ASYNC`.
- **Identity columns and sequences** – `autoincrement=True` generates `GENERATED BY DEFAULT AS IDENTITY (CACHE 65536)`; `Identity(always=True, cache=65536)` generates `GENERATED ALWAYS AS IDENTITY (CACHE 65536)`.

- **UUID primary keys** – Supports server-generated universally unique identifiers (UUIDs) through `gen_random_uuid()`.
- **Index extensions** – Supports the `INCLUDE` clause and the `NULLS NOT DISTINCT` index option.
- **Driver support** – Compatible with psycopg (v3, default and recommended), psycopg2, and asyncpg (through the `aurora-dsql-python-connector` directly).

Prerequisites

Before you use the dialect, make sure that you have the following:

- Python 3.10 or higher
- SQLAlchemy 2.0.42 or higher
- One of the following PostgreSQL drivers:
  - psycopg 3.2.0 or higher (`psycopg[binary]`) — default and recommended
  - psycopg2 2.9.8 or higher (`psycopg2-binary`)
- AWS credentials with `dsql:DbConnectAdmin` permission (admin) or `dsql:DbConnect` (non-admin)

> **Note**
>
> The `aurora-dsql-sqlalchemy` dialect supports psycopg and psycopg2. For asyncpg connections, use the `aurora-dsql-python-connector` directly.

Installation

```
PROMPT> pip install aurora-dsql-sqlalchemy
PROMPT> pip install "aurora-dsql-python-connector[psycopg]>=0.2.7"  # includes
 psycopg_pool and is_occ_error

# Install a driver (choose one):
PROMPT> pip install "psycopg[binary]"   # psycopg v3 (recommended)
PROMPT> pip install psycopg2-binary     # psycopg2
```

> **Note**
>
> Use `pip install "psycopg[binary]"`. The `psycopg-binary` package is a prebuilt-wheel distribution that isn't intended for direct installation.

Quick start

The following example creates an engine. In this example, replace *`cluster-id`* and *`region`* with the values for your cluster.

```
from aurora_dsql_sqlalchemy import create_dsql_engine

engine = create_dsql_engine(
    host="cluster-id.dsql.region.on.aws",
    user="admin",
    driver="psycopg",   # or "psycopg2"
)
```

When you call `create_dsql_engine`, it automatically does the following:

- Generates a short-lived IAM auth token via `aurora-dsql-python-connector`
- Configures SSL (`sslmode=verify-full`, `sslrootcert=system`)
- Enables direct TLS negotiation when libpq >= 17 is available
- Sets up connection pooling with sensible defaults

Configuration

`create_dsql_engine` parameters

Parameter	Type	Default	Description
`host`	`str`	required	Aurora DSQL cluster endpoint
`user`	`str`	required	Database user (for example, `"admin"`)

Parameter	Type	Default	Description
`driver`	`str`	`"psycopg"`	`"psycopg"` (v3, default) or `"psycopg2"`
`dbname`	`str`	`"postgres"`	Database name
`sslmode`	`str`	`"verify-full"`	SSL verification mode
`sslrootcert`	`str`	`"system"`	CA certificate path or `"system"`
`application_name`	`str`	`"sqlalchemy"`	Connection label visible in pg_stat_activity
`pool_size`	`int`	`5`	Connections to keep open in the pool
`max_overflow`	`int`	`10`	Extra connections allowed beyond `pool_size`
`connect_args`	`dict`	None	Extra args passed to the connector (for example, `region`, `profile`, `token_duration_secs`)
`**engine_kwargs`	Not applicable	Not applicable	Any additional keyword arguments for `create_engine()`

Manual engine creation (advanced)

If you need full control over the uniform resource locator (URL) and connector, you can create the engine manually. The following example creates a SQLAlchemy engine with the Aurora DSQL connector:

```
from sqlalchemy import create_engine
from sqlalchemy.engine import URL
import aurora_dsql_psycopg as dsql_connector

url = URL.create("auroradsql+psycopg", username="admin",
                 host="cluster-id.dsql.region.on.aws", database="postgres")
```

```
engine = create_engine(
    url,
    creator=lambda: dsql_connector.DSQLConnection.connect(
        host="cluster-id.dsql.region.on.aws",
        user="admin",
        sslmode="verify-full",
        sslrootcert="system",
    ),
    pool_size=5,
    max_overflow=10,
)
```

Basic usage

The following example defines a model, creates its table, and then inserts and queries a row:

```
from sqlalchemy import String
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import DeclarativeBase, Mapped, Session, mapped_column
from sqlalchemy.sql import text
from aurora_dsql_sqlalchemy import create_dsql_engine

engine = create_dsql_engine(host="cluster-id.dsql.region.on.aws", user="admin")

class Base(DeclarativeBase):
    pass

class Owner(Base):
    __tablename__ = "owner"
    id: Mapped[str] = mapped_column(UUID, primary_key=True,
 default=text("gen_random_uuid()"))
    name: Mapped[str] = mapped_column(String(30))
    city: Mapped[str] = mapped_column(String(80))

# Create tables
Base.metadata.create_all(engine)

# Insert
with Session(engine) as session:
    session.add(Owner(name="Mary Major", city="Seattle"))
    session.commit()
```

```
# Query
with Session(engine) as session:
    owners = session.query(Owner).filter_by(city="Seattle").all()
    print(owners)
```

Primary key generation

UUID (recommended)

Server-generated UUIDs are the recommended primary key type for Aurora DSQL.

```
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.sql import text

id: Mapped[str] = mapped_column(
    UUID,
    primary_key=True,
    default=text("gen_random_uuid()")
)
```

Sequences

```
from sqlalchemy import BigInteger, Sequence

id: Mapped[int] = mapped_column(
    BigInteger,
    primary_key=True,
    default=Sequence("my_seq", cache=65536)
)
```

Aurora DSQL requires sequence cache to be either 1 or >= 65536. The dialect automatically adds `CACHE 65536` (the minimum allowed non-trivial value) if you don't specify a cache, and upgrades `INTEGER` sequences to `BIGINT`.

Identity columns

```
from sqlalchemy import BigInteger, Identity

id: Mapped[int] = mapped_column(
    BigInteger,
    Identity(always=True, cache=65536),  # generates: GENERATED ALWAYS AS IDENTITY
 (CACHE 65536)
```

```
    primary_key=True,
)

# Or using autoincrement (dialect adds CACHE 65536 automatically):
id: Mapped[int] = mapped_column(BigInteger, primary_key=True, autoincrement=True)
```

Index options

`INCLUDE` clause

```
from sqlalchemy import Index

Index("idx_name", table.c.id, auroradsql_include=["name", "email"])
# # CREATE INDEX ASYNC idx_name ON table (id) INCLUDE (name, email)
```

`NULLS NOT DISTINCT`

```
Index("idx_unique_col", table.c.column, unique=True,
 auroradsql_nulls_not_distinct=True)
# # CREATE UNIQUE INDEX ASYNC idx_unique_col ON table (column) NULLS NOT DISTINCT
```

OCC retry pattern

Aurora DSQL uses optimistic concurrency control (OCC). Conflicting concurrent transactions raise a `DBAPIError` with one of two error codes:

- **`OC000`** — Data conflict: two transactions modified the same row concurrently.
- **`OC001`** — Schema conflict: the transaction's cached schema catalog is out of date.

Applications should catch both and retry. The following example retries a statement when an OCC conflict occurs:

```
import random
import time

from sqlalchemy import text
from sqlalchemy.exc import DBAPIError
from aurora_dsql_psycopg import is_occ_error  # requires aurora-dsql-python-connector
 >=0.2.7
```

```
def execute_with_retry(engine, statement, max_retries=5):
    with engine.connect() as conn:
        for attempt in range(max_retries):
            try:
                conn.execute(text(statement))
                conn.commit()
                return
            except DBAPIError as e:
                conn.rollback()
                if not is_occ_error(e.orig):
                    raise
                if attempt == max_retries - 1:
                    raise
                time.sleep(random.uniform(0, 0.1 * 2 ** attempt))  # exponential
 backoff with jitter
```

SSL configuration

Aurora DSQL requires TLS for all connections. The following table describes the supported `sslmode` values.

`sslmode` value	Behavior
`verify-full` (default)	Verifies server certificate **and** hostname — recommended
`verify-ca`	Verifies certificate only, not hostname
`require`	Encrypts connection but skips certificate verification

`sslrootcert="system"` uses the operating system's CA trust store. On Linux, the `psycopg[binary]` and `psycopg2-binary` wheels can't locate the system trust store. Set the `SSL_CERT_FILE` environment variable to your CA bundle, or pass the bundle path as `sslrootcert`. To pin a specific CA:

```
engine = create_dsql_engine(
    host="cluster-id.dsql.region.on.aws",
    user="admin",
    sslrootcert="/path/to/custom-ca.pem",
)
```

Important

The default `sslrootcert="system"` works only with `sslmode="verify-full"`. To use `verify-ca`, set `sslrootcert` to a CA file path. To use `require`, pass `sslrootcert=None` to clear the default:

```
engine = create_dsql_engine(
    host="cluster-id.dsql.region.on.aws",
    user="admin",
    sslmode="require",
    sslrootcert=None,
)
```

Connection pooling

`create_dsql_engine` creates a pooled engine by default.

```
engine = create_dsql_engine(
    host="cluster-id.dsql.region.on.aws",
    user="admin",
    pool_size=10,       # steady-state connections
    max_overflow=20,    # burst capacity above pool_size
)
```

Important

Aurora DSQL enforces a **one-hour maximum connection duration**. The cluster closes long-lived connections. Set `pool_recycle` below 3600 to avoid race conditions — 3300 provides a safe margin:

```
engine = create_dsql_engine(..., pool_recycle=3300)  # recycle before the 1-hour
 limit
```

Best practices

- Use UUID primary keys with `gen_random_uuid()` to avoid sequence hot-spots in distributed writes.

- The dialect supports foreign key constraints, including referential actions, match types, and deferrable foreign keys. Enforcement is OCC-based: foreign key (FK) conflicts raise a serialization error (`OC000` / `SQLSTATE 40001`) that you can retry in the same way as general write conflicts. FK constraint violations (referencing a non-existent row) raise a separate exception class and aren't retryable; don't use `is_occ_error` to gate retries on those.
- Aurora DSQL supports `SET CONSTRAINTS` for foreign keys only. For existing tables, add FK or CHECK constraints using `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`; the dialect appends `NOT VALID` automatically. Validating existing rows requires a separate async step. Record the `job_id` that `ALTER TABLE ASYNC` returns, then check the `succeeded` value that `sys.wait_for_job` returns:

  ```
  ALTER TABLE t ADD CONSTRAINT fk_name FOREIGN KEY (col) REFERENCES other(id) NOT
   VALID;
  ALTER TABLE ASYNC t VALIDATE CONSTRAINT fk_name;
  -- Use the job_id returned by the preceding statement:
  CALL sys.wait_for_job('returned_job_id');  -- verify succeeded is true
  ```

  Confirm that `sys.wait_for_job` returns `succeeded` as true. `convalidated` becomes `true` only after the job succeeds.
- Wrap write transactions in OCC retry loops to handle concurrent conflicts.
- Set `pool_recycle=3300` to stay within Aurora DSQL's one-hour connection duration limit with a safe margin.
- Use `INCLUDE` indexes for covering queries to reduce secondary reads.
- Use `Identity(always=True, cache=65536)` or an explicit `Sequence` with `cache=65536` for integer primary keys. Without an explicit `cache` argument, `Identity(always=True)` generates `GENERATED ALWAYS AS IDENTITY` without a CACHE clause. Aurora DSQL supports sequence cache values of 1 or at least 65536, so include an explicit supported cache value.
- Issue DDL and DML in separate transactions. Each transaction can contain only one DDL statement.

Dialect features reference

Capability	Supported	Notes
`CREATE INDEX ASYNC`	Yes	All indexes use async creation
`FOREIGN KEY` constraints	Yes	Supports referential actions, `MATCH SIMPLE` and `MATCH FULL`, and deferrable constraints. Conflicts raise `OC000` (`SQLSTATE 40001`). For details, see Best practices.
DDL + DML in same transaction	No	Use separate transactions for DDL and DML. Each transaction can contain only one DDL statement.
Sequences	Yes	Sequences must use `BIGINT`. The dialect adds `CACHE 65536`.
Identity columns	Yes	`autoincrement=True` → `GENERATED BY DEFAULT AS IDENTITY (CACHE 65536)`; `Identity(always=True, cache=65536)` → `GENERATED ALWAYS AS IDENTITY (CACHE 65536)`
`SAVEPOINT`	No	The dialect suppresses the init-time `SAVEPOINT` automatically.
`SKIP LOCKED`	No	For the supported locking clauses, see SELECT command.
`NULLS NOT DISTINCT`	Yes	Via `auroradsql_nulls_not_distinct` index option
`INCLUDE` clause	Yes	Via `auroradsql_include` index option
CHECK constraints	Yes	The dialect appends `NOT VALID` when you add a CHECK constraint to an existing table. To validate existing rows, see Best practices.

Capability	Supported	Notes
OCC conflict codes	`OC000` (data conflict), `OC001` (schema conflict)	Catch `DBAPIError` and check `e.orig`

Aurora DSQL adapter for Tortoise ORM

Like the other adapters and dialects for Amazon Aurora DSQL, the Aurora DSQL adapter for Tortoise ORM (`aurora-dsql-tortoise-orm`) integrates an object-relational mapping (ORM) framework with Aurora DSQL. It provides custom backends for both `asyncpg` and `psycopg` drivers, with Aurora DSQL-aware schema generators for asynchronous index creation, IDENTITY columns, and foreign key enforcement through optimistic concurrency control (OCC), and IAM authentication, so you can use the full Tortoise ORM API against a distributed Aurora DSQL cluster. For more information, see the Aurora DSQL adapter for Tortoise ORM and the Tortoise ORM pet clinic example application on the GitHub website.

About the adapter

Amazon Aurora DSQL is a serverless, distributed relational database service optimized for transactional workloads. Aurora DSQL has specific requirements, including IAM-based authentication, asynchronous index creation, universally unique identifier (UUID) primary keys by default, and OCC-based foreign key enforcement. Foreign key (FK) conflicts raise a serialization error that your application can retry.

With the Aurora DSQL adapter for Tortoise ORM, you can use the full Tortoise ORM API against Aurora DSQL without manually accounting for its behavioral differences. The adapter addresses these requirements with custom Tortoise ORM backends and schema generators.

What is Aurora DSQL authentication?

In Aurora DSQL, authentication involves:

- **IAM authentication:** All connections use IAM-based authentication with time-limited tokens.
- **Token generation:** The connector generates authentication tokens from your AWS credentials. Tokens have configurable lifetimes.

The adapter uses a password factory that calls `aurora-dsql-python-connector` for each connection to generate IAM tokens automatically. Connections use Secure Sockets Layer (SSL), but the connector doesn't verify the server certificate by default. To turn on verification, see SSL configuration.

Features

The adapter provides the following features:

- **Automatic IAM authentication** – Generates IAM tokens automatically from your AWS credentials.
- **Driver support** – Provides custom backends for both the `asyncpg` and `psycopg` drivers.
- **Async index creation** – Creates all indexes with `CREATE INDEX ASYNC`.
- **Foreign key support** – The adapter emits `REFERENCES` constraints in generated DDL, with OCC-based enforcement. For the foreign key options that Aurora DSQL supports, see Working with foreign key constraints in Aurora DSQL.
- **UUID primary keys** – `UUIDField` with `uuid.uuid4` is the recommended primary key strategy.
- **BigInt identity columns** – `BigIntField` primary keys use `GENERATED BY DEFAULT AS IDENTITY` with `CACHE 65536`.
- **aerich migrations** – Works with aerich migration tooling.
- **Connection uniform resource locator (URL) support** – Supports the `dsql+asyncpg://` and `dsql+psycopg://` URL schemes.
- **OCC error handling** – Surfaces `OC000` (data conflict) and `OC001` (schema conflict) so that your application can retry.

Prerequisites

Before you use the adapter, make sure that you have the following:

- Python 3.10 or higher
- AWS credentials with `dsql:DbConnectAdmin` permission (admin) or `dsql:DbConnect` (non-admin)
- Your cluster endpoint and user — the code examples below read these from `CLUSTER_ENDPOINT` and `CLUSTER_USER` environment variables, but you can supply them any way you prefer

Installation

```
PROMPT> pip install "aurora-dsql-python-connector>=0.2.7"  # provides is_occ_error for
 the retry example

# Install the adapter with one or both drivers:
PROMPT> pip install "aurora-dsql-tortoise-orm[asyncpg]"  # asyncpg backend
PROMPT> pip install "aurora-dsql-tortoise-orm[psycopg]"  # psycopg backend
```

Configuration

Tortoise.init() — programmatic

```
import os
from tortoise import Tortoise

CLUSTER_ENDPOINT = os.environ["CLUSTER_ENDPOINT"]
CLUSTER_USER = os.environ.get("CLUSTER_USER", "admin")

await Tortoise.init(
    config={
        "connections": {
            "default": {
                "engine": "aurora_dsql_tortoise.asyncpg",   # or .psycopg
                "credentials": {
                    "host": CLUSTER_ENDPOINT,
                    "user": CLUSTER_USER,
                },
            }
        },
        "apps": {
            "models": {
                "models": ["myapp.models"],
                "default_connection": "default",
            }
        },
    }
)
```

SSL configuration

Aurora DSQL requires TLS for all connections. To verify the server certificate, set `sslmode` to `verify-full` and `sslrootcert` to a certificate authority (CA) certificate file, such as Amazon Root CA 1. For more information, see Configuring SSL/TLS certificates for Aurora DSQL connections. The asyncpg backend takes these keys in `credentials` and doesn't accept `sslrootcert="system"`. The psycopg backend takes them in `server_settings`:

```
# asyncpg
"credentials": {
    "host": CLUSTER_ENDPOINT,
    "user": CLUSTER_USER,
    "sslmode": "verify-full",
    "sslrootcert": "/path/to/custom-ca.pem",
},

# psycopg
"credentials": {
    "host": CLUSTER_ENDPOINT,
    "user": CLUSTER_USER,
    "server_settings": {"sslmode": "verify-full", "sslrootcert": "/path/to/custom-
ca.pem"},
},
```

URL format

Before use, call `register_asyncpg()` to register the `dsql+asyncpg://` URL scheme, or `register_psycopg()` to register `dsql+psycopg://`. `register_backends()` registers both and requires both drivers. In the following example, replace *`cluster-id`* and *`region`* with the values for your cluster:

```
from aurora_dsql_tortoise import register_asyncpg
register_asyncpg()

await Tortoise.init(
    db_url="dsql+asyncpg://admin@cluster-id.dsql.region.on.aws/postgres",
    modules={"models": ["myapp.models"]},
)
# or, after register_psycopg():
db_url = "dsql+psycopg://admin@cluster-id.dsql.region.on.aws/postgres"
```

The adapter type-casts URL query parameters automatically (for example, `?max_size=10&timeout=30`).

Connection duration

Aurora DSQL enforces a **one-hour maximum connection duration**. The cluster closes long-lived connections.

asyncpg: `max_inactive_connection_lifetime` controls how long an *idle* connection stays in the pool before the pool closes it. It applies only to idle connections, so a continuously busy connection can still reach the one-hour limit. For workloads with long-running or continuously-used connections, recycle connections in application logic. Alternatively, switch to the **psycopg** backend. Its pool supports a `max_lifetime` parameter that caps total connection age regardless of activity.

> **psycopg backend**
>
> `psycopg_pool` defaults `max_lifetime` to `3600.0` seconds — exactly Aurora DSQL's hard limit. Set it explicitly to a lower value (for example, `3300`) to give the pool time to recycle connections before the cluster closes them:
>
> ```
> "credentials": {
>     "host": CLUSTER_ENDPOINT,
>     "user": CLUSTER_USER,
>     "max_lifetime": 3300,  # recycle before the 1-hour hard limit
> }
> ```

Schema per user

For non-admin users, set `CLUSTER_USER` to the non-admin user name. The adapter uses `dsql:DbConnect` instead of `dsql:DbConnectAdmin` for token generation automatically.

Defining models

UUID primary key (recommended)

```
import uuid
from tortoise import fields
from tortoise.models import Model
```

```
class Owner(Model):
    id   = fields.UUIDField(primary_key=True, default=uuid.uuid4)
    name = fields.CharField(max_length=30)
    city = fields.CharField(max_length=80)

    class Meta:
        table = "owner"
```

BigInt identity column (opt-in)

```
from tortoise import fields
from tortoise.models import Model

class Owner(Model):
    id   = fields.BigIntField(primary_key=True)   # # GENERATED BY DEFAULT AS IDENTITY
 (CACHE 65536)
    name = fields.CharField(max_length=30)

    class Meta:
        table = "owner"
```

> **Important**
>
> Use `UUIDField` or `BigIntField` as primary keys. `IntField` and `SmallIntField` primary keys raise a `ValueError` at schema generation time.

Foreign key relations

Aurora DSQL supports `NO ACTION`, `RESTRICT`, `CASCADE`, `SET NULL`, `SET DEFAULT`, `MATCH SIMPLE`, `MATCH FULL`, and deferrable foreign keys, and supports `SET CONSTRAINTS` for foreign keys only. Enforcement is OCC-based: FK conflicts raise a serialization error (`OC000` / `SQLSTATE 40001`), which you can detect with `is_occ_error` and retry. FK constraint violations (referencing a non-existent row) raise a distinct exception and aren't retryable. All DML on referenced or referencing tables incurs extra reads, and constraints added to existing tables use `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` followed by `ALTER TABLE ASYNC ... VALIDATE CONSTRAINT`; the adapter's aerich integration handles this automatically.

```
class Pet(Model):
    id     = fields.UUIDField(primary_key=True, default=uuid.uuid4)
```

```
    name   = fields.CharField(max_length=30)
    owner  = fields.ForeignKeyField("models.Owner", related_name="pets")

    class Meta:
        table = "pet"
```

The generated DDL includes an `owner_id` column with a `REFERENCES owner(id)` constraint. Enforcement is OCC-based — FK conflicts raise `0C000`, which your application can retry.

Many-to-many

```
class Vet(Model):
    id          = fields.UUIDField(primary_key=True, default=uuid.uuid4)
    name        = fields.CharField(max_length=30)
    specialties = fields.ManyToManyField("models.Specialty", related_name="vets")

    class Meta:
        table = "vet"

class Specialty(Model):
    id   = fields.UUIDField(primary_key=True, default=uuid.uuid4)
    name = fields.CharField(max_length=80)

    class Meta:
        table = "specialty"
```

Schema generation

`generate_schemas()` splits the DDL on `;` (respecting quoted strings, dollar-quoted blocks, and comments) and executes each statement individually:

```
await Tortoise.generate_schemas()
```

This is safe to call — the splitter is quote/comment/dollar-quote aware.

Basic usage

```
from tortoise import Tortoise
import uuid

# Create
owner = await Owner.create(name="John Doe", city="Seattle")
```

```
pet   = await Pet.create(name="Buddy", owner=owner)

# Query
found = await Pet.get(id=pet.id).prefetch_related("owner")
print(found.owner.name)

# Filter
owners = await Owner.filter(city="Seattle").all()

# Update
await Owner.filter(id=owner.id).update(city="Portland")

# Delete
await pet.delete()
await owner.delete()

# Close
await Tortoise.close_connections()
```

OCC retry pattern

Aurora DSQL uses optimistic concurrency control (OCC). Conflicting concurrent transactions raise a serialization failure with SQLSTATE `40001`. The error message contains `OC000` (data conflict: two transactions modified the same row) or `OC001` (schema conflict: the transaction's cached schema catalog has become out of date). Wrap write operations in a retry loop:

```
import asyncio
import random
from aurora_dsql_asyncpg import is_occ_error  # requires aurora-dsql-python-connector
 >=0.2.7
# psycopg backend: from aurora_dsql_psycopg import is_occ_error

async def with_retry(coro_fn, max_retries=5, base_delay=0.1):
    for attempt in range(max_retries):
        try:
            return await coro_fn()
        except Exception as e:
            if not is_occ_error(e):
                raise
            if attempt == max_retries - 1:
                raise
            await asyncio.sleep(random.uniform(0, base_delay * 2 ** attempt))  #
 exponential backoff with jitter
```

```
# Usage
result = await with_retry(lambda: Owner.create(name="Jane", city="Seattle"))
```

aerich migrations

aerich works with the Aurora DSQL backends. Install aerich, and then run `aerich init` to point aerich at an Aurora DSQL backend:

```
PROMPT> pip install "aerich>=0.9.2"
PROMPT> aerich init -t myapp.config.TORTOISE_ORM
```

```
# myapp/config.py
TORTOISE_ORM = {
    "connections": {
        "default": {
            "engine": "aurora_dsql_tortoise.asyncpg",
            "credentials": {"host": CLUSTER_ENDPOINT, "user": CLUSTER_USER},
        }
    },
    "apps": {
        "models": {
            "models": ["myapp.models", "aerich.models", "aurora_dsql_tortoise.aerich"],
            "default_connection": "default",
        }
    },
}
```

Include the `aurora_dsql_tortoise.aerich` module in the `models` list so that migrations succeed on Aurora DSQL. It applies the Aurora DSQL-specific patches to aerich: UUID primary key for the Aerich table, `RUN_IN_TRANSACTION=False`, and single-DDL statement splitting.

SQL splitting

The adapter includes a `split_sql(sql: str) -> list[str]` utility. It correctly handles the following SQL elements when it splits statements:

- Single-quoted strings (including escaped `''` quotes)
- Double-quoted identifiers
- Dollar-quoted strings (`$$...$$` and `$tag$...$tag$`)

- Single-line comments (`-- ...`)
- Multi-line comments (`/* ... */`)
- Trailing/consecutive semicolons

The following Python code example splits a multi-statement SQL script into individual statements:

```
from aurora_dsql_tortoise.common.config import split_sql

statements = split_sql("""
    CREATE TABLE owner (id UUID PRIMARY KEY DEFAULT gen_random_uuid());
    CREATE TABLE pet   (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), owner_id UUID);
    CREATE INDEX ASYNC idx_pet_owner ON pet (owner_id);
""")
# # ['CREATE TABLE owner ...', 'CREATE TABLE pet ...', 'CREATE INDEX ASYNC ...']
```

Adapter features reference

Capability	Supported	Notes
asyncpg backend	Yes	`aurora_dsql_tortoise.asyncpg`
psycopg backend	Yes	`aurora_dsql_tortoise.psycopg`
`CREATE INDEX ASYNC`	Yes	All indexes use async creation
`REFERENCES` constraints	Yes	The adapter emits these in generated DDL. Aurora DSQL supports referential actions, match types, and deferrable foreign keys, and OCC-based enforcement raises `OC000`, which your application can retry
UUID primary keys	Yes	Recommended — `UUIDField(primary_key=True, default=uuid.uuid4)`
BigInt identity primary keys (PKs)	Yes	`BigIntField(primary_key=True)` → `GENERATED BY DEFAULT AS IDENTITY (CACHE 65536)`

Capability	Supported	Notes
IntField / SmallIntField PKs	No	Raises `ValueError` at schema generation
`generate_schemas()`	Yes	The adapter splits the SQL into single statements before execution
aerich migrations	Yes	Works with the standard aerich config
URL scheme	Yes	`dsql+asyncpg://` and `dsql+psycopg://`
URL parameter type casting	Yes	int, float, bool params cast automatically
`SELECT ... FOR UPDATE`	Yes	Adds the selected rows to the commit-time conflict check instead of locking them. The transaction that commits second fails with SQLSTATE `40001`, so retry the whole transaction. Added in v0.3.0
OCC error codes	`0C000` (data conflict), `0C001` (schema conflict)	SQLSTATE `40001`; catch and retry
Python version	3.10+	Not applicable

Accessing Aurora DSQL with PostgreSQL-compatible clients

Aurora DSQL uses the PostgreSQL wire protocol. You can connect to PostgreSQL using a variety of tools and clients, such as AWS CloudShell, psql, DBeaver, and DataGrip. The following table summarizes how Aurora DSQL maps common PostgreSQL connection parameters:

PostgreSQL	Aurora DSQL	Notes
Role (also known as User or Group)	Database Role	Aurora DSQL creates a role for you named `admin`. When you create custom database

PostgreSQL	Aurora DSQL	Notes
		roles, you must use the `admin` role to associate them with IAM roles for authenticating when connecting to your cluster. For more information, see Using database roles and IAM authentication.
Host (also known as hostname or hostspec)	Cluster Endpoint	Aurora DSQL single-Region clusters provide a single managed endpoint and automatically redirect traffic if there is unavailability within the Region.
Port	N/A – use default `5432`	This is the PostgreSQL default.
Database (dbname)	use `postgres`	Aurora DSQL creates this database for you when you create the cluster.
SSL Mode	SSL is always enabled server-side	In Aurora DSQL, Aurora DSQL supports the `require` SSL Mode. Connections without SSL are rejected by Aurora DSQL.
Password	Authentication Token	Aurora DSQL requires temporary authentication tokens instead of long-lived passwords. To learn more, see Generating an authentication token in Amazon Aurora DSQL.

When connecting, Aurora DSQL requires a signed IAM authentication token in place of a traditional password. These temporary tokens are generated using AWS Signature Version 4 and are used only during connection establishment. Once connected, the session remains active until it ends or the client disconnects.

If you attempt to open a new session with an expired token, the connection request fails and a new token must be generated. For more information, see Generating an authentication token in Amazon Aurora DSQL.

Supported session parameters

Aurora DSQL automatically optimizes storage, connection management, and transaction processing, so it supports a focused set of session parameters. Aurora DSQL accepts the parameters listed in the following table. Parameters outside this set return the following error:

```
ERROR: setting configuration parameter "parameter_name" not supported
```

Aurora DSQL supports the following PostgreSQL session parameters. Unless otherwise noted, you can set these using `SET`, `SET SESSION`, or as connection string options. You can also use `SET LOCAL` to set any of these parameters for the duration of a single transaction. Use `RESET` to restore a parameter to its default value.

Supported session parameters

Parameter	Category	Description
`application_name`	Client identification	Sets the application name reported in connection metadata. Useful for identifying connections in monitoring.
`client_encoding`	Localization	Sets the client-side character encoding. The database uses UTF-8 internally.
`datestyle`	Output formatting	Sets the display format for date and time values (for example, `ISO, MDY`).
`extra_float_digits`	Output formatting	Sets the number of digits displayed for floating-point values. Some drivers, such as JDBC, use this parameter.
`intervalstyle`	Output formatting	Sets the display format for interval values.
`timezone`	Localization	Sets the time zone for the session. Aurora DSQL stores all timezone-aware dates and times internally in UTC. This parameter controls how Aurora DSQL displays values to the client.

Parameter	Category	Description
search_path	Schema resolution	Sets the schema search order for unqualified object names.
enable_bitmapscan	Query planner	Enables or disables the query planner's use of bitmap-scan plan types.
enable_hashjoin	Query planner	Enables or disables the query planner's use of hash-join plan types.
enable_indexonlyscan	Query planner	Enables or disables the query planner's use of index-only-scan plan types.
enable_indexscan	Query planner	Enables or disables the query planner's use of index-scan plan types.
enable_material	Query planner	Enables or disables the query planner's use of materialization.
enable_mergejoin	Query planner	Enables or disables the query planner's use of merge-join plan types.
enable_nestloop	Query planner	Enables or disables the query planner's use of nested-loop join plans.
dsql.enable_batched_nestloop	Query planner	Enables or disables the query planner's use of batched nested-loop join plans.
enable_seqscan	Query planner	Enables or disables the query planner's use of sequential scan plan types.
role	Session identity	Sets the current role. In Aurora DSQL, you can only set this parameter using SET LOCAL within a transaction block.

Aurora DSQL automatically manages the following aspects of your database, so you don't need to configure the corresponding PostgreSQL session parameters:

- **Connection and network parameters** – Aurora DSQL manages connection lifecycle and TCP settings internally, including `tcp_keepalives_idle`, `tcp_keepalives_interval`, and `tcp_keepalives_count`.
- **Memory and resource parameters** – Aurora DSQL automatically manages memory allocation, including `work_mem` and `shared_buffers`.
- **Timeout parameters** – Aurora DSQL enforces its own transaction duration limits, including `statement_timeout`, `lock_timeout`, and `idle_in_transaction_session_timeout`.
- **Replication and WAL parameters** – Aurora DSQL handles replication automatically through its built-in multi-Region architecture, including `wal_level` and `synchronous_commit`.
- **Logging parameters** – Aurora DSQL manages logging internally, including `log_statement` and `client_min_messages`.
- **Transaction parameters** – Aurora DSQL uses a fixed `REPEATABLE READ` isolation level for all transactions. If your driver requires specifying an isolation level, use `BEGIN ISOLATION LEVEL REPEATABLE READ`.

Access Aurora DSQL using SQL clients

Aurora DSQL supports multiple PostgreSQL-compatible clients for connecting to your cluster. The following sections describe how to connect using PostgreSQL with AWS CloudShell or your local command line, as well as GUI-based tools like DBeaver and JetBrains DataGrip. Each client requires a valid authentication token as described in the previous section.

Topics

- Use DBeaver to access Aurora DSQL
- Use JetBrains DataGrip to access Aurora DSQL
- Use the PostgreSQL interactive terminal (psql) to access Aurora DSQL
- Use Aurora DSQL driver for SQLTools
- Troubleshooting

Use DBeaver to access Aurora DSQL

DBeaver is a universal SQL client that can be used to manage any database that has a JDBC driver. It is widely used among developers and database administrators because of its robust data viewing,

editing, and management capabilities. Using DBeaver's cloud connectivity options, you can connect DBeaver to Aurora DSQL natively.

DBeaver Pro

DBeaver PRO products offer native integration with Aurora DSQL as of version 25.3. Follow the instructions from DBeaver Documentation to connect to your Aurora DSQL cluster.

DBeaver Community Edition

DBeaver Community Edition is the free and open-source version. Visit the download page for installation instructions. In order to connect to DSQL from DBeaver Community Edition, you need to install the Aurora DSQL Plugin for DBeaver.

The Aurora DSQL Plugin for DBeaver is built on top of the Aurora DSQL Connector for JDBC and enables IAM authentication to Aurora DSQL clusters. It is conveniently installed through DBeaver UI and eliminates the need to write token generation code or manually supply a valid IAM token, simplifying the authentication while eliminating security risks associated with traditional user-generated passwords.

Features

- IAM Authentication Support: Connect to Aurora DSQL clusters using AWS IAM credentials for secure, password-free authentication
- Automatic Driver Management: Seamlessly installs and configures the Aurora DSQL Connector for JDBC
- Flexible Connection Options: Choose between Host-based or JDBC URL-based connection configuration

Aurora DSQL Plugin for DBeaver Installation

1. With DBeaver opened, Go to the Drop down menu **Help** → **Install New Software**
2. Click **Add** to add a new repository
3. Enter:
   - **Name**: `Aurora DSQL Plugin`
   - **Location**: `https://awslabs.github.io/aurora-dsql-dbeaver-plugin/update-site/`
4. Check **Aurora DSQL Connector for JDBC**

5. Click **Next**, accept the license, and complete the installation
6. Restart DBeaver when prompted

Create an Aurora DSQL Connection

1. Click the **New Database Connection**
2. Select **Aurora DSQL**
3. Under **Server**, select one of the following for the **Connect by** setting
   - **Host**
     - to enable the user interface text inputs for the following fields:
       - **Endpoint:** DSQL Cluster Endpoint
       - **Username:** DSQL username (e.g. admin)
       - **AWS Profile:** e.g. default - The standard profile used when no specific profile is specified
       - **AWS Region (Optional):** must match the region where your DSQL cluster exists, otherwise authentication will fail
   - **URL**
     - JDBC URL in this format:

       ```
       jdbc:aws-dsql:postgresql://{cluster_endpoint}/{database}?
       user=admin&profile=default&region=us-east-1
       ```

     - Note: In this mode, only the URL input is enabled. In order to add parameters to the JDBC connection string, use the URL query parameters format starting with ? as the first parameter and append an & for subsequent parameters.
4. Click **Test Connection** to verify the Aurora DSQL connection works
5. Click **Finish**

Troubleshooting

Windows Trust Store Issue

Windows users may encounter issues downloading the Aurora DSQL Connector for JDBC driver from Maven Central.

Cause: Windows Trust Store may not include the certificates required to access Maven Central repository.

Solution:

1. Run DBeaver as "Administrator"
2. Uncheck this setting - Windows > Preferences > Connections > "Use Windows Trust store"

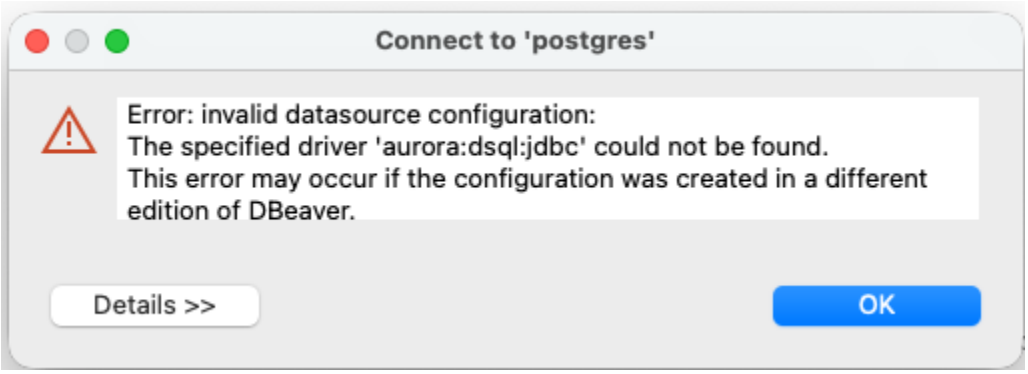
Missing Driver Error

If you see a missing driver icon or connection errors, the Aurora DSQL (Community Plugin) may not be installed in your current DBeaver version. See below some examples of errors and how to fix them:

- Creating a new connection with the missing driver:



- Attempting to connect without the driver:



Cause: When multiple DBeaver versions are installed, connection settings are shared but drivers are installed per application.

Solution: Reinstall the Aurora DSQL (Community plugin) by following the installation steps above.

> **Important**
>
> The administrative features provided by DBeaver for PostgreSQL databases (such as **Session Manager** and **Lock Manager**) don't apply to Aurora DSQL databases due to their

unique architecture. While accessible, these screens don't provide reliable information about database health or status.

Use JetBrains DataGrip to access Aurora DSQL

JetBrains DataGrip is a cross-platform IDE for working with SQL and databases, including PostgreSQL. DataGrip includes a robust GUI with an intelligent SQL editor. To download DataGrip, go to the download page on the *JetBrains* website.

To set up a new Aurora DSQL connection in JetBrains DataGrip

1. Choose **New Data Source** and choose PostgreSQL.
2. In the **Data Sources/General** tab, enter the following information:

   - **Host** – Use your cluster endpoint.

     Port – Aurora DSQL uses the PostgreSQL default: `5432`

     Database – Aurora DSQL uses the PostgreSQL default of `postgres`

     Authentication – Choose `User & Password` .

     Username – Enter `admin`.

     Password – Generate a token and paste it into this field.

     URL – Don't modify this field. It will be auto-populated based on the other fields.
3. **Password** – Provide this by generating an authentication token. Copy the resulting output of the token generator and paste it into the password field.

   > **Note**
   >
   > You must set SSL mode in the client connections. Aurora DSQL supports `PGSSLMODE=require` and `PGSSLMODE=verify-full`. Aurora DSQL enforces SSL communication on the server side and rejects non-SSL connections. For the `verify-full` option you will need to install the SSL certificates locally. For more information see SSL/TLS certificates.
4. You should be connected to your cluster and can start running SQL statements:

Important

Some views provided by DataGrip for PostgreSQL databases (such as Sessions) don't apply to Aurora DSQL databases because of their unique architecture. While accessible, these screens don't provide reliable information about the actual sessions connected to the database.

Use the PostgreSQL interactive terminal (psql) to access Aurora DSQL

Use AWS CloudShell to access Aurora DSQL with the PostgreSQL interactive terminal (psql)

Use the following procedure to access Aurora DSQL with the PostgreSQL interactive terminal from AWS CloudShell. For more information, see What is AWS CloudShell.

To connect using AWS CloudShell

1. Sign in to the Aurora DSQL console.
2. Choose the cluster for which you would like to open in CloudShell. If you haven't yet created a cluster, follow the steps in Step 1: Create an Aurora DSQL single-Region cluster or Create a multi-Region cluster.
3. Choose **Connect with Query Editor** and then choose **Connect with CloudShell**.
4. Choose whether you want to connect as an admin or with a custom database role.
5. Choose **Launch in CloudShell** and choose **Run** in the following CloudShell dialog.

Use the local CLI to access Aurora DSQL with the PostgreSQL interactive terminal (psql)

Use `psql`, a terminal-based front-end to PostgreSQL utility, to interactively enter in queries, issue them to PostgreSQL, and view the query results.

> **Note**
>
> To improve query response times, use the PostgreSQL version 17 client. If you use the CLI in a different environment, make sure you manually set up Python version 3.8+ and psql version 14+.

Download your operating system's installer from the PostgreSQL Downloads page. For more information about `psql`, see PostgreSQL Client Applications on the *PostgreSQL* website.

If you already have the AWS CLI installed, use the following example to connect to your cluster.

```
# 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
```

Use Aurora DSQL driver for SQLTools

The Aurora DSQL Driver for SQLTools is a Visual Studio Code extension for Amazon Aurora DSQL that integrates with SQLTools. It enables developers to connect to and query Aurora DSQL databases directly from VS Code. The driver is available for installation from Visual Studio Marketplace and Open VSX Registry. Kiro, Cursor and other VSCode-based IDEs can use the Open VSX Registry to install the driver following the standard installation procedure described in this page.

Features

- Automatic IAM Authentication
- Standard database operations like browsing schemas, tables, and executing SQL queries.

Installation

1. Open the Extensions view.
2. Search for "Aurora DSQL Driver for SQLTools".
3. Click "Install".

Note:

The SQLTools extension will be automatically installed if not already present.

Authentication

In Aurora DSQL all connections use **IAM-based authentication** with time-limited tokens. The driver automatically handles Aurora DSQL authentication using the Aurora DSQL Connector for node-postgres.

For more information on authentication in Aurora DSQL, see the user guide.

Create an Aurora DSQL Connection

Prerequisites

- AWS credentials configured (via AWS CLI, environment variables, or IAM roles)

Steps

1. Click the SQLTools icon in the left sidebar.
2. In the SQLTools pane, hover over CONNECTIONS and click the Add New Connection icon.
3. In the SQLTools Settings tab select Aurora DSQL Driver from the list.
4. Fill in the connection parameters.
   - AWS Region
     - Optional - the region will be parsed from the Aurora DSQL cluster endpoint.
     - Required when only a cluster ID is specified in the DSQL Cluster field.

- AWS Profile
  - Used for token generation.
  - Uses the default profile if not specified.

5. Click the "Test Connection button" to test the connection.
6. Click Save Connection.

Troubleshooting

Authentication credentials expiration for the SQL Clients

Established sessions remain authenticated for a maximum of 1 hour or until an explicit disconnect or a client-side timeout takes place. If new connections need to be established, a new authentication token must be generated and provided in the **Password** field of the connection. Trying to open a new session (for example, to list new tables, or open a new SQL console) forces a new authentication attempt. If the authentication token configured in the **Connection** settings is no longer valid, that new session will fail and all previously opened sessions will become invalid. Keep this in mind when choosing the duration of your IAM authentication token with the `expires-in` option, which can be set to 15 minutes by default and can be set to a maximum value of seven days.

Additionally, see the Troubleshooting section of the Aurora DSQL documentation.

Amazon Aurora DSQL cluster connectivity tools

Aurora DSQL is compatible with many third-party database drivers and ORM libraries. AWS provides two types of tools to simplify working with Aurora DSQL:

- **Connectors** – Authentication plugins that extend database drivers to handle IAM token generation automatically. Use connectors when working directly with database drivers.
- **Adapters and dialects** – Extensions for specific ORM frameworks that provide IAM authentication and improved Aurora DSQL compatibility. Use adapters when working with a supported ORM framework.

Aurora DSQL adapters and dialects

The following table shows the available adapters and dialects for Aurora DSQL. Each repository is available on the GitHub website.

Programming language	ORM/Framework	Repository link
Java	Hibernate	aurora-dsql-orms/java/hibernate
Python	Django	aurora-dsql-orms/python/django
Python	SQLAlchemy	aurora-dsql-orms/python/sqlalchemy
Python	Tortoise ORM	aurora-dsql-orms/python/tortoise-orm
TypeScript	Drizzle ORM	aurora-dsql-orms/node/drizzle

Database driver samples

The following table shows sample code for connecting to Aurora DSQL using third-party database drivers. Each sample repository is available on the GitHub website.

Programming language	Driver	Sample repository link
C++	libpq	aurora-dsql-samples/cpp/libpq
C# (.NET)	Npgsql	aurora-dsql-samples/dotnet/npgsql
Go	pgx	aurora-dsql-samples/go/pgx
Java	HikariCP + pgJDBC	aurora-dsql-samples/java/pgjdbc
JavaScript	node-postgres (AWS Lambda)	aurora-dsql-samples/lambda

Programming language	Driver	Sample repository link
JavaScript	node-postgres	aurora-dsql-samples/javascript/node-postgres
JavaScript	Postgres.js	aurora-dsql-samples/javascript/postgres-js
Python	asyncpg	aurora-dsql-samples/python/asyncpg
Python	Psycopg	aurora-dsql-samples/python/psycopg
Python	Psycopg2	aurora-dsql-samples/python/psycopg2
Ruby	pg	aurora-dsql-samples/ruby/ruby-pg
Rust	SQLx	aurora-dsql-samples/rust/sqlx

ORM and framework samples

The following table shows sample code for using third-party ORM libraries and frameworks with Aurora DSQL. Each sample repository is available on the GitHub website.

Programming language	ORM/Framework	Sample repository link
Java	Hibernate	aurora-dsql-orms/java/hibernate/examples/pet-clinic-app
Java	Liquibase	aurora-dsql-samples/java/liquibase
Java	Spring Boot	aurora-dsql-samples/java/spring_boot

Programming language	ORM/Framework	Sample repository link
Python	Django	aurora-dsql-orms/python/django/examples/pet-clinic-app
Python	SQLAlchemy	aurora-dsql-orms/python/sqlalchemy/examples/pet-clinic-app
Python	Tortoise ORM	aurora-dsql-orms/python/tortoise-orm/example
Ruby	Rails	aurora-dsql-samples/ruby/rails
TypeScript	Drizzle ORM	aurora-dsql-orms/node/drizzle/examples/veterinary-app
TypeScript	Prisma	aurora-dsql-samples/typescript/prisma-multi-region
TypeScript	Sequelize	aurora-dsql-samples/typescript/sequelize
TypeScript	TypeORM	aurora-dsql-samples/typescript/type-orm

Loading data into Aurora DSQL

Whether you are migrating from an existing database, importing files from Amazon Simple Storage Service, or loading data from your local system, Aurora DSQL provides multiple approaches for getting your data in. This section covers the recommended tools and techniques for data loads of all sizes, from gigabytes to hundreds of terabytes.

Choosing a loading approach

Aurora DSQL supports standard PostgreSQL data loading commands, but loading data efficiently at scale requires handling parallelization, connection management, and error recovery. The following table summarizes your options:

Approach	Best for	Considerations
Aurora DSQL Loader - Open source utility that makes it easy to parallelize inserts when using Aurora DSQL	Most data loading scenarios, especially migrations and bulk imports	Handles parallelization, connection pooling, conflict resolution, and IAM authentication automatically. Available as source code or binary.
PostgreSQL `\copy` - Client-side `psql` meta-command	Simple loads when you are already connected via `psql`	Reads files on the client and streams data over the connection; you manage parallelization yourself
`INSERT` transactions - Standard SQL DML	Small datasets or application-driven inserts	Simplest approach but slowest for bulk data

For most data loading tasks, use the Aurora DSQL Loader. It handles the operational complexity of loading data into a distributed database, including parallel execution across multiple connections and automatic retry of failed operations.

Aurora DSQL Loader

The Aurora DSQL Loader is an open-source command-line utility designed to efficiently load data into Aurora DSQL clusters. It manages connection pooling, parallelizes data transfer across multiple workers, and handles conflicts and retries automatically.

Key features

The Aurora DSQL Loader provides the following capabilities:

Parallel loading

Configurable worker threads enable concurrent data loading across multiple connections for improved performance.

Connection pooling

Manages a pool of connections to your Aurora DSQL cluster, handling IAM authentication and connection lifecycle automatically.

Multiple file format support

Supports CSV (comma-separated values), TSV (tab-separated values), and Apache Parquet columnar format. The loader automatically detects the file format based on the source URI extension.

Automatic schema inference

When used with the `--if-not-exists` flag, the loader can automatically create tables with appropriate column types based on the data.

Conflict handling

When your target table has unique constraints, configure how the loader handles conflicts using the `--on-conflict` option: skip duplicates, upsert records, or return an error.

Fault tolerance

Automatic retries and job resumption capabilities ensure that interrupted loads can continue from their stopping point rather than restarting entirely.

Local and S3 sources

Load data from local file system paths or directly from Amazon S3 buckets using S3 URIs.

Prerequisites

Before using the Aurora DSQL Loader, ensure you have the following:

- An active Aurora DSQL cluster with a valid endpoint.
- AWS credentials configured through the AWS CLI (**aws configure**), AWS Single Sign-On (**aws sso login**), or IAM roles.
- IAM permissions: `dsql:DbConnectAdmin` or `dsql:DbConnect` on your Aurora DSQL cluster.

- For S3 sources, appropriate permissions to read from the source bucket.

Installation

Download the latest release from the GitHub releases page. Pre-built binaries are available for common platforms. For instructions on building from source, see the Aurora DSQL Loader repository.

Usage examples

The following examples demonstrate common use cases for the Aurora DSQL Loader.

Example Loading a local CSV file

This example loads a CSV file from your local file system into an existing table:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table
```

Example Loading data from Amazon S3

This example loads a Parquet file from an Amazon S3 bucket:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/data.parquet \
  --table my_table
```

Example Automatic table creation

This example creates a new table automatically based on the data schema:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --if-not-exists
```

Example Validating before loading

This example validates your configuration without actually loading data:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --dry-run
```

Example Resuming an interrupted load

If a load operation is interrupted, you can resume it using the job ID from the previous run:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --resume-job-id job-id \
  --manifest-dir ./loader-state
```

> **Note**
>
> When resuming, the loader skips most already-completed work but may retry some records. If your target table has unique constraints, use the `--on-conflict` option to handle duplicates—for example, `DO NOTHING` to skip them or `DO UPDATE` to upsert.

Command-line options

The Aurora DSQL Loader supports the following command-line options:

`--endpoint`

(Required) The Aurora DSQL cluster endpoint. Example: *`cluster-id`*`.dsql.`*`region`*`.on.aws`

`--source-uri`

(Required) The path to the data file. Can be a local file path or an S3 URI (for example, `s3://`*`bucket-name`*`/`*`file.parquet`*).

`--table`

(Required) The name of the target table in your Aurora DSQL database.

`--if-not-exists`

(Optional) Automatically create the target table if it does not exist. The loader infers the schema from the data.

`--dry-run`

(Optional) Validate the configuration and data without actually loading it into the database.

`--resume-job-id`

(Optional) Resume a previously interrupted load operation using the specified job ID.

`--manifest-dir`

(Optional) Directory for storing job state and manifests, used for job resumption.

`--on-conflict`

(Optional) Specifies how to handle conflicts when inserting rows that violate unique constraints on the target table. Valid values are `error` (return an error), `do-nothing` (skip duplicate rows), or `do-update` (update existing rows with new values).

For a complete list of options and additional configuration parameters, run:

```
aurora-dsql-loader load --help
```

Best practices

- **Use dry-run for validation** – Always test your load configuration with `--dry-run` before loading data into production tables.
- **Define unique constraints for resumption** – If you need to resume interrupted loads, define unique constraints on your target tables and use the `--on-conflict` option to handle already-loaded records.
- **Use Parquet for large datasets** – Parquet's columnar format typically provides better compression and faster loading for large datasets compared to CSV or TSV.
- **Preserve manifest directories** – Keep the manifest directory for load jobs until you confirm the load completed successfully, enabling resumption if needed.

- **Pre-create tables when possible** – Define the target table with explicit column data types and primary keys before loading data. Pre-created schemas give you control over type precision and indexing, which typically results in better query performance compared to auto-inferred schemas.

Troubleshooting

Authentication errors

Verify your AWS credentials are configured correctly and that your IAM identity has the required `dsql:DbConnect` or `dsql:DbConnectAdmin` permissions on the target cluster.

S3 access errors

Ensure your IAM identity has appropriate S3 read permissions for the source bucket and objects.

Schema inference errors

When using `--if-not-exists`, ensure your data file has consistent column types. Mixed types in a column may cause schema inference to fail.

Duplicate key errors on resume

If you encounter duplicate key errors when resuming a load, add unique constraints to your target table so the loader can use `ON CONFLICT DO NOTHING` to skip already-loaded records.

For additional troubleshooting information, see the Aurora DSQL Loader GitHub repository.

Migration pathways

The following sections describe how to migrate data from common source systems into Aurora DSQL.

Migrating from PostgreSQL

To migrate data from an existing PostgreSQL database to Aurora DSQL:

1. Export your data from PostgreSQL to CSV or Parquet format. You can use the PostgreSQL `COPY` command to export each table:

```
COPY my_table TO '/path/to/my_table.csv' WITH (FORMAT csv, HEADER true);
```

2. Create the target table in Aurora DSQL. You can either create the schema manually or use the loader's `--if-not-exists` flag to infer the schema from your data.
3. Load the exported data using the Aurora DSQL Loader:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

Tip

For large migrations, consider exporting to Parquet format for better compression and faster loading. Tools like DuckDB can convert CSV files to Parquet efficiently.

Migrating from MySQL

To migrate data from MySQL to Aurora DSQL:

1. Export your data from MySQL to CSV format using `SELECT INTO OUTFILE` or a tool like **mysqldump** with the `--tab` option:

```
SELECT * FROM my_table
INTO OUTFILE '/path/to/my_table.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

2. Create the target table in Aurora DSQL with appropriate PostgreSQL-compatible data types.
3. Load the exported data using the Aurora DSQL Loader:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

> **Note**
>
> MySQL and PostgreSQL have different data type systems. Review your schema and adjust data types as needed when creating tables in Aurora DSQL.

Loading from Amazon S3

If your data is already in Amazon S3, you can load it directly without downloading to your local system. The Aurora DSQL Loader supports S3 URIs natively:

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/path/to/data.parquet \
  --table my_table
```

Ensure your IAM identity has `s3:GetObject` permission on the source objects.

Using PostgreSQL \copy

If you are already connected to Aurora DSQL through a `psql` session that handles IAM authentication, you can use the client-side `\copy` meta-command to load data from your local file system. Unlike the server-side `COPY` statement, `\copy` reads the file on the client machine and streams the data over the existing connection, so no server-side file access is required. This approach works well for simple, single-threaded loads.

Example Loading a CSV file with \copy

```
\copy my_table FROM '/path/to/data.csv' WITH (FORMAT csv, HEADER true);
```

When using `\copy` directly, you are responsible for:

- Managing parallelization if loading multiple files or large datasets
- Handling connection management and authentication token refresh
- Implementing retry logic for failed operations

Best practices for INSERT transactions

When using `INSERT` statements to load data into Aurora DSQL, follow these practices to improve throughput and reliability:

- **Batch rows into multi-row INSERTs** – Group multiple rows into a single `INSERT` statement to reduce round trips. For example, `INSERT INTO my_table VALUES (1, 'a'), (2, 'b'), (3, 'c')` is more efficient than three separate statements.
- **Use parameterized queries** – Use prepared statements with parameter binding instead of string concatenation. This avoids SQL injection risks and allows the database to reuse query plans.
- **Keep transactions small** – Aurora DSQL uses optimistic concurrency control, so large transactions that touch many rows are more likely to encounter conflicts. Aim for transactions of a few hundred rows rather than thousands.
- **Implement retry logic** – Transient errors such as optimistic concurrency control (OCC) conflicts are expected in a distributed system. Implement exponential backoff with retry for failed transactions.
- **Parallelize across connections** – Open multiple connections and distribute inserts across them. Each connection can process a different subset of data concurrently.

For most use cases, the Aurora DSQL Loader provides a simpler and more robust approach to data loading.

Additional resources

- Aurora DSQL Loader on GitHub – Source code, documentation, and issue tracking
- Generating an authentication token in Amazon Aurora DSQL – Learn about IAM authentication tokens for Aurora DSQL
- Accessing Aurora DSQL with PostgreSQL-compatible clients – Connect to Aurora DSQL using various clients and tools

Generative AI for Aurora DSQL

This section provides detailed instructions for how to use Generative AI tools with Aurora DSQL

AWS Labs Aurora DSQL MCP Server

An AWS Labs Model Context Protocol (MCP) server for Aurora DSQL

Features

- Converting human-readable questions and commands into structured Postgres-compatible SQL queries and executing them against the configured Aurora DSQL database.
- Read-only by default, transactions enabled with `--allow-writes`
- Connection reuse between requests for improved performance
- Built-in access to Aurora DSQL documentation, search, and best practice recommendations
- SQL validation for Aurora DSQL compatibility with optional auto-fix

Available Tools

Database Operations

- **readonly_query** - Execute read-only SQL queries against your DSQL cluster
- **transact** - Execute write operations in a transaction (requires `--allow-writes`)
- **get_schema** - Retrieve table schema information

Documentation and Recommendations

- **dsql_search_documentation** - Search Aurora DSQL documentation
  - Parameters: `search_phrase` (required), `limit` (optional)
- **dsql_read_documentation** - Read specific DSQL documentation pages
  - Parameters: `url` (required), `start_index` (optional), `max_length` (optional)
- **dsql_recommend** - Get recommendations for DSQL best practices
  - Parameters: `url` (required)

SQL Validation

- **dsql_lint** - Validate SQL for Aurora DSQL compatibility and optionally auto-fix issues
  - Parameters: `sql` (required), `fix` (optional)

Prerequisites

1. An AWS account with an Aurora DSQL Cluster
2. This MCP server can only be run locally on the same host as your LLM client.
3. Set up AWS credentials with access to AWS services
   - You need an AWS account with a role including these permissions:
     - `dsql:DbConnectAdmin` - Connect to DSQL clusters as the admin user
     - `dsql:DbConnect` - Connect to DSQL clusters with custom database roles (only needed if using non-admin users)
   - Configure AWS credentials with `aws configure` or environment variables

Installation

For most tools, updating the configuration by following the Default Installation instructions should be sufficient.

Separate instructions are outlined for Claude Code and Codex.

Default Installation: Updating the Relevant MCP Config File

Using uv

1. Install uv from Astral or the GitHub README
2. Install Python using `uv python install 3.10`

Configure the MCP server in your MCP client configuration (Finding the MCP Config File)

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "command": "uvx",
      "args": [
        "awslabs.aurora-dsql-mcp-server@latest",
        "--cluster_endpoint",
        "[your dsql cluster endpoint, e.g. abcdefghijklmnopqrst234567.dsql.us-
east-1.on.aws]",
        "--region",
        "[your dsql cluster region, e.g. us-east-1]",
        "--database_user",
```

```
        "[your dsql username, e.g. admin]",
        "--profile",
        "[your aws profile, e.g. default]"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR"
      },
      "disabled": false,
      "autoApprove": []
    }
  }
}
```

Windows Installation

For Windows users, the MCP server configuration format is slightly different:

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "disabled": false,
      "timeout": 60,
      "type": "stdio",
      "command": "uv",
      "args": [
        "tool",
        "run",
        "--from",
        "awslabs.aurora-dsql-mcp-server@latest",
        "awslabs.aurora-dsql-mcp-server.exe"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR",
        "AWS_PROFILE": "your-aws-profile",
        "AWS_REGION": "us-east-1"
      }
    }
  }
}
```

Finding the MCP Client Configuration File

For some of the most common Agentic development tools, you can find your MCP client configurations at the following file paths:

- Kiro:
  - User Config: `~/.kiro/settings/mcp.json`
  - Workspace Config: `/path/to/workspace/.kiro/settings/mcp.json`
- Claude Code: Refer to Claude Code Installation for detailed setup help
  - User Config: `~/.claude.json` in `"mcpServers"`
  - Project Config: `/path/to/project/.mcp.json`
  - Local Config: `~/.claude.json` in `"projects" -> "path/to/project" -> "mcpServers"`
- Cursor:
  - Global: `~/.cursor/mcp.json`
  - Project: `/path/to/project/.cursor/mcp.json`
- Codex: `~/.codex/config.toml`
  - Each MCP server is configured with a `[mcp_servers.<server-name>]` table in the config file. Refer to the Custom Codex Installation Instructions
- Warp:
  - File Editing: `~/.warp/mcp_settings.json`
  - Application Editor: `Settings > AI > Manage MCP Servers` and paste json
- Amazon Q Developer CLI: `~/.aws/amazonq/mcp.json`
- Cline: Usually a nested VS Code path - `~/.vscode-server/path/to/cline_mcp_settings.json`

Claude Code

Prerequisites

Important: MCP server management is only available through the Claude Code CLI terminal experience, not the VS Code native panel mode.

Install the Claude Code CLI first by following Claude's native installation recommended process.

Choosing the Right Scope

Claude Code offers 3 different scopes: local (default), project, and user and details which scope to choose based on credential sensitivity and need to share. Refer to the Claude Code documentation on MCP Installation Scopes for more details.

1. **Local-scoped** servers represent the default configuration level and are stored in `~/.claude.json` under your project's path. They're **both** private to you and only accessible within the current project directory. This is the default `scope` when creating MCP servers.
2. **Project-scoped** servers **enable team collaboration** while still only being accessible in a project directory. Project-scoped servers add a `.mcp.json` file at your project's root directory. This file is designed to be checked into version control, ensuring all team members have access to the same MCP tools and services. When you add a project-scoped server, Claude Code automatically creates or updates this file with the appropriate configuration structure.
3. **User-scoped** servers are stored in `~/.claude.json` and **provide cross-project accessibility**, making them available across all projects on your machine while remaining **private to your user account.**

Using the Claude CLI (recommended)

Using an interactive `claude` CLI session enables an improved troubleshooting experience, so this is the recommended path.

```
claude mcp add amazon-aurora-dsql \
  --scope [one of local, project, or user] \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

Troubleshooting: Using Claude Code with Bedrock on a different AWS Account

If you've configured Claude Code with a Bedrock AWS account or profile that is distinct from the profile needed to connect to your dsql cluster, you'll need to provide additional environment arguments:

```
  --env AWS_PROFILE="[dsql profile, eg. default]" \
  --env AWS_REGION="[dsql cluster region, eg. us-east-1]" \
```

Direct Modification in the Configuration File

Claude Code Requires alphanumeric naming, so we recommend naming your server: `aurora-dsql-mcp-server`.

Local-Scope

Update `~/.claude.json` within the project-specific `mcpServers` field:

```
{
  "projects": {
    "/path/to/project": {
      "mcpServers": {}
    }
  }
}
```

Project-Scope

Update `/path/to/project/root/.mcp.json` in the `mcpServers` field:

```
{
  "mcpServers": {}
}
```

User-Scope

Update `~/.claude.json` within the project-specific `mcpServers` field:

```
{
  "mcpServers": {}
}
```

Codex

Option 1: Codex CLI

If you have the Codex CLI installed, you can use the codex mcp command to configure your MCP servers.

```
codex mcp add amazon-aurora-dsql \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

Option 2: config.toml

For more fine grained control over MCP server options, you can manually edit the `~/.codex/config.toml` configuration file. Each MCP server is configured with a `[mcp_servers.<server-name>]` table in the config file.

```
[mcp_servers.amazon-aurora-dsql]
command = "uvx"
args = [
  "awslabs.aurora-dsql-mcp-server@latest",
  "--cluster_endpoint", "<DSQL_CLUSTER_ID>.dsql.<AWS_REGION>.on.aws",
  "--region", "<AWS_REGION>",
  "--database_user", "<DATABASE_USERNAME>"
]

[mcp_servers.amazon-aurora-dsql.env]
FASTMCP_LOG_LEVEL = "ERROR"
```

Verifying Installation

For Amazon Q Developer CLI, Kiro CLI, Claude CLI/TUI, or Codex CLI/TUI, run `/mcp` to see the status of the MCP server.

For the Kiro IDE, you can also navigate to the Kiro Panel's `MCP SERVERS` tab which shows all configured MCP servers and their connection status indicators.

Server Configuration Options

`--allow-writes`

By default, the dsql mcp server does not allow write operations ("read-only mode"). Any invocations of transact tool will fail in this mode. To use transact tool, allow writes by passing `--allow-writes` parameter.

We recommend using least-privilege access when connecting to DSQL. For example, users should use a role that is read-only when possible. The read-only mode has a best-effort client-side enforcement to reject mutations.

`--cluster_endpoint`

This is mandatory parameter to specify the cluster to connect to. This should be the full endpoint of your cluster, e.g., `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws`

`--database_user`

This is a mandatory parameter to specify the user to connect as. For example `admin`, or `my_user`. Note that the AWS credentials you are using must have permission to login as that user. For more information on setting up and using database roles in DSQL, see Using database roles with IAM roles.

`--profile`

You can specify the aws profile to use for your credentials. Note that this is not supported for docker installation.

Using the `AWS_PROFILE` environment variable in your MCP configuration is also supported:

```
"env": {
  "AWS_PROFILE": "your-aws-profile"
}
```

If neither is provided, the MCP server defaults to using the "default" profile in your AWS configuration file.

`--region`

This is a mandatory parameter to specify the region of your DSQL database.

`--knowledge-server`

Optional parameter to specify the remote MCP server endpoint for DSQL knowledge tools (documentation search, reading, and recommendations). By default it is pre-configured.

Example:

```
--knowledge-server https://custom-knowledge-server.example.com
```

Note: For security, only use trusted knowledge server endpoints. The server should be an HTTPS endpoint.

`--knowledge-timeout`

Optional parameter to specify the timeout in seconds for requests to the knowledge server.

Default: `30.0`

Example:

```
--knowledge-timeout 60.0
```

Increase this value if you experience timeouts when accessing documentation on slow networks.

Aurora DSQL steering: Plugins, skills, and powers

Use the `databases-on-aws` plugin to give your AI coding agent guidance for Aurora DSQL development. The plugin includes the `dsql` skill, which helps with schema design, queries, migrations, and troubleshooting. Install the plugin in Claude Code, Codex, or Cursor. You can also use the Kiro Power or install the skill on its own.

Overview

Choose the format that your AI tool supports:

- **Plugins** bundle skills with tools and hooks. The `databases-on-aws` plugin is available from Agent Plugins for AWS on the GitHub website.
- **Skills** give an agent instructions and reference files to load as needed. Use the Skills CLI to install the `dsql` skill on its own.

- **Powers** package guidance and Model Context Protocol (MCP) server configuration for Kiro. The Aurora DSQL power helps you set up tools and develop applications in the Kiro IDE.

Why use steering

Steering gives your agent Aurora DSQL guidance that you don't need to repeat in each prompt. After setup, start a new chat. Ask the agent to review an Aurora DSQL schema or plan a migration. The agent can load guidance for the task. You can also ask it to use the Aurora DSQL skill or power by name.

Recommended setup paths

Choose the setup path for your AI tool:

- the section called “Claude Code plugin”
- the section called “Codex plugin”
- the section called “Cursor plugin”
- the section called “Kiro Power”
- the section called “Other AI tools: Standalone skill”

Claude Code plugin

Install the `databases-on-aws` plugin in Claude Code.

Setup

Run these commands in Claude Code to add the marketplace and install the plugin:

```
/plugin marketplace add awslabs/agent-plugins
/plugin install databases-on-aws@agent-plugins-for-aws
```

For more information, see Claude Code in *Agent Plugins for AWS* on the GitHub website.

Codex plugin

Install the `databases-on-aws` plugin from the repository's marketplace:

1. Use Git to clone the Agent Plugins for AWS repository:

```
git clone https://github.com/awslabs/agent-plugins.git
```

2. Open the cloned `agent-plugins` directory in Codex. Restart Codex to discover the marketplace.
3. Open the plugin directory, choose the **Agent Plugins for AWS** marketplace, and install `databases-on-aws`.
4. Start a new chat to use the plugin's skills and tools.

For more information, see Codex in *Agent Plugins for AWS* on the GitHub website. To install only the skill, see the section called “Other AI tools: Standalone skill”.

Cursor plugin

Install AWS Databases from the Cursor Marketplace. This is the `databases-on-aws` plugin, which includes the Aurora DSQL skill.

For more information, see Cursor in *Agent Plugins for AWS* on the GitHub website.

Kiro Power

Use the Aurora DSQL power for guidance and MCP tools in the Kiro IDE.

Setup

1. Open the Aurora DSQL power from the Kiro Powers Registry. The link opens the power in the Kiro IDE.
2. Choose **Try power** for guided MCP server setup, or open a new Kiro chat and describe an Aurora DSQL task.
3. (Optional) To connect to an existing cluster, add your cluster details to the power's MCP configuration. Test the connection. For more information, see the section called “AWS Labs Aurora DSQL MCP Server”.

Other AI tools: Standalone skill

Use the Skills CLI to install the `dsql` skill in other supported AI tools. You can also choose this path when you want only the skill. It installs instructions and reference files, without the plugin's MCP servers or hooks. To set up database tools, see the section called “AWS Labs Aurora DSQL MCP Server”.

Skills CLI

You can install the skill for one or more coding agents, including Codex. For more information, see CLI Reference on the skills.sh website.

Setup

Install Node.js 22.20.0 or later and npm. Run this command to install the Aurora DSQL skill:

```
npx skills add https://github.com/awslabs/agent-plugins --skill dsql
```

Follow the prompts:

- Choose the coding agents that use the skill.
- Choose project scope for the current directory, or global scope for all projects.
- Use symlinks to share one copy of the skill, or keep a copy for each agent.

Managing skills

To list skills in the current project, run this command:

```
npx skills list
```

To list only global skills, run this command:

```
npx skills list --global
```

To update the Aurora DSQL skill in both the current project and global scope, run this command. The command defaults to both scopes.

```
npx skills update dsql
```

To update only the current project's Aurora DSQL skill, run this command:

```
npx skills update dsql --project
```

To update only the global Aurora DSQL skill, run this command:

```
npx skills update dsql --global
```

Get started with the Aurora DSQL Query Editor

With the Aurora DSQL Query Editor, you can securely connect to your Aurora DSQL clusters and run SQL queries directly from the AWS Management Console without installing or configuring external clients. It provides an intuitive workspace with built-in syntax highlighting, auto-completion, and intelligent code assistance. You can quickly explore schema objects, develop and run SQL queries, and view results, all within a single interface.

This topic walks you through the steps to connect to a cluster, run queries, view results, and explore advanced capabilities such as execution plans.

> **Note**
>
> The Query Editor is available in all Regions where Aurora DSQL is supported. For more details about regional availability, see AWS Regional Services.

Prerequisites

Before you begin, make sure that you meet the following requirements:

- You have at least one Aurora DSQL cluster available. For more details about creating clusters, see Step 1: Create an Aurora DSQL single-Region cluster.
- Your cluster endpoint is publicly accessible. The Query Editor doesn't support clusters that have public access blocked by resource-based policies or clusters managed through VPC endpoints. For more details about access restrictions, see Blocking public access with resource-based policies in Aurora DSQL and Managing and connecting to Amazon Aurora DSQL clusters using AWS PrivateLink.
- Your IAM user or role has the required permissions to access and connect to the cluster. For more details about permissions, see Using database roles and IAM authentication.

Working with the Query Editor

Open the Query Editor

To open the Query Editor

1. Open the Aurora DSQL console.

2. In the navigation pane, choose **Query Editor**.

Alternatively, from the **Clusters** page, select the cluster you want to query and choose **Connect with Query editor** to launch the editor directly.

> **Note**
>
> Work and connection state aren't saved. If you navigate away from the Aurora DSQL console, close the browser tab, or sign out, your connections, query text, and results are lost.

Connect to a cluster

To connect to a cluster

1. If no cluster connection exists, the editor displays **No cluster has been connected**. Choose **Connect** or select **+** (Add) in the **Cluster Explorer** pane to connect to an existing cluster.
2. (Optional) Connect to multiple clusters or to the same cluster by using different roles.

Explore cluster objects

The Cluster Explorer displays all available cluster connections and lets you browse objects such as databases, schemas, tables, and views. It also provides common actions like **Refresh**, **Create table**, and other context-specific options.

Run queries

To run a query

1. In the query editor tab pane, enter your SQL statement. For example:

```
SELECT * FROM public.orders LIMIT 10;
```

2. Verify the **Active Cluster Context** displayed on the upper right of the query tab. This indicates the cluster connection associated with the current query tab.

3. (Optional) Use the **connection** dropdown to review all available connections or switch to a different cluster. Changing the connection updates where your queries in that tab are executed.
4. Choose **Run** to run the query.

> **Note**
>
> Each query can return up to 10,000 rows in the results pane. For larger datasets, refine your query with filters or limits.

Review results and execution plans

After the query runs, review the output in the **Results panel** at the bottom of the editor. By default, each query execution displays the **Results (Table)** tab, showing tabular query output.

To get the query execution plan, run `EXPLAIN ANALYZE` or `EXPLAIN ANALYZE VERBOSE` to get additional insights into query performance. For more details about execution plans, see Reading Aurora DSQL EXPLAIN plans.

> **Tip**
>
> The `EXPLAIN ANALYZE VERBOSE` command surfaces DPU usage estimates including Compute, Read, Write, and Total DPU values, providing immediate visibility into the resources consumed by individual SQL statements.

Query Editors: Using JupyterLab with Aurora DSQL

This guide provides step-by-step instructions on how to connect and query Amazon Aurora DSQL using JupyterLab with Python. JupyterLab is a popular interactive computing environment that combines code, text, and visualizations in a single document. It's widely used for data science and research applications.

The instructions below will cover the basics of Aurora DSQL usage in both a local installation of JupyterLab as well as using Amazon SageMaker AI, a fully-managed machine learning service that provides a hosted environment with a UI for data workflows.

Getting started

Requirements

- An Aurora DSQL cluster
- AWS credentials configured (local installation only)
- Python version 3.9 or greater (local installation only)

Using local JupyterLab

To get started with JupyterLab, users must first install the application using Python's **pip**:

```
pip install jupyterlab
```

JupyterLab can then be opened by running **`jupyter lab`**. This will open the JupyterLab application at localhost:8888, accessible in a browser. Ensure you have AWS credentials configured in your local environment before proceeding.

Using Amazon SageMaker AI

In the AWS console, proceed to the Amazon SageMaker AI console page and then to the **Notebooks** section under **Applications and IDEs**. From there you can select **Create notebook instance** to begin creating a SageMaker environment. Select an instance type and platform before clicking **Create notebook instance**.

See Amazon SageMaker AI setup documentation for more information on setup and instance options.

> **Note**
>
> Warning: Using Amazon SageMaker AI may result in charges to your AWS account.

Once the SageMaker instance becomes active, you can open it from the **Notebook instances** section with **Open JupyterLab**. Before getting started with Aurora DSQL in your notebook you must provide access to your DSQL cluster in the SageMaker instance's IAM role. The simplest way to do so is to follow the link to the IAM role in the notebook instance page. From there you can edit the Policies attached to your SageMaker IAM role. See Authentication and authorization for more information on configuring an IAM policy to allow access to Aurora DSQL.

Connecting to Aurora DSQL using JupyterLab

After you have set up a JupyterLab instance, the steps to connect to Aurora DSQL are the same locally and in SageMaker AI. Create an empty Python 3 notebook, in which you can add cells with Python code.

In a Python cell, download the Amazon root certificate from the official trust store:

```
import urllib.request
urllib.request.urlretrieve('https://www.amazontrust.com/repository/AmazonRootCA1.pem',
 'root.pem')
```

To connect to Aurora DSQL, first install the Aurora DSQL Connector for Python and the Psycopg driver in a Python cell, and then import it:

```
pip install aurora_dsql_python_connector psycopg
```

```
import aurora_dsql_psycopg as dsql
```

With the connector imported, you can then create a DSQL configuration and connect. The Aurora DSQL Python Connector will automatically handle creation of an authentication token on each connection.

```
config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
    'region': "us-east-1",
    'user': "admin"
    }

conn = dsql.connect(**config)
```

Upon running your code you should now have a Psycopg connection to Aurora DSQL. You can then run queries using the Psycopg cursor and providing your SQL query. See the Psycopg documentation for more information on using Psycopg with a Postgres-compatible database. This query will result in a list of tuples in `results_list`.

```
with conn:
    with conn.cursor() as cur:
```

```
        cur.execute("SELECT * FROM table")
        results_list = cur.fetchall()
```

You can then use Python frameworks like Pandas to analyze or visualize your query results, for example:

```
pip install pandas

import pandas as pd

df = pd.DataFrame(tuples_list)
print(df)
print(f"Total records: {len(df)}")
```

Example notebook

A sample notebook using Aurora DSQL is available in the Aurora DSQL samples repository.

Further reading

Amazon SageMaker AI setup documentation

Aurora DSQL Connector for Python

Pandas documentation

Backup and restore for Amazon Aurora DSQL

Amazon Aurora DSQL helps you meet your regulatory compliance and business continuity requirements through integration with AWS Backup, a fully managed data protection service that makes it easy to centralize and automate backups across AWS services, in the cloud, and on premises. The service streamlines backup creation, management, and restoration for both single-Region and multi-Region Aurora DSQL clusters.

Key features include the following:

- Centralized backup management through the AWS Management Console, SDK, or AWS CLI
- Full cluster backups
- Automated backup schedules and retention policies
- Cross-Region and cross-account capabilities
- WORM (write-once, read-many) configuration for all the backups you store

For more information on the features of AWS Backup Vault Lock and an extensive list of available AWS Backup features for Aurora DSQL, see Vault lock benefits and AWS Backup feature availability in the *AWS Backup Developer Guide*.

Getting started with AWS Backup

AWS Backup creates complete copies of your Aurora DSQL clusters. You can get started using AWS Backup for Aurora DSQL by following the steps in Getting started with AWS Backup:

1. Create on-demand backups for immediate protection.
2. Establish backup plans for automated, scheduled backups.
3. Configure retention periods and cross-Region copying.
4. Set up monitoring and notifications for backup activities.

Restoring your backups

When you restore Aurora DSQL clusters, AWS Backup always creates new clusters to preserve your source data.

Restoring single-Region clusters

To restore an Aurora DSQL single-Region cluster, use the console: https://console.aws.amazon.com/backup or CLI to select the recovery point (backup) you wish to restore. Configure the settings for the new cluster that will be created from your backup. For detailed instructions, see Restore a single-Region Aurora DSQL cluster.

Restoring multi-Region clusters

Restoring an Aurora DSQL multi-Region cluster is supported through both the console: https://console.aws.amazon.com/backup and the AWS CLI. For detailed instructions, see Restore a multi-Region Aurora DSQL cluster.

To restore to a multi-Region Aurora DSQL cluster, you can use a backup taken in a single AWS Region. However, before you initiate the restore process, you must ensure there is an identical copy of your backup in all AWS Regions for your multi-Region clusters. If you don't yet have those copies, you must first copy the backup to another AWS Region that supports multi-Region clusters.

We recommend creating backup copies in key AWS Regions to enable robust disaster recovery options and meet compliance requirements. To view available AWS Regions for Aurora DSQL, see the section called “AWS Region availability”.

For detailed instructions on these steps, see Amazon Aurora DSQL restore documentation.

Monitoring and compliance

AWS Backup provides comprehensive visibility into backup and restore operations with the following resources.

- A centralized dashboard for tracking backup and restore jobs
- Integration with CloudWatch and CloudTrail.
- AWS Backup Audit Manager for compliance reporting and auditing.

See Logging Aurora DSQL operations using AWS CloudTrail to learn more about logging records of actions taken by a user, role, or an AWS service while using Aurora DSQL.

Additional resources

To learn more about AWS Backup features and and using it in tandem with Aurora DSQL, see the following resources:

- Managed policies for AWS Backup
- Amazon Aurora DSQL restore
- Supported services by AWS Region
- Encryption for backups in AWS Backup

By using AWS Backup for Aurora DSQL, you implement a robust, compliant, and automated backup strategy that protects your critical database resources while minimizing administrative overhead. Whether you manage a single cluster or a complex multi-Region deployment, AWS Backup provides the tools you need to ensure your data remains secure and recoverable.

Change data capture streams

For pricing information, see the Aurora DSQL pricing page.

Amazon Aurora DSQL change data capture (CDC) streams committed database changes in near real time directly to Amazon Kinesis Data Streams. Aurora DSQL delivers the final committed state of each changed row as a structured JSON record to a Kinesis data stream that you configure.

CDC is useful when you want to:

- **Keep downstream systems in sync** – Replicate changes to a search index, cache, data warehouse, or analytics system without batch jobs.
- **Build event-driven architectures** – Trigger workflows, notifications, or microservice actions in response to database changes.
- **Maintain an audit trail** – Capture the committed net effect of every transaction for compliance, debugging, or historical analysis.
- **Decouple producers from consumers** – Let the database focus on transactions while downstream systems process changes at their own pace.

How it works

Aurora DSQL reads committed transactions, formats the net row-level change as a structured JSON record, and delivers it to a Kinesis data stream that you configure. CDC automatically captures the committed effect of every `INSERT`, `UPDATE`, and `DELETE` across all user tables in the cluster. Apply filtering logic in your downstream apps by using the `source.schema` and `source.table` fields in each CDC record to focus on the tables or changes your app needs.

Aurora DSQL compacts the write set of each transaction before publishing CDC records and emits a single record for each row with a net change, reflecting the final state. For details, see Write-set compaction.

CDC streams are fully managed. Aurora DSQL manages all infrastructure required to capture change events, monitors stream health, and reports the status through the `GetStream` API operation and CloudWatch metrics.

CDC streams use a bring-your-own-target model. You create and manage the Kinesis data stream in your account, and Aurora DSQL assumes an IAM role that you configure to write CDC records on

your behalf. You're responsible for the target's capacity, encryption, and retention settings. For the latest supported targets, see the `TargetDefinition` parameter in CreateStream in the Amazon Aurora DSQL API Reference. For a complete list of CDC stream API operations, see the Amazon Aurora DSQL API Reference.

Topics on this page

- Ordering and delivery semantics
- Multi-Region CDC stream configuration
- Processing CDC records downstream

Related topics

- Getting started with CDC streams
- Configuring IAM
- Understanding CDC records
- Monitoring streams

Ordering and delivery semantics

Delivery guarantees

Aurora DSQL produces one CDC record for each row with a net change at commit time. For details about how Aurora DSQL compacts multiple operations on the same row, see Write-set compaction.

Aurora DSQL guarantees at-least-once delivery for every record it produces. Aurora DSQL can deliver the same record more than one time, so design your app to handle duplicates. For full and chunked main records with a `transaction` object, identify duplicate deliveries by comparing `source.txId` and `transaction.record_id`. For fragment records, compare `chunk_id` and `index`.

Ordering

CDC streams use `UNORDERED` mode. In practice, records arrive in approximate commit order because Aurora DSQL reads and publishes changes sequentially. However, Aurora DSQL doesn't guarantee strict ordering. Specifically:

- Aurora DSQL can deliver records from different transactions in any order.
- Records for the same primary key from different transactions can arrive out of commit order.
- Records from a single transaction can interleave with records from other transactions. Use the `source.txId` field to group records by transaction when your workflow requires it. For records that include a `transaction` object, you have received records for all rows with a net change in the transaction when the number of unique `transaction.record_id` values equals `transaction.total_records`. If the transaction includes a chunked main record, wait for and reassemble its fragments before processing the transaction.

Each CDC record includes a `source.ts_ns` field that contains the transaction commit timestamp in nanoseconds. Use this field to establish commit order on the receiving side.

Consumer strategies

Because records can arrive out of commit order and can appear more than one time, your app must account for both conditions.

> **Important**
>
> Define a primary key on all tables that participate in CDC. Without a primary key, your app can't reliably correlate deletes or out-of-order changes with the affected row. The `transaction` object identifies duplicate deliveries but doesn't identify the affected row.

Last-writer-wins (materialized views, caches)

Track the highest `source.ts_ns` value per primary key. Discard any record with a `source.ts_ns` less than or equal to the tracked value. This filters both duplicates and out-of-order records, keeping the most recent state for each key. When you process a delete (`op: "d"`), store a tombstone for the primary key that preserves the `source.ts_ns` value instead of removing the entry. The tombstone ensures that an insert or update with an earlier `source.ts_ns` that arrives after the delete doesn't incorrectly restore the row.

Every-change processing (audit logging, event sourcing)

For records that include a `transaction` object, group records by `source.txId` and remove duplicates by comparing `transaction.record_id`. You have received records for all rows

with a net change in the transaction when the number of unique record identifiers equals `transaction.total_records`. Reassemble any chunked records before processing the transaction. Use `source.ts_ns` as the commit-order key, but don't treat receipt of all changed rows as a watermark that all earlier transactions have arrived. CDC delivers only the final state of each row per committed transaction (after Write-set compaction). If your audit workflow requires a record for every SQL statement, perform each statement in its own transaction.

Multi-Region CDC stream configuration

A CDC stream is a regional resource. Each stream belongs to a single AWS Region and delivers changes to a Kinesis data stream in the same Region. On a multi-Region cluster, a CDC stream in any one Region captures committed writes from **all Regions** in the cluster. This means you only need one stream to capture every change, regardless of where the write originated. To deliver CDC records in more than one Region, create a separate stream in each Region. Each stream independently captures the full set of committed changes across the cluster.

All resources—the Aurora DSQL cluster, Kinesis data stream, IAM service role, and calling principal—must be in the same AWS account and Region.

Processing CDC records downstream

After CDC records arrive in your Kinesis data stream, you can process them directly or route them to other destinations by using AWS integration services. The following table summarizes common processing patterns.

Common processing patterns for CDC records

Pattern	How it works
Direct consumption	Read records from Kinesis by using the Amazon Kinesis Client Library (KCL), the AWS SDK, or a Kinesis Data Streams consumer. See Developing KCL consumers in the *Amazon Kinesis Data Streams Developer Guide*.
AWS Lambda	Configure a Lambda function as an event source for your Kinesis data stream to process each batch of CDC records as they arrive. See

Pattern	How it works
	Using AWS Lambda with Amazon Kinesis in the *AWS Lambda Developer Guide*.
Amazon Data Firehose	Deliver CDC records from Kinesis to Amazon S3, Amazon Redshift, Amazon OpenSearch Service, or other destinations for analytics and archival. See Sending data to a delivery stream in the *Amazon Data Firehose Developer Guide*.
Self-managed consumers	Run Apache Kafka Connect with the Kinesis source connector, Apache Flink, or other stream processing frameworks to transform and route records. For Apache Flink on AWS, see Configuring app input in the *Amazon Managed Service for Apache Flink Developer Guide*.

Each CDC record includes fields such as `source.schema`, `source.table`, and op that you can use to route and filter records in your processing logic. For the full record schema, see Understanding CDC records.

Getting started with CDC streams

This guide walks you through every step required to start streaming committed row-level changes from an Aurora DSQL cluster to an Amazon Kinesis data stream. By the end of this guide, you've created a working CDC pipeline and a Python script that reads and prints change records.

Prerequisites

Before you begin, confirm the following:

- You've created an Aurora DSQL cluster in `ACTIVE` status. If your cluster is idle, connect to it with any PostgreSQL-compatible client to wake it up before you create a CDC stream. `CreateStream` returns a validation error if the cluster isn't in `ACTIVE` status.
- Aurora DSQL requires all CDC resources—the cluster, Amazon Kinesis data stream, IAM service role, and calling principal—to be in the same AWS account.

- Your Amazon Kinesis data stream is in the same AWS Region as your Aurora DSQL cluster.
- You've installed and configured the AWS CLI with credentials that have permission to create IAM roles and Amazon Kinesis data streams.

Step 1: Create an Amazon Kinesis data stream

Create a Kinesis data stream in the same AWS account and Region as your Aurora DSQL cluster. CDC records are larger than the corresponding Aurora DSQL row because the JSON format includes column names, metadata, and encoding overhead.

Sizing the Kinesis data stream

Aurora DSQL CDC delivers the full row on every change. An update that touches a single column produces a record that contains every column in the row. Delete records are the exception—they include only the primary key columns.

Estimate average record size

Measure the average on-disk row size to understand the volume that CDC will produce and to anticipate oversized records. The following query returns the average tuple size in bytes for a table:

```
SELECT avg(pg_column_size(t.*)) FROM your_table t;
```

The CDC record envelope adds column names, metadata, and encoding overhead on top of the row size. For record payload examples, see Record payload. For how Aurora DSQL handles records that exceed the Kinesis record size limit, see Handling oversized records. For the full set of Kinesis service limits, see Amazon Kinesis Data Streams quotas and limits in the *Amazon Kinesis Data Streams Developer Guide*.

> **Important**
>
> When you create the Kinesis data stream, set the following:
>
> - `MaxRecordSizeInKiB` to `10240` (10 MiB). The default Kinesis maximum of 1 MiB isn't always large enough for Aurora DSQL CDC records. Any record that exceeds the configured Kinesis record size causes the CDC stream to become impaired with `KINESIS_OVERSIZE_RECORD`. Aurora DSQL splits oversized records into fragments that can approach 10 MiB each, so the Kinesis data stream needs to accept records of that size. For details, see Handling oversized records.

- `StreamMode` to `ON_DEMAND`. On-demand mode scales shard capacity automatically and protects you from under-provisioning during unexpected spikes. Kinesis can still return `WriteProvisionedThroughputExceeded` during sharp seconds-scale bursts as capacity scales up. Plan for brief throttling events.

Create CloudWatch alarms on `IncomingBytes` and `WriteProvisionedThroughputExceeded` in the `AWS/Kinesis` namespace. Kinesis throttling slows CDC delivery and increases replication lag. For Aurora DSQL-side metrics and alarm guidance, see Monitoring best practices.

The following example uses the AWS CLI. If your AWS CLI version doesn't support the `--max-record-size-in-ki-b` parameter, use an AWS SDK to call the Kinesis CreateStream operation.

```
aws kinesis create-stream \
  --stream-name my-cdc-stream \
  --stream-mode-details StreamMode=ON_DEMAND \
  --max-record-size-in-ki-b 10240 \
  --region region
```

Wait for the stream to become active:

```
aws kinesis describe-stream-summary \
  --stream-name my-cdc-stream \
  --region region \
  --query 'StreamDescriptionSummary.StreamStatus'
```

The command returns `"ACTIVE"` when the stream is ready.

Record the stream ARN from the output. You need it in the following steps. The ARN has the format `arn:aws:kinesis:region:account-id:stream/my-cdc-stream`.

Step 2: Create an IAM role for Aurora DSQL

Aurora DSQL assumes an IAM role to write CDC records to your Kinesis data stream. In this step, you create the role with a trust policy and attach a permissions policy. For a full explanation of each policy element, see Configuring IAM.

Create the trust policy file

Save the following JSON as `trust-policy.json`. Replace *`your-account-id`*, *`region`*, and *`cluster-id`* with your values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAccess",
            "Effect": "Allow",
            "Principal": {
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
                }
            }
        }
    ]
}
```

Create the role

Run the following command to create the IAM role:

```
aws iam create-role \
  --role-name dsql-cdc-role \
  --assume-role-policy-document file://trust-policy.json
```

Create the permissions policy file

Save the following JSON as `permissions-policy.json`. Replace the placeholder values with your Kinesis data stream ARN. The `KMSAccess` statement is only required if your Kinesis data stream uses an AWS KMS customer managed key, but you can include it preemptively so that adding a customer managed key later doesn't break your CDC stream. For a full explanation of each condition, see Service role permissions policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord",
                "kinesis:PutRecords"
            ],
            "Resource": "arn:aws:kinesis:region:your-account-id:stream/my-cdc-stream"
        },
        {
            "Sid": "KMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:GenerateDataKey"
            ],
            "Resource": "arn:aws:kms:*:*:key/*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "kinesis.region.amazonaws.com",
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:region:your-account-id:stream/my-cdc-stream",
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        }
    ]
}
```

Attach the permissions policy

Run the following command:

```
aws iam put-role-policy \
  --role-name dsql-cdc-role \
  --policy-name dsql-cdc-kinesis-access \
  --policy-document file://permissions-policy.json
```

Record the role ARN from the `create-role` output. The ARN has the format `arn:aws:iam::your-account-id:role/dsql-cdc-role`.

Step 3: Create the CDC stream

Use the AWS CLI to create a CDC stream that connects your Aurora DSQL cluster to the Kinesis data stream. Replace the placeholder values with the Kinesis stream ARN from Step 1, the IAM role ARN from Step 2, and your cluster identifier.

```
aws dsql create-stream \
  --cluster-identifier cluster-id \
  --target-definition '{"kinesis":{"streamArn":"kinesis-stream-arn","roleArn":"role-
arn"}}' \
  --ordering UNORDERED \
  --format JSON \
  --tags '{"Name":"my-cdc-stream"}' \
  --region region
```

The response includes a stream identifier and a status of `CREATING`. Stream creation typically takes one to three minutes.

Wait for the stream to become active

Poll the stream status until it reaches `ACTIVE`:

```
aws dsql get-stream \
  --cluster-identifier cluster-id \
  --stream-identifier stream-id \
  --region region \
  --query 'status'
```

You can also use the `StreamActive` waiter in the AWS SDKs to poll automatically.

After the stream reaches `ACTIVE`, Aurora DSQL begins delivering committed row-level changes to your Kinesis data stream.

> **Note**
>
> Each Aurora DSQL cluster has a maximum number of CDC streams. If you reach this limit, `CreateStream` returns a `ServiceQuotaExceededException`. For the default limit, see Quotas and limits.

Step 4: Verify that records are flowing

Insert a row into a table on your Aurora DSQL cluster. For example:

```
CREATE TABLE IF NOT EXISTS test_cdc (
    id INT PRIMARY KEY,
    message TEXT
);

INSERT INTO test_cdc VALUES (1, 'hello cdc');
```

Read from the Kinesis data stream to verify that the CDC record arrived. Because Aurora DSQL uses randomized partition keys, the record can land on any shard. The following script reads each shard on the stream up to the latest record, and requires `jq`. Replace *`my-cdc-stream`* and *`region`* with your values. CDC records can take a few seconds to arrive because of replication lag—if the output doesn't include your record, wait a moment and re-run the script.

```
for SHARD_ID in $(aws kinesis list-shards \
  --stream-name my-cdc-stream \
  --region region \
  --query 'Shards[].ShardId' --output text); do

  SHARD_ITERATOR=$(aws kinesis get-shard-iterator \
    --stream-name my-cdc-stream \
    --shard-id "$SHARD_ID" \
    --shard-iterator-type TRIM_HORIZON \
    --region region \
    --query 'ShardIterator' --output text)

  while [ -n "$SHARD_ITERATOR" ]; do
    RESPONSE=$(aws kinesis get-records \
      --shard-iterator "$SHARD_ITERATOR" \
      --region region \
      --output json) || break
    echo "$RESPONSE" | jq -r '.Records[].Data'
    [ "$(echo "$RESPONSE" | jq '.MillisBehindLatest')" = 0 ] && break
    SHARD_ITERATOR=$(echo "$RESPONSE" | jq -r '.NextShardIterator // empty')
  done
done
```

Each record's `Data` field contains a JSON payload. When you use the AWS CLI, the payload is Base64-encoded in the response. When you use the `boto3` SDK, the SDK decodes it automatically. The decoded JSON looks like the following:

```
{
    "type": "full",
    "op": "c",
    "before": null,
    "after": {"id": 1, "message": "hello cdc"},
    "source": {
        "version": "1.1",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
        "table": "test_cdc",
        "db": "postgres",
        "cluster": "cluster-id"
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291,
    "transaction": {
        "total_records": 1,
        "record_id": "8f14e45fceea167a5a36dedd4bea2543"
    }
}
```

For descriptions of the current fields, see Understanding CDC records.

Step 5: Consume records with a Python script

The following Python script reads CDC records from a Kinesis data stream and prints each change event. The script uses the `boto3` Amazon Kinesis client to iterate over shards and decode each record. Because Aurora DSQL CDC uses at-least-once delivery, the script might print the same record more than one time.

```
"""
Read CDC records from an Amazon Kinesis data stream.

Usage:
    pip install boto3
    python consume_cdc.py --stream-name my-cdc-stream --region us-east-1
```

```
"""
from __future__ import annotations

import argparse
import json

import boto3


def consume_cdc(stream_name: str, region: str) -> None:
    kinesis = boto3.client("kinesis", region_name=region)

    # List all shards (paginate if the stream has many shards)
    shard_ids: list[str] = []
    paginator = kinesis.get_paginator("list_shards")
    for page in paginator.paginate(StreamName=stream_name):
        shard_ids.extend(s["ShardId"] for s in page["Shards"])
    print(f"Reading from {stream_name} ({len(shard_ids)} shard(s))")

    for shard_id in shard_ids:
        iterator_response = kinesis.get_shard_iterator(
            StreamName=stream_name,
            ShardId=shard_id,
            ShardIteratorType="TRIM_HORIZON",
        )
        shard_iterator = iterator_response["ShardIterator"]

        while shard_iterator:
            records_response = kinesis.get_records(
                ShardIterator=shard_iterator, Limit=100
            )
            shard_iterator = records_response.get("NextShardIterator")

            for record in records_response["Records"]:
                # boto3 decodes Base64 automatically; record["Data"] is bytes.
                payload = json.loads(record["Data"])

                # A record's "type" field identifies its structure.
                # "full": inlined record with before/after values.
                # "chunked": main record that references fragments for a split image.
                # "fragment": one piece of a chunked image; reassemble in production
 code.
                # For details, see cdc-record-format.html#cdc-oversized-records.
                record_type = payload.get("type", "full")
```

```
                if record_type == "fragment":
                    print(f"[FRAGMENT] chunk_id={payload['chunk_id']}
 index={payload['index']}")
                    continue
                if record_type not in {"full", "chunked"}:
                    print(f"[UNKNOWN RECORD TYPE] type={record_type}")
                    continue

                source = payload["source"]
                op = payload["op"]
                ts_ns = source["ts_ns"]
                tx_id = source["txId"]
                table = f"{source['schema']}.{source['table']}"

                op_labels = {"c": "INSERT", "u": "UPDATE", "d": "DELETE"}
                print(
                    f"[{op_labels.get(op, op)}] {table} "
                    f"txId={tx_id} ts_ns={ts_ns} type={record_type}"
                )
                if payload.get("after"):
                    print(f"  after:  {json.dumps(payload['after'])}")
                if payload.get("before"):
                    print(f"  before: {json.dumps(payload['before'])}")
                if record_type == "chunked":
                    print(f"  chunked: {json.dumps(payload['chunked'])}")

            if not records_response["Records"]:
                break  # No more records in this shard


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Consume DSQL CDC records from Kinesis"
    )
    parser.add_argument("--stream-name", required=True, help="Kinesis stream name")
    parser.add_argument("--region", required=True, help="AWS Region")
    args = parser.parse_args()
    consume_cdc(args.stream_name, args.region)
```

Run the script:

```
pip install boto3
python consume_cdc.py \
```

```
  --stream-name my-cdc-stream \
  --region region
```

The script prints each change event as it arrives. You see output similar to the following:

```
Reading from my-cdc-stream (4 shard(s))
[INSERT] public.test_cdc txId=ffthunp5stx6ffs2vyfqoatmfu ts_ns=1705318200000000000
 type=full
  after:  {"id": 1, "message": "hello cdc"}
```

Adding last-writer-wins deduplication

Because Aurora DSQL CDC uses at-least-once delivery, production apps should deduplicate and order records. The following code example shows a high-water-mark approach: for each primary key, it tracks the highest `source.ts_ns` seen so far and discards any record with an equal or earlier timestamp. Set `PK_COLUMNS` to the primary key column names of the table you're processing. For strategies that handle multiple tables or deletes, see Consumer strategies.

```
# Set PK_COLUMNS to the primary key column(s) of your table.
PK_COLUMNS = ["id"]

# Maps each primary key value to the highest ts_ns seen for that key.
high_water: dict[tuple, int] = {}

def process_record(payload: dict) -> bool:
    """Return True if the record is new, False if it's a duplicate or stale.

    Skip fragment records; reassemble them into a full image before calling this.
    For a chunked record, put each reassembled image in the matching
    payload["before"] or payload["after"] field.
    """
    if payload.get("type") == "fragment":
        return False  # Fragments are reassembled upstream, not deduplicated here.

    source = payload["source"]
    ts_ns = source["ts_ns"]

    # For inserts/updates the row is in "after"; for deletes it's in "before".
    row = payload.get("after") or payload.get("before")
    if row is None:
        raise ValueError("Record has no row image")
    pk = tuple(row[col] for col in PK_COLUMNS)
```

```
    prev_ts = high_water.get(pk, -1)
    if ts_ns <= prev_ts:
        return False  # Duplicate or out-of-order record

    high_water[pk] = ts_ns
    return True
```

Managing CDC streams

Listing streams

To list all CDC streams for a cluster, use the `ListStreams` operation:

```
aws dsql list-streams \
  --cluster-identifier cluster-id \
  --region region
```

Deleting a stream

To delete a CDC stream, run the following command:

```
aws dsql delete-stream \
  --cluster-identifier cluster-id \
  --stream-identifier stream-id \
  --region region
```

You can use the `StreamNotExists` waiter to poll `GetStream` until a `ResourceNotFoundException` is returned, indicating that Aurora DSQL has fully deleted the stream.

Configuring IAM

CDC streams require two separate sets of IAM permissions:

- **Caller permissions** – The IAM principal that calls the CDC stream API operations (`CreateStream`, `GetStream`, `DeleteStream`, `ListStreams`) needs permission for these actions and for `iam:PassRole`.

- **Service role** – An IAM role that Aurora DSQL assumes at runtime to write CDC records to your target. You create this role, attach a trust policy that allows the Aurora DSQL service principal to assume it, and attach a permissions policy that grants write access to the target.

> **Note**
>
> The CDC service role is separate from any resource-based policy on your Aurora DSQL cluster. A cluster resource-based policy controls which principals can connect to and query the cluster. The CDC service role controls which target Aurora DSQL can write CDC records to.

Caller permissions

The IAM principal that calls the CDC stream API operations needs permissions for the relevant `dsql` actions and `iam:PassRole`. The `CreateStream` operation requires `iam:PassRole` because it passes the service role ARN to Aurora DSQL. The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLStreamActions",
            "Effect": "Allow",
            "Action": [
                "dsql:CreateStream",
                "dsql:GetStream",
                "dsql:ListStreams",
                "dsql:DeleteStream"
            ],
            "Resource": [
                "arn:aws:dsql:region:your-account-id:cluster/cluster-id",
                "arn:aws:dsql:region:your-account-id:cluster/cluster-id/stream/*"
            ]
        },
        {
            "Sid": "TagStreams",
            "Effect": "Allow",
            "Action": "dsql:TagResource",
```

```
            "Resource": "arn:aws:dsql:region:your-account-id:cluster/cluster-id/stream/
*"
        },
        {
            "Sid": "PassServiceRole",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::your-account-id:role/dsql-cdc-role",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "dsql.amazonaws.com"
                }
            }
        }
    ]
}
```

The `Resource` element includes both the cluster ARN (required by `CreateStream` and `ListStreams`) and the stream ARN pattern (required by `GetStream` and `DeleteStream`). The `TagStreams` statement is required only if you pass tags to `CreateStream`.

If the cluster uses an AWS KMS customer managed key, `CreateStream` also requires `kms:Decrypt` on that key.

For a full list of permissions required for each operation, see CreateStream, GetStream, DeleteStream, and ListStreams in the Amazon Aurora DSQL API Reference.

Service role

The service role is the IAM role that Aurora DSQL assumes to write CDC records to your target. You create this role and pass its ARN in the `targetDefinition.kinesis.roleArn` field when you call `CreateStream`. The role requires a trust policy and a permissions policy.

Service role trust policy

The trust policy must allow the Aurora DSQL service principal to assume the role. To protect against confused deputy attacks, use the `aws:SourceAccount` and `aws:SourceArn` condition keys.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
                }
            }
        }
    ]
}
```

The `aws:SourceArn` condition restricts the role to streams under a specific cluster. You must use the wildcard (`stream/*`) when creating a stream because Aurora DSQL hasn't assigned the stream identifier yet. After you create a stream, you can tighten the condition to the exact stream ARN (`arn:aws:dsql:region:your-account-id:cluster/cluster-id/stream/stream-id`) if the role serves a single stream.

To use the role with streams on any cluster in your account, use a broader wildcard: `arn:aws:dsql:region:your-account-id:cluster/*/stream/*`.

To learn more about confused deputy prevention, see Cross-service confused deputy prevention in this guide.

Service role permissions policy

The permissions policy grants the service role access to write records to your Kinesis data stream. The following policy includes both Kinesis write permissions and AWS KMS permissions. The `KMSAccess` statement is only required if your Kinesis data stream uses an AWS KMS customer managed key, but you can include it preemptively so that adding a customer managed key later doesn't break your CDC stream.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord",
                "kinesis:PutRecords"
            ],
            "Resource": "arn:aws:kinesis:region:your-account-id:stream/kinesis-stream-
name"
        },
        {
            "Sid": "KMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:GenerateDataKey"
            ],
            "Resource": "arn:aws:kms:*:*:key/*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "kinesis.region.amazonaws.com",
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:region:your-account-id:stream/kinesis-stream-name",
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        }
    ]
}
```

The conditions in the AWS KMS statement provide the following protections:

- `kms:ViaService` – Restricts key usage to requests that come through the Kinesis service in the specified Region.
- `kms:EncryptionContext:aws:kinesis:arn` – Restricts key usage to encryption operations for the specified Kinesis data stream.
- `aws:ResourceAccount` – The key must belong to the same AWS account as the calling principal, which prevents cross-account key usage.

> **Note**
>
> The AWS KMS key referenced here is the encryption key on the Kinesis data stream, not the cluster's AWS KMS key. The cluster's encryption key protects CDC data inside the Aurora DSQL boundary. The Kinesis encryption key protects CDC data after Aurora DSQL writes it to the Kinesis data stream.

Data protection

Aurora DSQL uses Transport Layer Security (TLS) to encrypt CDC data in transit between Aurora DSQL and your target. Within the Aurora DSQL boundary, Aurora DSQL encrypts CDC data at rest by using the cluster's encryption key.

If the cluster uses an AWS KMS customer managed key and that key becomes inaccessible, an `ACTIVE` or `IMPAIRED` stream transitions to `IMPAIRED` with the error code `CLUSTER_CMK_INACCESSIBLE`. If the key becomes inaccessible before the stream finishes creating, the stream transitions directly to `FAILED`.

For a detailed explanation of encryption in Aurora DSQL, see Data encryption for Amazon Aurora DSQL in this guide.

Understanding CDC records

Aurora DSQL CDC delivers each change as a JSON record. The record uses an envelope structure with operation type, before and after row images, and source metadata.

CDC envelope versioning

For CDC streams created on or after October 1, 2026, Aurora DSQL can evolve the CDC record format according to the following versioning policy. Streams created before October 1, 2026, continue to use the original envelope format.

To keep your applications working as the CDC record format evolves, check the `source.version` field before processing a record. Aurora DSQL increments the minor version when it makes an additive, backward-compatible change to the CDC record structure. This includes adding new fields to the record structure or adding new values to enum fields. To stay compatible with new minor versions of the record structure, your applications should ignore new fields. Applications should tolerate unrecognized enum values.

How records map to Amazon Kinesis

Aurora DSQL writes each CDC record as a single Kinesis record. The Kinesis record's `Data` field contains the JSON payload. Aurora DSQL uses a randomized Kinesis partition key to distribute CDC records evenly across shards. To read all changes, consume all shards on the Kinesis data stream. If a record exceeds the Kinesis record size limit, Aurora DSQL splits it across multiple Kinesis records. For details, see Handling oversized records.

> **Note**
>
> A Kinesis record has one `Data` blob. The primary key values appear in the JSON payload's `before` field for deletes, or the `after` field for inserts and updates. To extract the primary key for downstream processing, read it from the appropriate field in the payload.

Primary key in the payload

For tables with a primary key, the primary key column values appear in the payload:

- For **inserts** and **updates**, the payload includes the primary key columns along with all other columns in the `after` field.
- For **deletes**, the primary key columns appear in the `before` field.

For example, consider a table with a composite primary key:

```
CREATE TABLE order_items (
    order_id INT,
    item_id INT,
    quantity INT,
    price NUMERIC,
    PRIMARY KEY (order_id, item_id)
);
```

A delete on this table produces a payload where `"before": {"order_id": 1001, "item_id": 42}`.

Record payload

The following examples show the JSON envelope fields available in the current version.

INSERT example

The following example shows a CDC record for an INSERT operation:

```
{
    "type": "full",
    "op": "c",
    "before": null,
    "after": {"order_id": 1001, "item_id": 42, "quantity": 5, "price": "29.99"},
    "source": {
        "version": "1.1",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291,
    "transaction": {
        "total_records": 1,
        "record_id": "8f14e45fceea167a5a36dedd4bea2543"
    }
}
```

UPDATE example

The following example shows a CDC record for an UPDATE operation:

```
{
    "type": "full",
    "op": "u",
    "before": null,
    "after": {"order_id": 1001, "item_id": 42, "quantity": 10, "price": "29.99"},
    "source": {
        "version": "1.1",
        "ts_ms": 1705318300000,
        "ts_ns": 1705318300000000000,
        "txId": "qvtiesgmd55cvlfukm3dfuotji",
        "schema": "public",
        "table": "order_items",
```

```
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318300125,
    "ts_ns": 1705318300125483291,
    "transaction": {
        "total_records": 1,
        "record_id": "c9f0f895fb98ab9159f51fd0297e236d"
    }
}
```

DELETE example

For a DELETE on a table with a primary key, the `before` field contains the primary key values:

```
{
    "type": "full",
    "op": "d",
    "before": {"order_id": 1001, "item_id": 42},
    "after": null,
    "source": {
        "version": "1.1",
        "ts_ms": 1705318400000,
        "ts_ns": 1705318400000000000,
        "txId": "xyzabc123def456ghi789jklmno",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318400125,
    "ts_ns": 1705318400125483291,
    "transaction": {
        "total_records": 1,
        "record_id": "45c48cce2e2d7fbdea1afc51c7c6ad26"
    }
}
```

Payload fields

Field	Description
`type`	The record type. `full` for a complete record that includes inline `before` and `after` values. `chunked` for a main record that reference s fragment records for one or both images. `fragment` for an individual piece of a chunked image. For details, see Handling oversized records.
`op`	Operation type. `c` = create (insert), `u` = update, `d` = delete. The `op` reflects the row's net effect across the transaction: a row that didn't exist before the transaction receives `c`, a pre-exist ing row that the transaction modified receives `u`, and a pre-existing row that the transaction removed receives `d`. For details, see Write-set compaction.
`before`	For deletes on tables with a primary key, contains the primary key values of the deleted row. Aurora DSQL sets this field to `null` for inserts, updates, and deletes on tables without a primary key.
`after`	The full row state after the change, including all columns. Aurora DSQL sets this field to `null` for deletes.
`chunked`	Present only when `type` is `chunked`. Contains reassembly metadata for the `before` image, the `after` image, or both. Aurora DSQL omits the chunked image from the top-level `before` or `after` field and places it under `chunked`

Field	Description
	instead. For details, see Handling oversized records.
`source.version`	The CDC envelope format version. Beginning with version `1.1`, records include metadata about rows changed by a transaction. For information about compatible changes between versions, see CDC envelope versioning.
`source.ts_ms`	The transaction commit timestamp in milliseconds since the Unix epoch, Coordinated Universal Time (UTC).
`source.ts_ns`	Transaction commit timestamp in nanoseconds, UTC. The highest precision timestamp available. Use this field to establish a total order of transactions.
`source.txId`	A unique transaction identifier, encoded as base32. All records from the same transaction share the same `txId` value. Use this field to group records that belong to the same transaction.
`source.schema`	The PostgreSQL schema name (for example, `public`).
`source.table`	The table name.
`source.db`	The database name. Always `postgres` for Aurora DSQL.
`source.cluster`	The Aurora DSQL cluster identifier.

Field	Description
`transaction`	Metadata about rows changed by a transacti on. Beginning with version `1.1`, `full` and `chunked` main records include this object. Fragment records don't include it.
`transaction.total_records`	The number of rows with a net change across all user tables in the transaction after write-set compaction. The same value appears on every changed-row record from the transacti on. For an oversized record, the chunked main record represents one changed row; its fragment records aren't included in the total.
`transaction.record_id`	An opaque identifier for this changed-row record within the transaction. The identifier is stable across retries. Use it with `source.tx Id` to identify duplicate deliveries. Don't parse the identifier or depend on its format.
`ts_ms`	The time at which the CDC system processed the record, in milliseconds, UTC. The differenc e between `ts_ms` and `source.ts_ms` is a measure of replication lag.
`ts_ns`	The time at which the CDC system processed the record, in nanoseconds, UTC.

Format details

The following details describe how Aurora DSQL CDC formats records. Design your app to handle these behaviors.

- **Full after-image for inserts and updates.** Aurora DSQL includes the complete row state in the `after` field for all writes. The `before` field is `null` for inserts and updates. Aurora DSQL emits `op: "c"` for inserts and `op: "u"` for updates.

- **Write-set compaction.** Aurora DSQL compacts the write set of each transaction before publishing, emitting at most one record per row. For details and examples, see Write-set compaction.
- **Post-change row state only.** CDC records include the full row state after each change. The state of the row before an update isn't included. For deletes on tables with a primary key, the `before` field contains the primary key values.
- **Numeric types serialized as strings.** Aurora DSQL serializes `numeric` and `decimal` values as JSON strings to preserve exact precision.
- **Binary data encoded as Base64.** Aurora DSQL encodes `bytea` values as Base64 strings.
- **Special floating-point and numeric values.** Aurora DSQL serializes NaN and ±Infinity as the strings `"NaN"`, `"Infinity"`, and `"-Infinity"`. This applies to `real`, `double precision`, and `numeric` types.
- **JSON columns serialized as JSON strings.** Aurora DSQL serializes `json` column values as JSON strings that contain the raw JSON text stored in the column. Parse the string value in your app (for example, with `JSON.parse` in JavaScript or `json.loads` in Python) to access the underlying JSON value.
- **Overflow values emitted as null.** If a value can't be represented in the target JSON type during serialization, Aurora DSQL emits JSON `null` for that column. This applies to `interval` values whose total microseconds exceed the 64-bit signed integer range (±9,223,372,036,854,775,807 microseconds, approximately ±292,271 years). Design your app to handle unexpected `null` values in columns that aren't nullable in the database schema.
- **Oversized records split into chunks.** If a record exceeds the Amazon Kinesis record size limit, Aurora DSQL splits the affected `before` or `after` image into fragments and delivers them as separate Kinesis records so you still receive the change. Design your app to reassemble the images. For details, see Handling oversized records.

Write-set compaction

Aurora DSQL compacts the write set of each committed transaction before publishing CDC records. Aurora DSQL compares each row's state before the transaction to its state after, and emits at most one record per row reflecting the net effect.

Compaction rules

Aurora DSQL determines the op code based on the row's net state change:

op: "c" (create)

The row is new. It didn't exist before the transaction and exists after. The `after` field contains the final row state.

op: "u" (update)

The row existed before the transaction and still exists after. The `after` field contains the final row state. Aurora DSQL compacts operations, not row content.

op: "d" (delete)

The row existed before the transaction and doesn't exist after. The `before` field contains only the primary key values.

No record

The row didn't exist before the transaction and doesn't exist after (for example, an insert followed by a delete in the same transaction). Aurora DSQL recognizes zero net change and doesn't publish a record.

Handling oversized records

When a CDC record's serialized JSON exceeds 9 MiB, Aurora DSQL splits the `before` and/or `after` images, delivering multiple Kinesis records. Each record contains a top-level `type` field that indicates its structure: `full` for a complete record, `chunked` for a main record that references fragments, and `fragment` for an individual piece of a chunked image. The `op`, `source`, `transaction`, `ts_ms`, and `ts_ns` fields on a chunked main record behave the same as on a full record. Records that fit in a single Kinesis record have `type` set to `full` and don't require any extra handling.

A `chunk_id` is stable across retries. If Aurora DSQL redelivers a fragment, it carries the same `chunk_id` as the original delivery, so your app can continue buffering under the same identifier without handling partial sets from previous attempts. Identify duplicate fragment deliveries by comparing `chunk_id` and `index`.

Main record

A chunked main record replaces the top-level `before` or `after` field for the split image with a `chunked` object that describes how to reassemble it. Each entry under `chunked` has a `chunk_id`

(the identifier that links fragments to this record), `total_fragments` (the number of fragments that make up that image), and `crc32c` (a CRC32C checksum, as a decimal string, over the reassembled image text). If one image is inline and the other is chunked, the inline image still appears at the top level as either a value or `null`.

```
{
    "type": "chunked",
    "op": "c",
    "before": null,
    "after": null,
    "source": {
        "version": "1.1",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "cluster-id"
    },
    "chunked": {
        "after": {
            "chunk_id": "chunk-id",
            "total_fragments": 3,
            "crc32c": "2073618257"
        }
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291,
    "transaction": {
        "total_records": 1,
        "record_id": "8f14e45fceea167a5a36dedd4bea2543"
    }
}
```

Fragment record

Each fragment is its own Kinesis record with `type` set to `fragment` and three fields: `chunk_id` matches the value in the corresponding `chunked.before.chunk_id` or `chunked.after.chunk_id` on the main record, `index` is the zero-based position of the fragment within the image, and `data` is a segment of the image's JSON text split on UTF-8 character boundaries (each fragment's `data` value is a valid UTF-8 string on its own). Fragment records

don't include `source` or `transaction` metadata. Because Aurora DSQL CDC uses UNORDERED mode and randomized partition keys, fragments and the main record can arrive on different shards and in any order. To read all fragments, consume all shards on the Kinesis data stream. For more information about delivery ordering, see Ordering.

```
{
    "type": "fragment",
    "chunk_id": "chunk-id",
    "index": 0,
    "data": "partial-JSON-text"
}
```

To reassemble an oversized image, buffer each record with `type` `fragment` by its `chunk_id`. When you receive a main record with `type` `chunked`, wait until you have `total_fragments` fragments for each `chunk_id` referenced under `chunked.before` or `chunked.after`, sort the fragments by `index` ascending, and concatenate the `data` strings. The concatenated result is the original `before` or `after` object as JSON text—parse it to access the column values. To verify delivery integrity, compute CRC32C over the concatenated string and compare the result to `chunked.before.crc32c` or `chunked.after.crc32c`. A transaction that includes a chunked main record isn't ready to process until you have reassembled every image that the main record references.

Data type serialization

The following tables describe how Aurora DSQL serializes each PostgreSQL data type in CDC records.

Integer types

PostgreSQL type	JSON representation	Example
`smallint` (`int2`)	JSON number	`42`
`integer` (`int4`)	JSON number	`1001`
`bigint` (`int8`)	JSON number	`9223372036854775807`
`oid`	JSON number (unsigned)	`16384`

Values of `bigint` beyond ±2^53 might lose precision in JavaScript environments. Use BigInt or arbitrary-precision libraries in those cases.

Floating-point types

PostgreSQL type	JSON representation	Example	Notes
`real` (`float4`)	JSON number	`3.14159`	NaN and ±Infinity are serialized as the strings `"NaN"`, `"Infinity"` ,`"-Infinity"` .
`double precision` (`float8`)	JSON number	`3.141592653589793`	Same special value handling as `real`.
`numeric` / `decimal`	JSON string	`"123.45"`	Always a string to preserve exact precision. NaN and ±Infinity are serialized as the strings `"NaN"`, `"Infinity"` ,`"-Infinity"` .

Boolean

PostgreSQL type	JSON representation	Example
`boolean`	JSON boolean	`true` or `false`

Character types

PostgreSQL type	JSON representation	Example
`varchar` / `text`	JSON string	`"Hello, world!"`

PostgreSQL type	JSON representation	Example
`bpchar` (`char(n)`)	JSON string	`"ABC"` (trailing spaces stripped)
`name`	JSON string	`"pg_class"`
`"char"` (single-byte)	JSON string	`"A"`

Binary

PostgreSQL type	JSON representation	Example
`bytea`	JSON string (Base64)	`"SGVsbG8gV29ybGQh"`

Date and time types

PostgreSQL type	JSON representation	Example	Notes
`date`	JSON number (days since Unix epoch)	`19797`	`+infinity` and `-infinity` are represented as sentinel day counts derived from epoch-offset arithmeti c. These values don't correspond to meaningful calendar dates.
`time`	JSON number (microseconds since midnight)	`52200123456`	

PostgreSQL type	JSON representation	Example	Notes
`timetz`	JSON number (microseconds since midnight, UTC)	`52200123456`	The local time is adjusted to UTC by applying the stored timezone offset (seconds west of UTC). The result is wrapped to the range [0, 86400000000) microseconds.
`timestamp`	JSON number (microseconds since Unix epoch)	`171051060 0123456`	±Infinity maps to sentinel values: `922337203 6825200000` for `+infinity` and `-92233720 36832400000` for `-infinity` .
`timestamptz`	JSON number (microseconds since Unix epoch)	`171051060 0123456`	Stored and emitted in UTC. Same ±infinity sentinel values as `timestamp` .

PostgreSQL type	JSON representation	Example	Notes
`interval`	JSON number (approximate total microseconds)	`2802603000000`	Months are approximated as 30.4375 days (2,629,800 seconds). The total is computed as `(months × 2,629,800 + days × 86,400) × 1,000,000 + microseconds`. If the result exceeds the 64-bit signed integer range (±9,223,372,036,854,775,807 microseconds, approximately ±292,271 years), Aurora DSQL emits JSON `null` for the column.

Other types

PostgreSQL type	JSON representation	Example
`uuid`	JSON string (standard 8-4-4-4-12 hex format)	`"550e8400-e29b-41d4-a716-446655440000"`
`oidvector`	JSON empty array	`[]`
`json`	JSON string containing the raw JSON text	`"{\"key\": \"value\"}"`

NULL values

For any data type, `NULL` column values are represented as JSON `null`.

Schema evolution in CDC records

When you modify a table's schema—for example, by adding, dropping, or renaming a column—CDC records reflect the change starting from the transaction that committed the DDL change. Records from transactions committed before the DDL change use the previous schema. For example:

- If you add a column, records from earlier transactions don't include the new column. Records from the adding transaction onward include the new column.
- If you drop a column, records from the dropping transaction onward no longer include that column.
- If you rename a column, records from the renaming transaction onward use the new column name.

Track schema changes in your downstream consumer by inspecting the column names present in each record's `after` and `before` fields. The `source.version` field in each record identifies the CDC envelope format.

Monitoring streams

When Aurora DSQL encounters an error delivering a CDC record, the stream transitions to `IMPAIRED` status. An impaired stream continues to process and deliver other records—Aurora DSQL retries only the failing record. Aurora DSQL measures replication lag from the oldest undelivered record, and the lag grows until you resolve the issue. Aurora DSQL retains undelivered changes internally for one week.

If you resolve the underlying issue within this window, the next retry succeeds, the error state clears, and the stream transitions back to `ACTIVE`. Fix the external issue (IAM policy, AWS KMS key, Amazon Kinesis capacity, and so on) and Aurora DSQL retries automatically.

If the replication lag exceeds the failure threshold, the stream transitions to `FAILED`.

> **Important**
>
> A failed stream can't recover. You must delete the failed stream and create a new one.

Stream lifecycle

A stream transitions through the following statuses during its lifecycle:

- **`CREATING`** – Aurora DSQL is setting up the stream. Aurora DSQL doesn't deliver CDC records yet.
- **`ACTIVE`** – The stream is operational and delivering CDC records to the target.
- **`IMPAIRED`** – The stream has encountered an issue that requires your action. Aurora DSQL retries the failing record with exponential backoff, although other records can continue delivering. Aurora DSQL measures replication lag from the oldest undelivered record, and the lag grows until you resolve the issue. Aurora DSQL buffers undelivered changes internally for one week. See Error code reference.
- **`FAILED`** – The stream has encountered a persistent error and is no longer delivering CDC records. A failed stream can't recover and you must delete it. See Error code reference for the conditions that cause a stream to enter this state.
- **`DELETING`** – Aurora DSQL is removing stream resources.
- **`DELETED`** – Aurora DSQL has deleted the stream. After deletion completes, `GetStream` returns a `ResourceNotFoundException`.

Call `GetStream` to view the stream status at any time. When the stream is `IMPAIRED` or `FAILED`, the response includes a `statusReason` object with the error code and timestamp. For more details about the `GetStream` response fields, see GetStream in the Amazon Aurora DSQL API Reference.

Troubleshooting an impaired or failed stream

Follow these steps when a CDC stream becomes impaired or fails. If the stream is `FAILED`, you can't recover it—delete the stream, resolve the underlying issue, and create a new one.

1. **Get the stream status.** Call `GetStream` and verify the `status` field. If the status is `ACTIVE`, the stream is healthy.

```
aws dsql get-stream \
  --cluster-identifier cluster-id \
```

```
  --stream-identifier stream-id \
  --region region
```

2. **Read the error code.** If the status is `IMPAIRED` or `FAILED`, the response includes a `statusReason` object. The `error` field contains the error code.

```
{
    "status": "IMPAIRED",
    "statusReason": {
        "error": "KINESIS_THROUGHPUT_EXCEEDED",
        "updatedAt": "2025-01-15T14:30:00Z"
    }
}
```

3. **Follow the remediation.** If the stream is `IMPAIRED`, look up the error code in the following table and apply the recommended fix. Aurora DSQL retries automatically after you resolve the underlying issue. If the stream is `FAILED`, delete it, resolve the issue, and create a new stream.

Error code reference

The following table describes each error code, its cause, whether the stream can recover, and the steps to resolve it.

Error code	Cause	Recoverable?	How to resolve
`KINESIS_THROUGHPUT_EXCEEDED`	Your Kinesis data stream exceeded its throughput limit, or AWS KMS throttled encryption operations on the Kinesis data stream, and the replication lag has grown.	Yes	Increase the number of shards on your Kinesis data stream, or switch to on-demand capacity mode. If the Kinesis data stream uses an AWS KMS customer managed key, verify that the key's request quota is large enough. After you increase capacity,

Error code	Cause	Recoverable?	How to resolve
			Aurora DSQL retries automatically.
`KINESIS_STREAM_NOT_FOUND`	The target Kinesis data stream no longer exists.	No	The stream transitions directly to `FAILED`. Delete the CDC stream and create a new one pointing to a valid Kinesis data stream.
`ROLE_ACCESS_DENIED`	Aurora DSQL can't assume the IAM role specified in the target definition. The AWS STS `AssumeRole` call returned `AccessDenied`.	Yes	Verify the role's trust policy allows the Aurora DSQL service principal (`dsql.amazonaws.com`) to assume it. Verify the `aws:SourceAccount` and `aws:SourceArn` conditions match your cluster. For details, see Service role trust policy. After you fix the trust policy, Aurora DSQL retries automatically.

Error code	Cause	Recoverable?	How to resolve
KINESIS_ACCESS_DENIED	The assumed role doesn't have permission to write to the Kinesis data stream. Kinesis returned AccessDeniedException .	Yes	Add kinesis:PutRecord and kinesis:PutRecords permissions to the role's policy for the target Kinesis data stream Amazon Resource Name (ARN). After you fix the policy, Aurora DSQL retries automatically.

Error code	Cause	Recoverable?	How to resolve
`KINESIS_KMS_ACCESS_DENIED`	The assumed role doesn't have permission to use the AWS KMS key that encrypts the Kinesis data stream. This error covers AWS KMS access denial and invalid key states.	Yes	Verify the role has `kms:GenerateDataKey` permission on the AWS KMS key that the Kinesis data stream uses. Also verify that the AWS KMS key is in an enabled and valid state. This key is the encryption key on the Kinesis data stream, not the cluster's AWS KMS key. For details, see Service role permissions policy. After you fix the permissions or key state, Aurora DSQL retries automatically.

Error code	Cause	Recoverable?	How to resolve
KINESIS_OVERSIZE_RECORD	A CDC record exceeded the maximum record size configured on the Kinesis data stream.	Yes	Increase MaxRecordSizeInKiB on the Kinesis data stream to 10240 (10 MiB). You can update this setting on an existing Kinesis data stream without deleting it. After you increase the limit, Aurora DSQL retries the oversized record automatically and the stream transitions back to ACTIVE.
CLUSTER_CMK_INACCESSIBLE	The AWS KMS customer managed key that encrypts the Aurora DSQL cluster is inaccessible.	Yes	Verify the AWS KMS key policy and key state. Re-enable or restore access to the key. After the key becomes accessible again, the stream transitions back to ACTIVE.

The preceding table lists every StreamFailureErrorCode value. For details about the statusReason response field, see GetStream in the Amazon Aurora DSQL API Reference.

Recovering an impaired stream

Most errors first transition the stream to IMPAIRED. An impaired stream continues to process other records and retries the failing record automatically. A FAILED stream isn't recoverable—you must delete it and create a new one.

- **For recoverable errors:** fix the external issue (IAM policy, AWS KMS key, Kinesis capacity, or Kinesis record size limit). The next successful retry clears the error state and transitions the stream back to `ACTIVE`.
- **For `KINESIS_STREAM_NOT_FOUND`:** the stream transitions directly to `FAILED`. Delete the failed stream and create a new one pointing to a valid Kinesis data stream.

For all other error codes, if the replication lag exceeds the failure threshold before you resolve the issue, the stream transitions from `IMPAIRED` to `FAILED`. A failed stream can't transition back to `ACTIVE`. Delete the failed stream, resolve the underlying issue, and create a new one.

Monitoring stream health

Use CloudWatch metrics and the `GetStream` API to monitor stream health. CloudWatch metrics provide continuous visibility into CDC pipeline performance, and `GetStream` provides the specific error code when a stream is impaired or failed.

For the full list of CDC metrics, including `IsImpaired`, `BehindSourceLag`, `PublishedBytes`, and `PublishedRecords`, see CloudWatch metrics for CDC streams. For more details about the `GetStream` response fields, see GetStream in the Amazon Aurora DSQL API Reference.

CloudWatch metrics for CDC streams

Use the following CloudWatch metrics to monitor the health and throughput of each CDC stream. Aurora DSQL publishes these metrics in the `AWS/AuroraDSQL` namespace with the dimensions `ClusterId` and `StreamId`. The last metric is a standard Amazon Kinesis metric in the `AWS/Kinesis` namespace that measures downstream reading lag.

> **Note**
>
> Aurora DSQL also publishes the `BytesStreamed` and `StreamDPU` metrics in the `AWS/AuroraDSQL` namespace for usage and billing tracking. For descriptions, see CDC stream metrics.

Metric name	Useful statistic	Description
`IsImpaired`	Maximum	Indicates whether the stream is impaired. The value is 1

Metric name	Useful statistic	Description
		when the stream is in the `IMPAIRED` state, and `0` when the stream is healthy. Aurora DSQL emits this metric continuously for each active or impaired stream. Use this metric to create a CloudWatch alarm that notifies you when a stream becomes impaired.
`BehindSourceLag`	Average	The delay, in milliseconds, between when a transaction commits in Aurora DSQL and when the CDC system processes the resulting record. A rising value indicates that the CDC pipeline is falling behind the write workload.
`PublishedBytes`	Sum	The total bytes of CDC records that Aurora DSQL wrote to the target during the period. Use this metric together with your Kinesis shard count to determine whether you've provisioned enough write capacity.
`PublishedRecords`	Sum	The total number of CDC records that Aurora DSQL wrote to the target during the period. Each committed row change produces one record.

Metric name	Useful statistic	Description
`GetRecords.IteratorAgeMilliseconds` (`AWS/Kinesis`)	Average	A standard Kinesis metric that reports the age of the last record read from the Kinesis data stream by your downstream app, in milliseconds. Use the `StreamName` dimension. A rising value indicates that your downstream app can't keep up with the rate at which Aurora DSQL writes CDC records to Kinesis.

The Aurora DSQL console's **Monitoring** tab shows an **Average end-to-end latency** value that combines `BehindSourceLag` (CDC source latency) and `GetRecords.IteratorAgeMilliseconds` (Kinesis reader lag). This combined value represents the total delay from database commit to downstream read.

Monitoring best practices

Use the following practices to detect and resolve CDC pipeline issues before they affect your downstream systems.

Set alarms on `BehindSourceLag`

Create a CloudWatch alarm that fires when `BehindSourceLag` exceeds a threshold that matters to your workload. For example, set 60 seconds for a one-minute latency target. A sustained increase in this metric means the CDC pipeline is falling behind. If the lag reaches the failure threshold, the stream transitions to FAILED. Catching the trend gives you time to increase Kinesis capacity or investigate throughput bottlenecks before the stream degrades.

Monitor `GetRecords.IteratorAgeMilliseconds` on the Kinesis side

Even when Aurora DSQL delivers records on time, your downstream app can fall behind. Create a CloudWatch alarm on `GetRecords.IteratorAgeMilliseconds` (in the `AWS/Kinesis` namespace, dimension `StreamName`) to detect downstream lag independently. If this metric rises and `BehindSourceLag` stays flat, the bottleneck is in your downstream app, not in Aurora DSQL.

Track `PublishedBytes` against Kinesis shard capacity

Each Kinesis shard supports up to 1 MiB per second for writes. Compare the `PublishedBytes` Sum per minute against your total shard write capacity (number of shards × 60 MiB per minute). If usage approaches 80 percent, add shards or switch to on-demand capacity mode before throttling triggers `KINESIS_THROUGHPUT_EXCEEDED`.

Alarm on `IsImpaired` for instant impairment detection

Create a CloudWatch alarm that fires when `IsImpaired` Maximum is greater than or equal to `1` for one evaluation period. This gives you a direct signal when a stream enters the `IMPAIRED` state, without polling the API. After the alarm fires, call `GetStream` to read the `statusReason.error` field and follow the remediation steps in Troubleshooting an impaired or failed stream.

Poll `GetStream` for detailed status

The `IsImpaired` metric tells you that a stream is impaired, but the `GetStream` API provides the specific error code and timestamp. Poll `GetStream` on a schedule (for example, every five minutes) or in response to an `IsImpaired` alarm. The `statusReason.error` field tells you what went wrong. Pair this with the troubleshooting steps in Troubleshooting an impaired or failed stream for quick resolution.

Use dashboards to correlate metrics

Create a CloudWatch dashboard that shows `IsImpaired`, `BehindSourceLag`, `PublishedRecords`, `PublishedBytes`, and `GetRecords.IteratorAgeMilliseconds` side by side. Correlating these metrics helps you distinguish between a CDC pipeline issue (rising `BehindSourceLag`) and a downstream reading issue (rising `IteratorAge` with stable `BehindSourceLag`).

Monitoring and logging for Aurora DSQL

Monitoring and logging is an important part of maintaining the reliability, availability, and performance of your Amazon Aurora DSQL resources. You should monitor and collect logging data from all parts of your Aurora DSQL resources so you can easily debug a multi-point failure.

- *Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the Amazon CloudWatch User Guide.
- *Aurora DSQL Database Insights* gives you a deeper understanding of the performance and cost of your clusters. For more information, see Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Topics

- Monitoring Aurora DSQL with Amazon CloudWatch
- Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights
- Logging Aurora DSQL operations using AWS CloudTrail

Monitoring Aurora DSQL with Amazon CloudWatch

Monitor Aurora DSQL using CloudWatch, which collects raw data and processes it into readable, near real-time metrics. CloudWatch keeps these statistics for 15 months, helping you gain better perspective on your web application or service performance. Set alarms to watch for specific thresholds and send notifications or take actions when met. Review the following Usage and Observability metrics available for Aurora DSQL.

For more information, see the Amazon CloudWatch User Guide.

For a deeper understanding of the performance and cost of your clusters, see Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights.

Observability and performance

This table outlines observability metrics for Aurora DSQL. It includes metrics for tracking read-only and total transactions to provide overall workload characterization. Actionable metrics like query timeouts and OCC conflict rate are included to help identify performance issues and concurrency conflicts. Session-related metrics, both active and total, offer insights into the current load on the system.

CloudWatch Metric Name	Metric	Unit	Description
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.
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

CloudWatch Metric Name	Metric	Unit	Description
BytesRead	Bytes Read	bytes	Bytes read from storage
ComputeTime	QP compute time	milliseconds	QP wall clock time
ClusterStorageSize	Cluster Storage Size	bytes	Cluster size

Aurora DSQL also publishes the following metrics under the AWS/Usage namespace. Use these metrics to monitor connection attempts and active connection count for your clusters.

CloudWatch Metric Name	Type	Resource	ResourceId	Service	Unit	Description
ResourceCount	Resource	ClusterConnectionCount	cluster/<cluster-id>	AuroraDSQL	None	The number of active connections for a cluster.
CallCount	API	DbConnectAdmin or DbConnect	cluster/<cluster-id>	AuroraDSQL	None	The number of API calls for database connection operations (DbConnectAdmin or DbConnect).

Usage metrics

Aurora DSQL measures all request-based activity, such as query processing, reads, and writes, using a single normalized billing unit called Distributed Processing Unit (DPU).

CloudWatch Metric Name	Metric	Dimension: ResourceId	Unit	Description
WriteDPU	Write Units	<cluster-id>	DPU	Approximates the write active-use component of your Aurora DSQL cluster DPU usage.
MultiRegionWriteDPU	Multi-Region Write Units	<cluster-id>	DPU	Applicable for Multi-Region clusters: Approximates the multi-Region write active-use component of your Aurora DSQL cluster DPU usage.
ReadDPU	Read Units	<cluster-id>	DPU	Approximates the read active-use component of your Aurora DSQL cluster DPU usage.
ComputeDPU	Compute Units	<cluster-id>	DPU	Approximates the compute active-use component of your Aurora

CloudWatch Metric Name	Metric	Dimension: ResourceId	Unit	Description
				DSQL cluster DPU usage.
StreamDPU	Stream Units	<cluster-id>	DPU	Applicable for clusters with CDC streams: Approximates the CDC streaming active-use component of your Aurora DSQL cluster DPU usage, summed across all streams on the cluster.
TotalDPU	Total Units	<cluster-id>	DPU	Approximates the total active-use component of your Aurora DSQL cluster DPU usage. This is the sum of the write, multi-Region write, read, compute, and stream components. Use the Sum statistic.

CDC stream metrics

Aurora DSQL publishes the following metrics for change data capture (CDC) streams. These metrics use the `ClusterId` and `StreamId` dimensions, so you can monitor each CDC stream independently. For more information about CDC streams, see Change data capture (CDC) streams.

CloudWatch Metric Name	Metric	Unit	Description
IsImpaired	Is impaired	none	Indicates whether the stream is impaired. The value is 1 when the stream is in the `IMPAIRED` state, and `0` when the stream is healthy. Use this metric to create a CloudWatch alarm that notifies you when a stream becomes impaired.
PublishedBytes	Published bytes	bytes	The total number of bytes that Aurora DSQL wrote to the target Kinesis data stream.
PublishedRecords	Published records	none	The number of CDC records that Aurora DSQL wrote to the target Kinesis data stream.
BehindSourceLag	Behind source lag	milliseconds	The delay, in milliseconds, between when a transaction commits in Aurora DSQL

CloudWatch Metric Name	Metric	Unit	Description
			and when the CDC system processes the resulting record. A rising value indicates that the CDC pipeline is falling behind the write workload. If lag grows beyond the failure threshold, the stream transitions to `FAILED`.
BytesStreamed	Bytes streamed	bytes	The total bytes streamed through the CDC pipeline for billing purposes. This metric reflects the data volume used to calculate streaming charges.

CloudWatch Metric Name	Metric	Unit	Description
StreamDPU	Stream DPU	DPU	The Distributed Processing Units (DPU) consumed by the CDC stream. This metric reflects the processing cost of streaming change data. Aurora DSQL also publishes `StreamDPU` with only the `ClusterId` dimension, summed across all streams on the cluster.

Setting DPU cost alarms

To get notified when your cluster uses more DPUs than you expect, create a CloudWatch alarm on the `TotalDPU` metric in the `AWS/AuroraDSQL` namespace. Use the `ClusterId` dimension and the `Sum` statistic. Because the alarm evaluates DPU usage over a period, such as one hour, set the threshold as the DPU usage for that period.

To choose an hourly threshold from a monthly DPU budget, use the steps in the following table. Storage has its own charge, so leave it out of your DPU budget.

For example, assume a price of USD $8.00 per 1 million DPUs and a monthly DPU budget of USD $100.00. For the price in your AWS Region, see Aurora DSQL pricing on the AWS website.

Step	Calculation	Example
1. Find how many million DPUs your budget covers each month.	Monthly DPU budget ÷ price per 1 million DPUs	100 ÷ 8 = 12.5 million DPUs

Step	Calculation	Example
2. Find the hourly DPUs at 100% of your budget.	Result of step 1 ÷ 730 hours in a month	12,500,000 ÷ 730 = about 17,123 DPUs
3. Find the hourly DPUs for an earlier warning, such as at 80% of your budget.	Result of step 2 × 0.8	17,123 × 0.8 = about 13,699 DPUs

Use the result of step 3 as the hourly threshold for your alarm.

The easiest way to create a DPU cost alarm is to start from the Aurora DSQL console. The console fills in the metric, namespace, and cluster for you.

To start a DPU cost alarm (Aurora DSQL console)

1. Open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. In the list of clusters, select your cluster.
3. On the **Summary** tab, in the **Usage and cost** panel, choose **Create alarm**. CloudWatch opens with the `TotalDPU` metric for your cluster already selected.

You can also start from the CloudWatch console by choosing the metric yourself.

To choose the metric for a DPU cost alarm (CloudWatch console)

1. Open the Alarms page of the Amazon CloudWatch console.
2. Choose **Create alarm**, and then choose **Select metric**.
3. Choose **AuroraDSQL**, then **ClusterId**. Select the `TotalDPU` metric for your cluster, and then choose **Select metric**.

After you start the alarm from either console, configure it.

To configure the DPU cost alarm (console)

1. For **Statistic**, choose **Sum**. For **Period**, choose **1 hour**.
2. For **Threshold type**, select **Static**.
3. For **Whenever TotalDPU is**, select **Greater**.

4. For **than**, enter your hourly DPU threshold.
5. Under **Additional configuration**, for **Missing data treatment**, choose **Treat missing data as good (not breaching threshold)**. An idle cluster doesn't publish DPU metrics.
6. Choose **Next**. Choose an Amazon Simple Notification Service (Amazon SNS) topic to notify when the alarm changes state, and then finish creating the alarm.

The following AWS CLI command creates the same alarm. Replace *your-cluster-id* with your cluster ID, *hourly-dpu-threshold* with your hourly DPU threshold, and *sns-topic-arn* with the ARN of your Amazon SNS topic.

```
aws cloudwatch put-metric-alarm \
    --alarm-name "DSQL-Hourly-DPU-Budget-Warning" \
    --alarm-description "Alert when hourly DPU exceeds 80% of budget" \
    --metric-name TotalDPU \
    --namespace AWS/AuroraDSQL \
    --statistic Sum \
    --period 3600 \
    --evaluation-periods 1 \
    --datapoints-to-alarm 1 \
    --threshold hourly-dpu-threshold \
    --comparison-operator GreaterThanThreshold \
    --dimensions Name=ClusterId,Value=your-cluster-id \
    --alarm-actions sns-topic-arn \
    --ok-actions sns-topic-arn \
    --treat-missing-data notBreaching
```

Keep the following in mind when you set DPU cost alarms:

- The alarm tracks DPU usage for each period, not your total spend for the month. A short spike can trigger the alarm even if your monthly usage stays within budget. To track month-to-date spend, use AWS Budgets.
- To reduce alarms from short spikes, require more than one breaching data point, such as two out of three evaluation periods.
- A multi-Region cluster setup has a separate peered cluster in each Region, and each peered cluster reports its own `TotalDPU`. Create one alarm for each peered cluster, in that cluster's Region.
- If you use CDC streams, `TotalDPU` already includes `StreamDPU`, so your `TotalDPU` alarm covers stream costs. To track the cost of a single stream, you can also create a separate alarm

on `StreamDPU`. Aurora DSQL reports `StreamDPU` for each stream with the `ClusterId` and `StreamId` dimensions.

For more information about finding the statements that drive DPU usage, see Finding high-load queries with Aurora DSQL Database Insights.

Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights

Aurora DSQL database insights provides access to per-second sampled data for every active session on your cluster, collected by **DSQL Active Session History (DASH)** sampler which records the wait state and normalized SQL statement each session is running. Per-minute aggregated results from DASH are published as CloudWatch OpenTelemetry (OTel) metrics. Use this data to understand your database load, find the queries that consume the most resources, and diagnose performance issues.

You don't need to set up DASH. It is automatically enabled for every Aurora DSQL cluster, and per-minute aggreated data is available through Amazon CloudWatch Database Insights and through Prometheus Query Language (PromQL) queries run against CloudWatch OTel metrics at no additional cost.

Topics

- What is DASH?
- DASH metrics and wait events
- Accessing DASH data
- Related resources

What is DASH?

An active session represents a single connection to your cluster that has an open transaction. At any given moment, each active session is in one of three states:

- Using CPU
- Waiting for an external operation to complete, such as a storage read or a commit acknowledgment

- Idle in transaction, waiting for the next statement from your application in an open transaction

DASH tracks only sessions that have an open transaction. DASH captures this activity by sampling all active sessions on your cluster once per second. Each sample records two things:

- **The wait event** – the state the session was in at the moment of sampling. See DASH wait events for the full list.
- **The SQL statement** – the first 256 characters of the SQL the session was running, which is enough to identify most queries.

DASH aggregates these per-second samples and publishes them to CloudWatch once per minute, broken down by wait event and by SQL statement. The result is a single time-series metric, `db.active_sessions.avg`, that records the average number of active sessions over each sample period. This value is also known as **Average Active Sessions (AAS)**.

AAS is the foundational signal for Aurora DSQL performance analysis. The wait-event breakdown shows what your sessions spend time on, not only how busy the cluster is. The SQL breakdown attributes load to individual queries.

> **Tip**
>
> Aurora DSQL scales CPU elastically, so the most useful diagnostic signal is the shape of the wait profile (the proportional distribution of session time across wait events) rather than AAS against a fixed vCPU ceiling. Compare that shape against the baseline you observe during normal operation. A change in those proportions indicates a shift in workload or system behavior.

DASH metrics and wait events

DASH exposes a single metric whose dimensions let you slice database load by wait event and by SQL statement. The following table describes the DASH metric.

Metric	Unit	Description
`db.active_sessions.avg`	Count	The average number of active sessions on the cluster over

Metric	Unit	Description
		the sample period (AAS). Each active session is either running on CPU or in a named wait state.

The metric carries the following dimensions (labels), which you use to group and filter the data.

Dimension	Description
`db.wait.class`	The broader classification of wait events to identify the general type of resource contributing to the database load. For example, `class:oncpu` means the statement is actively running on the CPU, and `class:io` means that the statement is waiting for an input/output operation to complete.
`db.wait.event`	The wait state the sampled sessions were in. See DASH wait events for the full list of values.
`db.session.state`	The session state – `active` (executing a statement) or `idle in transaction` (waiting for the next command from the application while the transaction remains open).
`db.query.id`	The fingerprint of the normalized SQL text of the statement the sessions were running.
`db.query.normalized_text`	The normalized SQL text of the statement the sessions were running. DASH removes literal values so that it groups statements that differ only in their parameters.
`aws.auroradsql.session.role.arn`	The IAM role ARN assumed to connect to the Aurora DSQL cluster.

Dimension	Description
application.name	The application name that you set in connection parameters. You can override it at connect time. DASH includes this dimension only when you explicitly set it.

DASH wait events

Aurora DSQL sessions can experience the following wait events. The storage-related wait events – SequentialScanRead, ScatteredBatchRead, SingleRead, UniqueConstraintCheck, and FkExistenceCheck – represent communication between the query processing layer and the storage layer, and Commit represents communication with the commit service. This list might expand over time as Aurora DSQL identifies new wait events.

Wait event	Wait class	Description
OnCpu	class:oncpu	The session isn't waiting for external input and is actively executing on CPU – parsing, planning, evaluating expressions, or processing results.
ClientRead	class:client	The session is idle within an open transaction, waiting for the application to send the next SQL statement or a commit/rollback command. Frequent or long ClientRead waits often indicate excessive application round-trips or transactions that you hold open longer than necessary.
ClientWrite	class:client	The database sends results to the application over the

Wait event	Wait class	Description
		network. High `ClientWrite` can indicate large result sets or network latency between the application and the database.
`SequentialScanRead`	`class:io`	The session is reading a contiguous range of keys from storage. This isn't necessarily a full table scan – it might cover a relatively small range of contiguous keys.
`ScatteredBatchRead`	`class:io`	The session is performing batched reads from storage, retrieving multiple non-contiguous keys in a single call to storage.
`SingleRead`	`class:io`	The session is reading a single tuple (point lookup) from storage. `ScatteredBatchRead` with a batch size of 1 largely replaces this event, which is uncommon in current Aurora DSQL versions.

Wait event	Wait class	Description
`UniqueConstraintCheck`	`class:io`	The session is validating unique key constraints, which requires storage reads to check for duplicates. This applies to both unique constraints on non-primary-key columns and primary key constraints during the insertion of new rows.
`FkExistenceCheck`	`class:io`	The session is validating that a referenced foreign key row exists, which requires reads to confirm the relationship.
`StartTransaction`	`class:io`	The session is preparing for the distributed transaction to begin.
`Commit`	`class:io`	The session has initiated a commit and is waiting for acknowledgment from the commit service. The response is either a success or an abort (serialization error); a `Commit` wait precedes both outcomes.
`PgSleep`	`class:timeout`	The session is sleeping because the application explicitly called `pg_sleep()`. This is an application-initiated wait, not a database-imposed one.

Accessing DASH data

You can access DASH data in three ways:

- **Amazon CloudWatch Database Insights** – A curated, no-code user interface (UI) for exploring database load and top SQL. This is the starting point for most investigations.
- **PromQL** – Query the underlying `db.active_sessions.avg` metric programmatically for integration with third-party monitoring tools or for interactive exploration.
- **Aurora DSQL system diagnostics AI skill** – An artificial intelligence (AI)-powered health-check agent that automatically analyzes DASH data, compares wait event distributions across time frames, and generates diagnostic reports.

All of these access paths read from the same DASH dataset.

Using CloudWatch Database Insights

Amazon CloudWatch Database Insights presents DASH data through a Aurora DSQL-specific curated dashboard. Your Aurora DSQL clusters appear automatically in Database Insights – you don't need any setup beyond creating the cluster and running transactions on it.

To view DASH data in Database Insights

1. Open the CloudWatch console and choose **Database Insights** in the left navigation pane.
2. In the **Fleet Health** view, locate your Aurora DSQL cluster in the list of **Database resources**. Alternatively, you can navigate directly to the **Database Instance** page and select your cluster from the left panel.
3. Choose the **DB Identifier** to open the **Database Instance Dashboard**.
4. Use the **DB Load** chart to view Average Active Sessions over time. The chart is a stacked visualization in which each colored band represents one wait event, so the total height shows how busy the cluster is and the bands show what sessions are spending their time on.
5. Use the **Slice By** control on the **DB Load** chart to toggle between **Wait Events** and **SQL Text**.
6. The **DB Load Analysis** section shows AAS by **Top Wait Events** and **Top SQL** ranked by their contribution to AAS.
7. Use the time range picker at the top of the page to focus on a specific monitoring window, such as the period of a reported slowdown.

Using PromQL

DASH exposes data as the CloudWatch metric `db.active_sessions.avg`, which you can query with PromQL in CloudWatch Query Studio.

The following examples work in Query Studio, where the active workspace already scopes results to your account and Region, and the time picker drives the evaluation window. Because the metric name contains dots, the examples use the PromQL quoted-name selector form, `{"db.active_sessions.avg"}`.

> **Note**
>
> All examples include an `@resource.aws.auroradsql.cluster_id` label filter to scope results to a single cluster. Replace *`cluster-id`* with your cluster identifier. If you have only one cluster, you can omit this filter or use the Query Studio UI filters instead. To discover all of the labels available on the metric in your environment, run `{"db.active_sessions.avg"}` as a series query and inspect the label set on each returned series.

Each `db.active_sessions.avg` time series represents the average active sessions (AAS) for one unique set of metric labels. When you group results by a subset of those labels, use `sum by` to add the AAS values for all time series in each group.

Database load by wait event

Returns average active sessions (AAS) grouped by wait event – the same AAS-by-wait-event view available in the Database Insights DB Load chart, but accessible programmatically.

```
sum by ("db.wait.event") (
  {
    "db.active_sessions.avg",
    "@resource.aws.auroradsql.cluster_id"="cluster-id"
  }
)
```

The result has one series per distinct `db.wait.event` value, showing the total AAS for that wait event.

Top SQL by average active sessions

Ranks SQL statements by their total AAS contribution to database load. This is the PromQL equivalent of the Database Insights **Top SQL** view.

```
topk(5,
  sum by ("db.query.normalized_text") (
    {
      "db.active_sessions.avg",
      "@resource.aws.auroradsql.cluster_id"="cluster-id"
    }
  )
)
```

The nested `sum by` expression combines all time series for the same normalized SQL text before `topk` ranks the statements.

To see the top SQL statement and wait event combinations by their contribution to load, add `db.wait.event` to the grouping clause. Because this ranks every combination together, the results can include several wait events for the same high-contributing statement:

```
topk(5,
  sum by ("db.query.normalized_text", "db.wait.event") (
    {
      "db.active_sessions.avg",
      "@resource.aws.auroradsql.cluster_id"="cluster-id"
    }
  )
)
```

SQL statements performing the most storage reads

Returns the top five SQL statements spending the most time on storage read operations, by filtering to the storage-read wait events.

```
topk(5,
  sum by ("db.query.normalized_text") (
    {
      "db.active_sessions.avg",
      "db.wait.event"=~"S.*Read|.*Check",
      "@resource.aws.auroradsql.cluster_id"="cluster-id"
    }
  )
```

```
)
```

The regular expression `S.*Read` matches the storage-read events whose names start with S and end with Read (`SequentialScanRead`, `ScatteredBatchRead`, `SingleRead`). The pattern includes `.*Check` explicitly because the `UniqueConstraintCheck` and `FkExistenceCheck` wait events don't match the first pattern but are also storage-read wait events.

Using the Aurora DSQL system diagnostics AI skill

The Aurora DSQL system diagnostics AI skill automates health-check analysis of your Aurora DSQL cluster by reading DASH data, comparing wait event distributions across time frames, and generating a diagnostic report. The skill is part of the databases-on-aws plugin in Agent Plugins for AWS in the awslabs/agent-plugins repository and works with any supported AI coding agent.

To analyze cluster health, issue a prompt such as:

"Check the performance of my Aurora DSQL cluster `cluster-id` in `us-east-1` and write me a markdown report."

The skill uses the CloudWatch Model Context Protocol (MCP) server to analyze the `db.active_sessions.avg` metric across a selection of time frames and returns a Markdown report. You can direct this performance comparison window in the prompt:

"Check performance for the last 4 hours and compare against last Monday."

When specific queries look problematic, the skill initiates a deeper SQL-focused diagnostics workflow and reports potential solutions for that statement. It decides whether to drill into a query based on the wait event shift it detects, so you don't need additional prompting.

Related resources

- Query metrics with PromQL – overview of PromQL support in CloudWatch.
- Running PromQL queries in Query Studio – the interactive query workbench in the CloudWatch console.
- AWS vended metrics in OpenTelemetry format – how AWS service metrics, including those for Aurora DSQL, are exposed as PromQL-queryable time series.
- Amazon CloudWatch Database Insights – the CloudWatch UI for monitoring database load and performance.

- Agent Plugins for AWS on the GitHub website – source repository for the `databases-on-aws` plugin containing the Aurora DSQL system diagnostics AI skill.

Logging Aurora DSQL operations using AWS CloudTrail

Amazon Aurora DSQL is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. There are two types of events in CloudTrail: management events and data events. Management events are emitted to audit AWS resource configuration changes. Data events capture the AWS resource usage typically in the service data plane.

CloudTrail captures all API calls for Aurora DSQL as events. Aurora DSQL records console activity as management events. It also captures authenticated connection attempts to clusters as data events.

Using the information collected by CloudTrail, you can determine the request that was made to Aurora DSQL, the IP address from which the request was made, when it was made, the user identity making the request, and additional details.

CloudTrail is enabled by default in your AWS account when you create the account and you have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for recording the **Event history**.

To create an ongoing record of events in your AWS account, including events for Aurora DSQL, create a trail or an AWS CloudTrail Lake event data store (a centralized storage and analysis solution for AWS CloudTrail events). For more information on creating trails, see Working with CloudTrail trails. To learn about setting up and managing event data stores, see CloudTrail Lake event data stores.

Aurora DSQL management events in CloudTrail

CloudTrail Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail captures management events in the **Event history**.

Amazon Aurora DSQL logs all Aurora DSQL control plane operations as management events. For a list of the Amazon Aurora DSQL control plane operations that Aurora DSQL logs to CloudTrail, see the Aurora DSQL API reference.

Control plane logs

Amazon Aurora DSQL logs the following Aurora DSQL control plane operations to CloudTrail as management events.

- CreateCluster
- DeleteCluster
- GetCluster
- GetVpcEndpointServiceName
- ListClusters
- ListTagsForResource
- TagResource
- UntagResource
- UpdateCluster

CDC stream logs

Amazon Aurora DSQL logs the following CDC stream operations to CloudTrail as management events. For more information about CDC streams, see Change data capture (CDC) streams.

- CreateStream
- DeleteStream
- GetStream
- ListStreams

Backup and restore logs

Amazon Aurora DSQL logs the following Aurora DSQL backup and restore operations to CloudTrail as management events.

- `StartBackupJob`
- `StopBackupJob`
- `GetBackupJob`
- `StartRestoreJob`

- `StopRestoreJob`
- `GetRestoreJob`

For more on protecting your Aurora DSQL clusters using AWS Backup, see Backup and restore for Amazon Aurora DSQL .

AWS KMS logs

Amazon Aurora DSQL logs the following AWS KMS operations to CloudTrail as management events.

- `GenerateDataKey`
- `Decrypt`

To learn more about how CloudTrail logs track requests that Aurora DSQL sends to AWS KMS on your behalf, see Monitoring Aurora DSQL interaction with AWS KMS.

Aurora DSQL data events in CloudTrail

CloudTrail Data events typically provide information about the resource operations performed on or in a resource. These are also used to capture the service's data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

For more information about how to log data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

For Aurora DSQL, CloudTrail captures any connection attempt made to an Aurora DSQL cluster as a data event. The following table lists the Aurora DSQL resource types for which you can log data events. The **Resource type (console)** column shows the value to choose from the **Resource type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Resource type (console)	resources.type value	Data APIs logged to CloudTrail
Amazon Aurora DSQL	`AWS::DSQL::Cluster`	• `DbConnect` • `DbConnectAdmin`

You can configure advanced event selectors to filter on the `eventName` and `resources.ARN` fields to log only filtered events. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*.

The following example shows how to use AWS CLI to configure `dsql-data-events-trail` to receive data events for Aurora DSQL.

```
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"] } ]}]'
```

Security in Amazon Aurora DSQL

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Aurora DSQL, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Aurora DSQL. The following topics show you how to configure Aurora DSQL to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Aurora DSQL resources.

Topics

- AWS managed policies for Amazon Aurora DSQL
- Data protection in Amazon Aurora DSQL
- Data encryption for Amazon Aurora DSQL
- Identity and access management for Aurora DSQL
- Resource-based policies for Aurora DSQL
- Using service-linked roles in Aurora DSQL
- Using IAM condition keys with Amazon Aurora DSQL
- Incident response in Amazon Aurora DSQL
- Compliance validation for Amazon Aurora DSQL
- Resilience in Amazon Aurora DSQL

- Infrastructure Security in Amazon Aurora DSQL
- Configuration and vulnerability analysis in Amazon Aurora DSQL
- Cross-service confused deputy prevention
- Security best practices for Aurora DSQL

AWS managed policies for Amazon Aurora DSQL

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: AmazonAuroraDSQLFullAccess

You can attach `AmazonAuroraDSQLFullAccess` to your users, groups, and roles.

This policy grants permissions that allows full administrative access to Aurora DSQL. Principals with these permissions can:

- Create, delete, and update Aurora DSQL clusters, including multi-Region clusters
- Manage cluster inline policies (create, view, update, and delete policies)

- Create, update, delete, and manage CDC streams for clusters
- Add and remove tags from clusters and CDC streams
- List clusters and view information about individual clusters
- See tags attached to Aurora DSQL clusters
- Connect to the database as any user, including admin
- Pass IAM roles to CDC streams for data delivery to target destinations
- Perform backup and restore operations for Aurora DSQL clusters, including starting, stopping, and monitoring backup and restore jobs
- Use customer-managed AWS KMS keys for cluster encryption
- View any metrics from CloudWatch for their account
- Use AWS Fault Injection Service (AWS FIS) to inject failures into Aurora DSQL clusters for fault tolerance testing
- Create service-linked roles for the `dsql.amazonaws.com` service, which is required for creating clusters

Permissions details

This policy includes the following permissions.

- `dsql` – grants principals full access to Aurora DSQL.
- `cloudwatch` – grants permission to publish metric data points to Amazon CloudWatch.
- `iam` – grants permission to create a service-linked role and pass roles to CDC streams for data delivery.
- `backup and restore` – grants permissions to start, stop, and monitor backup and restore jobs for Aurora DSQL clusters.
- `kms` – grants permissions required to validate access to customer-managed keys used for Aurora DSQL cluster encryption when creating, updating, or connecting to clusters.
- `fis` – grants permissions to use AWS Fault Injection Service (AWS FIS) to inject failures into Aurora DSQL clusters for fault tolerance testing.

You can find the `AmazonAuroraDSQLFullAccess` policy in the IAM console and in the AWS Managed Policy Reference Guide.

AWS managed policy: AmazonAuroraDSQLReadOnlyAccess

You can attach `AmazonAuroraDSQLReadOnlyAccess` to your users, groups, and roles.

Allows read access to Aurora DSQL. Principals with these permissions can list clusters and view information about individual clusters. They can see the tags attached to Aurora DSQL clusters and CDC streams, view information about individual CDC streams, and view cluster inline policies. They can retrieve and see any metrics from CloudWatch on your account.

Permissions details

This policy includes the following permissions.

- `dsql` – grants read only permissions to all resources in Aurora DSQL.
- `cloudwatch` – grants permission to retrieve batch amounts of CloudWatch metric data and perform metric math on retrieved data
- `iam` – grants permissions to pass roles to CDC streams for data delivery configuration.

You can find the `AmazonAuroraDSQLReadOnlyAccess` policy in the IAM console and the AWS Managed Policy Reference Guide.

AWS managed policy: AmazonAuroraDSQLConsoleFullAccess

You can attach `AmazonAuroraDSQLConsoleFullAccess` to your users, groups, and roles.

Allows full administrative access to Amazon Aurora DSQL via the AWS Management Console. Principals with these permissions can:

- Create, delete, and update Aurora DSQL clusters, including multi-Region clusters, with the console
- Manage cluster inline policies through the console (create, view, update, and delete policies)
- List clusters and view information about individual clusters
- See tags on any resource on your account

- Connect to the database as any user, including the admin
- Perform backup and restore operations for Aurora DSQL clusters, including starting, stopping, and monitoring backup and restore jobs
- Use customer-managed AWS KMS keys for cluster encryption
- Launch AWS CloudShell from the AWS Management Console
- View any metrics from CloudWatch on your account
- Use AWS Fault Injection Service (AWS FIS) to inject failures into Aurora DSQL clusters for fault tolerance testing
- Create service linked roles for the `dsql.amazonaws.com` service, which is required for creating clusters

You can find the `AmazonAuroraDSQLConsoleFullAccess` policy on the IAM console and AmazonAuroraDSQLConsoleFullAccess in the AWS Managed Policy Reference Guide.

Permissions details

This policy includes the following permissions.

- `dsql` – grants full administrative permissions to all resources in Aurora DSQL via the AWS Management Console.
- `cloudwatch` – grants permission to retrieve batch amounts of CloudWatch metric data and perform metric math on retrieved data.
- `tag` – grants permission to returns tag keys and values currently in use in the specified AWS Region for the calling account.
- `backup and restore` – grants permissions to start, stop, and monitor backup and restore jobs for Aurora DSQL clusters.
- `kms` – grants permissions required to validate access to customer-managed keys used for Aurora DSQL cluster encryption when creating, updating, or connecting to clusters.
- `cloudshell` – grants permissions to launch AWS CloudShell to interact with Aurora DSQL.
- `ec2` – grants permission to view Amazon VPC endpoint information needed for Aurora DSQL connections.

- `fis` – grants permissions to use AWS FIS to inject failures into Aurora DSQL clusters for fault tolerance testing.
- `access-analyzer:ValidatePolicy` grants permission for the linter in the policy editor, which provides real-time feedback about errors, warnings, and security issues in the current policy.
- `fis` – grants permissions to use AWS Fault Injection Service (AWS FIS) to inject failures into Aurora DSQL clusters for fault tolerance testing.

You can find the `AmazonAuroraDSQLConsoleFullAccess` policy in the IAM console and the AWS Managed Policy Reference Guide.

AWS managed policy: AuroraDsqlServiceLinkedRolePolicy

You can't attach AuroraDsqlServiceLinkedRolePolicy to your IAM entities. Aurora DSQL attaches this policy to the service-linked role that it uses to access resources in your account.

With this policy, Aurora DSQL can do the following in your account:

- `cloudwatch` – Publish metrics to the `AWS/AuroraDSQL` and `AWS/Usage` CloudWatch namespaces.
- `logs` – Create the `/aws/dsql/normalized-sql-text` log group and its log streams, write log events to it, and set its retention. An upcoming Aurora DSQL feature uses these permissions to deliver normalized SQL text to your account. These permissions apply only to that log group in your own account, and they are limited to writing log data.

You can find the `AuroraDsqlServiceLinkedRolePolicy` policy on the IAM console and AuroraDsqlServiceLinkedRolePolicy in the AWS Managed Policy Reference Guide.

Aurora DSQL updates to AWS managed policies

View details about updates to AWS managed policies for Aurora DSQL since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Aurora DSQL Document history page.

Change	Description	Date
AuroraDsqlServiceLinkedRolePolicy update	Added the `logs:CreateLogGroup`, `logs:CreateLogStream`, `logs:PutLogEvents`, and `logs:PutRetentionPolicy` permissions on the `/aws/dsql/normalized-sql-text` log group and its log streams in your own account. With these permissions, an upcoming Aurora DSQL feature can deliver normalized SQL text to your account.	November 11, 2026
AmazonAuroraDSQLFullAccess and AmazonAuroraDSQLConsoleFullAccess update	Added support for AWS Fault Injection Service (AWS FIS) integration with Aurora DSQL. This allows you to inject failures into single-Region and multi-Region Aurora DSQL clusters to test fault tolerance of your applications. You can create experiment templates in the AWS FIS console to define failure scenarios and target specific Aurora DSQL clusters for testing. For more on these policies, see AmazonAuroraDSQLFullAccess and AmazonAuroraDSQLConsoleFullAccess.	August 19, 2025

Change	Description	Date
AmazonAuroraDSQLFullAccess, AmazonAuroraDSQLReadOnlyAccess, and AmazonAuroraDSQLConsoleFullAccess update	Added resource-based policy (RBP) support with new permissions: `PutClusterPolicy` , `GetClusterPolicy` , and `DeleteClusterPolicy` . These permissions allow managing inline policies attached to Aurora DSQL clusters for fine-grained access control. For more information, see AmazonAuroraDSQLFullAccess, AmazonAuroraDSQLReadOnlyAccess, and AmazonAuroraDSQLConsoleFullAccess.	October 15, 2025
AmazonAuroraDSQLFullAccess update	Adds the capability to perform backup and restore operations for Aurora DSQL clusters, including starting, stopping, and monitoring jobs. It also adds the capability to use customer-managed KMS keys for cluster encryption. For more information, see AmazonAuroraDSQLFullAccess and Using service-linked roles in Aurora DSQL .	May 21, 2025

Change	Description	Date
AmazonAuroraDSQLConsoleFullAccess update	Adds the capability to perform backup and restore operations for Aurora DSQL clusters through the AWS Console Home. This includes starting, stopping, and monitoring jobs. It also supports using customer-managed KMS keys for cluster encryption and launching AWS CloudShell. For more information, see AmazonAuroraDSQLConsoleFullAccess and Using service-linked roles in Aurora DSQL .	May 21, 2025

Change	Description	Date
AmazonAuroraDSQLFullAccess update	The policy adds four new permissions to create and manage database clusters across multiple AWS Regions: `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , and `RemovePeerCluster` . These permissions include resource-level controls and condition keys so you can control which clusters users you can modify. The policy also adds the `GetVpcEndpointServiceName` permission to help you connect to your Aurora DSQL clusters through AWS PrivateLink. For more information, see For more information, see AmazonAuroraDSQLFullAccess and Using service-linked roles in Aurora DSQL .	May 13, 2025

Change	Description	Date
AmazonAuroraDSQLReadOnlyAccess update	Includes the ability to determine the correct VPC endpoint service name when connecting to your Aurora DSQL clusters through AWS PrivateLink Aurora DSQL creates unique endpoints per cell, so this API helps ensure you can identify the correct endpoint for your cluster and avoid connection errors. For more information, see AmazonAuroraDSQLReadOnlyAccess and Using service-linked roles in Aurora DSQL .	May 13, 2025

Change	Description	Date
AmazonAuroraDSQLConsoleFullAccess update	Adds new permissions to Aurora DSQL to support multi-Region cluster management and VPC endpoint connection. The new permissions include: `PutMultiRegionProperties` `PutWitnessRegion` `AddPeerCluster` `RemovePeerCluster` `GetVpcEndpointServiceName` For more information, see AmazonAuroraDSQLConsoleFullAccess and Using service-linked roles in Aurora DSQL .	May 13, 2025
AuroraDsqlServiceLinkedRolePolicy update	Adds the ability to publish metrics to the `AWS/AuroraDSQL` and `AWS/Usage` `CloudWatch` namespaces to the policy. This allows the associated service or role to emit more comprehensive usage and performance data to your CloudWatch environment. For more information, see AuroraDsqlServiceLinkedRolePolicy and Using service-linked roles in Aurora DSQL.	May 8, 2025

Change	Description	Date
Page created	Started tracking AWS managed policies related to Amazon Aurora DSQL	December 3, 2024

Data protection in Amazon Aurora DSQL

The shared responsibility model applies to data protection in . As described in this model, is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the Shared Responsibility Model and GDPR blog post on the *Security Blog*.

For data protection purposes, we recommend that you protect credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management. That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using trails to capture activities, see Working with trails in the *User Guide*.
- Use encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.

We strongly recommend that you never put confidential or sensitive information, such as your customers email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with or other using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption

Amazon Aurora DSQL provides a highly durable storage infrastructure designed for mission-critical and primary data storage. Data is redundantly stored on multiple devices across multiple facilities in an Aurora DSQL Region.

Encryption in transit

By default, encryption in transit is configured for you. Aurora DSQL uses TLS to encrypt all traffic between your SQL client and Aurora DSQL.

Encryption and signing of data in transit between AWS CLI, SDK, or API clients and Aurora DSQL endpoints:

- Aurora DSQL provides HTTPS endpoints for encrypting data in transit.
- To protect the integrity of API requests to Aurora DSQL, API calls must be signed by the caller. Calls are signed by an X.509 certificate or the customer's AWS secret access key according to the Signature Version 4 Signing Process (Sigv4). For more information, see Signature Version 4 Signing Process in the *AWS General Reference*.
- Use the AWS CLI or one of the AWS SDKs to make requests to AWS. These tools automatically sign the requests for you with the access key that you specify when you configure the tools.

FIPS compliance

Aurora DSQL dataplane endpoints (cluster endpoints used for database connections) use FIPS 140-2 validated cryptographic modules by default. No separate FIPS endpoints are required for cluster connections.

For control plane operations, Aurora DSQL provides dedicated FIPS endpoints in supported regions. For more information about control plane FIPS endpoints, see Aurora DSQL endpoints and quotas in the *AWS General Reference*.

For encryption at rest, see Encryption at rest in Aurora DSQL.

Inter-network traffic privacy

Connections are protected both between Aurora DSQL and on-premises applications and between Aurora DSQL and other AWS resources within the same AWS Region.

You have two connectivity options between your private network and AWS:

- An AWS Site-to-Site VPN connection. For more information, see What is AWS Site-to-Site VPN?
- An Direct Connect connection. For more information, see What is Direct Connect?

You get access to Aurora DSQL through the network by using AWS-published API operations. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Data Protection in witness Regions

When you create a multi-Region cluster, a witness Region helps enable automated failure recovery by participating in synchronous replication of encrypted transactions. If a peered cluster becomes unavailable, the witness Region remains available to validate and process database writes, ensuring no loss of availability.

Witness Regions protect and secure your data through these design features:

- The witness Region receives and stores only encrypted transaction logs. It never hosts, stores or transmits your encryption keys.
- The witness Region focuses soley on write transaction logging and quorum functions. It can't read your data by design.
- The witness Region operates without cluster connection endpoints or query processors. This prevents user database access.

For more information on witness Regions, see Configuring multi-Region clusters.

Configuring SSL/TLS certificates for Aurora DSQL connections

Aurora DSQL requires all connections to use Transport Layer Security (TLS) encryption. To establish secure connections, your client system must trust the Amazon Root Certificate Authority (Amazon Root CA 1). This certificate is pre-installed on many operating systems. This section provides instructions for verifying the pre-installed Amazon Root CA 1 certificate on various operating systems, and guides you through the process of manually installing the certificate if it is not already present.

We recommend using PostgreSQL version 17.

> **Important**
>
> For production environments, we recommend using `verify-full` SSL mode to ensure the highest level of connection security. This mode verifies that the server certificate is signed by a trusted certificate authority and that the server hostname matches the certificate.

Verifying pre-installed certificates

In most operating systems, **Amazon Root CA 1** is already pre-installed. To validate this, you can follow the steps below.

Linux (RedHat/CentOS/Fedora)

Run the following command in your terminal:

```
trust list | grep "Amazon Root CA 1"
```

If the certificate is installed, you see the following output:

```
label: Amazon Root CA 1
```

macOS

1. Open Spotlight Search (**Command** + **Space**)
2. Search for **Keychain Access**
3. Select **System Roots** under **System Keychains**
4. Look for **Amazon Root CA 1** in the certificate list

Windows

> **Note**
>
> Due to a known issue with the psql Windows client, using system root certificates (`sslrootcert=system`) may return the following error: `SSL error: unregistered`

scheme. You can follow the Connecting from Windows as an alternative way to connect to your cluster using SSL.

If **Amazon Root CA 1** is not installed in your operating system, follow the steps below.

Installing certificates

If the `Amazon Root CA 1` certificate is not pre-installed on your operating system, you will need to manually install it in order to establish secure connections to your Aurora DSQL cluster.

Linux certificate installation

Follow these steps to install the Amazon Root CA certificate on Linux systems.

1. Download the Root Certificate:

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. Copy the certificate to the trust store:

```
sudo cp ./AmazonRootCA1.pem /etc/pki/ca-trust/source/anchors/
```

3. Update the CA trust store:

```
sudo update-ca-trust
```

4. Verify the installation:

```
trust list | grep "Amazon Root CA 1"
```

macOS certificate installation

These certificate installation steps are optional. The Linux certificate installation also work for a macOS.

1. Download the Root Certificate:

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. Add the certificate to the System keychain:

```
sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain
 AmazonRootCA1.pem
```

3. Verify the installation:

```
security find-certificate -a -c "Amazon Root CA 1" -p /Library/Keychains/
System.keychain
```

Connecting with SSL/TLS verification

Before configuring SSL/TLS certificates for secure connections to your Aurora DSQL cluster, ensure you have the following prerequisites.

- PostgreSQL version 17 installed
- AWS CLI configured with appropriate credentials
- Aurora DSQL cluster endpoint information

Connecting from Linux

1. Generate and set the authentication token:

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --hostname your-cluster-endpoint)
```

2. Connect using system certificates (if pre-installed):

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. Or, connect using a downloaded certificate:

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
```

```
--host your-cluster-endpoint
```

> **Note**
>
> For more on PGSSLMODE settings, see sslmode in the PostgresQL 17 Database Connection Control Functions documentation.

Connecting from macOS

1. Generate and set the authentication token:

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-cluster-region --hostname your-cluster-endpoint)
```

2. Connect using system certificates (if pre-installed):

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. Or, download the root certificate and save it as `root.pem` (if certificate is not pre-installed)

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

4. Connect using psql:

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

Connecting from Windows

Using Command Prompt

1. Generate the authentication token:

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

2. Set the password environment variable:

```
set "PGPASSWORD=token-from-above"
```

3. Set SSL configuration:

```
set PGSSLROOTCERT=C:\full\path\to\root.pem
set PGSSLMODE=verify-full
```

4. Connect to the database:

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres ^
--username admin ^
--host your-cluster-endpoint
```

Using PowerShell

1. Generate and set the authentication token:

```
$env:PGPASSWORD = (aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --expires-in=3600 --hostname=your-cluster-endpoint)
```

2. Set SSL configuration:

```
$env:PGSSLROOTCERT='C:\full\path\to\root.pem'
$env:PGSSLMODE='verify-full'
```

3. Connect to the database:

```
 "C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres `
```

```
--username admin `
--host your-cluster-endpoint
```

Additional resources

- PostgreSQL SSL documentation
- Amazon Trust Services

Data encryption for Amazon Aurora DSQL

Amazon Aurora DSQL encrypts all user data at rest. For enhanced security, this encryption uses AWS Key Management Service (AWS KMS). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. Encryption at rest helps you:

- Reduce the operational burden of protecting sensitive data
- Build security-sensitive applications that meet strict encryption compliance and regulatory requirements
- Add an extra layer of data protection by always securing your data in an encrypted cluster
- Comply with organizational policies, industry or government regulations, and compliance requirements

With Aurora DSQL, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements. The following sections explain how to configure encryption for new and existing Aurora DSQL databases and manage your encryption keys.

Topics

- KMS key types for Aurora DSQL
- Encryption at rest in Aurora DSQL
- Using AWS KMS and data keys with Aurora DSQL
- Authorizing use of your AWS KMS key for Aurora DSQL
- Aurora DSQL encryption context
- Monitoring Aurora DSQL interaction with AWS KMS
- Creating an encrypted Aurora DSQL cluster
- Removing or updating a key for your Aurora DSQL cluster

- Considerations for encryption with Aurora DSQL

KMS key types for Aurora DSQL

Aurora DSQL integrates with AWS KMS to manage the encryption keys for your clusters. To learn more about key types and states, see AWS Key Management Service concepts in the *AWS Key Management Service Developer Guide*. When you create a new cluster, you can choose from the following KMS key types to encrypt your cluster:

AWS owned key

Default encryption type. Aurora DSQL owns the key at no additional charge to you. Amazon Aurora DSQL transparently decrypts cluster data when you access an encrypted cluster. You don't need to change your code or applications to use or manage encrypted clusters, and all Aurora DSQL queries work with your encrypted data.

Customer managed key

You create, own, and manage the key in your AWS account. You have full control over the KMS key. AWS KMS charges apply.

Encryption at rest using the AWS owned key is available at no additional charge. However, AWS KMS charges apply for customer managed keys. For more information, see the AWS KMS Pricing page.

You can switch between these key types at any time. For more information about key types, see Customer managed keys and AWS owned keys in the *AWS Key Management Service Developer Guide*.

> **Note**
>
> Aurora DSQL encryption at rest is available in all AWS Regions where Aurora DSQL is available.

Encryption at rest in Aurora DSQL

Amazon Aurora DSQL uses 256-bit Advanced Encryption Standard (AES-256) to encrypt your data at rest. This encryption helps protect your data from unauthorized access to the underlying

storage. AWS KMS manages the encryption keys for your clusters. You can use the default AWS owned keys, or choose to use your own AWS KMS Customer managed keys. To learn more about specifying and managing keys for your Aurora DSQL clusters, see Creating an encrypted Aurora DSQL cluster and Removing or updating a key for your Aurora DSQL cluster.

Topics

- AWS owned keys
- Customer managed keys

AWS owned keys

Aurora DSQL encrypts all clusters by default with AWS owned keys. These keys are free to use and rotate annually to protect your account resources. You don't need to view, manage, use, or audit these keys, so there's no action required for data protection. For more information about AWS owned keys, see AWS owned keys in the *AWS Key Management Service Developer Guide*.

Customer managed keys

You create, own, and manage customer managed keys in your AWS account. You have full control over these KMS keys, including their policies, encryption material, tags, and aliases. For more information about managing permissions, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

When you specify a customer managed key for cluster-level encryption, Aurora DSQL encrypts the cluster and all its regional data with that key. To prevent data loss and maintain cluster access, Aurora DSQL needs access to your encryption key. If you disable your customer managed key, schedule your key for deletion, or have a policy that restricts your service access, the encryption status for your cluster changes to `KMS_KEY_INACCESSIBLE`. When Aurora DSQL can't access the key, users can't connect to the cluster, the encryption status for the cluster changes to `KMS_KEY_INACCESSIBLE`, and the service loses access to the cluster data.

For multi-Region clusters, customers can configure each region's AWS KMS encryption key separately, and each regional cluster uses its own cluster-level encryption key. If Aurora DSQL can't access the encryption key for a peer in a multi-Region cluster, the status for that peer becomes `KMS_KEY_INACCESSIBLE` and it becomes unavailable for read and write operations. Other peers continue normal operations.

Note

If Aurora DSQL can't access your customer managed key, your cluster encryption status changes to `KMS_KEY_INACCESSIBLE`. After you restore key access, service will automatically detect the restoration within 15 minutes. For more information, see Cluster idling.
For multi-Region clusters, if key access is lost for an extended time, the cluster restoration time depends on how much data was written while the key was inaccessible.

Using AWS KMS and data keys with Aurora DSQL

The Aurora DSQL encryption at rest feature uses an AWS KMS key and a hierarchy of data keys to protect your cluster data.

We recommend that you plan your encryption strategy before implementing your cluster in Aurora DSQL. If you store sensitive or confidential data in Aurora DSQL, consider including client-side encryption in your plan. This way you can encrypt data as close as possible to its origin, and ensure its protection throughout its lifecycle.

Topics

- Using AWS KMS keys with Aurora DSQL
- Using cluster keys with Aurora DSQL
- Cluster key caching

Using AWS KMS keys with Aurora DSQL

Encryption at rest protects your Aurora DSQL cluster under an AWS KMS key. By default, Aurora DSQL uses an AWS owned key, a multi-tenant encryption key that is created and managed in a Aurora DSQL service account. But you can encrypt your Aurora DSQL clusters under a customer managed key in your AWS account. You can select a different KMS key for each cluster, even if it participates in a multi-Region setup.

You select the KMS key for a cluster when you create or update the cluster. You can change the KMS key for a cluster at any time, either in the Aurora DSQL console or by using the `UpdateCluster` operation. The process of switching keys doesn't require downtime or degrade service.

> **Important**
>
> Aurora DSQL supports only symmetric KMS keys. You can't use an asymmetric KMS key to encrypt your Aurora DSQL clusters.

A customer managed key provides the following benefits.

- You create and manage the KMS key, including setting the key policies and IAM policies to control access to the KMS key. You can enable and disable the KMS key, enable and disable automatic key rotation, and delete the KMS key when it is no longer in use.
- You can use a customer managed key with imported key material or a customer managed key in a custom key store that you own and manage.
- You can audit the encryption and decryption of your Aurora DSQL cluster by examining the Aurora DSQL API calls to AWS KMS in AWS CloudTrail logs.

However, the AWS owned key is free of charge and its use doesn't count against AWS KMS resource or request quotas. Customer managed keys incur a charge for each API call and AWS KMS quotas apply to these keys.

Using cluster keys with Aurora DSQL

Aurora DSQL uses the AWS KMS key for the cluster to generate and encrypt a unique data key for the cluster, known as the **cluster key**.

The cluster key is used as a key encryption key. Aurora DSQL uses this cluster key to protect data encryption keys that are used to encrypt the cluster data. Aurora DSQL generates a unique data encryption key for each underlying structure in a cluster, but multiple cluster items might be protected by the same data encryption key.

To decrypt the cluster key, Aurora DSQL sends a request to AWS KMS when you first access an encrypted cluster. To keep the cluster available, Aurora DSQL periodically verifies decrypt access to the KMS key, even when you're not actively accessing the cluster.

Aurora DSQL stores and uses the cluster key and data encryption keys outside of AWS KMS. It protects all keys with Advanced Encryption Standard (AES) encryption and 256-bit encryption keys. Then, it stores the encrypted keys with the encrypted data so they are available to decrypt the cluster data on demand.

If you change the KMS key for your cluster, Aurora DSQL re-encrypts the existing cluster key with the new KMS key.

Cluster key caching

To avoid calling AWS KMS for every Aurora DSQL operation, Aurora DSQL caches the plaintext cluster keys for each caller in memory. If Aurora DSQL gets a request for the cached cluster key after 15 minutes of inactivity, it sends a new request to AWS KMS to decrypt the cluster key. This call will capture any changes made to the access policies of the AWS KMS key in AWS KMS or AWS Identity and Access Management (IAM) after the last request to decrypt the cluster key.

Authorizing use of your AWS KMS key for Aurora DSQL

If you use a customer managed key in your account to protect your Aurora DSQL cluster, the policies on that key must give Aurora DSQL permission to use it on your behalf.

You have full control over the policies on a customer managed key. Aurora DSQL does not need additional authorization to use the default AWS owned key to protect the Aurora DSQL clusters in your AWS account.

Key policy for a customer managed key

When you select a customer managed key to protect a Aurora DSQL cluster, Aurora DSQL needs permission to use the AWS KMS key on behalf of the principal who makes the selection. That principal, a user or role, must have the permissions on the AWS KMS key that Aurora DSQL requires. You can provide these permissions in a key policy, or an IAM policy.

At a minimum, Aurora DSQL requires the following permissions on a customer managed key:

- `kms:Encrypt`
- `kms:Decrypt`
- `kms:ReEncrypt*` (for kms:ReEncryptFrom and kms:ReEncryptTo)
- `kms:GenerateDataKey`
- `kms:DescribeKey`

For example, the following example key policy provides only the required permissions. The policy has the following effects:

- Allows Aurora DSQL to use the AWS KMS key in cryptographic operations, but only when it is acting on behalf of principals in the account who have permission to use Aurora DSQL. If the principals specified in the policy statement don't have permission to use Aurora DSQL, the call fails, even when it comes from the Aurora DSQL service.
- The `kms:ViaService` condition key allows the permissions only when the request comes from Aurora DSQL on behalf of the principals listed in the policy statement. These principals can't call these operations directly.

Before using an example key policy, replace the example principals with actual principals from your AWS account.

```
{
  "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 encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

Aurora DSQL uses the same encryption context in all AWS KMS cryptographic operations. If you use a customer managed key to protect your Aurora DSQL cluster, you can use the encryption context to identify use of the AWS KMS key in audit records and logs. It also appears in plaintext in logs such as those in AWS CloudTrail.

The encryption context can also be used as a condition for authorhorization in policies.

In its requests to AWS KMS, Aurora DSQL uses an encryption context with a key-value pair:

```
"encryptionContext": {
  "aws:dsql:ClusterId": "w4abucpbwuxx"
},
```

The key–value pair identifies the cluster that Aurora DSQL is encrypting. The key is `aws:dsql:ClusterId`. The value is the identifier of the cluster.

Monitoring Aurora DSQL interaction with AWS KMS

If you use a customer managed key to protect your Aurora DSQL clusters, you can use AWS CloudTrail logs to track the requests that Aurora DSQL sends to AWS KMS on your behalf.

Expand the following sections to learn how Aurora DSQL uses the AWS KMS operations `GenerateDataKey` and `Decrypt`.

GenerateDataKey

When you enable encryption at rest on a cluster, Aurora DSQL creates a unique cluster key. It sends a `GenerateDataKey` request to AWS KMS that specifies the AWS KMS key for the cluster.

The event that records the `GenerateDataKey` operation is similar to the following example event. The user is the Aurora DSQL service account. The parameters include the Amazon Resource Name (ARN) of the AWS KMS key, a key specifier that requires a 256-bit key, and the encryption context that identifies the cluster.


```
{
    "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": "AWS Internal",
    "eventCategory": "Management"
}
```

Decrypt

When you access an encrypted Aurora DSQL cluster, Aurora DSQL needs to decrypt the cluster key so that it can decrypt the keys below it in the hierarchy. It then decrypts the data in the cluster. To decrypt the cluster key, Aurora DSQL sends a `Decrypt` request to AWS KMS that specifies the AWS KMS key for the cluster.

The event that records the `Decrypt` operation is similar to the following example event. The user is the principal in your AWS account who is accessing the cluster. The parameters include the encrypted cluster key (as a ciphertext blob) and the encryption context that identifies the cluster. AWS KMS derives the ID of the AWS KMS key from the ciphertext.

```
{
  "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": "AWS Internal",
  "eventCategory": "Management"
}
```


Creating an encrypted Aurora DSQL cluster

All Aurora DSQL clusters are encrypted at rest. By default, clusters use an AWS owned key at no cost, or you can specify a custom AWS KMS key. Follow these steps to create your encrypted cluster from either the AWS Management Console or the AWS CLI.

Console

To create an encrypted cluster in the AWS Management Console

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. In the navigation pane, choose **Clusters**.
3. Choose **Create cluster**.
4. In **Cluster topology**, choose **Single Region**.
5. In **Cluster configuration**, choose **Advanced configuration**.
6. In **Cluster settings**, under **Cluster encryption**, choose one of the following options:
   - Accept the default settings to encrypt with an AWS owned key at no additional cost.

- Select **Customize encryption settings (advanced)** to specify a custom KMS key. Then, search for or enter the ID or alias of your KMS key. Alternatively, choose **Create an AWS KMS key** to create a new key in the AWS KMS console.

7. Choose **Create cluster**.

For a multi-Region cluster, in **Cluster topology**, choose **2 active Regions + Witness**. Then, in **Encryption key**, choose the encryption settings for each Region.

To confirm the encryption type, on the cluster details page, choose the **Cluster settings** tab. Under **Encryption**, **Cluster KMS key** shows **Aurora DSQL default key** for clusters that use AWS owned keys, or the key ID for other encryption types.

> **Note**
>
> If you select to own and manage your own key, make sure you set the KMS key policy appropriately. For examples and more information, see the section called “Key policy for a customer managed key”.

CLI

To create a cluster that's encrypted with the default AWS owned key

- Use the following command to create an Aurora DSQL cluster.

```
aws dsql create-cluster
```

As shown in the following encryption details, the encryption status for the cluster is enabled by default, and the default encryption type is AWS owned key. The cluster is now encrypted with the default AWS owned key in the Aurora DSQL service account.

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

To create a cluster that's encrypted with your customer managed key

- Use the following command to create an Aurora DSQL cluster, replacing the key ID in red text with the ID of your customer managed key.

```
aws dsql create-cluster \
--kms-encryption-key d41d8cd98f00b204e9800998ecf8427e
```

As shown in the following encryption details, the encryption status for the cluster is enabled by default, and the encryption type is customer managed KMS key. The cluster is now encrypted with your key.

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:kms:us-east-1:111122223333:key/
d41d8cd98f00b204e9800998ecf8427e",
  "encryptionStatus" : "ENABLED"
}
```

Removing or updating a key for your Aurora DSQL cluster

You can use the AWS Management Console or the AWS CLI to update or remove the encryption keys on existing clusters in Amazon Aurora DSQL. If you remove a key without replacing it, Aurora DSQL uses the default AWS owned key. Follow these steps to update the encryption keys of an existing cluster from the Aurora DSQL console or the AWS CLI.

Console

To update or remove an encryption key in the AWS Management Console

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. In the navigation pane on the left side of the console, choose **Clusters**.
3. From the list view, find and select row of the cluster that you want to update.
4. Select the **Actions** menu and then choose **Modify**.
5. In the **Cluster encryption settings**, choose one of the following options to modify your encryption settings.

- If you want to switch from a custom key to an AWS owned key, de-select the **Customize encryption settings (advanced)** option. The default settings will apply and encrypt your cluster with an AWS owned key at no cost.
- If you want to switch from one custom KMS key to another or from an AWS owned key to a KMS key, select the **Customize encryption settings (advanced)** option if it's not already selected. Then, search for and select the ID or alias of the key you want to use. Alternatively, choose **Create an AWS KMS key** to create a new key in the AWS KMS Console.

6. Choose **Save**.

CLI

The following examples show how to use the AWS CLI to update an encrypted cluster.

To update an encrypted cluster with the default AWS owned key

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key "AWS_OWNED_KMS_KEY"
```

The `EncryptionStatus` of the cluster description is set to `ENABLED` and the `EncryptionType` is `AWS_OWNED_KMS_KEY`.

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

This cluster is now encrypted using the default AWS owned key in the Aurora DSQL service account.

To update an encrypted cluster with a customer managed key for Aurora DSQL

Update the encrypted cluster, as in the following example:

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234a1b234
```

The `EncryptionStatus` of the cluster description transitions to `UPDATING` and the `EncryptionType` is `CUSTOMER_MANAGED_KMS_KEY`. After Aurora DSQL finishes propagating the new key through the platform, the encryption status will be transitioned to `ENABLED`

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:us-east-1:kms:key/abcd1234-abcd-1234-a123-ab1234a1b234",
  "encryptionStatus" : "ENABLED"
}
```

Note

If you select to own and manage your own key, make sure you set the KMS key policy appropriately. For examples and more information, see the section called “Key policy for a customer managed key”.

Considerations for encryption with Aurora DSQL

- Aurora DSQL encrypts all cluster data at rest. You can't disable this encryption or encrypt only some items in a cluster.
- AWS Backup encrypts your backups and any clusters restored from these backups. You can encrypt your backup data in AWS Backup using either the AWS owned key or a customer managed key.
- The following data protection states are enabled for Aurora DSQL:
  - **Data at rest** - Aurora DSQL encrypts all static data on persistent storage media
  - **Data in transit** - Aurora DSQL encrypts all communications using Transport Layer Security (TLS) by default

- When you transition to a different key, we recommend that you keep the original key enabled until the transition is complete. AWS needs the original key to decrypt data before it encrypts your data with the new key. The process is complete when the cluster's `encryptionStatus` is `ENABLED` and you see the `kmsKeyArn` of the new customer managed key.
- When you disable your Customer Managed Key or revoke access for Aurora DSQL to use your key, your cluster will go into `IDLE` state.
- The AWS Management Console and Amazon Aurora DSQL API use different terms for encryption types:
  - AWS Console – In the console, you'll see `KMS` when using a Customer managed key and `DEFAULT` when using an AWS owned key.
  - API – The Amazon Aurora DSQL API uses `CUSTOMER_MANAGED_KMS_KEY` for customer managed keys, and `AWS_OWNED_KMS_KEY` for AWS owned keys.
- If you don't specify an encryption key during cluster creation, Aurora DSQL automatically encrypts your data using the AWS owned key.
- You can switch between an AWS owned key and a Customer managed key at any time. Make this change using the AWS Management Console, AWS CLI, or the Amazon Aurora DSQL API.

Identity and access management for Aurora DSQL

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Aurora DSQL resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Aurora DSQL works with IAM
- Identity-based policy examples for Amazon Aurora DSQL
- Troubleshooting Amazon Aurora DSQL identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Amazon Aurora DSQL identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Amazon Aurora DSQL works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for Amazon Aurora DSQL)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Aurora DSQL works with IAM

Before you use IAM to manage access to Aurora DSQL, learn what IAM features are available to use with Aurora DSQL.

IAM features you can use with Amazon Aurora DSQL

IAM feature	Aurora DSQL support
Identity-based policies	Yes
Resource-based policies	Yes
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	Yes

To get a high-level view of how Aurora DSQL and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Aurora DSQL

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Aurora DSQL

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Resource-based policies within Aurora DSQL

Supports resource-based policies: Yes

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services. Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

To learn how to create and manage resource-based policies for Aurora DSQL clusters, see Resource-based policies for Aurora DSQL.

Policy actions for Aurora DSQL

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Aurora DSQL actions, see Actions Defined by Amazon Aurora DSQL in the *Service Authorization Reference*.

Policy actions in Aurora DSQL use the following prefix before the action:

```
dsql
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Policy resources for Aurora DSQL

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Aurora DSQL resource types and their ARNs, see Resources Defined by Amazon Aurora DSQL in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Amazon Aurora DSQL .

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Policy condition keys for Aurora DSQL

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Aurora DSQL condition keys, see Condition keys for Amazon Aurora DSQL in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon Aurora DSQL.

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

ACLs in Aurora DSQL

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Aurora DSQL

Supports ABAC (tags in policies): Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Aurora DSQL

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for Aurora DSQL

Supports forward access sessions (FAS): Yes

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Aurora DSQL

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Aurora DSQL functionality. Edit service roles only when Aurora DSQL provides guidance to do so.

Service-linked roles for Aurora DSQL

Supports service-linked roles: Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles for Aurora DSQL, see Using service-linked roles in Aurora DSQL.

Identity-based policy examples for Amazon Aurora DSQL

By default, users and roles don't have permission to create or modify Aurora DSQL resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Aurora DSQL, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Amazon Aurora DSQL in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the Aurora DSQL console
- Allow users to view their own permissions
- Allow cluster management and database connection
- Aurora DSQL resource access based on tags

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Aurora DSQL resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies

that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Aurora DSQL console

To access the Amazon Aurora DSQL console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Aurora DSQL resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Aurora DSQL console, also attach the Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess` or `AmazonAuroraDSQLReadOnlyAccess` AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
```

```
}
```

Allow cluster management and database connection

The following policy grants an IAM user permission to manage and connect to a specific Aurora DSQL cluster. The policy scopes cluster management and connection actions to a single cluster Amazon Resource Name (ARN), while allowing `dsql:ListClusters` on all resources because this action does not support resource-level permissions.

This example uses `dsql:DbConnectAdmin` to connect with the `admin` role. To connect with a custom database role instead, replace `dsql:DbConnectAdmin` with `dsql:DbConnect`. For more information, see Authentication and authorization for Aurora DSQL.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowClusterManagement",
            "Effect": "Allow",
            "Action": [
                "dsql:GetCluster",
                "dsql:UpdateCluster",
                "dsql:DeleteCluster",
                "dsql:DbConnectAdmin",
                "dsql:TagResource",
                "dsql:ListTagsForResource",
                "dsql:UntagResource"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/my-cluster-id"
        },
        {
            "Sid": "AllowListClusters",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        }
    ]
}
```

Aurora DSQL resource access based on tags

You can use conditions in your identity-based policy to control access to Aurora DSQL resources based on tags. The following example shows how you might create a policy that allows viewing a cluster. However, the policy grants permission only if the cluster tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListClustersInConsole",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        },
        {
            "Sid": "ViewClusterIfOwner",
            "Effect": "Allow",
            "Action": "dsql:GetCluster",
            "Resource": "arn:aws:dsql:*:*:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view an Aurora DSQL cluster, the cluster must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise IAM denies access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

The following policy allows a user to create clusters only if they tag the cluster with their own user name as the `Owner`. It also allows tagging only on clusters that the user already owns.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateTaggedCluster",
            "Effect": "Allow",
            "Action": "dsql:CreateCluster",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "${aws:username}"
                }
            }
        },
        {
            "Sid": "AllowTagOwnedClusters",
            "Effect": "Allow",
            "Action": "dsql:TagResource",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

Troubleshooting Amazon Aurora DSQL identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Aurora DSQL and IAM.

Topics

- I am not authorized to perform an action in Aurora DSQL
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Aurora DSQL resources

I am not authorized to perform an action in Aurora DSQL

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` tries to use the console to view details about the *`my-dsql-cluster`* resource but doesn't have the *`GetCluster`* permissions.

```
User: iam:::user/mateojackson is not authorized to perform: GetCluster on resource: my-dsql-cluster
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-dsql-cluster`* resource by using the *`GetCluster`* action.

If you need help, contact your administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Aurora DSQL.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Aurora DSQL. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Aurora DSQL resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Aurora DSQL supports these features, see How Amazon Aurora DSQL works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Resource-based policies for Aurora DSQL

Use resource-based policies for Aurora DSQL to restrict or grant access to your clusters through JSON policy documents that attach directly to your cluster resources. These policies provide fine-grained control over who can access your cluster and under what conditions.

Aurora DSQL clusters are accessible from the public internet by default, with IAM authentication as the primary security control. Resource-based policies enable you to add access restrictions, particularly to block access from the public internet.

Resource-based policies work alongside IAM identity-based policies. AWS evaluates both types of policies to determine the final permissions for any access request to your cluster. By default, Aurora

DSQL clusters are accessible within an account. If an IAM user or role has Aurora DSQL permissions, they can access clusters with no resource-based policy attached.

> **Note**
>
> Changes to resource-based policies are eventually consistent and typically take effect within one minute.

For more information about the differences between identity-based and resource-based policies, see Identity-based policies and resource-based policies in the *IAM User Guide*.

> **Warning**
>
> The AWS account root user of the account that owns the cluster always retains full access to its own resources, regardless of resource-based policy conditions, including VPC source restrictions. Resource-based policies don't restrict access for the root user. To limit root user access to your clusters, use IAM identity-based policies or avoid using root credentials for database connections. For more information, see Root user best practices for your AWS account.

When to use resource-based policies

Resource-based policies are particularly useful in these scenarios:

- *Network-based access control* — Restrict access based on the VPC or IP address that requests originate from, or block public internet access entirely. Use condition keys like `aws:SourceVpc` and `aws:SourceIp` to control network access.
- *Multiple teams or applications* — Grant access to the same cluster for multiple teams or applications. Rather than managing individual IAM policies for each principal, you define access rules once on the cluster.
- *Complex conditional access* — Control access based on multiple factors like network attributes, request context, and user attributes. You can combine multiple conditions in a single policy.
- *Centralized security governance* — Enable cluster owners to control access using familiar AWS policy syntax that integrates with your existing security practices.

> **Note**
>
> Cross-account access is not yet supported for Aurora DSQL resource-based policies but will be available in future releases.

When someone tries to connect to your Aurora DSQL cluster, AWS evaluates your resource-based policy as part of the authorization context, along with any relevant IAM policies, to determine whether the request should be allowed or rejected.

Resource-based policies can grant access to principals within the same AWS account as the cluster. For multi-Region clusters, each regional cluster has its own resource-based policy, allowing for Region-specific access controls when needed.

> **Note**
>
> Condition context keys may vary between Regions (such as VPC IDs).

Topics

- Creating clusters with resource-based policies
- Adding and editing resource-based policies for clusters
- Viewing resource-based policies
- Removing resource-based policies
- Common resource-based policy examples
- Blocking public access with resource-based policies in Aurora DSQL
- Aurora DSQL API Operations and Resource-Based Policies

Creating clusters with resource-based policies

You can attach resource-based policies when creating a new cluster to ensure access controls are in place from the start. Each cluster can have a single inline policy attached directly to the cluster.

AWS Management Console

To add a resource-based policy during cluster creation

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. Choose **Create cluster**.
3. In **Cluster topology**, choose **Single Region** or **2 active Regions + Witness**.
4. In **Cluster configuration**, choose **Advanced configuration**.
5. Select **Add resource-based policy**. For a single-Region cluster, it's in **Cluster settings**. For a multi-Region cluster, it's in **Resource-based policy - optional**, with one setting for each Region.
6. Enter your policy document in the JSON editor. For example, to block public internet access:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

7. You can use **Edit statement** or **Add new statement** to build your policy.
8. Complete the remaining cluster configuration and choose **Create cluster**.

AWS CLI

Use the `--policy` parameter when creating a cluster to attach an inline policy:

```
aws dsql create-cluster --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-1a2b3c4d5e6f7a8b9" }
        }
    }]
}'
```

AWS SDKs

Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-1a2b3c4d5e6f7a8b9" }
        }
    }]
}

response = client.create_cluster(
    policy=json.dumps(policy)
)
```

```
print(f"Cluster created: {response['identifier']}")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "StringNotEquals": { "aws:SourceVpc": "vpc-1a2b3c4d5e6f7a8b9" }
    }
  }]
}
""";

CreateClusterRequest request = CreateClusterRequest.builder()
    .policy(policy)
    .build();

CreateClusterResponse response = client.createCluster(request);
System.out.println("Cluster created: " + response.identifier());
```

Adding and editing resource-based policies for clusters

When you attach a policy or create a cluster with one, the policy must meet these requirements:

- It's no longer than 20,480 characters.
- It doesn't include service principals (the `Service` key in a `Principal` element).
- Each of its statements includes a `Principal` element and a `Resource` element. Aurora DSQL doesn't support `NotPrincipal` or `NotResource`.

- It doesn't grant public access. For more information, see Blocking public access with resource-based policies in Aurora DSQL.
- It doesn't deny you both `dsql:PutClusterPolicy` and `dsql:DeleteClusterPolicy`.

The last requirement makes sure that you can still change or remove the policy after you attach it. To attach a policy that denies you both actions anyway, set `bypassPolicyLockoutSafetyCheck` to `true`, or pass `--bypass-policy-lockout-safety-check` in the AWS CLI.

AWS Management Console

To add a resource-based policy to an existing cluster

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. Choose your cluster from the cluster list to open the cluster details page.
3. Choose the **Permissions** tab.
4. In the **Resource-based policy** section, choose **Add policy**.
5. Enter your policy document in the JSON editor. You can use **Edit statement** or **Add new statement** to build your policy.
6. Choose **Add policy**.

To edit an existing resource-based policy

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. Choose your cluster from the cluster list to open the cluster details page.
3. Choose the **Permissions** tab.
4. In the **Resource-based policy** section, choose **Edit**.
5. Modify the policy document in the JSON editor. You can use **Edit statement** or **Add new statement** to update your policy.
6. Choose **Save changes**.

AWS CLI

Use the `put-cluster-policy` command to attach a new policy or update an existing policy on a cluster:

```
aws dsql put-cluster-policy --identifier your_cluster_id --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": { "aws:SourceVpc": "true" }
        }
    }]
}'
```

To avoid overwriting a change that someone else made after you read the policy, pass the `policyVersion` value from `get-cluster-policy` in `--expected-policy-version`. If the policy changed after you read it, the command fails with a `ConflictException`.

AWS SDKs

Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
```

```
    }]
}

response = client.put_cluster_policy(
    identifier='your_cluster_id',
    policy=json.dumps(policy)
)
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.PutClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "Null": {"aws:SourceVpc": "true"}
    }
  }]
}
""";

PutClusterPolicyRequest request = PutClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .policy(policy)
    .build();

client.putClusterPolicy(request);
```

Viewing resource-based policies

You can view resource-based policies attached to your clusters to understand the current access controls in place.

AWS Management Console

To view resource-based policies

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. Choose your cluster from the cluster list to open the cluster details page.
3. Choose the **Permissions** tab.
4. View the attached policy in the **Resource-based policy** section.

AWS CLI

Use the `get-cluster-policy` command to view a cluster's resource-based policy:

```
aws dsql get-cluster-policy --identifier your_cluster_id
```

AWS SDKs

Python

```
import boto3
import json

client = boto3.client('dsql')

response = client.get_cluster_policy(
    identifier='your_cluster_id'
)

# Parse and pretty-print the policy
policy = json.loads(response['policy'])
print(json.dumps(policy, indent=2))
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyResponse;

DsqlClient client = DsqlClient.create();

GetClusterPolicyRequest request = GetClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

GetClusterPolicyResponse response = client.getClusterPolicy(request);
System.out.println("Policy: " + response.policy());
```

Removing resource-based policies

You can remove resource-based policies from clusters to change access controls.

> **Important**
>
> When you remove all resource-based policies from a cluster, access will be controlled entirely by IAM identity-based policies.

AWS Management Console

To remove a resource-based policy

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql/.
2. Choose your cluster from the cluster list to open the cluster details page.
3. Choose the **Permissions** tab.
4. In the **Resource-based policy** section, choose **Delete**.
5. In the confirmation dialog, type **`confirm`** to confirm the deletion.
6. Choose **Delete**.

AWS CLI

Use the `delete-cluster-policy` command to remove a policy from a cluster:

```
aws dsql delete-cluster-policy --identifier your_cluster_id
```

AWS SDKs

Python

```
import boto3

client = boto3.client('dsql')

response = client.delete_cluster_policy(
    identifier='your_cluster_id'
)

print("Policy deleted successfully")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

DeleteClusterPolicyRequest request = DeleteClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

client.deleteClusterPolicy(request);
System.out.println("Policy deleted successfully");
```

Common resource-based policy examples

These examples show common patterns for controlling access to your Aurora DSQL clusters. You can combine and modify these patterns to meet your specific access requirements.

Note

Use `"Resource": "*"` in cluster policies. In a cluster policy, * means the cluster that has the policy. If a statement names a different cluster's ARN, such as one copied from a peered cluster in another AWS Region, the statement doesn't apply. When you create a cluster with a policy, you also can't name the cluster's ARN, because the cluster doesn't have one yet.

Block public internet access

This policy blocks connections to your Aurora DSQL clusters from the public internet (non-VPC). The policy doesn't restrict which VPC your connections come from, only that they come from a VPC. To limit access to a specific VPC, use a `StringNotEquals` condition on `aws:SourceVpc` instead, as in Multi-Region cluster policies. This condition also blocks connections from the public internet, because they don't include `aws:SourceVpc`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

Note

This example uses only `aws:SourceVpc` to check for VPC connections. The `aws:VpcSourceIp` and `aws:SourceVpce` condition keys provide additional granularity but are not required for basic VPC-only access control.

To provide an exception for specific roles, use this policy instead:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyAccessFromOutsideVPC",
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        },
        "StringNotEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:role/ExceptionRole",
            "arn:aws:iam::123456789012:role/AnotherExceptionRole"
          ]
        }
      }
    }
  ]
}
```

Restrict access to AWS Organization

This policy restricts access to principals within an AWS Organization:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalOrgID": "o-exampleorgid"
        }
      }
    }
  ]
}
```

Restrict access to specific Organizational Unit

This policy restricts access to principals within a specific Organizational Unit (OU) in an AWS Organization, providing more granular control than organization-wide access:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringNotLike": {
          "aws:PrincipalOrgPaths": "o-exampleorgid/r-examplerootid/ou-exampleouid/*"
```

```
        }
      }
    }
  ]
}
```

Multi-Region cluster policies

For multi-Region clusters, each regional cluster maintains its own resource policy, allowing for Region-specific controls. In the following example, each policy denies connections that don't come from a specific VPC in its Region:

us-east-1 policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-0a1b2c3d4e5f67890"
        }
      }
    }
  ]
}
```

us-east-2 policy:

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

```
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-0f9e8d7c6b5a43210"
        }
      }
    }
  ]
}
```

> **Note**
>
> Condition context keys may vary between AWS Regions (such as VPC IDs).

Blocking public access with resource-based policies in Aurora DSQL

Block Public Access (BPA) is a feature that identifies and prevents the attaching of resource-based policies that grant public access to your Aurora DSQL clusters across your AWS accounts. With BPA, you can prevent public access to your Aurora DSQL resources. BPA performs checks during the creation or modification of a resource-based policy and helps improve your security posture with Aurora DSQL.

BPA uses automated reasoning to analyze the access granted by your resource-based policy and alerts you if such permissions are found at the time of administering a resource-based policy. The analysis verifies access across all resource-based policy statements, actions, and the set of condition keys used in your policies.

> **Important**
>
> BPA helps protect your resources by preventing public access from being granted through the resource-based policies that are directly attached to your Aurora DSQL resources, such

as clusters. In addition to using BPA, carefully inspect the following policies to confirm that they do not grant public access:

- Identity-based policies attached to associated AWS principals (for example, IAM roles)
- Resource-based policies attached to associated AWS resources (for example, AWS Key Management Service (KMS) keys)

You must ensure that the principal doesn't include a * entry or that one of the specified condition keys restrict access from principals to the resource. If the resource-based policy grants public access to your cluster across AWS accounts, Aurora DSQL will block you from creating or modifying the policy until the specification within the policy is corrected and deemed non-public.

You can make a policy non-public by specifying one or more principals inside the `Principal` block. The following resource-based policy example blocks public access by specifying two principals.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "dsql:*",
  "Resource": "*"
}
```

Policies that restrict access by specifying certain condition keys are also not considered public. Along with evaluation of the principal specified in the resource-based policy, the following trusted condition keys are used to complete the evaluation of a resource-based policy for non-public access:

- `aws:PrincipalAccount`
- `aws:PrincipalArn`
- `aws:PrincipalOrgID`
- `aws:PrincipalOrgPaths`
- `aws:SourceAccount`

- aws:SourceArn
- aws:SourceVpc
- aws:SourceVpce
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- aws:Ec2InstanceSourceVpc
- aws:SourceOrgID
- aws:SourceOrgPaths

Additionally, for a resource-based policy to be non-public, the values for Amazon Resource Name (ARN) and string condition keys must not contain wildcards or variables. If your resource-based policy uses the aws:PrincipalIsAWSService key, you must make sure that you've set the key value to true.

The following policy limits access to the user Ben in the specified account. The condition makes the Principal constrained and not be considered as public.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dsql:*",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/Ben"
    }
  }
}
```

The following example of a non-public resource-based policy constrains sourceVPC using the StringEquals operator.

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dsql:*",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
      }
    }
  ]
}
```

Aurora DSQL API Operations and Resource-Based Policies

Resource-based policies in Aurora DSQL control access to specific API operations. The following sections list all Aurora DSQL API operations organized by category, with an indication of which ones support resource-based policies.

The *Supports RBP* column indicates whether the API operation is subject to resource-based policy evaluation when a policy is attached to the cluster.

Tag APIs

API Operation	Description	Supports RBP
ListTagsForResource	Lists the tags for a Aurora DSQL resource	Yes
TagResource	Adds tags to a Aurora DSQL resource	Yes
UntagResource	Removes tags from a Aurora DSQL resource	Yes

Cluster management APIs

API Operation	Description	Supports RBP
CreateCluster	Creates a new cluster	No
DeleteCluster	Deletes a cluster	Yes
GetCluster	Retrieves information about a cluster	Yes
GetVpcEndpointServiceName	Retrieves the VPC endpoint service name for a cluster	Yes
ListClusters	Lists clusters in your account	No
UpdateCluster	Updates the configuration of a cluster	Yes

Multi-Region property APIs

API Operation	Description	Supports RBP
AddPeerCluster	Adds a peer cluster to a multi-region configuration	Yes
PutMultiRegionProperties	Sets multi-region properties for a cluster	Yes
PutWitnessRegion	Sets the witness region for a multi-region cluster	Yes

Stream APIs

API Operation	Description	Supports RBP
CreateStream	Creates a new change data capture stream	Yes
DeleteStream	Deletes a change data capture stream	No

API Operation	Description	Supports RBP
GetStream	Retrieves information about a change data capture stream	No
ListStreams	Lists the change data capture streams in your account	Yes

Resource-based policy APIs

API Operation	Description	Supports RBP
DeleteClusterPolicy	Deletes the resource-based policy from a cluster	Yes
GetClusterPolicy	Retrieves the resource-based policy for a cluster	Yes
PutClusterPolicy	Creates or updates the resource-based policy for a cluster	Yes

AWS Fault Injection Service APIs

API Operation	Description	Supports RBP
InjectError	Injects errors for fault injection testing	No

Backup and restore APIs

API Operation	Description	Supports RBP
GetBackupJob	Retrieves information about a backup job	No
GetRestoreJob	Retrieves information about a restore job	No
StartBackupJob	Starts a backup job for a cluster	Yes

API Operation	Description	Supports RBP
StartRestoreJob	Starts a restore job from a backup	No
StopBackupJob	Stops a running backup job	No
StopRestoreJob	Stops a running restore job	No

Using service-linked roles in Aurora DSQL

Aurora DSQL uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Aurora DSQL. Service-linked roles are predefined by Aurora DSQL and include all the permissions that the service requires to call AWS services on behalf of your Aurora DSQL cluster.

Service-linked roles make the setup process easier because you don't have to manually add the necessary permissions to use Aurora DSQL. When you create a cluster, Aurora DSQL automatically creates a service-linked role for you. You can delete the service-linked role only after you delete all of your clusters. This protects your Aurora DSQL resources because you can't inadvertently remove permissions needed for access to the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked roles are available in all supported Aurora DSQL Regions.

Service-linked role permissions for Aurora DSQL

Aurora DSQL uses the service-linked role named `AWSServiceRoleForAuroraDsql` – Allows Amazon Aurora DSQL to create and manage AWS resources on your behalf. This service-linked role is attached to the following managed policy: AuroraDsqlServiceLinkedRolePolicy.

With this service-linked role, Aurora DSQL publishes metrics to CloudWatch. It can also write to the `/aws/dsql/normalized-sql-text` log group in Amazon CloudWatch Logs in your account, which an upcoming Aurora DSQL feature uses to deliver normalized SQL text.

Note

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. You might encounter the following error message: `You don't have the permissions to create an Amazon Aurora DSQL service-linked role`. If you see this message, make sure that you have the following permissions enabled:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateDsqlServiceLinkedRole",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "dsql.amazonaws.com"
                }
            }
        }
    ]
}
```

For more information, see Service-linked role permissions.

Create a service-linked role

You don't need to manually create an AuroraDSQLServiceLinkedRolePolicy service-linked role. Aurora DSQL creates the service-linked role for you. If the AuroraDSQLServiceLinkedRolePolicy service-linked role has been deleted from your account, Aurora DSQL creates the role when you create a new Aurora DSQL cluster.

Edit a service-linked role

Aurora DSQL doesn't allow you to edit the AuroraDSQLServiceLinkedRolePolicy service-linked role. After you create a service-linked role, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using the IAM console, the AWS Command Line Interface (AWS CLI), or IAM API.

Delete a service-linked role

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way, you don't have an unused entity that is not actively monitored or maintained.

Before you can delete a service-linked role for an account, you must delete any clusters in the account.

You can use the IAM console, the AWS CLI, or the IAM API to delete a service-linked role. For more information, see Create a service-linked role in the IAM User Guide.

Supported Regions for Aurora DSQL service-linked roles

Aurora DSQL supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Using IAM condition keys with Amazon Aurora DSQL

The `Condition` element (or `Condition` block) specifies the conditions under which a policy statement is in effect. You build conditional expressions using condition operators such as `StringEquals` or `StringLike` to match keys in the policy against values in the request. Condition keys are different from resource ARN scoping, which restricts actions to specific resources using the `Resource` element.

Amazon Aurora DSQL supports both service-specific condition keys and AWS global condition keys. You can use these keys to refine the conditions under which your IAM policy statements apply. For more information about AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Available condition keys for Amazon Aurora DSQL

Amazon Aurora DSQL supports the service-specific condition keys `dsql:WitnessRegion`, `dsql:FisActionId`, and `dsql:FisTargetArns`. You can also use AWS global condition keys such as `aws:RequestTag`, `aws:ResourceTag`, and `aws:TagKeys`. For a complete list of Amazon Aurora DSQL actions, condition keys, and resources that you can specify in policies, see Actions, resources, and condition keys for Amazon Aurora DSQL in the *Service Authorization Reference*.

Example: Restrict the witness Region for multi-Region clusters

The following policy uses the `dsql:WitnessRegion` condition key to restrict multi-Region cluster creation to cases where the witness Region is US West (Oregon). Without this condition, you can specify any Region as the witness Region.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateWithSpecificWitness",
            "Effect": "Allow",
            "Action": [
                "dsql:CreateCluster",
                "dsql:PutWitnessRegion"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": "us-west-2"
                }
            }
        },
        {
            "Sid": "AllowMultiRegionSetup",
            "Effect": "Allow",
            "Action": [
                "dsql:PutMultiRegionProperties",
                "dsql:AddPeerCluster"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
```

```
        }
    ]
}
```

The first statement applies the `dsql:WitnessRegion` condition to the actions that support it (`dsql:CreateCluster` and `dsql:PutWitnessRegion`). The second statement grants `dsql:PutMultiRegionProperties` and `dsql:AddPeerCluster` without the condition, because the `dsql:WitnessRegion` condition key applies only to `dsql:CreateCluster` and `dsql:PutWitnessRegion`.

Example: Require tags when creating clusters

The following policy uses the `aws:RequestTag` and `aws:TagKeys` condition keys to require an `Environment` tag when you create clusters.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateWithRequiredTag",
            "Effect": "Allow",
            "Action": "dsql:CreateCluster",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Environment": [
                        "production",
                        "staging",
                        "development"
                    ]
                },
                "ForAllValues:StringEquals": {
                    "aws:TagKeys": [
                        "Environment",
                        "Owner"
                    ]
                }
            }
```

```
        }
    ]
}
```

This policy uses two condition keys together. The `aws:RequestTag/Environment` condition requires that the `Environment` tag has one of the allowed values. The `aws:TagKeys` condition with the `ForAllValues` set operator ensures that the request includes only the `Environment` and `Owner` tag keys.

Example: Restrict fault injection to specific clusters

The following policy uses the `dsql:FisActionId` and `dsql:FisTargetArns` condition keys to restrict AWS FIS fault injection to a specific action and target cluster.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowSpecificFaultInjection",
            "Effect": "Allow",
            "Action": "dsql:InjectError",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "dsql:FisActionId": "aws:dsql:cluster-connection-failure"
                },
                "ForAllValues:ArnLike": {
                    "dsql:FisTargetArns": "arn:aws:dsql:us-
east-1:123456789012:cluster/*"
                }
            }
        }
    ]
}
```

The `dsql:FisActionId` condition restricts the fault injection actions that you can perform. The `dsql:FisTargetArns` condition uses the `ForAllValues` set operator and the `ArnLike`

operator. Together, they ensure that every cluster ARN in the request matches the allowed pattern. This blocks any request that includes cluster ARNs outside the intended scope.

Incident response in Amazon Aurora DSQL

Security is the highest priority at AWS. As part of the AWS Cloud shared responsibility model, AWS manages a data center, network, and software architecture that meets the requirements of the most security-sensitive organizations. AWS is responsible for any incident response with respect to the Amazon Aurora DSQL service itself. Also, as an AWS customer, you share a responsibility for maintaining security in the cloud. This means that you control the security you choose to implement from the AWS tools and features you have access to. In addition, you're responsible for incident response on your side of the shared responsibility model.

By establishing a security baseline that meets the objectives for your applications running in the cloud, you're able to detect deviations that you can respond to. To help you understand the impact that incident response and your choices have on your corporate goals, we encourage you to review the following resources:

- AWS Security Incident Response Guide
- AWS Best Practices for Security, Identity, and Compliance
- Security Perspective of the AWS Cloud Adoption Framework (CAF) whitepaper

Amazon GuardDuty is a managed threat detection service continuously monitoring malicious or unauthorized behavior to help customers protect AWS accounts and workloads and identify suspicious activity potentially before it escalates into an incident. It monitors activity such as unusual API calls or potentially unauthorized deployments indicating possible account or resource compromise or reconnaissance by bad actors. For example, Amazon GuardDuty is able to detect suspicious activity in Amazon Aurora DSQL APIs, such as a user logging in from a new location and creating a new cluster.

Compliance validation for Amazon Aurora DSQL

Our new AWS sign-up experience is not designed for regulated workloads. If you're using our new AWS sign-up experience, but you want to use AWS for regulated workloads, you can sign up for AWS (advanced) or activate advanced features for your AWS environment.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in Amazon Aurora DSQL

The AWS global infrastructure is built around AWS Regions and Availability Zones (AZ). AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures. Aurora DSQL is designed so that you can take advantage of AWS Regional infrastructure while providing the highest database availability. By default, single-Region clusters in Aurora DSQL have Multi-AZ availability, providing tolerance to major component failures and infrastructure disruptions that might impact access to a full AZ. Multi-Region clusters provide all of the benefits from Multi-AZ resiliency while still providing the strongly consistent database availability, even in cases in which AWS Region is inaccessible to application clients.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Aurora DSQL offers several features to help support your data resiliency and backup needs.

Backup and restore

Aurora DSQL supports backup and restore with AWS Backup console. You can perform a full backup and restore for your single-Region and multi-Region clusters. For more information, see Backup and restore for Amazon Aurora DSQL.

Replication

By design, Aurora DSQL commits all write transactions to a distributed transaction log and synchronously replicates all committed log data to user storage replicas in three AZs. Multi-Region clusters provide full cross-Region replication capabilities between read and write Regions.

A designated witness Region supports transaction log-only writes and doesn't consume storage. Witness Regions don't have an endpoint. This means that witness Regions store only encrypted transaction logs, require no administration or configuration, and aren't accessible by users. If the witness Region becomes impaired, there is no impact to cluster availability. Write transactions might experience a small increase in latency until the witness Region recovers.

Aurora DSQL transaction logs and user storage are distributed with all data presented to Aurora DSQL query processors as a single logical volume. Aurora DSQL automatically splits, merges, and replicates data based on database primary key range and access patterns. Aurora DSQL automatically scales read replicas, both up and down, based on read access frequency.

Cluster storage replicas are distributed across a multi-tenant storage fleet. If a component or AZ becomes impaired, Aurora DSQL automatically redirects access to surviving components and asynchronously repairs missing replicas. Once Aurora DSQL fixes the impaired replicas, Aurora DSQL automatically adds them back to the storage quorum and makes them available to your cluster.

High availability

By default, single-Region and multi-Region clusters in Aurora DSQL are active-active, and you don't need to manually provision, configure, or reconfigure any clusters. Aurora DSQL fully automates cluster recovery, which eliminates the need for traditional primary-secondary failover operations. Replication is always synchronous and done in multiple AZs, so there is no risk of data loss due to replication lag or failover to an asynchronous secondary database during failure recovery.

Single-Region clusters provide a Multi-AZ redundant endpoint that automatically enables concurrent access with strong data consistency across three AZs. This means that user storage replicas on any of these three AZs always return the same result to one or more readers and are always available to receive writes. This strong consistency and Multi-AZ resiliency is available across all Regions for Aurora DSQL multi-Region clusters. This means that multi-Region clusters provide two strongly consistent Regional endpoints, so clients can read or write indiscriminately to either Region with zero replication lag on commit.

Aurora DSQL provides 99.99% availability for single-Region clusters and 99.999% for multi-Region clusters.

Fault injection testing

Amazon Aurora DSQL integrates with AWS Fault Injection Service (AWS FIS), a fully managed service for running controlled fault injection experiments to improve an application's resilience. Using AWS FIS, you can:

- Create experiment templates that define specific failure scenarios
- Inject failures (elevated cluster connection error rates) to validate application error handling and recovery mechanisms
- Test multi-Region application behavior to validate application traffic shift between AWS Regions when one AWS Region is experiencing high connection error rates

For example, in a multi-Region cluster spanning US East (N. Virginia) and US East (Ohio), you can run an experiment in US East (Ohio) to test failures there while US East (N. Virginia) continues normal operations. This controlled testing helps you identify and resolve potential issues before they affect production workloads.

See Action targets in the *AWS FIS user guide* for a complete list of AWS FIS supported actions.

For information about Amazon Aurora DSQL actions available in AWS FIS, see Aurora DSQL actions reference in the *AWS FIS User Guide*.

To get started running fault injection experiments, see Planning your AWS FIS experiments in the *AWS FIS User Guide*.

Infrastructure Security in Amazon Aurora DSQL

As a managed service, Amazon Aurora DSQL is protected by the AWS global network security procedures that are described in Best Practices for Security, Identity, & Compliance.

You use AWS published API calls to access Aurora DSQL through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Managing and connecting to Amazon Aurora DSQL clusters using AWS PrivateLink

With AWS PrivateLink for Amazon Aurora DSQL, you can provision interface Amazon VPC endpoints (interface endpoints) in your Amazon Virtual Private Cloud. These endpoints are directly accessible from applications that are on premises over Amazon VPC and Direct Connect, or in a different AWS Region over Amazon VPC peering. Using AWS PrivateLink and interface endpoints, you can simplify private network connectivity from your applications to Aurora DSQL.

Applications within your Amazon VPC can access Aurora DSQL using Amazon VPC interface endpoints without requiring public IP addresses.

Interface endpoints are represented by one or more elastic network interfaces (ENIs) that are assigned private IP addresses from subnets in your Amazon VPC. Requests to Aurora DSQL over interface endpoints stay on the AWS network. For more information about how to connect your Amazon VPC with your on-premises network, see the Direct Connect User Guide and the AWS Site-to-Site VPN VPN User Guide.

For general information about interface endpoints, see Access an AWS service using an interface Amazon VPC endpoint in the AWS PrivateLink User Guide.

Types of Amazon VPC endpoints for Aurora DSQL

Aurora DSQL requires two different types of AWS PrivateLink endpoints.

1. *Management endpoint*— This endpoint is used for administrative operations, such as `get`, `create`, `update`, `delete`, and `list` on Aurora DSQL clusters. See Managing Aurora DSQL clusters using AWS PrivateLink.
2. *Connection endpoint*— This endpoint is used for connecting to Aurora DSQL clusters through PostgreSQL clients. See Connecting to Aurora DSQL clusters using AWS PrivateLink.

Considerations when using AWS PrivateLink for Aurora DSQL

Amazon VPC considerations apply to AWS PrivateLink for Aurora DSQL. For more information, see Access an AWS service using an interface VPC endpoint and AWS PrivateLink quotas in the AWS PrivateLink Guide.

Managing Aurora DSQL clusters using AWS PrivateLink

You can use the AWS Command Line Interface or AWS Software Development Kits (SDKs) to manage Aurora DSQL clusters through Aurora DSQL interface endpoints.

Creating an Amazon VPC endpoint

To create an Amazon VPC interface endpoint, see Create an Amazon VPC endpoint in the AWS PrivateLink Guide.

```
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 \
```

To use the default Regional DNS name for Aurora DSQL API requests, do not disable private DNS when you create the Aurora DSQL interface endpoint. When private DNS is enabled, requests to the Aurora DSQL service made from within your Amazon VPC will automatically resolve to the private IP address of the Amazon VPC endpoint, rather than the public DNS name. When private DNS is enabled, Aurora DSQL requests made within your Amazon VPC will automatically resolve to your Amazon VPC endpoint.

If private DNS is not enabled, use the `--region` and `--endpoint-url` parameters with AWS CLI commands to manage Aurora DSQL clusters through Aurora DSQL interface endpoints.

Listing clusters using an endpoint URL

In the following example, replace the AWS Region `us-east-1` and the DNS name of the Amazon VPC endpoint ID `vpce-1a2b3c4d-5e6f.dsql.us-east-1.vpce.amazonaws.com` with your own information.

```
aws dsql --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsql.us-
east-1.vpce.amazonaws.com list-clusters
```

API Operations

Refer to the Aurora DSQL API reference for documentation on managing resources in Aurora DSQL.

Managing endpoint policies

By thoroughly testing and configuring the Amazon VPC endpoint policies, you can help ensure that your Aurora DSQL cluster is secure, compliant, and aligned with your organization's specific access control and governance requirements.

Example: Full Aurora DSQL access policy

The following policy grants full access to all Aurora DSQL actions and resources through the specified Amazon VPC endpoint.

```
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": "*"
        }
      ]
    }'
```

Example: Restricted Aurora DSQL Access Policy

The following policy only permits these Aurora DSQL actions.

- `CreateCluster`
- `GetCluster`
- `ListClusters`

All other Aurora DSQL actions are denied.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "*"
    }
  ]
}
```

Connecting to Aurora DSQL clusters using AWS PrivateLink

Once your AWS PrivateLink endpoint is set up and active, you can connect to your Aurora DSQL cluster using a PostgreSQL client. The connection instructions below outline the steps to construct the proper hostname for connecting through the AWS PrivateLink endpoint.

Setting up an AWS PrivateLink connection endpoint

Step 1: Get the service name for your cluster

When creating an AWS PrivateLink endpoint for connecting to your cluster, you first need to fetch the cluster-specific service name.

AWS CLI

```
aws dsql get-vpc-endpoint-service-name \
--region us-east-1 \
--identifier your-cluster-id
```

Example response

```
{
    "serviceName": "com.amazonaws.us-east-1.dsql-fnh4"
}
```

The service name includes an identifier, such as `dsql-fnh4` in the example. This identifier is also needed when constructing the hostname for connecting to your cluster.

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);
```

The service identifier is cluster-specific. Different clusters can return different identifiers, even in the same AWS Region or account. Always use the service name returned for the cluster that you want to connect to.

Step 2: Create the Amazon VPC endpoint

Using the service name obtained in the previous step, create an Amazon VPC endpoint.

> **Important**
>
> The connection instructions below only work for connecting to clusters when private is DNS enabled. Do not use the `--no-private-dns-enabled` flag when creating the endpoint, as this will prevent the connection instructions below from working properly. If you disable private DNS, you will need to create your own wildcard private DNS record that points to the created endpoint.

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
```

Example response

```
{
    "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 (Boto3)

```
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

Use an endpoint URL for Aurora DSQL APIs

```
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);
```

Additional setup when connecting via Direct Connect or Amazon VPC peering

Some additional setup may be needed to connect to Aurora DSQL clusters using an AWS PrivateLink connection endpoint from on-premise devices via Amazon VPC peering or Direct Connect. This setup is not required if your application is running in the same Amazon VPC as your AWS PrivateLink endpoint. The private DNS entries created above will not resolve correctly outside the endpoint's Amazon VPC, but you can create your own private DNS records which resolve to your AWS PrivateLink connection endpoint.

Create a private CNAME DNS record which points to the AWS PrivateLink endpoint's fully-qualified domain name. The domain name of the created DNS record should be constructed from the following components:

1. The service identifier from the service name. For example: `dsql-fnh4`
2. The AWS Region

Create the CNAME DNS record with a domain name in the following format: `*.`*`service-identifier`*`.`*`region`*`.on.aws`

The format of the domain name is important for two reasons:

1. The hostname used to connect to Aurora DSQL must match Aurora DSQL's server certificate when using the `verify-full` SSL mode. This ensures the highest level of connection security.
2. Aurora DSQL uses the cluster ID portion of the hostname used to connect to Aurora DSQL to identify the connecting cluster.

If creating private DNS records is not possible, you can still connect to Aurora DSQL. See Connecting to an Aurora DSQL cluster using an AWS PrivateLink endpoint without private DNS.

Connecting to an Aurora DSQL cluster using an AWS PrivateLink connection endpoint

Once your AWS PrivateLink endpoint is set up and active (check that the `State` is `available`), you can connect to your Aurora DSQL cluster using a PostgreSQL client. For instructions on using the AWS SDKs, you can follow the guides in Programming with Aurora DSQL. You must change the cluster endpoint to match the hostname format.

Constructing the hostname

The hostname for connecting through AWS PrivateLink differs from the public DNS hostname. You need to construct it using the following components.

1. `Your-cluster-id`
2. The service identifier from the service name. For example: `dsql-fnh4`
3. The AWS Region. For example: `us-east-1`

Use the following format: *`cluster-id`*`.`*`service-identifier`*`.`*`region`*`.on.aws`

Example: Connection Using PostgreSQL

```
# 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
```

Connecting to an Aurora DSQL cluster using an AWS PrivateLink endpoint without private DNS

The connection instructions above rely on private DNS records. If your application is running in the same Amazon VPC as your AWS PrivateLink endpoint, the DNS records are created for you. Alternatively, if you are connecting from on-premise devices via Amazon VPC peering or Direct Connect, then you can create your own private DNS records. However, DNS record setup is not always possible due to network restrictions imposed by your security teams. If your application must connect using Direct Connect or from a peered Amazon VPC, and DNS record setup is not possible, you can still connect to Aurora DSQL.

Aurora DSQL uses the cluster ID portion of your hostname to identify the connecting cluster, but if DNS record setup is not possible, Aurora DSQL supports specifying the target cluster using the `amzn-cluster-id` connection option. With this option, it is possible to use your AWS PrivateLink endpoint's fully-qualified domain name as your hostname when connecting.

> **Important**
>
> When connecting with your AWS PrivateLink endpoint's fully-qualified domain name or IP address, the `verify-full` SSL mode is not supported. For this reason, setting up private DNS is preferred.

Example: Specifying the cluster ID connection option using PostgreSQL

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export HOSTNAME=vpce-04037adb76c111221-d849uc2p.dsql-fnh4.us-east-1.vpce.amazonaws.com
 # This should match your endpoint's fully-qualified domain name

# Construct the hostname used to generate the authentication token
export AUTH_HOSTNAME="$CLUSTERID.dsql.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $AUTH_HOSTNAME)

# Specify the amzn-cluster-id connection option
export PGOPTIONS="-c amzn-cluster-id=$CLUSTERID"

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

Troubleshooting issues with AWS PrivateLink

Common Issues and Solutions

The following table lists common issues and solutions relating to AWS PrivateLink with Aurora DSQL.

Issue	Possible Cause	Solution
Connection timeout	Security group not properly configured	Use Amazon VPC Reachability Analyzer to ensure your networking setup allows traffic on port 5432.
DNS resolution failure	Private DNS not enabled	Verify that the Amazon VPC endpoint was created with private DNS enabled.
Authentication failure	Incorrect credentials or expired token	Generate a new authentication token and verify the user name.
Service name not found	Incorrect cluster ID	Double-check your cluster ID and AWS Region when fetching the service name.

Related Resources

For more information, see the following resources:

- Amazon Aurora DSQL User Guide
- AWS PrivateLink Documentation
- Access AWS services through AWS PrivateLink

Configuration and vulnerability analysis in Amazon Aurora DSQL

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- Shared responsibility model
- Amazon Web Services: Overview of security processes (whitepaper)

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon Aurora DSQL gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know

the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:dsql:*:123456789012:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of `aws:SourceArn` must be the ARN of the Aurora DSQL resource that the service role acts on behalf of.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Aurora DSQL to prevent the confused deputy problem.

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"
      }
    }
  }
}
```

CDC stream service role

Change data capture (CDC) streams require an IAM service role that Aurora DSQL assumes to write CDC records to your target. When you create this role, use `aws:SourceAccount` and `aws:SourceArn` conditions in the trust policy to ensure that only CDC streams in your account can assume the role.

Set `aws:SourceArn` to the stream ARN pattern for the cluster that uses the role. Because Aurora DSQL hasn't assigned the stream identifier when you create the stream, use a wildcard for the stream portion of the ARN:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
                }
            }
        }
    ]
}
```

After you create a stream, you can tighten `aws:SourceArn` to the exact stream ARN if the role serves a single stream. For a full explanation of the trust policy and permissions policy for CDC service roles, see Configuring IAM.

Security best practices for Aurora DSQL

Aurora DSQL provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Topics

- Detective security best practices for Aurora DSQL
- Preventative security best practices for Aurora DSQL

Detective security best practices for Aurora DSQL

In addition to the following ways to securely use Aurora DSQL, see Security in AWS Well-Architected Tool to learn about how cloud technologies improve your security.

Amazon CloudWatch Alarms

Using Amazon CloudWatch alarms, you watch a single metric over a time period that you specify. If the metric exceeds a given threshold, a notification is sent to an Amazon SNS topic or AWS Auto Scaling policy. CloudWatch alarms do not invoke actions because they are in a particular state. Rather the state must have changed and been maintained for a specified number of periods.

Tag your Aurora DSQL resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they enable you to categorize resources by purpose, owner, environment, or other criteria. The following are some examples:

- Security – Used to determine requirements such as encryption.
- Confidentiality – An identifier for the specific data-confidentiality level a resource supports.
- Environment – Used to distinguish between development, test, and production infrastructure.

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they let you categorize resources by purpose, owner, environment, or other criteria. The following are some examples.

- Security – used to determine requirements such as encryption.
- Confidentiality – an identifier for the specific data-confidentiality level a resource supports.
- Environment – used to distinguish between development, test, and production infrastructure.

For more information, see Best Practices for Tagging AWS Resources.

Preventative security best practices for Aurora DSQL

In addition to the following ways to securely use Aurora DSQL, see Security in AWS Well-Architected Tool to learn about how cloud technologies improve your security.

Use IAM roles to authenticate access to Aurora DSQL.

Users, applications, and other AWS services that access Aurora DSQL must include valid AWS credentials in AWS API and AWS CLI requests. You shouldn't store AWS credentials directly in the application or EC2 instances. These are long-term credentials that aren't automatically rotated. There is significant business impact if these credentials are compromised. An IAM role lets you obtain temporary access keys that you can use to access AWS services and resources.

For more information, see Authentication and authorization for Aurora DSQL.

Use IAM policies for Aurora DSQL base authorization.

When you grant permissions, you decide who is getting them, which Aurora DSQL API operations they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM roles and grant permissions to perform operations on Aurora DSQL resources. Also available are permissions boundaries for IAM entities, which let you set the maximum permissions that an identity-based policy can grant to an IAM entity.

Similar to the root user best practices for your AWS account, don't use the `admin` role in Aurora DSQL to perform everyday operations. Instead, we recommend that you create custom database roles to manage and connect to your cluster. For more information, see Accessing Aurora DSQL and Understanding authentication and authorization for Aurora DSQL.

Use `verify-full` in production environments.

This setting verifies that the server certificate is signed by a trusted certificate authority and that the server hostname matches the certificate.

Update your PostgreSQL client

Regularly update your PostgreSQL client to the latest version to benefit from security improvements. We recommend using PostgreSQL version 17.

Tagging resources in Aurora DSQL

In AWS, tags are user-defined key-value pairs that you define and associate with Aurora DSQL resources such as clusters and CDC streams. Tags are optional. If you provide a key, the value is optional.

You can use the AWS Management Console, the AWS CLI, or the AWS SDKs to add, list, and delete tags on Aurora DSQL clusters and CDC streams. You can add tags during and after resource creation by using the AWS console. To tag a resource after creation with the AWS CLI, use the `TagResource` operation.

Tagging clusters with a Name

Aurora DSQL creates clusters with a globally unique identifier assigned as the Amazon Resource Name (ARN). If you want to assign a user friendly name to your cluster, we recommend that you use a Tag.

If you create a console with the Aurora DSQL console, Aurora DSQL automatically creates a tag. This tag has a key of **Name** and an automatically generated value that represents the name of the cluster. This value is configurable, so you can assign a more friendly name to your cluster. If a cluster has a Name tag with an associated value, you can see the value throughout the Aurora DSQL console.

Tagging requirements

Tags have the following requirements:

- Keys can't be prefixed with `aws:`.
- Keys must be unique per tag set.
- A key must be between 1 and 128 allowed characters.
- A value must be between 0 and 256 allowed characters.
- Values do not need to be unique per tag set.
- Allowed characters for keys and values are letters, digits, white space, and any of the following symbols: _ . : / = + - @.
- Keys and values are case sensitive.

Tagging CDC streams

CDC streams are independently taggable resources. You can add tags when you create a stream by passing the `--tags` parameter to `CreateStream`, and you can read, add, or remove tags on an existing stream by using `ListTagsForResource`, `TagResource`, and `UntagResource` with the stream ARN. Tags on a CDC stream are separate from tags on the parent cluster and from tags on the destination Amazon Kinesis data stream.

A CDC stream ARN has the format `arn:aws:dsql:`*`region`*`:`*`account-id`*`:cluster/`*`cluster-id`*`/stream/`*`stream-id`*. For more information about CDC streams, see Change data capture (CDC) streams.

Tagging usage notes

When using tags in Aurora DSQL, consider the following.

- When using the AWS CLI or Aurora DSQL API operations, make sure to provide the Amazon Resource Name (ARN) for the Aurora DSQL resource to work with. For more information, see Amazon Resource Name (ARNs) format for Aurora DSQL resources.
- Each resource has one tag set, which is a collection of one or more tags assigned to the resource.
- Each resource can have up to 50 tags per tag set.
- If you delete a resource, any associated tags are deleted.
- You can add tags when you create a resource, you can view and modify tags using the following API operations: `TagResource`, `UntagResource`, and `ListTagsForResource`.
- You can use tags with IAM policies. You can use them to manage access to Aurora DSQL clusters and to control what actions can be applied to those resources. To learn more, see Controlling access to AWS resources using tags.
- You can use tags for various other activities across AWS. To learn more, see Common tagging strategies.

Considerations for working with Amazon Aurora DSQL

Consider the following behaviors when you work with Amazon Aurora DSQL. For more information about PostgreSQL compatibility and support, see SQL feature compatibility in Aurora DSQL. For quotas and limits, see Cluster quotas and database limits in Amazon Aurora DSQL.

- Storage limit calculations may take time to reflect freed storage after running a `DROP TABLE` command. If you need additional storage capacity, see Cluster quotas to request quota updates.
- For large tables in Aurora DSQL, use the system catalog to retrieve table row counts instead of `COUNT(*)` operations. For more information, see Using systems tables and commands in Aurora DSQL.
- Aurora DSQL manages permissions through schema-level grants. Admin users create schemas using `CREATE SCHEMA` and grant access to other roles using `GRANT USAGE ON SCHEMA`. Admin users manage objects in the public schema, while non-admin users create objects in user-created schemas. The admin role can grant itself any other role to obtain permissions on user-created objects. For more information, see Authorizing database roles to use SQL in your database.
- When drivers call `PG_PREPARED_STATEMENTS`, Aurora DSQL provides a cluster-wide view of cached prepared statements. You may see more prepared statements per connection than expected for the same cluster and IAM role. Aurora DSQL manages statement names dynamically during preparation.
- When connecting from IPv4-only instances, ensure your client is configured for IPv4 connections. Some PostgreSQL clients attempt both IPv4 and IPv6 connections in dualstack mode. If the IPv4 connection experiences throttling, the client may attempt IPv6 and return a `NetworkUnreachable` error on IPv4-only hosts. Configure your client to use IPv4 explicitly to avoid this behavior.
- After an admin user creates a new schema, `GRANT` and `REVOKE` changes propagate to existing connections within the connection lifetime (up to one hour). For immediate effect, establish a new connection after permission changes.
- During rare multi-Region linked-cluster recovery scenarios, automated cluster recovery operations maintain high availability, but you may experience transient concurrency control or connection errors. In most cases, only a percentage of your workload is affected. When you encounter these transient errors, retry your transaction or reconnect with your client.
- Some SQL clients, such as Datagrip, request extensive system metadata to populate schema information. Aurora DSQL provides core metadata for SQL query functionality. Schema display in these clients may show limited information compared to their full feature set.

- To ensure queries recognize newly created schemas and tables, refresh your connection after creating or dropping database objects. This includes scenarios where you see `Schema Already Exists` errors after dropping a schema, or when querying objects created in another connection. Disconnect and reconnect, or run `SET search_path` again to refresh the catalog cache.
- For complex queries, use `EXPLAIN ANALYZE VERBOSE` to identify high-latency operations and optimize query plans. Covering indexes can significantly reduce DPU costs by enabling index-only scans instead of full table scans. For more information, see Working with Aurora DSQL EXPLAIN plans.
- Connection limits are managed at the cluster level. See Cluster quotas to request quota updates.

Cluster quotas and database limits in Amazon Aurora DSQL

The following sections describe the cluster quotas and database limits for Aurora DSQL.

Cluster quotas

Your AWS account has the following cluster quotas in Aurora DSQL. To request an increase to the service quotas for single-Region and multi-Region clusters within a specific AWS Region, use the Service Quotas console page. For other quota increases, contact AWS Support.

Description	Default limit	Configurable?	Aurora DSQL error code
Maximum single-Region clusters per AWS account per AWS Region	20 clusters	Yes	API error code `ServiceQuotaExceededException`
Maximum multi-Region clusters per AWS account per AWS Region	5 clusters	Yes	API error code `ServiceQuotaExceededException`
Maximum storage per cluster	10 TiB default limit, up to 256 TiB with approved limit increase	Yes	`DISK_FULL(53100)`

Description	Default limit	Configurable?	Aurora DSQL error code
Maximum connections per cluster	10,000 connections	Yes	`TOO_MANY_CONNECTIONS(53300)`
Maximum connection rate per cluster	100 connections per second	No	`CONFIGURED_LIMIT_EXCEEDED(53400)`
Maximum connection burst capacity per cluster	1,000 connections	No	No error code
Maximum concurrent restore jobs	4	No	No error code
Connection Refill Rate	100 connections per second	No	No error code
Maximum CDC streams per cluster	5 streams	No	API error code `ServiceQuotaExceededException`

Database limits in Aurora DSQL

The following table describes the database limits in Aurora DSQL.

Description	Default limit	Configurable?	Aurora DSQL error code	Error message
Maximum combined size of the columns used in a primary key	1 KiB	No	54000	ERROR: key size too large
Maximum combined size of the columns in a secondary index	1 KiB	No	54000	ERROR: key size too large
Maximum size of a row in a table	2 MiB	No	54000	ERROR: maximum row size excee
Maximum size of a column that's not part of an index	1 MiB	No	54000	ERROR: maximum column size ex
Maximum number of columns in a primary key or a secondary index	8	No	54011	ERROR: more than 8 column key are not supported
Maximum number of columns in a table	255 (1600 total columns including dropped columns)	No	54011	ERROR: tables can have at mos

Description	Default limit	Configurable?	Aurora DSQL error code	Error message
Maximum number of indexes in a table	24	No	54000	ERROR: more than 24 indexes p allowed
Maximum number of extended statistics objects in a table	5	No	54000	ERROR: more than 5 extended s table are not allowed
Maximum size of all data modified in a write transacti on	10 MiB	No	54000	ERROR: transaction size limit DETAIL: Current transaction s 10mb
Maximum number of table rows that can be mutated in a transaction block	3,000 rows per transacti on. See Aurora DSQL considerations for PostgreSQ L compatibi lity.	No	54000	ERROR: transaction row limit
Maximum base amount of memory that a query operation can use	128 MiB per transaction	No	53200	ERROR: query requires too muc out of memory.

Description	Default limit	Configurable?	Aurora DSQL error code	Error message
Maximum number of schemas defined in a database	10	No	54000	ERROR: more than 10 schemas n
Maximum number of tables in a database	1,000 tables	No	54000	ERROR: creating more than 100 allowed
Maximum number of databases in a cluster	1	No	No error code	ERROR: unsupported statement
Maximum transaction time	5 minutes	No	54000	ERROR: transaction age limit exceeded
Maximum connection duration	60 minutes	No	No error code	No error message
Maximum number of views in a database	5,000	No	54000	ERROR: creating more than 500 allowed
Maximum view definition size	2 MiB	No	54000	ERROR: view definition too la

Description	Default limit	Configurable?	Aurora DSQL error code	Error message
Maximum number of sequences	5,000	No	54000	ERROR: creating more than 500 not allowed
Maximum message size	10 MiB	No	08P01	FATAL: invalid message length

For data type limits specific to Aurora DSQL, see Supported data types in Aurora DSQL.

Aurora DSQL API reference

In addition to the AWS Management Console and the AWS Command Line Interface (AWS CLI), Aurora DSQL also provides an API interface. You can use the API operations to manage your resources in Aurora DSQL.

For an alphabetical list of API operations, see Actions.

For an alphabetical list of data types, see Data types.

For a list of common query parameters, see Common parameters.

For descriptions of the error codes, see Common errors.

For more information about the AWS CLI, see AWS Command Line Interface reference for Aurora DSQL.

Troubleshooting issues in Aurora DSQL

> **Note**
>
> The following topics provide troubleshooting advice for errors and issues that you might encounter when using Aurora DSQL. If you find an issue that is not listed here, reach out to AWS support

Topics

- Troubleshooting connection errors
- Troubleshooting authentication errors
- Troubleshooting authorization errors
- Troubleshooting SQL errors
- Troubleshooting concurrency control responses
- Troubleshooting SSL/TLS connections
- Troubleshooting missing metrics from the Amazon CloudWatch Database Insights console

Troubleshooting connection errors

error: unrecognized SSL error code: 6 or **unable to accept connection, sni was not received**

You might be using a psql version earlier than version 14, which doesn't support Server Name Indication (SNI). The SNI is required when connecting to Aurora DSQL.

You can check your client version with `psql --version`.

error: NetworkUnreachable

A `NetworkUnreachable` error during connection attempts might indicate that your client doesn't support IPv6 connections, rather than signaling an actual network problem. This error commonly occurs on IPv4-only instances because of how PostgreSQL clients handle dual-stack connections. When a server supports dual-stack mode, these clients first resolve hostnames to both IPv4 and IPv6 addresses. They attempt an IPv4 connection first, then try IPv6 if the initial connection fails. If your system doesn't support IPv6, you'll see a general `NetworkUnreachable` error instead of a clear "IPv6 not supported" message.

Troubleshooting authentication errors

IAM authentication failed for user "..."

When you generate an Aurora DSQL IAM authentication token, the maximum duration you can set is 1 week. After one week, you can't authenticate with that token.

Additionally, Aurora DSQL rejects your connection request if your assumed role has expired. For example, if you try to connect with a temporary IAM role even if your authentication token hasn't expired, Aurora DSQL will reject the connection request.

To learn more about how IAM works with Aurora DSQL, see Understanding authentication and authorization for Aurora DSQL and AWS Identity and Access Management in Aurora DSQL.

An error occurred (InvalidAccessKeyId) when calling the GetObject operation: The AWS Access Key ID you provided does not exist in our records

IAM rejected your request. For more information, see Why requests are signed.

IAM role <role> does not exist

Aurora DSQL couldn't find your IAM role. For more information, see IAM roles.

IAM role must look like an IAM ARN

See IAM Identifiers - IAM ARNs for more information.

Wrong user to action mapping

This error occurs when the authentication token type does not match the database role. Aurora DSQL uses two token types: `DbConnectAdmin` for the `admin` role and `DbConnect` for custom database roles.

- If you see `Wrong user to action mapping. user: admin, action: DbConnect`, use `generate-db-connect-admin-auth-token` instead of `generate-db-connect-auth-token`.
- If you see `Wrong user to action mapping. user: myusername, action: DbConnectAdmin`, use `generate-db-connect-auth-token` instead of `generate-db-connect-admin-auth-token`.

Troubleshooting authorization errors

Role <role> not supported

Aurora DSQL doesn't support the `GRANT` operation. See Supported subsets of SQL commands in Aurora DSQL.

Cannot establish trust with role <role>

Aurora DSQL doesn't support the `GRANT` operation. See Supported subsets of SQL commands in Aurora DSQL.

Role <role> does not exist

Aurora DSQL couldn't find specified database user. See Authorize custom database roles to connect to a cluster.

ERROR: permission denied to grant IAM trust with role <role>

To grant access to a database role, you must be connected to your cluster with the admin role. To learn more, see Authorize database roles to use SQL in a database.

ERROR: role <role> must have the LOGIN attribute

Any database roles you create must have the `LOGIN` permission.

To address this error, make sure that you've created the PostgreSQL Role with the `LOGIN` permission. For more information, see CREATE ROLE and ALTER ROLE in the PostgreSQL documentation.

ERROR: role <role> cannot be dropped because some objects depend on it

Aurora DSQL returns an error if you drop a database role with an IAM relationship until you revoke the relationship using `AWS IAM REVOKE`. To learn more, see Revoking authorization.

Troubleshooting SQL errors

Error: Not supported

Aurora DSQL doesn't support all PostgreSQL-based dialect. To learn about what is supported, see Supported PostgreSQL features in Aurora DSQL .

Error: use `CREATE INDEX ASYNC` instead

To create an index on a table with existing rows, you must use the `CREATE INDEX ASYNC` command. To learn more, see Creating indexes asynchronously in Aurora DSQL.

Error: server unavailable (`SQLSTATE XX000`)

A server unavailable error indicates a temporary service condition. Your connection remains active, so you can retry without reconnecting.

Always retry the entire transaction from the beginning, including the `COMMIT` command, with exponential backoff and jitter.

> ⚠ **Uncertain commit outcome**
>
> If the error occurs while committing a transaction, the transaction might have committed even though your application received an error.

Design transactions to be idempotent when possible. For non-idempotent transactions, confirm whether Aurora DSQL applied the write before you retry the transaction.

Monitor the rate of server unavailable errors. Expect occasional retries. If the errors become sustained or affect application performance, contact AWS Support.

Troubleshooting concurrency control responses

OC000 "ERROR: change conflicts with another transaction (OC000)"

This transaction attempted to modify the same tuples as another concurrent transaction. This indicates contention on the modified tuples. To learn more, refer to Concurrency control in Aurora DSQL .

OC001 "ERROR: schema has been updated by another transaction (OC001)"

Your session had a cached copy of the schema catalog at version V1, loaded at time T1.

A separate transaction updated the catalog to version V2 at time T2.

At time T3, when your session runs a query, it detects that it's behind and attempts to rebase onto the new catalog changes. In some situations the rebase can't succeed, and Aurora DSQL returns

a 40001 OC001 response. The time between T2 and T3 can range from milliseconds to minutes, because query processors discover catalog changes reactively rather than receiving proactive updates.

When you retry from the same session, Aurora DSQL refreshes the catalog cache. The retried transaction uses catalog V2 and succeeds as long as no further catalog changes have occurred since T2.

Troubleshooting SSL/TLS connections

SSL error: certificate verify failed

This error indicates that the client cannot verify the server's certificate. Ensure that:

1. The Amazon Root CA 1 certificate is properly installed. See Configuring SSL/TLS certificates for Aurora DSQL connections for instructions on how to validate and install this certificate.
2. The PGSSLROOTCERT environment variable points to the correct certificate file.
3. The certificate file has the correct permissions.

Unrecognized SSL error code: 6

This error occurs with PostgreSQL clients below version 14. Upgrade your PostgreSQL client to version 17 to resolve this issue.

SSL error: unregistered scheme (Windows)

This is a known issue with the Windows psql client when using system certificates. Use the downloaded certificate file method described in the Connecting from Windows instructions.

Troubleshooting missing metrics from the Amazon CloudWatch Database Insights console

Aurora DSQL cluster not appearing in the Amazon CloudWatch Database Insights console

Amazon CloudWatch Database Insights populates its cluster selector using activity data from Aurora DSQL database insights. For more information about Aurora DSQL database insights, see

Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights. A cluster becomes visible in Database Insights only after the cluster has generated load within the last eight days that was captured by Aurora DSQL active session history (DASH) sampler.

DASH uses 1-second sampling, which might miss fast, infrequent transactions that complete in milliseconds.

Use the following steps to confirm whether this explains what you're seeing:

1. Confirm that you have run transactions on the cluster within the last eight days. A cluster that has been idle longer than that doesn't appear in Database Insights regardless of how you used it previously. To check for recent activity, view the `TotalTransactions` metric for your cluster in CloudWatch. For more information about this metric, see Observability and performance.
2. Run a sustained workload against the cluster so that at least one session stays active. Examples include a load-testing script, a batch of inserts, or a long-running query. Because DASH samples once per second, a brief workload might not be captured, so the more continuous the activity, the more likely it is to appear.
3. Wait a few minutes after running that workload, then refresh the Database Insights console.
4. Verify that you're viewing the same AWS Region and account where the cluster was created. Selecting the wrong Region or account is a common reason a cluster appears to be missing.

Providing feedback on Amazon Aurora DSQL

If you encounter features that are critical for your migration but not currently supported in Aurora DSQL, AWS provides several channels for feedback:

Feedback channels

Aurora DSQL Discord server

Join the Aurora DSQL Discord server to connect with the AWS team and community. Share feature requests, discuss migration challenges, and get real-time feedback.

AWS Support

If you have an AWS Support plan, create a support case to discuss your specific requirements and timeline needs.

AWS re:Post

Use AWS re:Post to ask questions and share feedback with the community and AWS experts.

Effective feature requests

When requesting features, provide:

- **Use case description:** Explain what you're trying to accomplish and why
- **Current workaround:** Describe any alternatives you've tried
- **Business impact:** Explain how the missing feature affects your migration timeline or application functionality
- **Priority level:** Indicate whether this is blocking your migration or would be a nice-to-have improvement

Release notes for Aurora DSQL

This page describes new features, service launches, and important updates for Aurora DSQL. For documentation-specific changes, see the Document history.

2026

October 2026

October 16, 2026

Feature — **CDC transaction field** — CDC envelope version `1.1` adds a `transaction` field, which you can use to reconstruct the net changes made by a transaction across rows and tables. For more information, see Record payload.

September 2026

September 15, 2026

PostgreSQL Compatibility — **Partial indexes** — Aurora DSQL now supports partial indexes. Add a `WHERE` clause to `CREATE INDEX` to index only the rows that satisfy a predicate, tailoring the index to common query filter patterns. Aurora DSQL can use a partial index for a query when it can prove that the query's `WHERE` conditions imply the index's predicate. For more information, see CREATE INDEX.

September 11, 2026

Feature — **Batched nested-loop joins** — Aurora DSQL can now use batched nested-loop joins for eligible queries. This join strategy batches multiple outer rows and probes the inner side for the batch, which can reduce round trips between the compute and storage layers for selective joins. In EXPLAIN output, this node appears as `Nested Loop (Batched Join)`. You can enable or disable planner use in a session with `dsql.enable_batched_nestloop`. For more information, see Batched nested-loop joins in Aurora DSQL EXPLAIN plans.

August 2026

August 27, 2026

PostgreSQL Compatibility — **Extended statistics** — Aurora DSQL now supports the `CREATE STATISTICS`, `ALTER STATISTICS`, and `DROP STATISTICS` commands. You can create extended statistics on correlated columns and expressions so that the query planner produces more accurate row estimates. For more information, see CREATE STATISTICS, ALTER STATISTICS, and DROP STATISTICS.

August 26, 2026

PostgreSQL Compatibility — **Foreign key constraints** — Aurora DSQL now supports foreign key constraints. You can define foreign key constraints when you create a table, or add them to an existing table with `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`. For more information, see Working with foreign key constraints.

August 26, 2026

PostgreSQL Compatibility — **Compression for character data types** — Aurora DSQL now compresses large values in `text`, `varchar`, and `bpchar` columns, storing text-heavy payloads more efficiently. Aurora DSQL compresses these values only in columns that aren't part of a key. Values in primary key columns and in the key columns of a secondary index are always stored uncompressed. For more information, see Character data types.

August 25, 2026

Developer Tools — **Aurora DSQL Adapter for Drizzle ORM** — Added the Aurora DSQL Adapter for Drizzle ORM, a TypeScript object-relational mapping (ORM) framework for Node.js. With this adapter, you can authenticate with IAM tokens and pool connections automatically. The adapter also provides opt-in optimistic concurrency control (OCC) retry. A migrator applies one DDL statement per transaction and waits for asynchronous DDL jobs. A CLI rewrites migration SQL for Aurora DSQL compatibility. For more information, see Aurora DSQL adapters and dialects.

August 25, 2026

PostgreSQL Compatibility — **`numeric` precision and scale limits** — Aurora DSQL now supports higher precision and scale limits for `numeric` data type. The max precision is now 1000 and scale can be between -1000 to 1000 when you declare numerics with explicit precision and scale.

August 25, 2026

PostgreSQL Compatibility — **`SELECT FOR KEY SHARE`** — Aurora DSQL now supports the `SELECT ... FOR KEY SHARE` clause, which declares that a transaction depends on the key columns of the selected rows. For more information about how Aurora DSQL detects conflicts between reads and concurrent writes, see Concurrency control in Aurora DSQL.

August 24, 2026

PostgreSQL Compatibility — **`SELECT FOR UPDATE` expanded support** — Now in Aurora DSQL you can use `FOR UPDATE` in a `SELECT` without specifying equality predicates on all primary key columns. You can also use `FOR UPDATE` with queries that span multiple tables. Transaction limits still apply.

August 17, 2026

Monitoring — **Aurora DSQL Database Insights** — Aurora DSQL now supports Aurora DSQL Database Insights, per-statement cluster-level performance monitoring capability, which samples active sessions to give visibility into database load by wait event and SQL statement. This release also adds troubleshooting guidance for clusters that don't appear in the Amazon CloudWatch Database Insights console. For more information, see Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights and Troubleshooting missing metrics from the Amazon CloudWatch Database Insights console.

August 13, 2026

PostgreSQL Compatibility — **Indexes on expressions** — Aurora DSQL now supports indexes on expressions. An index key can be an expression computed from one or more columns of the table row, so a query that filters on the same expression can use the index. For example, an index on `lower(title)` gives efficient case-insensitive searches. All functions and operators used in an index definition must be immutable. For more information, see CREATE INDEX.

August 3, 2026

PostgreSQL Compatibility — **`ALTER TABLE DROP COLUMN`** — Aurora DSQL now supports `ALTER TABLE ... DROP COLUMN` to remove a column from an existing table, You can drop multiple columns in a single statement. Dropping a primary key column isn't supported. For more information, see ALTER TABLE.

August 3, 2026

PostgreSQL Compatibility — **`ALTER TABLE ADD CONSTRAINT` and `VALIDATE CONSTRAINT`** — Aurora DSQL now supports `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` for

adding `CHECK` constraints without validating existing data, and `ALTER TABLE ASYNC ... VALIDATE CONSTRAINT` for validating existing data against constraints later. The `NOT VALID` option allows adding constraints to large tables without blocking operations. The validation runs as an asynchronous DDL job that can be monitored using `sys.jobs`. For more information, see ALTER TABLE.

July 2026

July 31, 2026

Region Expansion — **Aurora DSQL available in four additional Regions** — Multi-Region clusters are now available in Europe (Stockholm), Europe (Spain), Asia Pacific (Mumbai), and Asia Pacific (Singapore). For a complete list of supported Regions, see Region availability for Aurora DSQL.

July 30, 2026

Feature — **Cost visibility and alerting** — With the Aurora DSQL console, you can now view accumulated Aurora DSQL distributed processing units (DPU) and more easily establish real-time alerts based on DPU usage.

July 15, 2026

AI Integration — **Aurora DSQL Assist in the Aurora DSQL Playground** — Aurora DSQL Assist is a new artificial intelligence (AI) assistant in the Aurora DSQL Playground that helps you go from idea to working query. You can generate and fix SQL from natural language and design schemas and indexes for the distributed architecture of Aurora DSQL. You can also explore errors and documentation conversationally. For more information about Aurora DSQL Assist, see the Aurora DSQL Playground website.

July 6, 2026

PostgreSQL Compatibility — **Additional ALTER TABLE operations** — Aurora DSQL now supports additional `ALTER TABLE` operations: `SET DEFAULT`, `DROP DEFAULT`, `DROP NOT NULL`, `DROP EXPRESSION`, `ADD GENERATED AS IDENTITY`, and `DROP CONSTRAINT`. These metadata DDL operations enable more flexible schema evolution without requiring table recreation. For more information, see ALTER TABLE.

July 8, 2026

Feature — **Change data capture now generally available** — Change data capture (CDC) for Aurora DSQL is now generally available. This release adds distinct operation types

(op: "u" for updates, op: "c" for inserts) and documents write-set compaction, which consolidates multiple changes to the same row within a transaction into a single CDC record. If an application inserts and deletes a row in the same transaction, Aurora DSQL recognizes zero net change and omits the row from the stream. For more information, see CDC streams.

June 2026

June 8, 2026

PostgreSQL Compatibility — **JSONB data type with compression** — Aurora DSQL now supports the PostgreSQL JSONB data type, with optional compression for stored data. You can use JSONB columns to store semi-structured data such as system configuration metadata, API parameters, and event logs alongside relational data, and existing code and tools that depend on PostgreSQL's JSONB type work with Aurora DSQL without modification. Aurora DSQL enables PostgreSQL compression by default, storing larger JSONB payloads more efficiently. For more information, see Supported data types in Aurora DSQL.

May 2026

May 14, 2026

Feature — **Change data capture (Preview)** — Aurora DSQL introduces support for change data capture (CDC) in preview, enabling you to stream real-time database changes directly to Amazon Kinesis Data Streams. This fully managed capability removes the need to build or maintain custom streaming pipelines, making it easier to build event-driven applications, power real-time analytics pipelines, and synchronize data across systems. For more information, see Getting started with CDC.

May 11, 2026

Region Expansion — **Aurora DSQL available in five additional Regions** — Single-Region clusters are now available in Asia Pacific (Hong Kong), Asia Pacific (Mumbai), Asia Pacific (Singapore), Europe (Stockholm), and South America (São Paulo). For a complete list of supported Regions, see Region availability for Aurora DSQL.

May 4, 2026

PostgreSQL Compatibility — **JSON data type with compression** — Aurora DSQL now supports the PostgreSQL JSON data type with optional compression. You can store semi-structured

data such as API payloads, configuration objects, or event logs alongside relational data. Aurora DSQL enables PostgreSQL compression by default, storing larger `JSON` payloads more efficiently.

April 2026

April 28, 2026

Feature — **Cost Estimator in the Aurora DSQL Playground** — The Aurora DSQL Playground now includes a Cost Estimator that helps you estimate the cost of running your workload on Aurora DSQL. You can pin queries to build a representative workload and review the estimated Distributed Processing Units (DPUs) those queries consume. The estimator also shows how the AWS Free Tier allowance of 100,000 DPUs per month applies to your workload.

To get started with the Cost Estimator, see the Aurora DSQL Playground website. For more information about how Aurora DSQL meters usage, see Billing and Metering in Aurora DSQL.

April 24, 2026

PostgreSQL Compatibility — **ALTER DEFAULT PRIVILEGES support** — Aurora DSQL now supports PostgreSQL's `ALTER DEFAULT PRIVILEGES` for tables, sequences, functions, and types. Newly created objects automatically inherit configured default privileges, removing the need for manual `GRANT` statements after each object creation.

April 24, 2026

PostgreSQL Compatibility — **CREATE SCHEMA with nested statements** — Aurora DSQL now supports nested `CREATE SEQUENCE` and `GRANT` statements within a `CREATE SCHEMA` command, enabling developers to define schema objects and their permissions in a single atomic operation.

April 24, 2026

PostgreSQL Compatibility — **ALTER USER/ROLE RENAME support** — Aurora DSQL now supports `ALTER USER/ROLE/GROUP ... RENAME TO`, enabling administrators to rename database roles without dropping and recreating them. The rename propagates across all active sessions through catalog version notifications.

April 24, 2026

PostgreSQL Compatibility — **ALTER ROUTINE support** — Aurora DSQL now supports `ALTER ROUTINE ... SET SCHEMA`, `SET OWNER`, and `RENAME TO`, along with `GRANT`, `REVOKE`,

`COMMENT ON`, and `DROP ROUTINE`. The `ROUTINE` keyword is a PostgreSQL umbrella term covering both functions and procedures, enabling object-relational mapping (ORM) and migration tools that emit `ROUTINE` syntax to work without modification.

April 21, 2026

Feature — **AWS Backup adds Aurora DSQL support for AWS Organizations backup policies** — AWS Backup now supports Aurora DSQL clusters as a resource type in AWS Organizations backup policies. Organization administrators can define backup policy rules that directly target Aurora DSQL clusters across member accounts, providing more precise control over which resources are included in Organization-wide backup plans.

April 13, 2026

Developer Tools — **Aurora DSQL Connector for PHP (`PDO_PGSQL`)** — Added the Aurora DSQL Connector for PHP that handles IAM token generation, Secure Sockets Layer (SSL) configuration, and connection pooling for PHP applications. The connector provides opt-in optimistic concurrency control (OCC) retry with exponential backoff. For more information, see Connecting to Aurora DSQL clusters with a PHP connector.

March 2026

March 31, 2026

Developer Tools — **Aurora DSQL Connectors for .NET (`Npgsql`) and Rust (`SQLx`)** — Added connectors for .NET and Rust that handle IAM token generation, SSL configuration, and connection pooling. Both connectors provide opt-in OCC retry with exponential backoff, custom IAM credential providers, and AWS profile support. For more information, see Aurora DSQL connectors.

March 30, 2026

Performance — **Faster INSERT ON CONFLICT statements** — Aurora DSQL improves write performance for `INSERT ON CONFLICT` statements. The improvement scales with the number of rows proposed for insertion per transaction, so workloads that batch many `INSERT ON CONFLICT` statements into a single transaction see the largest gains.

March 30, 2026

Performance — **Faster uniqueness validation for large insert workloads** — Aurora DSQL now validates secondary unique-index constraints in batches rather than one key at a time, accelerating large insert and migration workloads. The speedup is most pronounced for

transactions that touch many rows or tables with multiple unique secondary indexes — workloads where uniqueness validation was previously the bottleneck. For example, inserting 1,000 rows into a table with one unique secondary index now completes in 577 ms, down from 1,041 ms (1.8x faster).

March 26, 2026

Developer Tools — **Aurora DSQL Connector for Ruby (pg gem)** — Added the Aurora DSQL Connector for Ruby that handles IAM token generation, SSL configuration, and connection pooling. The connector provides opt-in OCC retry with exponential backoff and custom IAM credential providers. For more information, see Connecting to Aurora DSQL clusters with a Ruby pg connector.

February 2026

February 25, 2026

Developer Tools — **Integrations for Visual Studio Code SQLTools and DBeaver** — Added the Aurora DSQL Driver for SQLTools and the Aurora DSQL Plugin for DBeaver Community Edition. Both integrations simplify database connectivity by automatically handling IAM authentication and transparently managing access tokens. The SQLTools driver is also available on Open VSX Registry for use with VS Code-compatible editors such as Cursor and Kiro.

February 25, 2026

Feature — **Aurora DSQL Playground** — AWS launches a browser-based playground that enables developers to interact with an Aurora DSQL database without requiring an AWS account. With zero setup or infrastructure configuration, developers can create schemas, load data, and execute SQL queries directly from their browser with built-in sample datasets. To start exploring, visit the Aurora DSQL Playground.

February 25, 2026

Developer Tools — **Support for Tortoise, Flyway, and Prisma** — Added integrations for popular ORM and database migration tools: an adapter for Tortoise (Python ORM), a dialect for Flyway (schema management tool), and CLI tools for Prisma (Node.js ORM). These integrations handle IAM authentication and Aurora DSQL-specific compatibility requirements automatically.

February 19, 2026

Developer Tools — **Go, Python, and Node.js connectors with WebSocket support** — Added Aurora DSQL Connectors for Go (`pgx`), Python (`asyncpg`), and Node.js (WebSocket

for `Postgres.js`) that simplify IAM authentication. Each connector automatically generates tokens for each connection. The `Postgres.js` connector additionally supports WebSocket protocol for environments where Transmission Control Protocol (TCP) connections aren't available. For more information, see Aurora DSQL connectors.

February 18, 2026

AI Integration — **Aurora DSQL integrates with Kiro powers and AI agent skills** — Aurora DSQL integration with Kiro powers and AI agent skills enables developers to build Aurora DSQL-backed applications faster with AI-assisted development. The Kiro power bundles the Aurora DSQL Model Context Protocol (MCP) server with development best practices for one-click installation. The Aurora DSQL skill extends the same capabilities to additional AI coding agents through the Skills CLI. For more information, see Aurora DSQL Steering: Skills and Powers.

February 13, 2026

PostgreSQL Compatibility — **Identity columns and sequence objects** — Aurora DSQL now supports identity columns and sequence objects, enabling developers to generate auto-incrementing, integer-based IDs directly in the database using familiar SQL patterns. For more information, see Sequences and identity columns.

February 11, 2026

Region Expansion — **Aurora DSQL available in Canada and Australia Regions** — Aurora DSQL is now available with single-Region clusters in Asia Pacific (Melbourne), Asia Pacific (Sydney), Canada (Central), and Canada West (Calgary). For a complete list of supported Regions, see Region availability for Aurora DSQL.

February 5, 2026

Developer Tools — **Aurora DSQL Connector for Go** — Added the Aurora DSQL Connector for Go, which wraps `pgx` with automatic IAM authentication for token generation, SSL configuration, and connection management. For more information, see Connecting to Aurora DSQL clusters with a Go connector.

February 3, 2026

PostgreSQL Compatibility — **Index support for the NUMERIC data type** — Aurora DSQL now supports indexes on the `NUMERIC` data type, allowing customers to use `NUMERIC` columns as primary keys and in secondary indexes for improved query performance on decimal and financial data.

January 2026

January 15, 2026

Integration — **Aurora DSQL available in v0 by Vercel** — AWS databases including Aurora DSQL are now available in v0 by Vercel, enabling developers to quickly spin up and connect to Aurora DSQL databases directly from the Vercel platform.

2025

December 2025

December 19, 2025

AI Integration — **AWS Labs Aurora DSQL MCP Server updates** — Updated MCP Server with installation approaches for Claude Code and Codex, including CLI-based setup. For more information, see AWS Labs Aurora DSQL MCP Server.

December 17, 2025

Integration — **Aurora DSQL available in the Vercel Marketplace** — AWS databases including Aurora DSQL are now available on the Vercel Marketplace, allowing developers to provision Aurora DSQL clusters in seconds directly from Vercel's developer platform.

December 11, 2025

Feature — **Fast cluster creation** — Aurora DSQL now supports cluster creation in seconds, enabling developers to instantly provision Aurora DSQL databases to rapidly prototype new ideas and leverage the integrated query editor in the AWS Management Console to immediately start building.

December 11, 2025

Feature — **AWS PrivateLink with Direct Connect support** — Added private Domain Name System (DNS) setup and the cluster ID connection option for AWS PrivateLink with Direct Connect or Amazon VPC peering. For more information, see Managing and connecting to Aurora DSQL clusters using AWS PrivateLink.

November 2025

November 21, 2025

Developer Tools — **Python, Node.js, and JDBC Connectors** — Aurora DSQL launches new Python, Node.js, and Java Database Connectivity (JDBC) connectors that simplify IAM authorization, enabling developers to connect to Aurora DSQL clusters using standard PostgreSQL drivers with automatic IAM token management. For more information, see Aurora DSQL connectors.

November 21, 2025

Feature — **Storage volume increased to 256 TiB** — Aurora DSQL database clusters now support up to 256 terabytes of storage volume, doubling the previous 128 terabyte limit and allowing customers to store and manage larger databases within a single database cluster.

November 21, 2025

Feature — **Integrated Query Editor in AWS Management Console** — Aurora DSQL now provides an integrated query editor for browser-based SQL access in the AWS Management Console. Customers can securely connect to their Aurora DSQL clusters and run SQL queries directly from the console without installing or configuring external clients. For more information, see Getting started with the Aurora DSQL Query Editor.

November 20, 2025

Feature — **Statement-level cost estimates in query plans** — Aurora DSQL now provides statement-level cost estimates and query plans, surfacing per-category breakdowns including compute, read, write, and multi-Region costs with total estimated DPUs to help developers immediately get visibility into their estimated cost profile. For more information, see Working with Aurora DSQL explain plans.

November 20, 2025

Integration — **Using JupyterLab with Aurora DSQL** — Published a step-by-step guide for connecting Aurora DSQL with JupyterLab for both local and Amazon SageMaker AI environments. For more information, see Using JupyterLab with Aurora DSQL.

October 2025

October 31, 2025

Feature — **FIPS 140-3 compliant endpoints** — Aurora DSQL now supports Federal Information Processing Standards (FIPS) 140-3 compliant endpoints, enabling customers in regulated industries and government sectors to meet federal security requirements when connecting to Aurora DSQL clusters.

October 24, 2025

Feature — **Resource-based policy support** — Aurora DSQL now supports resource-based policies, allowing customers to attach inline policies directly to Aurora DSQL clusters for fine-grained access control with new permissions: `PutClusterPolicy`, `GetClusterPolicy`, and `DeleteClusterPolicy`. For more information, see Resource-based policies for Aurora DSQL.

October 23, 2025

Region Expansion — **Aurora DSQL available in Europe (Frankfurt)** — Aurora DSQL is now available in the Europe (Frankfurt) Region, expanding availability for European customers who need to deploy distributed SQL databases with data residency in the European Union (EU).

August 2025

August 26, 2025

Feature — **AWS Fault Injection Service integration** — Aurora DSQL now integrates with AWS Fault Injection Service (AWS FIS), allowing customers to inject failures into single-Region and multi-Region Aurora DSQL clusters to test fault tolerance and validate application error handling and recovery mechanisms.

July 2025

July 29, 2025

Feature — **AWS Backup improves Aurora DSQL multi-Region restore workflow** — AWS Backup improves the Aurora DSQL multi-Region restore workflow, simplifying the process of restoring Aurora DSQL backups across AWS Regions for disaster recovery and data migration scenarios. For more information, see Backup and restore for Aurora DSQL.

July 3, 2025

Region Expansion — **Aurora DSQL available in Asia Pacific (Seoul)** — Aurora DSQL is now available in the Asia Pacific (Seoul) Region, expanding availability for customers in South Korea and the broader Asia Pacific region.

May 2025

May 27, 2025

Major Release — **General availability of Aurora DSQL** — Aurora DSQL is now generally available (GA). Aurora DSQL is a serverless, distributed SQL database with PostgreSQL compatibility, active-active high availability, and virtually unlimited scale. The GA release includes CloudWatch monitoring, AWS Backup integration, and data encryption with customer-managed KMS keys. For more information, see What is Aurora DSQL?

May 8, 2025

Feature — **AWS PrivateLink for Aurora DSQL** — Aurora DSQL now supports AWS PrivateLink, enabling customers to simplify private network connectivity between virtual private clouds (VPCs), Aurora DSQL, and on-premises data centers using interface Amazon VPC endpoints and private IP addresses. For more information, see Managing and connecting to Aurora DSQL clusters using AWS PrivateLink.

2024

December 2024

December 3, 2024

Major Release — **Public preview launch of Aurora DSQL** — Aurora DSQL launches in public preview as a new serverless distributed SQL database. Aurora DSQL offers virtually unlimited scale, PostgreSQL v16 compatibility, multi-Region active-active support with a 99.999% availability service level agreement (SLA), IAM authentication, and optimistic concurrency control. For more information, see What is Aurora DSQL?

Document history for the Amazon Aurora DSQL User Guide

The following table describes the documentation releases for Aurora DSQL.

Change	Description	Date
AuroraDsqlServiceLinkedRolePolicy update	Aurora DSQL updated the `AuroraDsqlServiceLinkedRolePolicy` managed policy to add the `logs:CreateLogGroup` , `logs:CreateLogStream` , `logs:PutLogEvents` , and `logs:PutRetentionPolicy` permissions on the `/aws/dsql/normalized-sql-text` log group and its log streams in your own account. With these permissions, an upcoming Aurora DSQL feature can deliver normalized SQL text to your account. For more information, see Aurora DSQL updates to AWS managed policies and Using service-linked roles in Aurora DSQL.	November 11, 2026
New content: Aurora DSQL adapters and dialects	Added documentation for the Aurora DSQL dialect for Hibernate, adapter for Django, dialect for SQLAlchemy, and adapter for Tortoise ORM. These extensions provide IAM	October 29, 2026

	authentication and improved Aurora DSQL compatibility for object-relational mapping (ORM) frameworks. For more information, see Adapters and dialects for Aurora DSQL.	
CDC transaction field	CDC envelope version `1.1` adds a `transaction` field that helps consumers reconstruct the net changes made by a transaction across rows and tables. It is available in all AWS Regions where Aurora DSQL is available. See the AWS Region table on the AWS website for regional availability. For more information about the `transaction` field, see Record payload.	October 16, 2026
CDC envelope versioning	Added guidance for handling CDC envelope versions for streams created on or after October 1, 2026, including backward-compatible additions to fields and enum values. For more information, see CDC envelope versioning.	September 22, 2026

New content: CREATE FUNCTION and ALTER FUNCTION	Added documentation for `CREATE FUNCTION` and `ALTER FUNCTION`. `CREATE FUNCTION` only supports SQL-language functions. `ALTER FUNCTION` supports the `RENAME TO`, `OWNER TO`, and `SET SCHEMA` forms. For more information, see CREATE FUNCTION and ALTER FUNCTION.	September 17, 2026
New content: partial indexes	Added documentation for partial indexes. Add a `WHERE` clause to `CREATE INDEX` to index only the rows that satisfy a predicate, which reduces index size and maintenance cost. Aurora DSQL can use a partial index for a query when it can prove that the query's `WHERE` conditions imply the index's predicate. For more informati on, see CREATE INDEX.	September 15, 2026

New content: batched nested-loop joins in EXPLAIN plans	Added documentation for batched nested-loop joins in Aurora DSQL, including when the planner chooses the `Nested Loop (Batched Join)` node, how to interpret `Recheck Cond` in EXPLAIN output, and how to control planner use with `dsql.enable_batched_nestloop` . Updated the session parameter reference to include the new setting. For more information, see Batched nested-loop joins in Aurora DSQL EXPLAIN plans.	September 11, 2026
New troubleshooting guidance for server unavailable errors	Added guidance for handling `ERROR: server unavailable (SQLSTATE XX000)`. Applications can retry the entire transaction without reconnecting. The guidance also explains how to handle uncertain commit outcomes and non-idempotent transactions. For more information, see Error: server unavailable (SQLSTATE XX000).	September 8, 2026

New content: extended statistics commands	Added documentation for the `CREATE STATISTICS`, `ALTER STATISTICS`, and `DROP STATISTICS` commands in Aurora DSQL. You can create extended statistics to give the query planner more accurate row estimates for correlated columns and expressions. For more information, see CREATE STATISTICS, ALTER STATISTICS, and DROP STATISTICS.	August 27, 2026
Updated content: Compression for character data types	Aurora DSQL now compresses large values in `text`, `varchar`, and `bpchar` columns that aren't part of a key. For more information, see Supported data types in Aurora DSQL.	August 26, 2026

New content: foreign key constraints	Added documentation for foreign key constraints in Aurora DSQL. You can define foreign key constraints when you create a table, or add them to an existing table with `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`. You can then validate existing data with `VALIDATE CONSTRAINT .` Updated CREATE TABLE, ALTER TABLE, and added a new SET CONSTRAINTS reference. For more informati on, see Working with foreign key constraints.	August 26, 2026
New content: Drizzle ORM adapter	Added support for Drizzle ORM, a TypeScript object-relational mapping (ORM) framework for Node.js. With the Aurora DSQL Adapter for Drizzle ORM, you can use Drizzle ORM with Aurora DSQL clusters. For more information, see Aurora DSQL adapters and dialects.	August 26, 2026
Updated numeric data type precision and scale limits	Updated precision and scale limits for `numeric` data type in Aurora DSQL to 1000. For more information, see Supported data types in Aurora DSQL	August 25, 2026

New content: SELECT FOR KEY SHARE and concurrency control	Added documentation for the SELECT ... FOR KEY SHARE clause and a new Data conflict types section that explains how Aurora DSQL detects conflicts between SELECT ... FOR UPDATE and SELECT ... FOR KEY SHARE reads and concurrent writes at commit time. For more information, see Concurrency control in Aurora DSQL.	August 25, 2026
Updated SELECT FOR UPDATE documentation	The SELECT ... FOR UPDATE entry now reflects support for broader query patterns. For more information, see Supported SQL.	August 24, 2026
Updated SQL reference	Updated the supported SQL overview to include ALTER DEFAULT PRIVILEGES, CREATE SCHEMA, role rename operations, and routine management commands. Also updated the CREATE INDEX entry to describe expression indexes and other supported clauses. For more information, see Supported SQL for Aurora DSQL.	August 20, 2026

New content: Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights	Added documentation for Aurora DSQL database insights, which samples active sessions on Aurora DSQL cluster to help customers understand database load by wait event and SQL statement, and describes how to access this data through Amazon CloudWatch Database Insights and Prometheus Query Language (PromQL). Added troubleshooting guidance for clusters that don't appear in the CloudWatch Database Insights console. For more information, see Monitoring Aurora DSQL clusters with Aurora DSQL Database Insights and Troubleshooting missing metrics from the Amazon CloudWatch Database Insights console.	August 17, 2026

Updated content: Reorganized CREATE INDEX documentation	Moved the `CREATE INDEX` syntax and parameter reference to a new CREATE INDEX page alongside the other supported subsets of SQL commands. The Asynchronous indexes in Aurora DSQL page now focuses on asynchronous build behavior, job monitoring, and examples, and links to the new page for syntax and parameters.	August 13, 2026
New content: Indexes on expressions	Added documentation for indexes on expressions. An index key can be an expression computed from one or more columns of the table row, such as `lower(title)` , which gives fast access to data based on a transformation of the underlying values. All functions and operators used in an index definition must be immutable. Expressions aren't supported as `INCLUDE` columns. For more information, see CREATE INDEX.	August 13, 2026

New content: ALTER TABLE DROP COLUMN	Added documentation for `ALTER TABLE ... DROP COLUMN`, which removes a column from an existing table without requiring table recreation. You can drop multiple columns in a single statement. Dropping a primary key column isn't supported. For more information, see ALTER TABLE.	August 3, 2026
New content: ALTER TABLE ADD CONSTRAINT and VALIDATE CONSTRAINT	Added support for `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` to add `CHECK` constraints without validating existing data, and `ALTER TABLE ASYNC ... VALIDATE CONSTRAINT` to validate existing data against constraints later. The validation runs as an asynchronous DDL job. This allows adding constraints to large tables without blocking concurrent operations. For more information, see ALTER TABLE.	August 3, 2026

Updated content: CDC streams generally available	CDC streams are now generally available. This release adds distinct op codes for inserts and updates, and a new Write-set compaction section with compaction rules. For more information, see CDC streams.	July 1, 2026
New content: JSONB data type	Added documentation for the `jsonb` data type in Aurora DSQL, including automatic compression behavior, the 1 MiB compressed size limit, and supported JSONB operators. Updated the query runtime data types section to reflect that `jsonb` is now available as a native storage type. For more information, see Supported data types in Aurora DSQL.	June 5, 2026
New content: Change data capture (CDC) streams	Added documentation for Aurora DSQL change data capture (CDC) streams, which capture committed row-level changes and deliver them to Amazon Kinesis Data Streams. For more information, see Change data capture (CDC) streams.	May 13, 2026

New content: STORAGE keyword support in CREATE TABLE and ALTER TABLE	Added documentation for the `STORAGE` clause in `CREATE TABLE` and `ALTER TABLE`, which sets the storage mode (`PLAIN`, `EXTERNAL`, `EXTENDED`, `MAIN`, or `DEFAULT`) for a column. This can be used to control compression behavior for variable-length data types such as `json`. For more information, see CREATE TABLE and ALTER TABLE.	May 1, 2026
New content: JSON data type	Added documentation for the `json` data type in Aurora DSQL, including automatic compression behavior, the 1 MiB compressed size limit, and supported JSON functions and operators. Updated the JSONB runtime type guidance to reflect the availability of native `json` storage. For more informati on, see Supported data types in Aurora DSQL.	May 1, 2026
New content: ALTER TABLE ADD CONSTRAINT USING INDEX	Added documentation for `ALTER TABLE ... ADD` *`table_constraint_u sing_index`* , which adds a `UNIQUE` constraint to a table based on an existing unique index. For more information, see ALTER TABLE.	April 20, 2026

New content: Aurora DSQL Connector for PHP PDO_PGSQL	Added documentation for the Aurora DSQL Connector for PHP PDO_PGSQL , which wraps PDO_PGSQL with automatic IAM authentication. The connector handles token generation, SSL configuration, and connection managemen t for PHP applications. For more information, see Connecting to Aurora DSQL clusters with a PHP connector.	April 10, 2026
New content: Aurora DSQL Connector for Rust SQLx	Added documentation for the Aurora DSQL Connector for Rust SQLx, which wraps SQLx with automatic IAM authentication. The connector handles token generatio n, SSL configuration, and connection management for Rust applications. For more information, see Connecting to Aurora DSQL clusters with a Rust connector.	March 27, 2026

New content: Aurora DSQL Connector for .NET Npgsql	Added documentation for the Aurora DSQL Connector for .NET Npgsql, which wraps Npgsql with automatic IAM authentication. The connector handles token generatio n, SSL configuration, and connection managemen t for .NET applications. For more information, see Connecting to Aurora DSQL clusters with a .NET Npgsql connector.	March 20, 2026
Updated content: SQL command reference and system queries	Added START TRANSACTI ON and ROLLBACK to the transaction control commands reference, with END and ABORT as aliases. Added useful system queries for retrieving Aurora DSQL and PostgreSQL version information. For more information, see PostgreSQL compatibility reference.	March 13, 2026
Updated content: Loading data into Aurora DSQL	Updated the data loading guide with client-side \copy usage, INSERT best practices , and guidance for pre-creat ing tables before loading. For more information, see Loading data into Aurora DSQL.	March 13, 2026

New content: Aurora DSQL Connector for Ruby pg	Added documentation for the Aurora DSQL Connector for Ruby pg, which wraps the pg gem with automatic IAM authentication. The connector handles token generatio n, SSL configuration, and connection management for Ruby applications. For more information, see Connecting to Aurora DSQL clusters with a Ruby pg connector.	March 12, 2026
Updated content: Asynchron ous DDL jobs	Updated the `sys.jobs` documentation with expanded details on monitoring and managing asynchronous DDL operation s. For more information, see PostgreSQL compatibility reference.	March 6, 2026
Updated content: PHP authentication token generation	Added a PHP SDK tab to the authentication token generation page. For more information, see Generating authentication tokens.	March 5, 2026
New content: Loading data into Aurora DSQL	Added a guide for loading data into Aurora DSQL clusters, including usage of the Aurora DSQL loader utility. For more information, see Loading data into Aurora DSQL.	March 5, 2026

New content: Sequences and identity columns	Added support for sequences and identity columns. New SQL command reference pages for CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE, and sequence manipulation functions . Updated CREATE TABLE and ALTER TABLE to include identity column syntax. Added a new guide for choosing identifier types and cache sizes. For more information, see Sequences and identity columns.	February 11, 2026
New content: Aurora DSQL Connector for Go	Added documentation for the Aurora DSQL Connector for Go, which wraps pgx with automatic IAM authentication. The connector handles token generation, SSL configuration, and connection management for Go applications. For more information, see Connecting to Aurora DSQL clusters with a Go connector.	February 5, 2026

Updated content: Amazon Aurora DSQL cluster connectivity tools	Reorganized the cluster connectivity tools documentation to clarify the distinction between AWS-provided connectors, adapters, and third-party tools. Added missing links to code samples. For more information, see Amazon Aurora DSQL cluster connectivity tools.	January 26, 2026
New content: Aurora DSQL Plugin for DBeaver Community Edition	Added documentation for the Aurora DSQL Plugin for DBeaver Community Edition, which enables IAM authentication and simplified connection setup for Aurora DSQL clusters. Includes installation instructions, connection configuration, and troubleshooting guidance. For more information, see Use DBeaver to access Aurora DSQL.	January 26, 2026
New content: Aurora DSQL Driver for SQLTools	Added documentation for the Aurora DSQL Driver for SQLTools, a Visual Studio Code extension that enables developers to connect to and query Aurora DSQL databases directly from VS Code with automatic IAM authentication. For more information, see Use Aurora DSQL driver for SQLTools.	January 26, 2026

Updated content: Aurora DSQL Steering: Skills and Powers	Added documentation to support installation using the Skills CLI for agent-agnostic support. The Skills CLI provides a simplified setup method that works across multiple AI coding assistants including Claude Code, Cursor, Copilot, Gemini, and others. For more information, see Aurora DSQL steering: Plugins, skills, and powers.	January 23, 2026
New content: Aurora DSQL Adapter for Tortoise ORM	Added support for Tortoise ORM, a Python async ORM framework. The Aurora DSQL Adapter for Tortoise ORM enables developers to use Tortoise ORM with Aurora DSQL clusters. For more information, see Aurora DSQL adapters and dialects.	January 23, 2026
New content: Aurora DSQL Steering: Skills and Powers	Added new documentation for configuring AI steering with Aurora DSQL using skills and powers. Includes setup instructions for Kiro Powers, Claude Skills, Gemini Skills, and Codex Skills. For more information, see Aurora DSQL steering: Plugins, skills, and powers.	January 16, 2026

Numeric data type index support	Added index support for the `numeric` data type in Aurora DSQL. You can now use `numeric` columns as primary keys and in secondary indexes. For more informati on, see Supported data types in Aurora DSQL.	January 13, 2026
Updated content: Supported subsets of SQL commands	Reorganized SQL command documentation into separate pages for improved navigatio n and clarity. Each command (CREATE TABLE, ALTER TABLE, CREATE VIEW, ALTER VIEW, DROP VIEW) now has its own dedicated page. For more information, see Supported subsets of SQL commands.	January 6, 2026
Updated content: AWS Labs Aurora DSQL MCP Server	Updated the MCP Server documentation with detailed installation approaches for Claude Code and Codex, including CLI-based setup and configuration file examples. Added comprehensive guidance for finding MCP client configuration files across different development tools. For more information, see AWS Labs Aurora DSQL MCP Server.	December 19, 2025

Updated content: Migrating from PostgreSQL to Aurora DSQL	Redrafted the PostgreSQL compatibility section as a comprehensive migration guide. Includes framework compatibility information, common migration patterns, architectural differences, and AI-assisted migration guidance. Added new chapter for providing feedback on Aurora DSQL. For more information, see Migrating from PostgreSQL to Aurora DSQL.	December 16, 2025
Updated content: Connecting to Aurora DSQL using AWS PrivateLink	Added documentation for private DNS setup and the cluster ID connection option to support customers using AWS PrivateLink with Direct Connect or Amazon VPC peering. Includes guidance for connecting without private DNS using the `amzn-cluster-id` connection option. For more information, see Managing and connecting to Aurora DSQL clusters using AWS PrivateLink.	December 11, 2025

Updated content: Aurora DSQL cluster lifecycle	Updated documentation for Aurora DSQL cluster lifecycle management. Explains cluster status definitions, state transitions, and operations available during Idle and Inactive states. For more information, see Aurora DSQL cluster lifecycle.	December 4, 2025
New content: Aurora DSQL connectors for Python and Node.js	Added documentation for Aurora DSQL connectors for Python (psycopg, psycopg2, asyncpg) and Node.js (node-postgres, Postgres.js). These connectors integrate IAM authentication for connecting applications to Aurora DSQL clusters. For more information, see Aurora DSQL connectors.	November 21, 2025
New content: Using JupyterLab with Aurora DSQL	Added step-by-step guide for connecting and querying Aurora DSQL using JupyterLab with Python. Includes instructions for both local JupyterLab installations and Amazon SageMaker AI environments. For more information, see Using JupyterLab with Aurora DSQL.	November 20, 2025

Updated content: Quotas for Aurora DSQL	Updated maximum cluster storage quota from 128 TiB to 256 TiB. For more informati on, see Quotas for Aurora DSQL.	November 19, 2025
New content: Getting started with the Aurora DSQL Query Editor	Added documentation for using the Aurora DSQL Query Editor in the AWS Management Console. Includes prerequisites, connection setup, and query execution instructions. For more information, see Getting started with the Aurora DSQL Query Editor.	November 18, 2025

Resource-based policy support for Amazon Aurora DSQL	Added resource-based policy (RBP) support with new permissions: `PutClusterPolicy`, `GetClusterPolicy`, and `DeleteClusterPolicy`. These permissions allow managing inline policies attached to Aurora DSQL clusters for fine-grained access control. Updated managed policies AmazonAuroraDSQLFullAccess, AmazonAuroraDSQLReadOnlyAccess, and AmazonAuroraDSQLConsoleFullAccess to include RBP capabilities. For more information, see AWS managed policies for Amazon Aurora DSQL.	October 15, 2025
Aurora DSQL JDBC Connector	Added documentation for the Aurora DSQL JDBC Connector, a PgJDBC connector that integrates IAM Authentication for connecting Java applications to Amazon Aurora DSQL clusters. For more information, see Connecting to Aurora DSQL clusters with a JDBC connector.	September 2, 2025

AWS managed policy updates for AWS FIS integration	Updated `AmazonAuroraDSQLFullAccess` and `AmazonAuroraDSQLConsoleFullAccess` policies to support AWS Fault Injection Service integration with Aurora DSQL. This allows you to inject failures into single-Region and multi-Region Aurora DSQL clusters to test fault tolerance of your applications. For more on these policies, see AWS managed policy updates.	August 19, 2025
General availability (GA) of Amazon Aurora DSQL	Amazon Aurora DSQL is now generally available with added support for CloudWatch monitoring, enhanced data protection features, and AWS Backup integration. For more information, see Monitoring Aurora DSQL with CloudWatch, Backup and restore for Amazon Aurora DSQL, and Data encryption for Amazon Aurora DSQL.	May 27, 2025

AmazonAuroraDSQLFullAccess update	Adds the capability to perform backup and restore operations for Aurora DSQL clusters, including starting, stopping, and monitoring jobs. It also adds the capability to use customer-managed KMS keys for cluster encryption. For more information, see AmazonAuroraDSQLFullAccess and Using service-linked roles in Aurora DSQL .	May 21, 2025
AmazonAuroraDSQLConsoleFullAccess update	Adds the capability to perform backup and restore operations for Aurora DSQL clusters through the AWS Console Home. This includes starting, stopping, and monitoring jobs. It also supports using customer-managed KMS keys for cluster encryption and launching AWS CloudShell. For more information, see AmazonAuroraDSQLConsoleFullAccess and Using service-linked roles in Aurora DSQL.	May 21, 2025

AmazonAuroraDSQLReadOnlyAccess update	Includes the ability to determine the correct VPC endpoint service name when connecting to your Aurora DSQL clusters through AWS PrivateLink Aurora DSQL creates unique endpoints per cell, so this API helps ensure you can identify the correct endpoint for your cluster and avoid connection errors. For more information, see AmazonAuroraDSQLReadOnlyAccess and Using service-linked roles in Aurora DSQL .	May 13, 2025

AmazonAuroraDSQLFullAccess update	The policy adds four new permissions to create and manage database clusters across multiple AWS Regions: `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , and `RemovePeerCluster` . These permissions include resource-level controls and condition keys so you can control which clusters users you can modify. The policy also adds the `GetVpcEndpointServiceName` permission to help you connect to your Aurora DSQL clusters through AWS PrivateLink. For more information, see AmazonAuroraDSQLConsoleFullAccess and Using service-linked roles in Aurora DSQL.	May 13, 2025

AmazonAuroraDSQLConsoleFullAccess update	Adds new permissions to Aurora DSQL to support multi-Region cluster management and VPC endpoint connection. The new permissions include: `PutMultiRegionProperties` `PutWitnessRegion` `AddPeerCluster` `RemovePeerCluster` `GetVpcEndpointServiceName` . See AmazonAuroraDSQLConsoleFullAccess and Using service-linked roles in Aurora DSQL.	May 13, 2025
AuroraDsqlServiceLinkedRolePolicy update	Adds the ability to publish metrics to the `AWS/AuroraDSQL` and `AWS/Usage` `CloudWatch` namespaces to the policy. This allows the associated service or role to emit more comprehensive usage and performance data to your CloudWatch environment. For more information, see AuroraDsqlServiceLinkedRolePolicy and Using service-linked roles in Aurora DSQL.	May 8, 2025

AWS PrivateLink for Amazon Aurora DSQL	Aurora DSQL now supports AWS PrivateLink. With AWS PrivateLink, you can simplify private network connectivity between virtual private clouds (VPCs), Aurora DSQL, and your on-premises data centers using interface Amazon VPC endpoints and private IP addresses. For more information, see Managing and connecting to Amazon Aurora DSQL clusters using AWS PrivateLink.	May 8, 2025
Initial release	Initial release of the Amazon Aurora DSQL User Guide.	December 3, 2024