ユーザーガイド

Amazon Aurora DSQL

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Aurora DSQL: ユーザーガイド

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスはAmazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

Amazon Aurora DSQL とは 1
どのようなときに使うか 1
主な特徴 1
AWS リージョン の可用性 3
料金 4
次のステップ 4
入門 5
前提条件 5
Aurora DSQL へのアクセス 6
コンソールアクセス 6
SQL クライアント 6
PostgreSQL プロトコル 10
単一リージョンクラスターを作成する 11
クラスターに接続する 12
SQL コマンドを実行する 13
マルチリージョンクラスターを作成する 15
認証と認可 19
クラスターの管理 19
クラスターへの接続 19
PostgreSQL ロールと IAM ロール 20
Aurora DSQL での IAM ポリシーアクションの使用 21
IAM ポリシーアクションを使用したクラスターへの接続 21
IAM ポリシーアクションを使用したクラスターの管理 22
IAM と PostgreSQL を使用した認可の取り消し 23
認証トークンを生成する 24
コンソール 24
AWS CloudShell 25
AWS CLI 26
Aurora DSQL SDK 27
データベースロールと IAM 認証 36
IAM ロール 36
IAM ユーザー 36
接続 36
クエリ 37

取り消し ... 37
Aurora DSQL および PostgreSQL ... 38
互換性のハイライト ... 38
アーキテクチャ上の主な相違点 ... 39
SQL 互換性 ... 40
サポートされているデータ型 ... 40
サポートされている SQL の機能 ... 45
サポートされている SQL コマンドのサブセット ... 48
サポートされていない PostgreSQL 機能 ... 59
同時実行制御 ... 63
トランザクションの競合 ... 64
トランザクションパフォーマンスを最適化するためのガイドライン ... 64
DDL および分散トランザクション ... 64
プライマリキー ... 66
データ構造とストレージ ... 66
プライマリキーを選択するためのガイドライン ... 66
非同期インデックス ... 67
構文 ... 68
パラメータ ... 68
使用に関する注意事項 ... 69
インデックスの作成 ... 70
インデックスのクエリ ... 70
一意のインデックスビルドの失敗 ... 72
一意性違反 ... 72
システムテーブルとコマンド ... 74
システムテーブル ... 75
ANALYZE コマンド ... 84
Aurora DSQL クラスターの管理 ... 85
単一リージョンクラスター ... 85
クラスターの作成 ... 85
クラスターの説明 ... 86
クラスターの更新 ... 87
クラスターの削除 ... 87
クラスターの一覧表示 ... 88
マルチリージョンクラスター ... 88
マルチリージョンクラスターへの接続 ... 89

マルチリージョンクラスターの作成 89
マルチリージョンクラスターの削除 93
AWS CloudFormation 95
Aurora DSQL を使用したプログラミング 97
..... 97
AWS SDKとサンプルコード 98
AWS CLI 104
CreateCluster 104
GetCluster 105
UpdateCluster 105
DeleteCluster 106
ListClusters 106
マルチリージョンクラスターの GetCluster 107
クラスターの作成、読み取り、更新、削除 108
クラスターを作成する 108
クラスターを取得する 140
クラスターを更新する 149
クラスターを削除 158
チュートリアル 182
AWS Lambda チュートリアル 182
バックアップと復元 188
AWS Backup の開始方法 188
バックアップの復元 188
モニタリングとコンプライアンス 189
追加リソース 190
モニタリングとログ記録 191
クラスターステータスの表示 191
クラスターのステータス 191
クラスターステータスの表示 192
CloudWatch によるモニタリング 193
オブザーバビリティ 194
使用方法 195
CloudTrail によるログ記録 196
管理イベント 197
データイベント 199
セキュリティ 201

AWS マネージドポリシー 202
AmazonAuroraDSQLFullAccess 202
AmazonAuroraDSQLReadOnlyAccess 203
AmazonAuroraDSQLConsoleFullAccess 204
AuroraDSQLServiceRolePolicy 205
ポリシーの更新 205
データ保護 210
データ暗号化 211
SSL/TLS 証明書 212
データ暗号化 211
KMS キーの種類 219
保管中の暗号化 220
KMS およびデータキーの使用 221
KMS キーを許可する 223
暗号化コンテキスト 225
AWS KMS のモニタリング 226
暗号化されたクラスターの作成 229
キーの削除または更新 231
考慮事項 233
Identity and Access Management 234
対象者 234
アイデンティティによる認証 235
ポリシーを使用したアクセス権の管理 239
Aurora DSQL と IAM の連携 241
アイデンティティベースのポリシーの例 248
トラブルシューティング 251
サービスリンクロールの使用 253
Aurora DSQL のサービスにリンクされたロールのアクセス許可 254
サービスにリンクされたロールの作成 254
サービスにリンクされたロールの編集 255
サービスにリンクされたロールを削除 255
Aurora DSQL のサービスにリンクされたロールをサポートするリージョン 255
IAM 条件キーの使用 255
特定のリージョンにクラスターを作成します。 255
特定のリージョンにマルチリージョンクラスターを作成する 256
特定の監視リージョンを持つマルチリージョンクラスターを作成する 257

インシデントへの対応 258
コンプライアンス検証 258
レジリエンス 260
バックアップと復元 260
レプリケーション 260
高可用性 261
インフラストラクチャセキュリティ 261
AWS PrivateLink を使用したクラスターの管理 262
設定と脆弱性の分析 271
サービス間の混乱した代理の防止 272
セキュリティに関するベストプラクティス 273
セキュリティ問題の検出ベストプラクティス 273
予防的セキュリティのベストプラクティス 274
リソースのタグ付け 276
[Name tag] (名前タグ) 276
タグ付け要件 276
タグ付けの使用に関する注意事項 277
考慮事項 278
クォータと制限 279
クラスタークォータ 279
データベース制限 280
API リファレンス 284
トラブルシューティング 285
接続エラー 285
認証エラー 286
認可エラー 286
SQL エラー 287
OCC エラー 288
SSL/TLS 接続 288
ドキュメント履歴 290

Amazon Aurora DSQL とは

Amazon Aurora DSQL は、トランザクションワークロード用に最適化されたサーバーレスの分散リレーショナルデータベースサービスです。Aurora DSQL は実質的に無制限のスケールを提供し、インフラストラクチャを管理する必要はありません。アクティブ/アクティブ高可用性アーキテクチャは、99.99% の単一リージョンと 99.999% のマルチリージョンの可用性を提供します。

どのようなときに Aurora DSQL を使うか

Aurora DSQL は、ACID トランザクションとリレーショナルデータモデルの恩恵を受けるトランザクションワークロード向けに最適化されています。Aurora DSQL はサーバーレスであるため、マイクロサービス、サーバーレス、イベント駆動型アーキテクチャのアプリケーションパターンに最適です。Aurora DSQL は PostgreSQL と互換性があるため、使い慣れたドライバー、オブジェクトリレーショナルマッピング (ORM)、フレームワーク、SQL 機能を使用できます。

Aurora DSQL は、システムインフラストラクチャを自動的に管理し、ワークロードに基づいてコンピューティング、I/O、ストレージをスケーリングします。プロビジョニングや管理するサーバーがないため、プロビジョニング、パッチ適用、インフラストラクチャのアップグレードに関連するメンテナンスのダウンタイムについて心配する必要はありません。

Aurora DSQL は、あらゆる規模で常に利用可能なエンタープライズアプリケーションを構築および維持するのに役立ちます。アクティブ/アクティブサーバーレス設計は障害復旧を自動化するため、従来のデータベースフェイルオーバーについて心配する必要はありません。アプリケーションにはマルチ AZ とマルチリージョンの可用性のメリットがあり、フェイルオーバーに関連する最終的な一貫性やデータの欠落について心配する必要はありません。

Aurora DSQL の主な特徴

以下の主要な機能は、高可用性アプリケーションをサポートするサーバーレス分散データベースを作成するのに役立ちます。

分散アーキテクチャ

Aurora DSQL は、次のマルチテナントコンポーネントで構成されています。

- リレーと接続
- コンピューティングとデータベース

- トランザクションログ、同時実行制御、分離
- [Storage (ストレージ)]

コントロールプレーンは、前述のコンポーネントを調整します。各コンポーネントは、3 つのアベイラビリティーゾーン (AZ) にまたがる冗長性を提供し、コンポーネントに障害が発生した場合にクラスターの自動スケーリングと自己修復を行います。このアーキテクチャが高可用性をサポートする方法の詳細については、「Amazon Aurora DSQL の耐障害性」を参照してください。

単一リージョンクラスターとマルチリージョンクラスター

Aurora DSQL クラスターには次の利点があります。

- 同期データレプリケーション
- 一貫した読み取りオペレーション
- 障害からの自動復旧
- 複数の AZ またはリージョン間でのデータ整合性

インフラストラクチャコンポーネントに障害が発生した場合、Aurora DSQL は手動で介入することなく、正常なインフラストラクチャにリクエストを自動的にルーティングします。Aurora DSQL は、強力な整合性、スナップショット分離、原子性、クロス AZ およびクロスリージョン耐久性を備えた原子性、整合性、分離、耐久性 (ACID) トランザクションを提供します。

マルチリージョンピア接続クラスターは、単一リージョンクラスターと同じ耐障害性と接続性を提供します。ただし、ピア接続された各クラスターリージョンに 1 つずつ、2 つのリージョンエンドポイントを提供することで可用性が向上します。ピア接続されたクラスターの両方のエンドポイントには、単一の論理データベースがあります。これらは同時読み取りおよび書き込みオペレーションに利用でき、強力なデータ整合性を提供します。パフォーマンスとレジリエンスのために、複数のリージョンで同時に実行されるアプリケーションを構築できます。また、リーダーは常に同じデータを参照できます。

PostgreSQL データベースとの互換性

Aurora DSQL の分散データベースレイヤー (コンピューティング) は、PostgreSQL の現在のメジャーバージョンに基づいています。`psql` などの使い慣れた PostgreSQL ドライバーとツールを使用して Aurora DSQL に接続できます。Aurora DSQL は現在 PostgreSQL バージョン 16 と互換性があり、PostgreSQL の機能、式、およびデータ型のサブセットをサポートしています。サポートされる SQL 機能の詳細については、「Aurora DSQL での SQL 機能の互換性」を参照してください。

Aurora DSQL の利用可能なリージョン

Amazon Aurora DSQL を使用すると、データベースインスタンスを複数の AWS リージョンにデプロイして、グローバルアプリケーションをサポートし、データレジデンシー要件を満たすことができます。リージョンの可用性によって、Aurora DSQL データベースクラスターを作成および管理できる場所が決まります。可用性が高く、グローバルに分散されたデータベースシステムを設計する必要があるデータベース管理者やアプリケーションアーキテクトは、多くの場合、ワークロードのリージョンサポートを理解する必要があります。一般的なユースケースには、クロスリージョンディザスタリカバリの設定、レイテンシーを低減するための地理的に近いデータベースインスタンスからのユーザーへの提供、コンプライアンスのために特定のロケーションにデータコピーを維持することなどがあります。

次の表は、現時点で Aurora DSQL を利用できる AWS リージョンと AWS リージョン別のエンドポイントの一覧です。

サポート対象の AWS リージョン および エンドポイント

リージョン名	リージョン	エンドポイント	プロトコル
米国東部 (バージニア北部)	us-east-1	dsql.us-east-1.api.aws	HTTPS
米国東部 (オハイオ)	us-east-2	dsql.us-east-2.api.aws	HTTPS
米国西部 (オレゴン)	us-west-2	dsql.us-west-2.api.aws	HTTPS
欧州 (ロンドン)	eu-west-2	dsql.eu-west-2.api.aws	HTTPS
欧州 (アイルランド)	eu-west-1	dsql.eu-west-1.api.aws	HTTPS
欧州 (パリ)	eu-west-3	dsql.eu-west-3.api.aws	HTTPS
アジアパシフィック (大阪)	ap-northeast-3	dsql.ap-northeast-3.api.aws	HTTPS
アジアパシフィック (東京)	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS

> **Note**
>
> Aurora DSQL マルチリージョンクラスターは、現在次の 3 つの AWS リージョンでサポートされています。
>
> - 米国東部 (バージニア北部)
> - 米国東部 (オハイオ)
> - 米国西部 (オレゴン)

Aurora DSQL の料金

コスト情報については、「Aurora DSQL の料金」を参照してください。

次のステップ

Aurora DSQL のコアコンポーネントの詳細とサービスの開始方法については、以下を参照してください。

- Aurora DSQL の開始方法
- Aurora DSQL での SQL 機能の互換性
- Aurora DSQL へのアクセス
- Aurora DSQL および PostgreSQL

Aurora DSQL の開始方法

Amazon Aurora DSQL は、トランザクションワークロード用に最適化されたサーバーレスの分散リレーショナルデータベースです。以下のセクションでは、単一リージョンおよびマルチリージョンの Aurora DSQL クラスターを作成し、それらに接続して、サンプルスキーマを作成してロードする方法について説明します。AWS Management Consoleを使用してクラスターにアクセスし、`psql` ユーティリティを使用してデータベースを操作します。最終的に、動作中の Aurora DSQL クラスターがセットアップされ、テストまたは本番ワークロードで使用できるようになります。

トピック

- 前提条件
- Aurora DSQL へのアクセス
- ステップ 1: Aurora DSQL 単一リージョンクラスターを作成する
- ステップ 2: Aurora DSQL クラスターに接続する
- ステップ 3: Aurora DSQL でサンプル SQL コマンドを実行する
- ステップ 4: マルチリージョンクラスターを作成する

前提条件

Aurora DSQL の使用を開始する前に、以下の前提条件を満たしていることを確認してください。

- IAM ID には、AWS Management Consoleにサインインするためのアクセス許可が必要です。
- IAM ID は、以下のいずれかの条件を満たしている必要があります。
  - AWS アカウント内の任意のリソースに対して任意のアクションを実行できるアクセス許可
  - IAM アクセス許可 `iam:CreateServiceLinkedRole` と IAM ポリシーアクション `dsql:*` へのアクセス許可
- Unix 系環境で AWS CLI を使用する場合は、Python バージョン 3.8 以降と `psql` バージョン 14 以降がインストールされていることを確認してください。アプリケーションのバージョンを確認するには、以下のコマンドを実行します。

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

AWS CLI を別の環境で使用する場合は、Python バージョン 3.8 以降と `psql` バージョン 14 以降を手動で設定してください。

- AWS CloudShell を使用して Aurora DSQL にアクセスする場合、Python バージョン 3.8 以降と `psql` バージョン 14 以降が追加の設定なしで提供されます。AWS CloudShell の詳細については、「AWS CloudShell とは」をご参照ください。
- GUI を使用して Aurora DSQL にアクセスする場合は、DBeaver または JetBrains DataGrip を使用します。詳細については、「DBeaver を使用した Aurora DSQL へのアクセス」および「JetBrains DataGrip を使用した Aurora DSQL へのアクセス」を参照してください。

Aurora DSQL へのアクセス

Aurora DSQL には、以下の方法でアクセスできます。CLI、API、SDK の使用方法については、「Aurora DSQL へのアクセス」を参照してください。

トピック

- AWS Management Console 経由での Aurora DSQL へのアクセス
- SQL クライアントを使用した Aurora DSQL へのアクセス
- Aurora DSQL での PostgreSQL プロトコルの使用

AWS Management Console 経由での Aurora DSQL へのアクセス

Aurora DSQL のAWS Management Consoleには、https://console.aws.amazon.com/dsql からアクセスできます。

SQL クライアントを使用した Aurora DSQL へのアクセス

Aurora DSQL は PostgreSQL プロトコルを使用します。クラスターに接続するときに、署名付き IAM 認証トークンをパスワードとして指定して、任意のインタラクティブクライアントを使用できます。認証トークンは、Aurora DSQL が AWS 署名バージョン 4 を使用して動的に生成する一意の文字列です。

Aurora DSQL はトークンを認証にのみ使用します。トークンは、接続が確立された後は接続に影響を与えません。期限切れのトークンを使用して再接続しようとすると、接続リクエストは拒否されます。詳細については、「Amazon Aurora DSQL での認証トークンの生成」を参照してください。

トピック

- psql (PostgreSQL インタラクティブターミナル) を使用した Aurora DSQL へのアクセス
- DBeaver を使用した Aurora DSQL へのアクセス
- JetBrains DataGrip を使用した Aurora DSQL へのアクセス

psql (PostgreSQL インタラクティブターミナル) を使用した Aurora DSQL へのアクセス

psql ユーティリティは、PostgreSQL へのターミナルベースのフロントエンドです。これにより、クエリをインタラクティブに入力し、PostgreSQL に発行して、クエリ結果を表示できます。クエリの応答時間を改善するには、PostgreSQL バージョン 17 クライアントを使用します。

PostgreSQL のダウンロードページからオペレーティングシステムのインストーラーをダウンロードします。psql の詳細については、「https://www.postgresql.org/docs/current/app-psql.htm」を参照してください。

AWS CLI が既にインストールされている場合は、次の例を使用してクラスターに接続します。psql がプリインストールされている AWS CloudShell を使用することも、psql を直接インストールすることもできます。

```
# 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
```

DBeaver を使用した Aurora DSQL へのアクセス

DBeaver は、オープンソースの GUI ベースのデータベースツールです。これを使用してデータベースに接続し、管理することができます。DBeaver をダウンロードするには、DBeaver Community ウェブサイトの「download page」を参照してください。次の手順では、DBeaver を使用してクラスターに接続する方法について説明します。

DBeaver で新しい Aurora DSQL 接続を設定するには

1. [新しいデータベース接続] を選択します。
2. [新しいデータベース接続] ウィンドウで、[PostgreSQL] を選択します。
3. [接続の設定/メイン] タブで、[接続方法: ホスト] を選択し、次の情報を入力します。
   - ホスト - クラスターエンドポイントを使用します。

     データベース – `postgres` と入力します。

     認証 - `Database Native` を選択します。

     ユーザー名 - `admin` と入力します。

     パスワード - 認証トークンを生成します。生成されたトークンをコピーし、パスワードとして使用します。
4. 警告が表示された場合は無視し、認証トークンを [DBeaver Password] フィールドに貼り付けます。

   > **Note**
   > クライアント接続で SSL モードを設定する必要があります。Aurora DSQL は `SSLMODE=require` をサポートしています。Aurora DSQL はサーバー側で SSL 通信を強制し、SSL 以外の接続を拒否します。

5. クラスターに接続し、SQL ステートメントの実行を開始できます。

> **Important**
> DBeaver が PostgreSQL データベース向けに提供する管理機能 (Session Manager や Lock Manager など) は、独自のアーキテクチャのため、データベースには適用されません。これ

らの画面はアクセス可能ですが、データベースのヘルスやステータスに関する信頼できる情報を提供しません。

DBeaver の認証情報の有効期限

確立されたセッションは、最大 1 時間、または DBeaver によって切断されるかタイムアウトが発生するまで認証されたままになります。新しい接続を確立する場合は、[接続設定] の [パスワード] フィールドに有効な認証トークンを指定する必要があります。新しいセッション (新しいテーブルの一覧表示や新しい SQL コンソールなど) を開こうとすると、新しい認証が強制的に試行されます。[接続設定] で設定された認証トークンが無効になった場合、その新しいセッションは失敗し、DBeaver は以前に開かれたすべてのセッションを無効にします。`expires-in` オプションを使用して IAM 認証トークンの有効期間を選択するときは、この点に注意してください。

JetBrains DataGrip を使用した Aurora DSQL へのアクセス

JetBrains DataGrip は、PostgreSQL を含む SQL とデータベースを操作するためのクロスプラットフォーム IDE です。DataGrip には、インテリジェントな SQL エディタを備えた堅牢な GUI が搭載されています。DataGrip をダウンロードするには、JetBrains ウェブサイトのダウンロードページにアクセスしてください。

JetBrains DataGrip で新しい Aurora DSQL 接続を設定するには

1. [新しいデータソース] を選択し、[PostgreSQL] を選択します。
2. [データソース/全般] タブに、次の情報を入力します。

   - ホスト - クラスターエンドポイントを使用します。

     ポート - Aurora DSQL は PostgreSQL のデフォルトを使用します: `5432`

     データベース - Aurora DSQL は PostgreSQL のデフォルトの `postgres` を使用します。

     認証 - `User & Password` を選択します。

     ユーザー名 - `admin` と入力します。

     パスワード - トークンを生成し、このフィールドに貼り付けます。

     URL - このフィールドは変更しないでください。他のフィールドに基づいて自動的に入力されます。

3. パスワード - 認証トークンを生成してこれを指定します。トークンジェネレーターの結果出力をコピーし、[パスワード] フィールドに貼り付けます。

> **Note**
>
> クライアント接続で SSL モードを設定する必要があります。Aurora DSQL は `PGSSLMODE=require` をサポートしています。Aurora DSQL はサーバー側で SSL 通信を強制し、SSL 以外の接続を拒否します。

4. クラスターに接続し、SQL ステートメントの実行を開始できます。

> **Important**
>
> DataGrip が PostgreSQL データベース向けに提供する一部のビュー (Sessions など) は、独自のアーキテクチャのため、データベースには適用されません。これらはアクセス可能ですが、データベースに接続されている実際のセッションに関する信頼できる情報は提供されません。

認証情報の有効期限

確立されたセッションは、最大 1 時間、または明示的な切断またはクライアント側のタイムアウトが発生するまで認証されたままになります。新しい接続を確立する必要がある場合は、新しい認証トークンを生成し、[データソースプロパティ] の [パスワード] フィールドに指定する必要があります。新しいセッション (新しいテーブルの一覧表示や新しい SQL コンソールなど) を開こうとすると、新しい認証が強制的に試行されます。[接続] 設定で設定された認証トークンが無効になった場合、その新しいセッションは失敗し、以前に開かれたすべてのセッションは無効になります。

Aurora DSQL での PostgreSQL プロトコルの使用

PostgreSQL は、クライアントとサーバー間の通信にメッセージベースのプロトコルを使用します。プロトコルは、TCP/IP および Unix ドメインソケットの両方でサポートされています。次の表は、Aurora DSQL が PostgreSQL プロトコルをどのようにサポートしているかを示しています。

PostgreSQL	Aurora DSQL	メモ
ロール (ユーザーまたはグループとも呼ばれます)	データベースロール	Aurora DSQL は、`admin` という名前のロールを作成します。カスタムデータベースロー

PostgreSQL	Aurora DSQL	メモ
		ルを作成する場合は、`admin` ロールを使用して、それをクラスターへの接続時の認証に使用する IAM ロールに関連付ける必要があります。詳細については、「カスタムデータベースロールを設定する」を参照してください。
ホスト (hostname または hostspec とも呼ばれます)	クラスターエンドポイント	Aurora DSQL の単一リージョンクラスターは、単一のマネージドエンドポイントを提供し、リージョン内で利用できない場合はトラフィックを自動的にリダイレクトします。
ポート	該当なし - デフォルトの `5432` を使用	これは PostgreSQL のデフォルトです。
データベース (dbname)	`postgres` を使用	Aurora DSQL は、クラスターの作成時にこのデータベースを作成します。
SSL モード	SSL は常にサーバー側で有効	Aurora DSQL では、Aurora DSQL は `require` SSL モードをサポートしています。SSL を使用しない接続は Aurora DSQL によって拒否されます。
パスワード	認証トークン	Aurora DSQL には、存続期間の長いパスワードの代わりに一時的な認証トークンが必要です。詳細については Amazon Aurora DSQL での認証トークンの生成を参照してください。

ステップ 1: Aurora DSQL 単一リージョンクラスターを作成する

Aurora DSQL の基本単位は、データを保存するクラスターです。このタスクでは、単一の AWS リージョンにクラスターを作成します。

Aurora DSQL で単一リージョンクラスターを作成するには

1. AWS Management Consoleにサインインし、https://console.aws.amazon.com/dsql で Aurora DSQL コンソールを開きます。
2. [クラスターを作成] を選択し、[シングルリージョン] を選択します。
3. (オプション) [クラスター設定] で、次のいずれかのオプションを選択します。
   - [暗号化の設定をカスタマイズ (詳細設定)] を選択して、AWS KMS keyを選択または作成します。
   - 削除オペレーションでクラスターが削除されないようにするために、[削除保護を有効にする] を選択します。デフォルトでは、削除保護が選択されています。
4. (オプション) [タグ] で、このクラスターのタグを選択または入力します。
5. [クラスターを作成] を選択します。

ステップ 2: Aurora DSQL クラスターに接続する

クラスターエンドポイントは、クラスター ID とリージョンに基づいて Aurora DSQL クラスターを作成すると自動的に生成されます。名前の形式は `clusterid.dsql.region.on.aws` です。クライアントはエンドポイントを使用して、クラスターへのネットワーク接続を作成します。

認証は IAM を使用して管理されるため、データベースに認証情報を保存する必要はありません。認証トークンは動的に生成される一意の文字列です。トークンは認証にのみ使用され、確立後の接続には影響しません。接続を試みる前に、「前提条件」で説明されているように、IAM ID に `dsql:DbConnectAdmin` アクセス許可があることを確認してください。

> **Note**
> データベース接続速度を最適化するには、PostgreSQL バージョン 17 クライアントを使用し、`PGSSLNEGOTIATION` を direct に設定します: `PGSSLNEGOTIATION=direct`。

認証トークンを使用してクラスターに接続するには

1. Aurora DSQL コンソールで、接続先のクラスターを選択します。
2. [接続] を選択します。
3. [エンドポイント (ホスト)] からエンドポイントをコピーします。

4. [認証トークン (パスワード)] セクションで、[管理者として接続] が選択されていることを確認します。
5. 生成された認証トークンをコピーします。このトークンの有効期間は 15 分です。
6. オペレーティングシステムのコマンドラインで、次のコマンドを使用して `psql` を起動し、クラスターに接続します。*`your_cluster_endpoint`* を、先ほどコピーしたクラスターエンドポイントに置き換えます。

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

パスワードの入力を求められたら、先ほどコピーした認証トークンを入力します。期限切れのトークンを使用して再接続しようとすると、接続リクエストは拒否されます。詳細については、「Amazon Aurora DSQL での認証トークンの生成」を参照してください。

7. [Enter] キーを押します。PostgreSQL プロンプトが表示されます。

```
postgres=>
```

アクセス拒否エラーが発生した場合は、IAM ID に `dsql:DbConnectAdmin` アクセス許可があることを確認してください。アクセス許可があるにもかかわらずアクセス拒否エラーが引き続き発生する場合は、「IAM をトラブルシューティングする」および「IAM ポリシーで、アクセス拒否または操作が許可されていないことを示すエラーをトラブルシューティングする方法を教えてください」を参照してください。

ステップ 3: Aurora DSQL でサンプル SQL コマンドを実行する

SQL ステートメントを実行して Aurora DSQL クラスターをテストします。次のサンプルステートメントには、`department-insert-multirow.sql` および `invoice.csv` という名前のデータファイルが必要です。これらのファイルは、GitHub の aws-samples/aurora-dsql-samples リポジトリからダウンロードできます。

Aurora DSQL でサンプル SQL コマンドを実行するには

1. `example` という名前のスキーマを作成します。

```
CREATE SCHEMA example;
```

2. 自動的に生成された UUID をプライマリキーとして使用する請求書テーブルを作成します。

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

3. 空のテーブルを使用するセカンダリインデックスを作成します。

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

4. 部門テーブルを作成します。

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

5. `psql \include` コマンドを使用して、GitHub の aws-samples/aurora-dsql-samples リポジトリからダウンロードした `department-insert-multirow.sql` という名前のファイルをロードします。*my-path* をローカルコピーへのパスに置き換えます。

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

6. `psql \copy` コマンドを使用して、GitHub の aws-samples/aurora-dsql-samples リポジトリから、ダウンロードした `invoice.csv` という名前のファイルをロードします。*my-path* をローカルコピーへのパスに置き換えます。

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

7. 部門をクエリし、合計売上でソートします。

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

次の出力例は、部門 3 の売上が最も多いことを示しています。

```
           name           |     sum_amount
--------------------------+--------------------
 Example Department Three |  54061.67752854594
 Example Department Seven |  53869.65965365204
 Example Department Eight |  52199.73742066634
 Example Department One   | 52034.078869900826
 Example Department Six   |  50886.15556256385
 Example Department Two   |  50589.98422247931
 Example Department Five  | 49549.852635496005
 Example Department Four  |  49266.15578027619
(8 rows)
```

ステップ 4: マルチリージョンクラスターを作成する

マルチリージョンクラスターを作成する際は、以下のリージョンを指定します。

リモートリージョン

これは、2 番目のクラスターを作成するリージョンです。このリージョンに 2 番目のクラスターを作成し、最初のクラスターにピア接続します。Aurora DSQL は、最初のクラスターに対するすべての書き込みをリモートクラスターにレプリケートします。任意のクラスターで読み取りと書き込みが可能です。

ウィットネスリージョン

このリージョンは、マルチリージョンクラスターに書き込まれるすべてのデータを受け取ります。ウィットネスリージョンはクライアントエンドポイントをホストせず、ユーザーデータアクセスも提供しません。暗号化されたトランザクションログの制限されたウィンドウが、監視リージョンで保持されます。このログにより、復旧が容易になり、リージョンが利用できない場合のトランザクションクォーラムがサポートされます。

次の例は、最初のクラスターを作成し、別のリージョンに 2 番目のクラスターを作成し、2 つのクラスターをピアリングしてマルチリージョンクラスターを作成する方法を示しています。さらに、クロスリージョン書き込みレプリケーションと、両方のリージョンエンドポイントからの一貫した読み取りも示しています。

マルチリージョンクラスターを作成するには

1. AWS Management Consoleにサインインし、https://console.aws.amazon.com/dsql で Aurora DSQL コンソールを開きます。
2. ナビゲーションペインで [Clusters] (クラスター) を選択してください。
3. [クラスターを作成] を選択し、[マルチリージョン] を選択します。
4. (オプション) [クラスター設定] で、最初のクラスターに次のいずれかのオプションを選択します。
   - [暗号化の設定をカスタマイズ (詳細設定)] を選択して、AWS KMS keyを選択または作成します。
   - 削除オペレーションでクラスターが削除されないようにするために、[削除保護を有効にする] を選択します。デフォルトでは、削除保護が選択されています。
5. [マルチリージョン設定] で、最初のクラスターに次のオプションを選択します。
   - [ウィットネスリージョン] で、リージョンを選択します。現在、マルチリージョンクラスターのウィットネスリージョンでは、米国を拠点とするリージョンのみがサポートされています。
   - (オプション) [リモートリージョンクラスター ARN] に、別のリージョンにある既存のクラスターの ARN を入力します。マルチリージョンクラスターの 2 番目のクラスターとして機能するクラスターが存在しない場合は、最初のクラスターを作成した後にセットアップを完了します。
6. (オプション) 最初のクラスターのタグを選択します。
7. 最初のクラスターを作成するには、[クラスターの作成] を選択します。前のステップで ARN を入力しなかった場合、コンソールには [クラスターのセットアップは保留中です] の通知が表示されます。
8. [クラスターのセットアップは保留中です] の通知で、[マルチリージョンクラスターのセットアップを完了する] を選択します。このアクションにより、別のリージョンで 2 番目のクラスターの作成が開始されます。
9. 2 番目のクラスターについて、次のいずれかのオプションを選択します。
   - リモートリージョンクラスター ARN を追加する – クラスターが存在し、マルチリージョンクラスターの 2 番目のクラスターにする場合は、このオプションを選択します。
   - 別のリージョンでクラスターを作成する – 2 番目のクラスターを作成するには、このオプションを選択します。[リモートリージョン] で、この 2 番目のクラスターのリージョンを選択します。

10. [***your-second-region*** でクラスターを作成する] を選択します。ここで、*your-second-region* は 2 番目のクラスターの場所です。コンソールが 2 番目のリージョンで開きます。
11. (オプション) 2 番目のクラスターのクラスター設定を選択します。例えば、AWS KMS keyを選択できます。
12. 2 番目のクラスターを作成するには、[クラスターの作成] を選択します。
13. [***initial-cluster-region*** でピアリングする] を選択します。ここで、*initial-cluster-region* は作成した最初のクラスターをホストするリージョンです。
14. プロンプトが表示されたら、[確認] を選択します。このステップで、マルチリージョンクラスターの作成を完了します。

2 番目のクラスターに接続してするには

1. Aurora DSQL コンソールを開き、2 番目のクラスターのリージョンを選択します。
2. [クラスター] を選択します。
3. マルチリージョンクラスター内の 2 番目のクラスターの行を選択します。
4. [アクション] で、[CloudShell で開く] を選択します。
5. [管理者として接続] を選択します。
6. [CloudShell の起動] を選択します。
7. [Run] (実行) を選択します。
8. 「ステップ 3: Aurora DSQL でサンプル SQL コマンドを実行する」の手順に従って、サンプルスキーマを作成します。

   トランザクションの例

   Example

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

9. `psql copy` および `include` コマンドを使用してサンプルデータをロードします。詳細については、「ステップ 3: Aurora DSQL でサンプル SQL コマンドを実行する」を参照してください。

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

初期クラスターをホストするリージョンから 2 番目のクラスターのデータをクエリするには

1. Aurora DSQL コンソールで、最初のクラスターのリージョンを選択します。
2. [クラスター] を選択します。
3. マルチリージョンクラスター内の 2 番目のクラスターの行を選択します。
4. [アクション] で、[CloudShell で開く] を選択します。
5. [管理者として接続] を選択します。
6. [CloudShell の起動] を選択します。
7. [Run] (実行) を選択します。
8. 2 番目のクラスターに挿入したデータをクエリします。

   Example

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

Aurora DSQL の認証および認可

Aurora DSQL はクラスター認可に IAM ロールとポリシーを使用します。データベース認可のために、IAM ロールを PostgreSQL データベースロールに関連付けます。このアプローチは、IAM の利点と PostgreSQL の権限を組み合わせたものです。Aurora DSQL は、これらの機能を使用して、クラスター、データベース、データに対する包括的な認可とアクセスポリシーを提供します。

IAM を使用したクラスターの管理

クラスターを管理するには、認証と認可に IAM を使用します。

IAM 認証

Aurora DSQL クラスターを管理するときに IAM ID を認証するには、IAM を使用する必要があります。AWS Management Console、AWS CLI、または AWS SDK を使用して認証を提供できます。

IAM 認可

Aurora DSQL クラスターを管理するには、Aurora DSQL の IAM アクションを使用して認可を付与します。例えば、クラスターを作成するには、次のポリシーアクションの例のように、IAM ID に IAM アクション `dsql:CreateCluster` のアクセス許可があることを確認します。

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

詳細については、「IAM ポリシーアクションを使用したクラスターの管理」を参照してください。

IAM を使用したクラスターへの接続

クラスターに接続するには、認証と認可に IAM を使用します。

IAM 認証

クラスターに接続する認可を持つ IAM ID を使用して一時認証トークンを生成します。詳細については[Amazon Aurora DSQL での認証トークンの生成](#)を参照してください。

IAM 認可

クラスターのエンドポイントへの接続を確立するために使用する IAM ID に、次の IAM ポリシーアクションを付与します。

- `dsql:DbConnectAdmin` ロールを使用している場合は、`admin` を使用します。Aurora DSQL は、このロールを自動的に作成および管理します。次のサンプル IAM ポリシーアクションは、`admin` が *`my-cluster`* に接続することを許可します。

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

- カスタムデータベースロールを使用している場合は、`dsql:DbConnect` を使用します。データベースで SQL コマンドを使用して、このロールを作成および管理します。次のサンプル IAM ポリシーアクションは、カスタムデータベースロールが *`my-cluster`* に最大 1 時間接続することを許可します。

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

接続を確立すると、ロールは接続に対して最大 1 時間承認されます。

PostgreSQL データベースロールと IAM ロールを使用したデータベースの操作

PostgreSQL は、ロールの概念を使用してデータベースアクセス許可を管理します。ロールは、ロールの設定方法に応じて、データベースユーザーまたはデータベースユーザーのグループと考えることができます。SQL コマンドを使用して PostgreSQL ロールを作成します。データベースレベルの認可を管理するには、PostgreSQL データベースロールに PostgreSQL アクセス許可を付与します。

Aurora DSQL は、admin ロールとカスタムロールの 2 種類のデータベースロールをサポートしています。Aurora DSQL は、Aurora DSQL クラスターで事前定義された admin ロールを自動的に作成します。admin ロールは変更できません。admin としてデータベースに接続すると、SQL を発行して、IAM ロールに関連付ける新しいデータベースレベルのロールを作成できます。IAM ロールがデータベースに接続できるようにするには、カスタムデータベースロールを IAM ロールに関連付けます。

認証

admin ロールを使用してクラスターに接続します。データベースを接続したら、次の例のように、AWS IAM GRANT コマンドを使用して、カスタムデータベースロールをクラスターへの接続が承認された IAM ID に関連付けます。

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

詳細については[クラスターへの接続をデータベースロールに許可する]を参照してください。

Authorization

admin ロールを使用してクラスターに接続します。SQL コマンドを実行してカスタムデータベースロールを設定し、アクセス許可を付与します。詳細については、「PostgreSQL ドキュメント」の「PostgreSQL データベースロール」および「PostgreSQL の権限」を参照してください。

Aurora DSQL での IAM ポリシーアクションの使用

使用する IAM ポリシーアクションは、クラスターへの接続に使用するロールが、admin またはカスタムデータベースロールのいずれかによって異なります。このポリシーは、このロールに必要な IAM アクションにも依存します。

IAM ポリシーアクションを使用したクラスターへの接続

デフォルトのデータベースロール admin を使用してクラスターに接続する場合は、認可付きの IAM ID を使用して、次の IAM ポリシーアクションを実行します。

```
"dsql:DbConnectAdmin"
```

カスタムデータベースロールを使用してクラスターに接続するときは、まず IAM ロールをデータベースロールに関連付けます。クラスターへの接続に使用する IAM ID には、次の IAM ポリシーアクションを実行する権限が必要です。

```
"dsql:DbConnect"
```

カスタムデータベースロールの詳細については、「データベースロールと IAM 認証の使用」を参照してください。

IAM ポリシーアクションを使用したクラスターの管理

Aurora DSQL クラスターを管理するときは、ロールが実行する必要があるアクションに対してのみポリシーアクションを指定します。例えば、ロールがクラスター情報のみを取得する必要がある場合は、次のサンプルポリシーのように、ロールのアクセス許可を `GetCluster` および `ListClusters` アクセス許可のみに制限できます。

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

次のポリシー例は、クラスターを管理するために使用可能なすべての IAM ポリシーアクションを示しています。

```
{
"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" : "*"
    }
  ]
}
```

IAM と PostgreSQL を使用した認可の取り消し

データベースレベルのロールにアクセスするための IAM ロールのアクセス許可を取り消すことができます。

クラスターへの接続に関する管理者権限の取り消し

admin ロールを使用してクラスターに接続する認可を取り消すには、IAM アイデンティティの dsql:DbConnectAdmin へのアクセスを取り消します。IAM ポリシーを編集するか、アイデンティティからポリシーをデタッチします。

IAM ID から接続認可を取り消すと、Aurora DSQL はその IAM ID からのすべての新しい接続試行を拒否します。IAM ID を使用するアクティブな接続は、接続の期間中は認可されたままになる場合があります。接続の期間の詳細については、「クォータと制限」を参照してください。

クラスターへの接続に関するカスタムロール認可の取り消し

admin 以外のデータベースロールへのアクセスを取り消すには、IAM アイデンティティの dsql:DbConnect へのアクセスを取り消します。IAM ポリシーを編集するか、アイデンティティからポリシーをデタッチします。

データベースの AWS IAM REVOKE コマンドを使用して、データベースロールと IAM の関連付けを削除することもできます。データベースロールからのアクセスの取り消しの詳細については、「IAM ロールからのデータベース認可の取り消し」を参照してください。

事前定義された admin データベースロールのアクセス許可を管理することはできません。カスタムデータベースロールのアクセス許可を管理する方法については、「PostgreSQL の権限」を参照してください。権限の変更は、Aurora DSQL が変更トランザクションを正常にコミットした後、次のトランザクションで有効になります。

Amazon Aurora DSQL での認証トークンの生成

SQL クライアントを使用して Amazon Aurora DSQL に接続するには、パスワードとして使用する認証トークンを生成します。このトークンは、接続の認証にのみ使用されます。接続が確立されると、認証トークンの有効期限が切れても接続は有効です。

AWS コンソールを使用して認証トークンを作成すると、トークンはデフォルトで 1 時間で自動的に期限切れになります。AWS CLI または SDK を使用してトークンを作成する場合、デフォルトは 15 分です。最大継続時間は 604,800 秒 (1 週間) です。クライアントから Aurora DSQL に再度接続するには、有効期限が切れていない場合は同じ認証トークンを使用するか、新しいトークンを生成できます。

トークンの生成を開始するには、IAM ポリシー作成し、Aurora DSQL でクラスターを作成します。次に、AWS コンソール、AWS CLI、または AWS SDK を使用してトークンを生成します。

少なくとも、接続に使用するデータベースロールに応じて、IAM を使用したクラスターへの接続 にリストされている IAM アクセス許可が必要です。

トピック

- AWS コンソールを使用して Aurora DSQL で認証トークンを生成する
- AWS CloudShell を使用して Aurora DSQL で認証トークンを生成する
- AWS CLI を使用して Aurora DSQL で認証トークンを生成する
- SDK を使用して Aurora DSQL でトークンを生成する

AWS コンソールを使用して Aurora DSQL で認証トークンを生成する

Aurora DSQL は、パスワードではなくトークンを使用してユーザーを認証します。コンソールからトークンを生成できます。

認証トークンを生成するには

1. AWS Management Consoleにサインインし、https://console.aws.amazon.com/dsql で Aurora DSQL コンソールを開きます。
2. 認証トークンを生成するクラスターのクラスター ID を選択します。クラスターをまだ作成していない場合は、「ステップ 1: Aurora DSQL 単一リージョンクラスターを作成する」または「ステップ 4: マルチリージョンクラスターを作成する」の手順に従います。
3. [接続] を選択し、[トークンを取得] を選択します。

4. admin として接続するか、カスタムデータベースロールを使用して接続するかを選択します。
5. 生成された認証トークンをコピーし、「SQL クライアントを使用した Aurora DSQL へのアクセス」で使用します。

Aurora DSQL のカスタムデータベースロールと IAM の詳細については、「認証と認可」を参照してください。

AWS CloudShell を使用して Aurora DSQL で認証トークンを生成する

AWS CloudShell を使用して認証トークンを生成する前に、以下を行うことを確認してください。

- Aurora DSQL クラスターを作成
- 組織外の AWS アカウントからオブジェクトを取得するための Amazon S3 オペレーション `get-object` を実行するアクセス許可を追加 詳細については、「Amazon S3 ユーザーガイド」を参照してください。

AWS CloudShell を使用して認証トークンを生成するには

1. AWS Management Consoleにサインインし、https://console.aws.amazon.com/dsql で Aurora DSQL コンソールを開きます。
2. AWS コンソールの左下で、AWS CloudShell を選択します。
3. AWS CLI をインストールするには、「AWS CLI の最新バージョンのインストールまたは更新」の手順に従います。

```
sudo ./aws/install --update
```

4. 次のコマンドを実行して、admin ロールの認証トークンを生成します。*us-east-1* をリージョンに置き換え、*your_cluster_endpoint* を独自のクラスターのエンドポイントに置き換えます。

> **Note**
>
> admin として接続しない場合は、代わりに `generate-db-connect-auth-token` を使用します。

```
aws dsql generate-db-connect-admin-auth-token \
```

```
  --expires-in 3600 \
  --region us-east-1 \
  --hostname your_cluster_endpoint
```

問題が発生した場合は、「IAM をトラブルシューティングする」および「IAM ポリシーで、アクセス拒否または操作が許可されていないことを示すエラーをトラブルシューティングする方法を教えてください」を参照してください。

5. 次のコマンドを使用し、psql を使用してクラスターへの接続を開始します。

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

6. パスワードを指定するプロンプトが表示されます。生成したトークンをコピーし、追加のスペースや文字が含まれていないことを確認します。psql から次のプロンプトに貼り付けます。

```
Password for user admin:
```

7. [Enter] キーを押します。PostgreSQL プロンプトが表示されます。

```
postgres=>
```

アクセス拒否エラーが発生した場合は、IAM ID に dsql:DbConnectAdmin アクセス許可があることを確認してください。アクセス許可があるにもかかわらずアクセス拒否エラーが引き続き発生する場合は、「IAM をトラブルシューティングする」および「IAM ポリシーで、アクセス拒否または操作が許可されていないことを示すエラーをトラブルシューティングする方法を教えてください」を参照してください。

Aurora DSQL のカスタムデータベースロールと IAM の詳細については、「認証と認可」を参照してください。

AWS CLI を使用して Aurora DSQL で認証トークンを生成する

クラスターが ACTIVE の場合、aws dsql コマンドを使用して CLI で認証トークンを生成できます。以下のいずれかの対策を使用します。

- admin ロールに接続する場合は、generate-db-connect-admin-auth-token オプションを使用します。

- カスタムデータベースロールに接続する場合は、`generate-db-connect-auth-token` オプションを使用します。

次の例では、次の属性を使用して `admin` ロールの認証トークンを生成します。

- *`your_cluster_endpoint`* – クラスターのエンドポイント。これは、例 `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws` のように、*`your_cluster_identifier`*`.dsql.`*`region`*`.on.aws` 形式に従います。
- *`region`* – `us-east-2` や `us-east-1` などの AWS リージョン。

次の例では、トークンの有効期限を 3,600 秒 (1 時間) で設定します。

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
```

SDK を使用して Aurora DSQL でトークンを生成する

クラスターが `ACTIVE` ステータスの場合、クラスターの認証トークンを生成できます。SDK の例では、次の属性を使用して、`admin` ロールの認証トークンを生成します。

- *`your_cluster_endpoint`* (または *`yourClusterEndpoint`*) – Aurora DSQL クラスターのエンドポイント。命名形式は、例 `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws` のように *`your_cluster_identifier`*`.dsql.`*`region`*`.on.aws` です。
- *`region`* (または *`RegionEndpoint`*) – `us-east-2` や `us-east-1` など、クラスターが配置されている AWS リージョン。

Python SDK

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、generate_db_connect_admin_auth_token を使用します。
- カスタムデータベースロールで接続する場合は、generate_connect_auth_token を使用します。

```
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

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、GenerateDBConnectAdminAuthToken を使用します。
- カスタムデータベースロールで接続する場合は、GenerateDBConnectAuthToken を使用します。

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

using namespace Aws;
using namespace Aws::DSQL;

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

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

    std::cout << token << std::endl;

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

JavaScript SDK

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、getDbConnectAdminAuthToken を使用します。
- カスタムデータベースロールで接続する場合は、getDbConnectAuthToken を使用します。

```
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

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、generateDbConnectAdminAuthToken を使用します。
- カスタムデータベースロールで接続する場合は、generateDbConnectAuthToken を使用します。

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

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

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

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

Rust SDK

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、db_connect_admin_auth_token を使用します。
- カスタムデータベースロールで接続する場合は、db_connect_auth_token を使用します。

```
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

次の方法でトークンを生成できます。

- admin ロールに接続する場合は、generate_db_connect_admin_auth_token を使用します。
- カスタムデータベースロールで接続する場合は、generate_db_connect_auth_token を使用します。

```
require 'aws-sdk-dsql'

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

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

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

```
          :region => region
      })
  rescue => error
    puts error.full_message
  end
end
```

.NET

> **Note**
>
> 公式の SDK for .NET には、Aurora DSQL の認証トークンを生成するための組み込み API コールは含まれていません。代わりに、ユーティリティクラスの `DSQLAuthTokenGenerator` を使用する必要があります。次のコードサンプルは、.NET の認証トークンを生成する方法を示しています。

次の方法でトークンを生成できます。

- `admin` ロールに接続する場合は、`DbConnectAdmin` を使用します。
- カスタムデータベースロールで接続する場合は、`DbConnect` を使用します。

次の例では、`DSQLAuthTokenGenerator` ユーティリティクラスを使用して、`admin` ロールを持つユーザーの認証トークンを生成します。*`insert-dsql-cluster-endpoint`* をクラスターエンドポイントに置き換えます。

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

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

AWSCredentials credentials = FallbackCredentialsFactory.GetCredentials();

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

Console.WriteLine(token);
```

Golang

> **Note**
>
> Golang SDK では、署名付きトークンを生成するための組み込みメソッドは提供されません。次のコード例に示すように、署名付きリクエストを手動で作成する必要があります。

次のコード例では、PostgreSQL ユーザーに基づいて `action` を指定します。

- `admin` ロールに接続する場合は、`DbConnectAdmin` アクションを使用します。
- カスタムデータベースロールに接続する場合は、`DbConnect` アクションを使用します。

次の例では、*`yourClusterEndpoint`* と*`#####`*に加えて、*`#####`*を使用します。PostgreSQL ユーザーに基づいて*`#####`*を指定します。

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

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

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

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

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

 return url, nil
}
```

データベースロールと IAM 認証の使用

Aurora DSQL は、IAM ロールと IAM ユーザーの両方を使用した認証をサポートしています。どちらの方法でも、Aurora DSQL データベースを認証してアクセスできます。

IAM ロール

IAM ロールは、特定のアクセス許可を持つ AWS アカウント内のアイデンティティですが、特定のユーザーには関連付けられていません。IAM ロールを使用して一時的なセキュリティ認証情報を提供します。IAM ロールは、いくつかの方法で一時的に引き受けることができます。

- AWS Management Consoleでロールを切り替える
- AWS CLI または AWS API オペレーションを呼び出す
- カスタム URL を使用する

ロールを引き受けると、ロールの一時的な認証情報を使用して Aurora DSQL にアクセスできます。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「IAM アイデンティティ」を参照してください。

IAM ユーザー

IAM ユーザーは、特定の許可を持つ AWS アカウント内のアイデンティティで、1 人のユーザーまたは 1 つのアプリケーションに関連付けられています。IAM ユーザーには、Aurora DSQL へのアクセスに使用できるパスワードやアクセスキーなどの長期的な認証情報があります。

> **Note**
>
> IAM 認証で SQL コマンドを実行するには、以下の例の IAM ロール ARN または IAM ユーザー ARN のいずれかを使用できます。

クラスターへの接続をデータベースロールに許可する

IAM ロールを作成し、IAM ポリシーアクション `dsql:DbConnect` を使用して接続認可を付与します。

IAM ポリシーは、クラスターリソースにアクセスするためのアクセス許可も付与する必要があります。ワイルドカード (*) を使用するか、「Amazon Aurora DSQL での IAM 条件キーの使用」の手順に従います。

データベースで SQL を使用するためのデータベースロールの許可

クラスターに接続するには、認可付きの IAM ロールを使用する必要があります。

1. SQL ユーティリティを使用して Aurora DSQL クラスターに接続します。

   IAM アクション `dsql:DbConnectAdmin` がクラスターに接続することを許可されている IAM ID で `admin` データベースロールを使用します。
2. 新しいデータベースロールを作成し、必ず `WITH LOGIN` オプションを指定します。

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

3. データベースロールを IAM ロール ARN に関連付けます。

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

4. データベースレベルのアクセス許可をデータベースロールに付与する

   次の例では、`GRANT` コマンドを使用してデータベース内で認可を提供します。

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

詳細については、「PostgreSQL ドキュメント」の「PostgreSQL GRANT」および「PostgreSQL の権限」を参照してください。

IAM ロールからのデータベース認可の取り消し

データベース認可を取り消すには、`AWS IAM REVOKE` オペレーションを使用します。

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

認可の取り消しの詳細については、「IAM と PostgreSQL を使用した認可の取り消し」を参照してください。

Aurora DSQL および PostgreSQL

Aurora DSQL は、トランザクションワークロード用に設計された PostgreSQL 互換の分散リレーショナルデータベースです。Aurora DSQL は、パーサー、プランナー、オプティマイザー、型システムなどの主要な PostgreSQL コンポーネントを使用します。

Aurora DSQL 設計により、サポートされているすべての PostgreSQL 構文によって互換性のある動作が提供されるため、同一のクエリ結果が得られます。例えば、Aurora DSQL は PostgreSQL と同様の型変換、算術演算、数値の精度とスケールを提供します。偏差は文書化されます。

Aurora DSQL では、オプティミスティック同時実行制御や分散スキーマ管理などの高度な機能も導入されています。これらの機能により、PostgreSQL の使い慣れたツールを使用しながら、最新のクラウドネイティブな分散アプリケーションに必要なパフォーマンスとスケーラビリティを活用できます。

PostgreSQL の互換性に関するハイライト

Aurora DSQL は現在 PostgreSQL バージョン 16 に基づいています。主な互換性は次のとおりです。

ワイヤプロトコル

Aurora DSQL は、標準の PostgreSQL v3 ワイヤプロトコルを使用します。これにより、標準の PostgreSQL クライアント、ドライバー、ツールとの統合が可能になります。例えば、Aurora DSQL は `psql`、`pgjdbc`、および `psycopg` と互換性があります。

SQL 互換性

Aurora DSQL は、トランザクションワークロードで一般的に使用される標準 PostgreSQL の式と関数を幅広くサポートしています。サポートされている SQL 式は、以下を含む PostgreSQL と同じ結果を生成します。

- null の処理
- ソート順の動作
- 数値オペレーションのスケールと精度
- 文字列オペレーションの同等性

詳細については、「Aurora DSQL での SQL 機能の互換性」を参照してください。

トランザクション管理

Aurora DSQL は、ACID トランザクションや PostgreSQL Repeatable Read に相当する分離レベルなど、PostgreSQL の主な特性を保持します。詳細については、「Aurora DSQL での同時実行制御」を参照してください。

アーキテクチャ上の主な相違点

Aurora DSQL の分散型で共有なしの設計では、従来の PostgreSQL といくつかの基本的な違いがあります。これらの違いは Aurora DSQL アーキテクチャに不可欠であり、パフォーマンスとスケーラビリティにおいて多くの利点を提供します。主な違いは次のとおりです。

オプティミスティック同時実行制御 (OCC)

Aurora DSQL は、オプティミスティック同時実行制御モデルを使用します。このロックフリーアプローチにより、トランザクションが互いにブロックされるのを防ぎ、デッドロックを排除し、高スループットの並列実行を可能にします。これらの機能により、Aurora DSQL は大規模で一貫したパフォーマンスを必要とするアプリケーションに特に役立ちます。その他の例については、「Aurora DSQL での同時実行制御」を参照してください。

非同期 DDL オペレーション

Aurora DSQL は DDL オペレーションを非同期的に実行します。これにより、スキーマの変更中に中断のない読み取りと書き込みが可能になります。分散アーキテクチャにより、Aurora DSQL は次のアクションを実行できます。

- DDL オペレーションをバックグラウンドタスクとして実行し、中断を最小限に抑えます。
- カタログの変更を強力な整合性のある分散トランザクションとして調整します。これにより、障害時や同時操作時でも、すべてのノードでアトミックな可視性が確保されます。
- コンピューティングレイヤーとストレージレイヤーが分離された複数のアベイラビリティーゾーン間で、完全に分散されたリーダーレスな方法で運用します。

詳細については、「Aurora DSQL での DDL および分散トランザクション」を参照してください。

Aurora DSQL での SQL 機能の互換性

Aurora DSQL と PostgreSQL は、すべての SQL クエリに対して同じ結果を返します。Aurora DSQL は、`ORDER BY` 句のない PostgreSQL とは異なることに注意してください。以下のセクションでは、PostgreSQL データ型と SQL コマンドの Aurora DSQL サポートについて説明します。

トピック

- Aurora DSQL でサポートされているデータ型
- Aurora DSQL でサポートされている SQL
- Aurora DSQL でサポートされている SQL コマンドのサブセット
- Aurora DSQL でサポートされていない PostgreSQL 機能

Aurora DSQL でサポートされているデータ型

Aurora DSQL は、一般的な PostgreSQL タイプのサブセットをサポートしています。

トピック

- 数値データ型
- 文字データ型
- 日付と時刻のデータ型
- その他のデータ型
- クエリランタイムデータ型

数値データ型

Aurora DSQL は、次の PostgreSQL 数値データ型をサポートしています。

名前	エイリアス	範囲と精度	ストレージサイズ	インデックスのサポート
`smallint`	`int2`	-32768 ~ +32767	2 バイト	はい
`integer`	`int`, `int4`	-2147483648 ~ +2147483647	4 バイト	はい

名前	エイリアス	範囲と精度	ストレージサイズ	インデックスのサポート
bigint	int8	-9223372036854775808 ~ +9223372036854775807	8 バイト	はい
real	float4	小数点以下 6 桁の精度	4 バイト	はい
double precision	float8	小数点以下 15 桁の精度	8 バイト	はい
numeric [(*p*, *s*)]	decimal [(*p*, *s*)] dec[(*p*,*s*)]	精度の選択が可能な真数。最大精度は 38 で、最大スケールは 37 です。[2] デフォルトは numeric (18,6) です。	精度桁あたり 8 バイト + 2 バイト。最大サイズは 27 バイトです。	いいえ

[2] – CREATE TABLE または ALTER TABLE ADD COLUMN の実行時にサイズを明示的に指定しない場合、Aurora DSQL はデフォルトを適用します。Aurora DSQL は、INSERT または UPDATE ステートメントを実行するときに制限を適用します。

文字データ型

Aurora DSQL は、次の PostgreSQL 文字データ型をサポートしています。

名前	エイリアス	説明	Aurora DSQL の制限	ストレージサイズ	インデックスのサポート
character [(*n*)]	char [(*n*)]	固定長のキャラクタ文字列	4096 バイト[1]	最大 4100 バイトの変数	はい
character varying [(*n*)]	varchar [(*n*)]	可変長の文字列	65535 バイト[1]	最大 65539 バイトの変数	はい
bpchar [(*n*)]		固定長の場合、これは char の エイ	4096 バイト[1]	最大 4100 バイトの変数	はい

名前	エイリアス	説明	Aurora DSQL の制限	ストレージサイズ	インデックスのサポート
		リアスです。可変長の場合、これは varchar のエイリアスであり、末尾のスペースは意味的に重要ではありません。			
text		可変長の文字列	1 MiB[1]	最大 1 MiB の変数	はい

[1] – CREATE TABLE または ALTER TABLE ADD COLUMN の実行時にサイズを明示的に指定しない場合、Aurora DSQL はデフォルトを適用します。Aurora DSQL は、INSERT または UPDATE ステートメントを実行するときに制限を適用します。

日付と時刻のデータ型

Aurora DSQL は、次の PostgreSQL 日付と時刻のデータ型をサポートしています。

名前	エイリアス	説明	Range	解決方法	ストレージサイズ	インデックスのサポート
date		カレンダー日付 (年、月、日)	4713 BC ~ 5874897 AD	1 日	4 バイト	はい
time [(*p*)] [without time zone]	times	時刻 (タイムゾーンなし)	0 ~ 1	1 マイクロ秒	8 バイト	はい

名前	エイリアス	説明	Range	解決方法	ストレージサイズ	インデックスのサポート
`time [ (p) ] with time zone`	`time`	時刻 (タイムゾーンを含む)	00:00:00+1559 ~ 24:00:00 –1559	1 マイクロ秒	12 バイト	いいえ
`timestamp [ (p) ] [ without time zone ]`		日付と時刻 (タイムゾーンなし)	4713 BC ~ 294276 AD	1 マイクロ秒	8 バイト	はい
`timestamp [ (p) ] with time zone`	`timestamptz`	日付と時刻 (タイムゾーンを含む)	4713 BC ~ 294276 AD	1 マイクロ秒	8 バイト	はい
`interval [ fields ] [ (p) ]`		期間	-178000000 年 ~ 178000000 年	1 マイクロ秒	16 バイト	いいえ

その他のデータ型

Aurora DSQL は、以下のその他の PostgreSQL データ型をサポートしています。

名前	エイリアス	説明	Aurora DSQL の制限	ストレージサイズ	インデックスのサポート
`boolean`	`bool`	論理ブール演算型 (true/false)		1 バイト	はい

名前	エイリアス	説明	Aurora DSQL の制限	ストレージサイズ	インデックスのサポート
bytea		バイナリデータ ("バイト配列")	1 MiB[1]	最大 1 MiB の変数制限	いいえ
UUID		Universally Unique Identifier		16 バイト	はい

[1] – CREATE TABLE または ALTER TABLE ADD COLUMN の実行時にサイズを明示的に指定しない場合、Aurora DSQL はデフォルトを適用します。Aurora DSQL は、INSERT または UPDATE ステートメントを実行するときに制限を適用します。

クエリランタイムデータ型

クエリランタイムデータ型は、クエリ実行時に使用される内部データ型です。これらのタイプは、スキーマで定義する varchar や integer などの PostgreSQL 互換タイプとは異なります。代わりに、これらのタイプは Aurora DSQL がクエリを処理するときに使用するランタイム表現です。

次のデータ型は、クエリランタイム中にのみサポートされます。

配列型

Aurora DSQL は、サポートされているデータ型の配列をサポートしています。例えば、整数の配列を持つことができます。以下の例に示すとおり、関数 string_to_array は、カンマ区切り文字 (,) を使用して、文字列を PostgreSQL スタイルの配列に分割します。クエリの実行中に、式、関数出力、または一時的な計算で配列を使用できます。

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

関数は次のようなレスポンスを返します。

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

inet 型

データ型は、IPv4, IPv6 ホストアドレス、およびそれらのサブネットを表します。このタイプは、ログを解析したり、IP サブネットでフィルタリングしたり、クエリ内でネットワーク計算を行ったりする場合に便利です。詳細については、PostgreSQL ドキュメントの「inet」を参照してください。

Aurora DSQL でサポートされている SQL

Aurora DSQL は、主要な PostgreSQL SQL 機能を幅広くサポートしています。以下のセクションでは、PostgreSQL 式の一般的なサポートについて説明します。これはすべてを網羅したリストではありません。

SELECT コマンド

Aurora DSQL は、`SELECT` コマンドの次の句をサポートしています。

プライマリ句	サポートされている句
`FROM`	
`GROUP BY`	`ALL`, `DISTINCT`
`ORDER BY`	`ASC`, `DESC`, `NULLS`
`LIMIT`	
`DISTINCT`	
`HAVING`	
`USING`	
`WITH` (テーブル共通表現)	
`INNER JOIN`	`ON`
`OUTER JOIN`	`LEFT`, `RIGHT`, `FULL`, `ON`
`CROSS JOIN`	`ON`

プライマリ句	サポートされている句
UNION	ALL
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR UPDATE	

データ定義言語 (DDL)

Aurora DSQL は、次の PostgreSQL DDL コマンドをサポートしています。

コマンド	プライマリ句	サポートされている句
CREATE	TABLE	CREATE TABLE コマンドでサポートされている構文については、「CREATE TABLE」を参照してください。
ALTER	TABLE	ALTER TABLE コマンドでサポートされている構文については、「ALTER TABLE」を参照してください。
DROP	TABLE	
CREATE	[UNIQUE] INDEX ASYNC	このコマンドを、ON、NULLS FIRST、NULLS LAST のパラメータとともに使用できます。 CREATE INDEX ASYNC コマンドでサポートされている構文については、「Aurora DSQL の非同期インデックス」を参照してください。
DROP	INDEX	

コマンド	プライマリ句	サポートされている句
CREATE	VIEW	CREATE VIEW コマンドでサポートされている構文について詳しくは、「CREATE VIEW」を参照してください。
ALTER	VIEW	ALTER VIEW コマンドでサポートされている構文については、「ALTER VIEW」を参照してください。
DROP	VIEW	DROP VIEW コマンドでサポートされている構文については、「DROP VIEW」を参照してください。
CREATE	ROLE, WITH	
CREATE	FUNCTION	LANGUAGE SQL
CREATE	DOMAIN	

データ操作言語 (DML)

Aurora DSQL は、次の PostgreSQL DML コマンドをサポートしています。

コマンド	プライマリ句	サポートされている句
INSERT	INTO	VALUES SELECT
UPDATE	SET	WHERE (SELECT) FROM, WITH
DELETE	FROM	USING, WHERE

データ制御言語 (DCL)

Aurora DSQL は、次の PostgreSQL DCL コマンドをサポートしています。

コマンド	サポートされている句
GRANT	ON, TO
REVOKE	ON, FROM, CASCADE, RESTRICT

トランザクション制御言語 (TCL)

Aurora DSQL は、次の PostgreSQL TCL コマンドをサポートしています。

コマンド	サポートされている句
COMMIT	
BEGIN	[WORK | TRANSACTION] [READ ONLY | READ WRITE]

ユーティリティコマンド

Aurora DSQL は、次の PostgreSQL ユーティリティコマンドをサポートしています。

- EXPLAIN
- ANALYZE (リレーション名のみ)

Aurora DSQL でサポートされている SQL コマンドのサブセット

この PostgreSQL セクションでは、サポートされている式に関する詳細情報を提供し、広範なパラメータセットとサブコマンドを持つコマンドに焦点を当てます。例えば、PostgreSQL の CREATE TABLE には句とパラメータが多数あります。このセクションでは、Aurora DSQL がこれらのコマンドでサポートするすべての PostgreSQL 構文要素について説明します。

トピック

- CREATE TABLE
- ALTER TABLE

- CREATE VIEW
- ALTER VIEW
- DROP VIEW

CREATE TABLE

CREATE TABLE は、新しいテーブルを定義します。

```
CREATE TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )

where column_constraint is:

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

and table_constraint is:

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

and like_option is:

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

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

ALTER TABLE

ALTER TABLE は、テーブルの定義を変更します。

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

where action is one of:

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

CREATE VIEW

CREATE VIEW は、新しい永続ビューを定義します。Aurora DSQL では、一時ビューはサポートされていません。永続ビューのみがサポートされています。

サポートされている構文

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

説明

CREATE VIEW は、クエリのビューを定義します。ビューは物理的にマテリアライズされていません。代わりに、クエリでビューが参照されるたびにクエリが実行されます。

CREATE or REPLACE VIEW は似ていますが、同じ名前のビューが既に存在する場合は置き換えられます。新しいクエリは、既存のビュークエリによって生成されたのと同じ列 (つまり、同じ順序で同じデータ型を持つ同じ列名) を生成する必要がありますが、リストの最後に列を追加する場合があります。出力列を生成する計算は異なる場合があります。

スキーマ名を指定すると (`CREATE VIEW myschema.myview ...` など)、指定したスキーマでビューが作成されます。指定しない場合、現在のスキーマに作成されます。

ビューの名前は、同じスキーマ内の他のリレーション (テーブル、インデックス、ビュー) の名前とは異なる必要があります。

パラメータ

`CREATE VIEW` は、自動更新可能なビューの動作を制御するさまざまなパラメータをサポートしています。

RECURSIVE

再帰的ビューを作成します。構文: `CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;` は `CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...) SELECT column_names FROM view_name;` と同等です。

再帰的ビューには、ビューの列名リストを指定する必要があります。

name

作成するビューの名前。オプションでスキーマ修飾されている場合があります。再帰的ビューには列名リストを指定する必要があります。

column_name

ビューの列に使用されるオプションの名前リスト。列名を指定しない場合、列名はクエリから推定されます。

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

この句は、ビューのオプションパラメータを指定します。次のパラメータがサポートされています。

- `check_option (enum)` – このパラメータは `local` または `cascaded` のいずれかであり、`WITH [ CASCADED | LOCAL ] CHECK OPTION` を指定した場合と同等です。
- `security_barrier (boolean)` – これは、ビューで行レベルのセキュリティを提供することを目的とする場合に使用します。現在、Aurora DSQL は行レベルのセキュリティをサポートしていませんが、このオプションでは、ビューの `WHERE` 条件 (および `LEAKPROOF` としてマークされている演算子を使用する条件) を最初に強制的に評価します。

- `security_invoker (boolean)` – このオプションでは、基盤となるベースリレーションが、ビュー所有者ではなくビューのユーザーの権限と照合されます。詳細については、以下のメモを参照してください。

上記のオプションはすべて、`ALTER VIEW` を使用して既存のビューで変更できます。

query

ビューの列と行を提供する `SELECT` または `VALUES` コマンド。

- `WITH [ CASCADED | LOCAL ] CHECK OPTION` – このオプションは、自動更新可能なビューの動作を制御します。このオプションを指定すると、ビューの `INSERT` および `UPDATE` コマンドがチェックされ、新しい行がビュー定義条件を満たしていることが確認されます (つまり、新しい行がチェックされ、ビューを通じて表示されていることを確認します)。そうでない場合、更新は拒否されます。`CHECK OPTION` が指定されていない場合、ビューの `INSERT` および `UPDATE` コマンドは、ビューでは表示されない行を作成できます。以下のチェックオプションがサポートされています。
- `LOCAL` – 新しい行は、ビュー自体で直接定義された条件に対してのみチェックされます。基盤となるベースビューで定義された条件はチェックされません (`CHECK OPTION` も指定しない限り)。
- `CASCADED` – 新しい行は、ビューおよび基盤となるすべてのベースビューの条件と照合されます。`CHECK OPTION` が指定されていて、`LOCAL` も `CASCADED` も指定されていない場合は、`CASCADED` が指定されたと見なされます。

> **Note**
>
> `CHECK OPTION` は `RECURSIVE` ビューでは使用できません。`CHECK OPTION` は、自動更新可能なビューでのみサポートされます。

メモ

ビューを削除するには、`DROP VIEW` ステートメントを使用します。

ビューの列の名前とデータ型は慎重に検討する必要があります。例えば、CREATE VIEW vista AS SELECT 'Hello World'; は、列名のデフォルトが `?column?;` であるため、お勧めしません。また、列のデータ型はデフォルトで `text` になりますが、これは必要ではない可能性があります。

より良い方法は、`CREATE VIEW vista AS SELECT text 'Hello World' AS hello;` のように列名とデータ型を明示的に指定することです。

デフォルトでは、ビューで参照される基盤となるベースリレーションへのアクセスは、ビュー所有者のアクセス許可によって決まります。場合によっては、これは基盤となるテーブルへの安全で制限されたアクセスを提供するために使用できます。ただし、すべてのビューが改ざんから保護されているわけではありません。

- ビューの `security_invoker` プロパティが true に設定されている場合、基盤となるベースリレーションへのアクセスは、ビューの所有者ではなく、クエリを実行するユーザーのアクセス許可によって決定されます。したがって、セキュリティ実行者ビューのユーザーは、ビューおよび基盤となるベースリレーションに対する適切なアクセス許可を持っている必要があります。
- 基盤となるベースリレーションのいずれかがセキュリティ実行者ビューである場合、元のクエリから直接アクセスされたものとして処理されます。したがって、セキュリティ実行者ビューは、`security_invoker` プロパティなしのビューからアクセスされた場合でも、常に現在のユーザーのアクセス許可を使用して、基盤となるベースリレーションをチェックします。
- ビューで呼び出された関数は、ビューを使用するクエリから直接呼び出された場合と同様に処理されます。したがって、ビューのユーザーは、ビューで使用されるすべての関数を呼び出すアクセス許可を持っている必要があります。ビュー内の関数は、関数が `SECURITY INVOKER` または `SECURITY DEFINER` として定義されているかに応じて、クエリを実行するユーザーの権限または関数の所有者の権限で実行されます。例えば、ビューで直接 `CURRENT_USER` を呼び出すと、ビューの所有者ではなく、常に実行ユーザーを返します。これはビューの `security_invoker` 設定の影響を受けないため、`security_invoker` が false に設定されたビューは `SECURITY DEFINER` 関数と同等ではありません。
- ビューを作成または置換するユーザーは、スキーマ内の参照オブジェクトを検索するために、ビュークエリで参照されるスキーマに対する `USAGE` 権限を持っている必要があります。ただし、この参照はビューが作成または置換された場合にのみ行われることに注意してください。したがって、ビューのユーザーは、セキュリティ実行者ビューの場合でも、ビュークエリで参照されるスキーマではなく、ビューを含むスキーマに対する `USAGE` 権限のみを必要とします。
- `CREATE OR REPLACE VIEW` が既存のビューで使用さると、ビューを定義する `SELECT` ルールに加えて、`WITH ( ... )` パラメータとその `CHECK OPTION` のみが変更されます。所有権、アクセス許可、SELECT 以外のルールなど、他のビュープロパティは変更されません。ビューを置き換えるには、ビューの所有者でなければなりません (所有ロールのメンバーである場合も含まれます)。

更新可能ビュー

簡単なビューは自動更新可能です。システムは、ビューに対する INSERT、UPDATE、DELETE ステートメントを通常のテーブルの場合と同じ方法で使用できるようにします。ビューは、以下の条件をすべて満たすと、自動更新可能になります。

- ビューの FROM リストに、テーブルまたは更新可能な別のビューの項目が 1 つだけ含まれている必要があります。
- ビューの定義の最上位レベルには、WITH、DISTINCT、GROUP BY、HAVING、LIMIT または OFFSET 句を含めることはできません。
- ビューの定義の最上位レベルには、セットオペレーション (UNION、INTERSECT または EXCEPT) を含めることはできません。
- ビューの選択リストに、集約関数、ウィンドウ関数、集合を返す関数を含めることはできません。

自動更新可能ビューでは、更新可能な列と更新不可能な列を混在させることができます。基盤となるベースリレーションの更新可能な列を単純に参照する列は更新可能です。そうでなければ、列は読み取り専用で、INSERT または UPDATE ステートメントでその列に値を割り当てようとしたらエラーが発生します。

ビューが自動更新可能であれば、システムはビューに対する INSERT、UPDATE または DELETE ステートメントを基盤となるベースリレーションへの対応するステートメントに変換します。ON CONFLICT UPDATE 句を持つ INSERT ステートメントは完全にサポートされます。

自動更新可能ビューが WHERE 条件を持つ場合、その条件は、ビューの UPDATE および DELETE ステートメントによって変更できるベースリレーションの行を制限します。ただし、UPDATE は、WHERE 条件を満たさないように行を変更し、ビューで非表示にすることができます。同様に、INSERT コマンドは、WHERE 条件を満たさず、そのためビューを通して参照できない行をベースリレーションに挿入する可能性があります。ON CONFLICT UPDATE は、ビューを通して見えない既存の行にも同様に影響を与える可能性があります。

CHECK OPTION は、INSERT および UPDATE コマンドがビューに表示されない行を作るのを防ぐために使用できます。

自動更新可能ビューが security_barrier プロパティを持つ場合、ビューのすべての WHERE 条件 (および LEAKPROOF の演算子を使ったすべての条件) が必ず、ビューのユーザーが追加した条件より前に評価されます。このため、最終的には、戻されない行 (ユーザーの WHERE 条件を満たさないため) はロックされてしまう可能性があります。EXPLAIN を使って、リレーションのレベルで適用されている条件 (その結果、行をロックしない) と適用されていない条件を確認できます。

これらの条件をすべて満たさないより複雑なビューは、デフォルトで読み取り専用です。システムは、ビューに対する挿入、更新、削除を許可しません。

> **Note**
>
> ビューに対する挿入、更新、削除を実行するユーザーには、ビューに対する対応する挿入、更新、または削除の権限が必要です。デフォルトでは、ビューの所有者は基盤となるベースリレーションに対する適切な権限を持っている必要がありますが、更新を行うユーザーは基盤となるベースリレーションに対するアクセス許可を必要としません。ただし、ビューの security_invoker が true に設定されている場合、ビューの所有者ではなく更新を行うユーザーが基盤となるベースリレーションに対する適切な権限を持っている必要があります。

例

すべてのコメディ映画からなるビューを作成します。

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

これは、ビューを作成した時点で `film` テーブル内にある列を持つビューを作成します。ビューを作成するために `*` が使用されましたが、その後にテーブルに追加された列はビューには含まれません。

`LOCAL CHECK OPTION` でビューを作成します。

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

これにより、新しい行の `kind` と `classification` の両方を確認するビューが作成されます。

更新可能な列と更新不可能な列が混在するビューを作成します。

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
```

```
            (SELECT avg(r.rating)
             FROM user_ratings r
             WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

このビューは、INSERT、UPDATE、および DELETE をサポートします。フィルムテーブルのすべての列は更新可能ですが、計算される列 country と avg_rating は読み取り専用です。

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

> **Note**
>
> この CREATE では再帰的ビューの名前はスキーマ修飾されていますが、その内側の自己参照はスキーマ修飾されていません。これは、暗黙的に作成される共通テーブル式 (CTE) の名前はスキーマ修飾できないためです。

互換性

CREATE OR REPLACE VIEW は PostgreSQL の言語拡張です。WITH (...) 句も拡張であり、セキュリティバリアビューとセキュリティ実行者ビューも同様です。Aurora DSQL は、これらの言語拡張をサポートしています。

ALTER VIEW

ALTER VIEW ステートメントは、既存のビューのさまざまなプロパティの変更を許可し、Aurora DSQL はこのコマンドのすべての PostgreSQL 構文をサポートします。

サポートされている構文

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

説明

ALTER VIEW は、ビューの各種補助プロパティを変更します。(ビューを定義するクエリを変更する場合は、CREATE OR REPLACE VIEW を使用します。) ALTER VIEW を使用するには、ビューの所有者である必要があります。ビューのスキーマを変更するには、新しいスキーマに対する CREATE 権限も必要です。所有者を変更するには、新しい所有者ロールに対して SET ROLE ができなければなりません。また、そのロールにはビューのスキーマに対する CREATE 権限が必要です。(これらの制限は、ビューの削除および再作成によりユーザーが実行できないことを、所有者の変更により実行できないようにするためのものです。)

パラメータ

ALTER VIEW 個のパラメータ

name

既存のビューの名前 (オプションでスキーマ修飾可) です。

column_name

既存の列の新しい名前です。

IF EXISTS

ビューが存在しなかったとしてもエラーになりません。この場合、通知が発行されます。

SET/DROP DEFAULT

これらのフォームは、列のデフォルト値を設定または削除します。ビュー列のデフォルト値は、ビューを対象とした任意の INSERT または UPDATE コマンドに代入されます。ビューのデフォルトは、基になるリレーションのデフォルト値よりも優先されます。

new_owner

ビューの新しい所有者のユーザー名です。

new_name

ビューの新しい名前です。

new_schema

ビューの新しいスキーマです。

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

ビューのオプションを設定またはリセットします。サポートされているオプションは以下のとおりです。

- check_option (enum) – ビューのチェックオプションを変更します。値は local または cascaded である必要があります。
- security_barrier (boolean) – ビューのセキュリティバリアプロパティを変更します。値は、true や false などのブール値である必要があります。
- security_invoker (boolean) – ビューのセキュリティバリアプロパティを変更します。値は、true や false などのブール値である必要があります。

メモ

PostgreSQL の歴史上の理由から、ALTER TABLE はビューでも使用できますが、ビューで使用できる ALTER TABLE のバリエーションは、上記で示したものと同じものだけです。

例

ビュー foo の名前を bar に変更します。

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

更新可能なビューにデフォルトの列値を付与します。

```
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
```

互換性

ALTER VIEW は、Aurora DSQL がサポートする SQL 標準の PostgreSQL 拡張です。

DROP VIEW

DROP VIEW ステートメントは既存のビューを削除します。Aurora DSQL は、このコマンドの完全な PostgreSQL 構文をサポートしています。

サポートされている構文

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

説明

`DROP VIEW` は既存のビューを削除します。このコマンドを実行できるのは、ビューの所有者のみです。

パラメータ

`IF EXISTS`

ビューが存在しなかったとしてもエラーになりません。この場合、通知が発行されます。

`name`

削除するビューの名前 (オプションでスキーマ修飾可) です。

`CASCADE`

削除するビューに依存しているオブジェクト (他のビューなど) を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します。

`RESTRICT`

依存するオブジェクトがある場合は、ビューの削除を拒否します。これがデフォルトです。

例

```
DROP VIEW kinds;
```

互換性

このコマンドは SQL 標準に準拠していますが、標準ではコマンドごとに 1 つのビューしか削除できないこと、および Aurora DSQL がサポートする PostgreSQL 拡張である `IF EXISTS` オプションを除きます。

Aurora DSQL でサポートされていない PostgreSQL 機能

Aurora DSQL は PostgreSQL と互換性があります。つまり、Aurora DSQL は ACID トランザクション、セカンダリインデックス、結合、挿入、更新などのコアリレーショナル機能をサポートしていま

す。サポートされている SQL 機能の概要については、「Supported SQL expressions」を参照してください。

以下のセクションでは、Aurora DSQL で現在サポートされていない PostgreSQL 機能について説明します。

サポートされていないオブジェクト

Aurora DSQL でサポートされていないオブジェクトは次のとおりです。

- 単一の Aurora DSQL クラスター上の複数のデータベース
- 一時テーブル
- トリガー
- 型 (部分的なサポート)
- テーブルスペース
- SQL 以外の言語で記述された関数
- シーケンス
- パーティション

サポートされていない制約

- 外部キー
- 排他制約

サポートされていないコマンド

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

> **Note**
>
> Aurora DSQL はバキューム処理を必要としません。システムは統計を維持し、手動バキュームコマンドを使用せずにストレージの最適化を自動的に管理します。

サポートされていない拡張機能

Aurora DSQL は PostgreSQL 拡張機能をサポートしていません。以下の表は、サポートされていない拡張機能を示しています。

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

サポートされていない SQL 式

次の表は、Aurora DSQL でサポートされていない句を示しています。

カテゴリ	プライマリ句	サポートされていない句
CREATE	INDEX ASYNC	ASC | DESC
CREATE	INDEX [1]	
TRUNCATE		
ALTER	SYSTEM	すべての ALTER SYSTEM コマンドがブロックされます。
CREATE	TABLE	COLLATE, AS SELECT, INHERITS, PARTITION
CREATE	FUNCTION	LANGUAGE *non-sql-lang* 、*non-sql-lang* は SQL 以外の言語です。
CREATE	TEMPORARY	TABLES
CREATE	EXTENSION	

カテゴリ	プライマリ句	サポートされていない句
CREATE	SEQUENCE	
CREATE	MATERIALIZED	VIEW
CREATE	TABLESPACE	
CREATE	TRIGGER	
CREATE	TYPE	
CREATE	DATABASE	追加のデータベースを作成することはできません。

[1] 指定したテーブルの列にインデックスを作成するには、「Aurora DSQL の非同期インデックス」を参照してください。

PostgreSQL 互換性に関する Aurora DSQL の考慮事項

Aurora DSQL を使用する場合は、次の互換性の制限を考慮してください。一般的な考慮事項については、「Amazon Aurora DSQL の使用に関する考慮事項」を参照してください。クォータと制限については、「Amazon Aurora DSQL のクラスタークォータとデータベース制限」を参照してください。

- Aurora DSQL は、`postgres` という名前の単一の組み込みデータベースを使用します。追加のデータベースを作成することや、`postgres` データベースを名前変更、削除することはできません。
- `postgres` データベースは UTF-8 文字エンコードを使用します。エンコードを変更することはできません。
- データベースは `C` のみを使用します。
- Aurora DSQL はシステムのタイムゾーンに `UTC` を使用します。パラメータまたは `SET TIMEZONE` などの SQL ステートメントを使用してタイムゾーンを変更することはできません。
- トランザクション分離レベルは PostgreSQL `Repeatable Read` で固定されています。
- トランザクションには次の制約があります。
  - トランザクションでは DDL オペレーションと DML オペレーションを混在させることはできません。

  - トランザクションに含めることができる DDL ステートメントは 1 つのみです。
  - トランザクションは、セカンダリインデックスの数に関係なく、3,000 行まで変更できます。
  - 3,000 行の制限は、すべての DML ステートメント (`INSERT`、`UPDATE`、`DELETE`) に適用されます。
- データベース接続は 1 時間後にタイムアウトします。
- Aurora DSQL では現在、`GRANT [permission] ON DATABASE` を実行できません。そのステートメントを実行しようとすると、Aurora DSQL はエラーメッセージ `ERROR: unsupported object type in GRANT` を返します。
- Aurora DSQL は、管理者以外のユーザーロールに `CREATE SCHEMA` コマンドの実行を許可しません。`GRANT [permission] on DATABASE` コマンドを実行して、データベースに対する `CREATE` アクセス許可を付与することはできません。管理者以外のユーザーロールがスキーマを作成しようとすると、Aurora DSQL はエラーメッセージ `ERROR: permission denied for database postgres` を返します。
- 管理者以外のユーザーは、パブリックスキーマにオブジェクトを作成できません。パブリックスキーマにオブジェクトを作成できるのは、管理者ユーザーのみです。管理者ユーザーロールには、これらのオブジェクトへの読み取り、書き込み、変更のアクセス許可を管理者以外のユーザーに付与する権限がありますが、パブリックスキーマ自体に `CREATE` アクセス許可を付与することはできません。管理者以外のユーザーは、オブジェクトの作成に、ユーザーが作成した別のスキーマを使用する必要があります。
- Aurora DSQL は `ALTER ROLE [] CONNECTION LIMIT` コマンドをサポートしていません。接続制限の引き上げが必要な場合は、AWS サポートにお問い合わせください。
- Aurora DSQL は、Python 用の非同期 PostgreSQL データベースドライバーである asyncpg をサポートしていません。

Aurora DSQL での同時実行制御

同時実行により、複数のセッションがデータの整合性と一貫性を損なうことなく、同時にデータにアクセスして変更することができます。Aurora DSQL は、最新かつロックフリーの同時実行制御メカニズムを実装しながら、PostgreSQL との互換性を提供します。スナップショットの分離を通じて完全な ACID コンプライアンスを維持し、データ整合性および信頼性を確保します。

Aurora DSQL の主な利点は、一般的なデータベースパフォーマンスのボトルネックを排除するロックフリーのアーキテクチャです。Aurora DSQL は、遅いトランザクションが他のオペレーションをブロックするのを防ぎ、デッドロックのリスクを排除します。このアプローチにより、Aurora DSQL

は、パフォーマンスとスケーラビリティが重要な高スループットアプリケーションに特に役立ちます。

トランザクションの競合

Aurora DSQL は、オプティミスティック同時実行制御 (OCC) を使用します。これは、従来のロックベースのシステムとは動作が異なります。OCC はロックを使用する代わりに、コミット時に競合を評価します。同じ行の更新中に複数のトランザクションが競合すると、Aurora DSQL は次のようにトランザクションを管理します。

- コミット時間が最も早いトランザクションは、Aurora DSQL によって処理されます。
- 競合するトランザクションは、再試行する必要があることを示す PostgreSQL シリアル化エラーを受け取ります。

競合を処理する再試行ロジックを実装するようにアプリケーションを設計します。理想的な設計パターンはべき等性であり、可能な限り最初の手段としてトランザクションの再試行を可能にします。推奨されるロジックは、標準 PostgreSQL のロックタイムアウトまたはデッドロック状況におけるアボートおよび再試行ロジックに似ています。ただし、OCC では、アプリケーションでこのロジックをより頻繁に実行する必要があります。

トランザクションパフォーマンスを最適化するためのガイドライン

パフォーマンスを最適化するには、単一キーまたは小さなキー範囲における高い競合を最小限に抑えます。この目標を達成するには、次のガイドラインを使用して、クラスターキー範囲に更新を分散するようにスキーマを設計します。

- テーブルのランダムなプライマリキーを選択します。
- 単一キーでの競合が増加するパターンは避けてください。このアプローチにより、トランザクション量が増えても最適なパフォーマンスが保証されます。

Aurora DSQL での DDL および分散トランザクション

データ定義言語 (DDL) の動作は、Aurora DSQL では PostgreSQL とは異なります。Aurora DSQL は、マルチテナントのコンピューティングおよびストレージフリート上に構築されたマルチ AZ 分散型およびシェアードナッシングのデータベースレイヤーを特徴としています。単一のプライマリデータベースノードまたはリーダーが存在しないため、データベースカタログは分散されます。したがって、Aurora DSQL は、DDL スキーマの変更を分散トランザクションとして管理します。

具体的には、DDL の動作は Aurora DSQL では次のように異なります。

同時実行制御エラー

Aurora DSQL は、あるトランザクションを実行し、別のトランザクションがリソースを更新すると、同時実行制御違反エラーを返します。例えば、次の一連のアクションを考えてみましょう。

1. セッション 1 では、ユーザーはテーブル `mytable` に列を追加します。
2. セッション 2 では、ユーザーは `mytable` に行を挿入しようとしています。

   Aurora DSQL は `SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (OC001).` エラーを返します。

同じトランザクション内の DDL と DML

Aurora DSQL のトランザクションには 1 つの DDL ステートメントのみを含めることができます。DDL ステートメントと DML ステートメントの両方を含めることはできません。この制限は、テーブルを作成し、同じトランザクション内の同じテーブルにデータを挿入できないことを意味します。例えば、Aurora DSQL は次のシーケンシャルトランザクションをサポートしています。

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

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

Aurora DSQL は、`CREATE` および `INSERT` ステートメントの両方を含む次のトランザクションをサポートしていません。

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

非同期 DDL

標準の PostgreSQL では、`CREATE INDEX` などの DDL オペレーションが影響を受けるテーブルをロックすることで、他のセッションからの読み取りと書き込みができなくなります。Aurora DSQL では、これらの DDL ステートメントはバックグラウンドマネージャーを使用して非同期的に実行されます。影響を受けるテーブルへのアクセスはブロックされません。したがって、大きなテーブルの DDL はダウンタイムやパフォーマンスに影響を与えることなく実行できます。Aurora DSQL の非同期ジョブマネージャーの詳細については、「Aurora DSQL の非同期インデックス」を参照してください。

Aurora DSQL のプライマリキー

Aurora DSQL では、プライマリキーはテーブルデータを物理的に整理する機能です。これは、PostgreSQL の `CLUSTER` オペレーションや他のデータベースのクラスター化されたインデックスに似ています。プライマリキーを定義すると、Aurora DSQL はテーブル内のすべての列を含むインデックスを作成します。Aurora DSQL のプライマリキー構造により、効率的なデータアクセスと管理が保証されます。

データ構造とストレージ

プライマリキーを定義すると、Aurora DSQL はテーブルデータをプライマリキーの順序で保存します。このインデックス組織構造により、従来の B ツリーインデックスのようにデータへのポインタに従う代わりに、プライマリキールックアップですべての列値を直接取得できます。データを 1 回だけ再編成する PostgreSQL の `CLUSTER` オペレーションとは異なり、Aurora DSQL はこの順序を自動的かつ継続的に維持します。このアプローチにより、プライマリキーのアクセスに依存するクエリのパフォーマンスが向上します。

Aurora DSQL は、プライマリキーを使用して、テーブルとインデックスの各行にクラスター全体の一意のキーを生成します。この一意のキーは、分散データ管理の基盤にも使用されます。これにより、複数のノードにまたがるデータの自動パーティショニングが可能になり、スケーラブルなストレージと高い同時実行性がサポートされます。その結果、プライマリキー構造により、Aurora DSQL は自動的にスケールされ、同時ワークロードを効率的に管理できます。

プライマリキーを選択するためのガイドライン

Aurora DSQL でプライマリキーを選択して使用する場合は、次のガイドラインを考慮してください。

- テーブルの作成時にプライマリキーを定義します。このキーを後で変更したり、新しいプライマリキーを追加したりすることはできません。プライマリキーは、データのパーティショニングと書き込みスループットの自動スケーリングに使用されるクラスター全体のキーの一部になります。プライマリキーを指定しない場合、Aurora DSQL は合成非表示 ID を割り当てます。
- 書き込み量が多いテーブルでは、一定間隔で増加する整数をプライマリキーに使用しないでください。これが原因で、すべての新しい挿入を 1 つのパーティションに送信することで、パフォーマンスの問題が生じる可能性があります。代わりに、ランダム分散のプライマリキーを使用して、ストレージパーティション間で書き込みが均等に分散されるようにします。
- 頻繁に変更されないテーブルや読み取り専用のテーブルでは、昇順キーを使用できます。昇順キーの例は、タイムスタンプまたはシーケンス番号です。高密度キーには、間隔の狭い値または重複した値が多くあります。書き込みパフォーマンスの重要性が低いため、高密度であっても昇順キーを使用できます。
- フルテーブルスキャンがパフォーマンス要件を満たさない場合は、より効率的なアクセス方法を選択します。ほとんどの場合、これは、最も一般的な結合キーとルックアップキーに一致するプライマリキーをクエリで使用することを意味します。
- プライマリキーの列の最大合計サイズは 1 キビバイトです。詳細については、「Database limits in Aurora DSQL」および「Supported data types in Aurora DSQL」を参照してください。
- プライマリキーまたはセカンダリインデックスには、最大 8 つの列を含めることができます。詳細については、「Database limits in Aurora DSQL」および「Supported data types in Aurora DSQL」を参照してください。

Aurora DSQL の非同期インデックス

`CREATE INDEX ASYNC` コマンドは、指定されたテーブルの 1 つ以上の列にインデックスを作成します。このコマンドは、他のトランザクションをブロックしない非同期 DDL オペレーションです。`CREATE INDEX ASYNC` を実行すると、Aurora DSQL はすぐに `job_id` を返します。

この非同期ジョブのステータスは、`sys.jobs` システムビューでモニタリングできます。インデックス作成ジョブが進行中の場合は、次のプロシージャとコマンドを使用できます。

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

指定されたジョブが完了または失敗するまでセッションをブロックします。成功または失敗を示すブール値を返します。

`DROP INDEX`

進行中のインデックスビルドジョブをキャンセルします。

Aurora DSQL が非同期インデックスの作成を完了すると、システムカタログが更新され、インデックスがアクティブであることが示されます。

> **Note**
>
> この更新中に同じ名前空間内のオブジェクトにアクセスする同時トランザクションでは、同時実行エラーが発生する可能性があります。

Aurora DSQL が非同期インデックスタスクを完了すると、システムカタログが更新され、インデックスがアクティブであることが示されます。この時点で他のトランザクションが同じ名前空間内のオブジェクトを参照している場合、同時実行エラーが表示されることがあります。

構文

`CREATE INDEX ASYNC` は以下の構文を使用します。

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

パラメータ

`UNIQUE`

Aurora DSQL がインデックスを作成する際、およびデータを追加するたびに、テーブル内の重複した値をチェックすることを示します。このパラメータを指定した場合、エントリが重複するオペレーションを挿入および更新すると、エラーが発生します。

`IF NOT EXISTS`

同じ名前のインデックスが既に存在する場合、Aurora DSQL が例外をスローすべきではないことを示します。この場合、Aurora DSQL は新しいインデックスを作成しません。作成しようとしているインデックスは、存在するインデックスとは大きく異なる構造を持つ可能性があることに注意してください。このパラメータを指定する場合、インデックス名は必須です。

`name`

インデックスの名前。このパラメータにスキーマの名前を含めることはできません。

Aurora DSQL は、親テーブルと同じスキーマにインデックスを作成します。インデックスの名前は、スキーマ内のテーブルやインデックスなどの他のオブジェクトの名前とは異なる必要があります。

名前を指定しない場合、Aurora DSQL は親テーブルとインデックス付き列の名前に基づいて名前を自動的に生成します。例えば、`CREATE INDEX ASYNC on table1 (col1, col2)` を実行すると、Aurora DSQL は自動的にインデックス `table1_col1_col2_idx` に名前を付けます。

`NULLS FIRST | LAST`

null 列と null 列以外の列のソート順。`FIRST` は、Aurora DSQL が null 列以外の列の前に null 列をソートする必要があることを示します。`LAST` は、Aurora DSQL が null 列以外の列の後に null 列をソートする必要があることを示します。

`INCLUDE`

非キー列としてインデックスに含める列のリスト。インデックススキャン検索条件にキー以外の列を使用することはできません。Aurora DSQL は、インデックスの一意性の観点から列を無視します。

`NULLS DISTINCT | NULLS NOT DISTINCT`

Aurora DSQL が一意のインデックスで null 値を個別と見なすかどうかを指定します。デフォルトは `DISTINCT` です。つまり、一意のインデックスには列に複数の null 値を含めることができます。`NOT DISTINCT` は、インデックスが列に複数の null 値を含めることができないことを示します。

使用に関する注意事項

以下のガイドラインを検討します。

- `CREATE INDEX ASYNC` コマンドはロックを導入しません。また、Aurora DSQL がインデックスの作成に使用するベーステーブルにも影響しません。
- スキーマ移行オペレーション中は、`sys.wait_for_job(job_id)`*`'your_index_creation_job_id'`* のプロシージャが役立ちます。これにより、後続の DDL および DML オペレーションが新しく作成されたインデックスをターゲットにします。
- Aurora DSQL は、新しい非同期タスクを実行するたびに、`sys.jobs` ビューをチェックし、30 分以上ステータスが `completed` または `failed` のタスクを削除します。したがって、`sys.jobs` では主に進行中のタスクが表示され、古いタスクに関する情報は含まれません。

- Aurora DSQL が非同期インデックスの構築に失敗した場合、インデックスは INVALID のままになります。一意のインデックスの場合、インデックスを削除するまで、DML オペレーションは一意性の制約を受けます。無効なインデックスを削除して再作成することをお勧めします。

インデックスの作成: 例

次の例は、スキーマ、テーブル、インデックスを作成する方法を示しています。

1. test.departments という名前のテーブルを作成します。

```
CREATE SCHEMA test;

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

2. テーブルに行を挿入します。

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

3. 非同期インデックスを作成します。

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

   CREATE INDEX コマンドは、次に示すようにジョブ ID を返します。

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

   job_id は、Aurora DSQL がインデックスを作成するための新しいジョブを送信したことを示します。sys.wait_for_job(job_id)*'your_index_creation_job_id'* のプロシージャを使用して、ジョブが終了またはタイムアウトするまでセッションの他の作業をブロックできます。

インデックス作成のステータスのクエリ: 例

次の例に示すように、sys.jobs システムビューをクエリしてインデックスの作成ステータスを確認します。

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

Aurora DSQL は、次のようなレスポンスを返します。

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

[ステータス] 列には、次のいずれかの値を指定することができます。

submitted	processing	failed	completed
タスクは送信されましたが、Aurora DSQL はまだタスクの処理を開始していません。	Aurora DSQL はタスクを処理しています。	タスクが失敗しました。詳細については、[details] 列を参照してください。Aurora DSQL がインデックスの構築に失敗した場合、Aurora DSQL ではインデックス定義を自動的に削除しません。`DROP INDEX` コマンドを使用してインデックスを手動で削除する必要があります。	Aurora DSQL

カタログテーブル `pg_index` および `pg_class` を使用して、インデックスの状態をクエリすることもできます。具体的には、属性 `indisvalid` と `indisimmediate` により、インデックスの状態を知ることができます。Aurora DSQL がインデックスを作成する間、初期ステータスは `INVALID` です。インデックスの `indisvalid` フラグは、`FALSE` または `f` を返します。これはインデックスが有効でないことを示します。フラグが `TRUE` または `t` を返すと、インデックスは準備完了です。

```
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)
```

一意のインデックスビルドの失敗

非同期の一意インデックスビルドジョブが詳細 `Found duplicate key while validating index for UCVs` で失敗状態を示している場合、これは一意性の制約違反のために一意インデックスを構築できなかったことを示します。

一意のインデックスビルドの失敗を解決するには

1. 一意のセカンダリインデックスで指定されたキーのエントリが重複しているプライマリテーブルの行を削除します。
2. 失敗したインデックスを削除します。
3. 新しいインデックス作成コマンドを発行します。

プライマリテーブルでの一意性違反の検出

次の SQL クエリは、テーブルの指定された列で重複する値を識別するのに役立ちます。これは、現在プライマリキーとして設定されていない列や、ユーザーテーブルの E メールアドレスなどの一意の制約がない列に一意性を適用する必要がある場合に特に便利です。

以下の例は、サンプルユーザーテーブルを作成し、既知の重複を含むテストデータを入力してから、検出クエリを実行する方法を示しています。

テーブルスキーマを定義する

```
-- 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
);
```

重複する E メールアドレスのセットを含むサンプルデータを挿入する

```
-- 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');
```

重複検出のクエリを実行する

```
-- 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;
```

重複した E メールアドレスを持つすべてのレコードを表示する

```
 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)
```

ここでインデックス作成ステートメントを試すと、失敗します。

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

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

Aurora DSQL のシステムテーブルとコマンド

Aurora DSQL でサポートされているシステムテーブルとカタログについては、以下のセクションを参照してください。

システムテーブル

Aurora DSQL は PostgreSQL と互換性があるため、PostgreSQL の多くのシステムカタログテーブルとビューが Aurora DSQL にも存在します。

重要な PostgreSQL カタログテーブルとビュー

次の表は、Aurora DSQL で使用できる最も一般的なテーブルとビューを示しています。

名前	説明
pg_namespace	すべてのスキーマに関する情報
pg_tables	すべてのテーブルに関する情報
pg_attribute	すべての属性に関する情報
pg_views	(事前) 定義ビューに関する情報
pg_class	すべてのテーブル、列、インデックス、および同様のオブジェクトに関する説明
pg_stats	プランナー統計の表示
pg_user	ユーザーに関する情報
pg_roles	ユーザーとグループに関する情報
pg_indexes	すべてのインデックスのリスト
pg_constraint	テーブルの制約のリスト

サポートされているカタログテーブルとサポートされていないカタログテーブル

次の表は、Aurora DSQL でサポートされているテーブルとサポートされていないテーブルを示しています。

名前	Aurora DSQL に適用可能
pg_aggregate	いいえ

名前	Aurora DSQL に適用可能
pg_am	あり
pg_amop	いいえ
pg_amproc	いいえ
pg_attrdef	はい
pg_attribute	はい
pg_authid	いいえ (pg_roles を使用)
pg_auth_members	はい
pg_cast	あり
pg_class	あり
pg_collation	あり
pg_constraint	はい
pg_conversion	いいえ
pg_database	いいえ
pg_db_role_setting	はい
pg_default_acl	あり
pg_depend	あり
pg_description	はい
pg_enum	いいえ
pg_event_trigger	いいえ
pg_extension	いいえ

名前	Aurora DSQL に適用可能
pg_foreign_data_wrapper	いいえ
pg_foreign_server	いいえ
pg_foreign_table	いいえ
pg_index	はい
pg_inherits	はい
pg_init_privs	いいえ
pg_language	いいえ
pg_largeobject	いいえ
pg_largeobject_metadata	はい
pg_namespace	はい
pg_opclass	なし
pg_operator	あり
pg_opfamily	なし
pg_parameter_acl	あり
pg_partitioned_table	いいえ
pg_policy	いいえ
pg_proc	いいえ
pg_publication	いいえ
pg_publication_namespace	いいえ
pg_publication_rel	いいえ

名前	Aurora DSQL に適用可能
pg_range	あり
pg_replication_origin	いいえ
pg_rewrite	いいえ
pg_seclabel	いいえ
pg_sequence	いいえ
pg_shdepend	はい
pg_shdescription	はい
pg_shseclabel	なし
pg_statistic	あり
pg_statistic_ext	いいえ
pg_statistic_ext_data	いいえ
pg_subscription	いいえ
pg_subscription_rel	いいえ
pg_tablespace	いいえ
pg_transform	いいえ
pg_trigger	いいえ
pg_ts_config	はい
pg_ts_config_map	あり
pg_ts_dict	あり
pg_ts_parser	あり

名前	Aurora DSQL に適用可能
pg_ts_template	あり
pg_type	はい
pg_user_mapping	いいえ

サポートされているシステムビューとサポートされていないシステムビュー

次の表は、Aurora DSQL でサポートされているビューとサポートされていないビューを示しています。

名前	Aurora DSQL に適用可能
pg_available_extensions	いいえ
pg_available_extension_versions	いいえ
pg_backend_memory_contexts	あり
pg_config	いいえ
pg_cursors	いいえ
pg_file_settings	いいえ
pg_group	あり
pg_hba_file_rules	いいえ
pg_ident_file_mappings	いいえ
pg_indexes	あり
pg_locks	いいえ
pg_matviews	いいえ
pg_policies	いいえ

名前	Aurora DSQL に適用可能
pg_prepared_statements	いいえ
pg_prepared_xacts	いいえ
pg_publication_tables	いいえ
pg_replication_origin_status	いいえ
pg_replication_slots	いいえ
pg_roles	あり
pg_rules	いいえ
pg_seclabels	いいえ
pg_sequences	いいえ
pg_settings	はい
pg_shadow	あり
pg_shmem_allocations	あり
pg_stats	はい
pg_stats_ext	いいえ
pg_stats_ext_exprs	いいえ
pg_tables	はい
pg_timezone_abbrevs	あり
pg_timezone_names	あり
pg_user	はい
pg_user_mappings	なし

名前	Aurora DSQL に適用可能
pg_views	あり
pg_stat_activity	いいえ
pg_stat_replication	いいえ
pg_stat_replication_slots	いいえ
pg_stat_wal_receiver	いいえ
pg_stat_recovery_prefetch	いいえ
pg_stat_subscription	いいえ
pg_stat_subscription_stats	いいえ
pg_stat_ssl	あり
pg_stat_gssapi	いいえ
pg_stat_archiver	いいえ
pg_stat_io	いいえ
pg_stat_bgwriter	いいえ
pg_stat_wal	いいえ
pg_stat_database	いいえ
pg_stat_database_conflicts	いいえ
pg_stat_all_tables	いいえ
pg_stat_all_indexes	いいえ
pg_statio_all_tables	いいえ
pg_statio_all_indexes	いいえ

名前	Aurora DSQL に適用可能
pg_statio_all_sequences	いいえ
pg_stat_slru	いいえ
pg_statio_user_tables	いいえ
pg_statio_user_sequences	いいえ
pg_stat_user_functions	いいえ
pg_stat_user_indexes	いいえ
pg_stat_progress_analyze	いいえ
pg_stat_progress_basebackup	いいえ
pg_stat_progress_cluster	いいえ
pg_stat_progress_create_index	いいえ
pg_stat_progress_vacuum	いいえ
pg_stat_sys_indexes	いいえ
pg_stat_sys_tables	いいえ
pg_stat_xact_all_tables	いいえ
pg_stat_xact_sys_tables	いいえ
pg_stat_xact_user_functions	いいえ
pg_stat_xact_user_tables	いいえ
pg_statio_sys_indexes	いいえ
pg_statio_sys_sequences	いいえ
pg_statio_sys_tables	いいえ

名前	Aurora DSQL に適用可能
pg_statio_user_indexes	いいえ

sys.jobs および sys.iam_pg_role_mappings ビュー

Aurora DSQL は、次のシステムビューをサポートしています。

sys.jobs

sys.jobs は、非同期ジョブに関するステータス情報を提供します。例えば、非同期インデックスを作成すると、Aurora DSQL は job_uuid を返します。この job_uuid と sys.jobs を使って、ジョブのステータスを検索できます。

```
SELECT * FROM sys.jobs WHERE job_id = 'example_job_uuid';

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

sys.iam_pg_role_mappings

ビュー sys.iam_pg_role_mappings には、IAM ユーザーに付与されたアクセス許可に関する情報が表示されます。例えば、DQSLDBConnect が Aurora DSQL に管理者以外のユーザーへのアクセスを許可する IAM ロールで、testuser という名前のユーザーに DQSLDBConnect ロールと対応するアクセス許可が付与されている場合、sys.iam_pg_role_mappings ビューをクエリして、どのユーザーにどのアクセス許可が付与されているかを確認できます。

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

pg_class テーブル

pg_class テーブルには、データベースオブジェクトに関するメタデータが保存されます。テーブル内の行数の概算数を取得するには、次のコマンドを実行します。

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

このコマンドにより、以下のような出力が返されます。

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

ANALYZE コマンド

ANALYZE コマンドは、データベース内のテーブルの内容に関する統計を収集し、結果を pg_stats システムビューに保存します。その後、クエリプランナーはこれらの統計を使用して、最も効率的なクエリの実行プランを決定します。

Aurora DSQL では、明示的なトランザクション内で ANALYZE コマンドを実行することはできません。ANALYZE はデータベーストランザクションのタイムアウト制限の対象ではありません。

手動介入の必要性を減らし、統計を一貫して最新の状態に保つために、Aurora DSQL はバックグラウンドプロセスとして ANALYZE を自動的に実行します。このバックグラウンドジョブは、テーブルで観測された変化率に基づいて自動的にトリガーされます。これは、前回の分析以降に挿入、更新、または削除された行 (タプル) の数にリンクされます。

ANALYZE はバックグラウンドで非同期的に実行され、そのアクティビティは次のクエリを使用してシステムビュー sys.jobs でモニタリングできます。

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

主な考慮事項

> **Note**
>
> ANALYZE ジョブは、Aurora DSQL の他の非同期ジョブと同様に請求されます。テーブルを変更すると、自動バックグラウンド統計収集ジョブが間接的にトリガーされ、関連するシステムレベルのアクティビティが原因で計測料金が発生する可能性があります。

バックグラウンド ANALYZE ジョブは自動的にトリガーされ、手動 ANALYZE で収集するタイプの統計と同じものを収集し、デフォルトでユーザーテーブルに適用します。システムテーブルとカタログテーブルは、この自動プロセスから除外されます。

Aurora DSQL クラスターの管理

Aurora DSQL には、ニーズに合った適切なデータベースインフラストラクチャを確立するのに役立ついくつかの設定オプションが用意されています。Aurora DSQL クラスターインフラストラクチャをセットアップするには、以下のセクションを確認してください。

トピック

- 単一リージョンクラスターの設定
- マルチリージョンクラスターの設定

このガイドで説明されている機能により、Aurora DSQL 環境の回復力、応答性が向上し、アプリケーションの成長および進化をサポートできるようになります。

単一リージョンクラスターの設定

クラスターの作成

クラスターを作成するには、create-cluster コマンドを使用します。

> **Note**
> クラスターの作成は非同期演算です。ステータスが `ACTIVE` に変わるまで GetCluster API を呼び出します。クラスターは、アクティブになった後に接続できます。

Example コマンド

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

> **Note**
> 作成時に削除保護を無効にするには、`--no-deletion-protection-enabled` フラグを含めます。

Example レスポンス

```
{
    "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"
    }
}
```

クラスターの説明

get-cluster コマンドを使用してクラスターに関する情報を取得します。

Example コマンド

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

Example レスポンス

```
{
    "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"
    }
}
```

クラスターの更新

update-cluster コマンドを使用して既存のクラスターを更新します。

> **Note**
>
> 更新は非同期オペレーションです。ステータスが ACTIVE に変わるまで GetCluster API を呼び出して、変更を確認します。

Example コマンド

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

Example レスポンス

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

クラスターの削除

delete-cluster コマンドを使用して既存のクラスターを削除します。

> **Note**
>
> 削除保護が無効になっている DB クラスターのみ削除できます。新しいクラスターを作成する場合は、デフォルトで削除保護が有効になっています。

Example コマンド

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

```
  --identifier your_cluster_id
```

Example レスポンス

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

クラスターの一覧表示

list-clusters コマンドを使用してクラスターを一覧表示します。

Example コマンド

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

Example レスポンス

```
{
    "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"
        }
    ]
}
```

マルチリージョンクラスターの設定

この章では、複数の AWS リージョンにまたがるクラスターを設定および管理する方法を説明します。

マルチリージョンクラスターへの接続

マルチリージョンピアリングクラスターは、ピアリングクラスターの AWS リージョンごとに 1 つずつ、2 つのリージョンエンドポイントを提供します。どちらのエンドポイントも、強力なデータ整合性を持つ同時読み取りおよび書き込みオペレーションをサポートする単一の論理データベースを提供します。ピア接続されたクラスターに加えて、マルチリージョンクラスターには、暗号化されたトランザクションログを限られた期間保存するウィットネスリージョンもあり、マルチリージョンの耐久性と可用性を向上させるために使用されます。マルチリージョンウィットネスリージョンにはエンドポイントがありません。

マルチリージョンクラスターの作成

マルチリージョンクラスターを作成するには、まずウィットネスリージョンを使用してクラスターを作成します。次に、このクラスターを、最初のクラスターと同じウィットネスリージョンを共有する 2 番目のクラスターとピア接続します。次の例は、米国東部 (バージニア北部) と米国東部 (オハイオ) で、米国西部 (オレゴン) を監視リージョンとしてクラスターを作成する方法を示しています。

ステップ 1: 米国東部 (バージニア北部) でクラスター 1 を作成する

マルチリージョンプロパティを使用して米国東部 (バージニア北部) AWS リージョンにクラスターを作成するには、以下のコマンドを使用します。

```
aws dsql create-cluster \
--region us-east-1 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example レスポンス:

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

> **Note**
>
> API オペレーションが成功すると、クラスターは PENDING_SETUP 状態になります。クラスターの作成は、ピアクラスターの ARN でクラスターを更新するまで PENDING_SETUP のままです。

ステップ 2: 米国東部 (オハイオ) でクラスター 2 を作成する

マルチリージョンプロパティを使用して米国東部 (オハイオ) AWS リージョンにクラスターを作成するには、以下のコマンドを使用します。

```
aws dsql create-cluster \
--region us-east-2 \
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example レスポンス:

```
{
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5",
    "status": "PENDING_SETUP",
    "creationTime": "2025-05-06T06:51:16.145000-07:00",
    "deletionProtectionEnabled": true,
    "multiRegionProperties": {
        "witnessRegion": "us-west-2",
        "clusters": [
            "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"
        ]
    }
}
```

API オペレーションが成功すると、クラスターは PENDING_SETUP 状態に移行します。クラスターの作成は、ピアリング用に別のクラスターの ARN で更新するまで PENDING_SETUP 状態のままです。

ステップ 3: 米国東部 (バージニア北部) と米国東部 (オハイオ) のピアクラスター

米国東部 (バージニア北部) クラスターを米国東部 (オハイオ) クラスターとピアリングするには、`update-cluster` コマンドを使用します。米国東部 (バージニア北部) クラスター名と、米国東部 (オハイオ) クラスターの ARN を持つ JSON 文字列を指定します。

```
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 レスポンス

```
{
    "identifier": "foo0bar1baz2quux3quuxquux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

ステップ 4: 米国東部 (オハイオ) と米国東部 (バージニア北部) のピアクラスター

米国東部 (オハイオ) クラスターを米国東部 (バージニア北部) クラスターとピアリングするには、`update-cluster` コマンドを使用します。米国東部 (オハイオ) クラスター名と、米国東部 (バージニア北部) クラスターの ARN を持つ JSON 文字列を指定します。

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 レスポンス

```
{
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5",
    "status": "UPDATING",
    "creationTime": "2025-05-06T06:51:16.145000-07:00"
}
```

> **Note**
> ピアリングが成功すると、両方のクラスターは「PENDING_SETUP」から「CREATING」に移行し、使用する準備が整うと、最終的に「ACTIVE」ステータスに移行します。

マルチリージョンクラスターのプロパティの表示

クラスターを説明すると、異なる AWS リージョンのクラスターのマルチリージョンプロパティを表示できます。

Example

```
aws dsql get-cluster \
--region us-east-1 \
--identifier 'foo0bar1baz2quux3quuxquux4'
```

Example レスポンス

```
{
    "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"
       ]
    }
}
```

作成中のピアクラスター

クラスターの作成時にピアリング情報を含めることで、ステップの数を減らすことができます。米国東部 (バージニア北部) で最初のクラスターを作成した後 (ステップ 1)、米国東部 (オハイオ) で 2 番

目のクラスターを作成し、最初のクラスターの ARN を含めることで同時にピアリングプロセスを開始できます。

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"]}'
```

これにより、ステップ 2 と 4 が組み合わせられますが、ピアリング関係を確立するには、ステップ 3 (最初のクラスターを 2 番目のクラスターの ARN で更新) を完了する必要があります。すべてのステップが完了すると、両方のクラスターは標準プロセスと同じ状態に移行します。[PENDING_SETUP] から [CREATING] に移行し、使用する準備ができたら最後に [ACTIVE] に移行します。

マルチリージョンクラスターの削除

マルチリージョンクラスターを削除するには、2 つのステップを完了する必要があります。

1. 各クラスターの削除保護をオフにします。
2. ピアリングされた各クラスターをそれぞれ対応する AWS リージョンで個別に削除する

米国東部 (バージニア北部) のクラスターを更新および削除する

1. `update-cluster` コマンドを使用して削除保護をオフにします。

   ```
   aws dsql update-cluster \
     --region us-east-1 \
     --identifier 'foo0bar1baz2quux3quuxquux4' \
     --no-deletion-protection-enabled
   ```

2. `delete-cluster` コマンドを使用してクラスターを削除します。

   ```
   aws dsql delete-cluster \
     --region us-east-1 \
     --identifier 'foo0bar1baz2quux3quuxquux4'
   ```

   このコマンドは、次のレスポンスを返します。

```
{
    "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**
>
> クラスターは PENDING_DELETE ステータスに移行します。削除は、米国東部 (オハイオ) でピアリングされたクラスターを削除するまで完了しません。

米国東部 (オハイオ) でクラスターを更新および削除する

1. `update-cluster` コマンドを使用して削除保護をオフにします。

```
aws dsql update-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quux4quuux' \
--no-deletion-protection-enabled
```

2. `delete-cluster` コマンドを使用してクラスターを削除します。

```
aws dsql delete-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5'
```

このコマンドは、次のレスポンスを返します。

```
{
    "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**
>
> クラスターは PENDING_DELETE ステータスに移行します。数秒後、システムは検証後に両方のピアリングされたクラスターを DELETING ステータスに自動的に移行します。

AWS CloudFormation を使用したマルチリージョンクラスターの設定

同じ AWS CloudFormation リソース AWS::DSQL::Cluster を使用して、単一リージョンおよびマルチリージョンの Aurora DSQL クラスターをデプロイおよび管理できます。

AWS::DSQL::Cluster リソースを使用して、クラスターを作成、変更、管理する方法については、「Amazon Aurora DSQL resource type reference」を参照してください。

初期クラスター設定の作成

まず、マルチリージョンクラスターを定義する AWS CloudFormation テンプレートを作成します。

```
---
Resources:
  MRCluster:
    Type: AWS::DSQL::Cluster
    Properties:
      DeletionProtectionEnabled: true
      MultiRegionProperties:
        WitnessRegion: us-west-2
```

次の AWS CLI コマンドを使用して、両方のリージョンにスタックを作成します。

```
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
```

クラスター識別子の検索

クラスターの物理リソース ID を取得します。

```
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"
]
```

クラスター設定の更新

AWS CloudFormation テンプレートを更新して、以下の両方のクラスター ARN を含めます。

```
---
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
```

更新した設定を両方のリージョンに適用します。

```
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 を使用したプログラミング

Aurora DSQL には、Aurora DSQL リソースをプログラムで管理するための以下のツールが用意されています。

AWS Command Line Interface (AWS CLI)

リソースを作成および管理するには、コマンドラインシェルの AWS CLI を使用します。AWS CLI は、Aurora DSQL などの AWS のサービスの API に直接アクセスできます。Aurora DSQL のコマンドの構文と例については、「AWS CLI コマンドリファレンス」の「dsql」を参照してください。

AWS Software Development Kits (SDK)

AWS では、多くの一般的なテクノロジーとプログラミング言語用の SDK を提供しています。これにより、その言語またはテクノロジーでアプリケーション内から、AWS のサービスを簡単に呼び出すことができます。これらの SDK の詳細については、「AWS でアプリケーションを開発および管理するためのツール」を参照してください。

Aurora DSQL API

この API は、Aurora DSQL の別のプログラミングインターフェイスです。この API を使用する場合は、すべての HTTPS リクエストを正しくフォーマットし、すべてのリクエストに有効なデジタル署名を追加する必要があります。詳細については、「API リファレンス」を参照してください。

AWS CloudFormation

AWS::DSQL::Cluster は、インフラストラクチャの一部として Aurora DSQL クラスターをコードとして作成および管理できる AWS CloudFormation リソースです。AWS CloudFormation は、AWS 環境全体をコードで定義し、インフラストラクチャを一貫して信頼性の高い方法でプロビジョニング、更新、レプリケートすることを容易にします。

AWS CloudFormation テンプレートで AWS::DSQL::Cluster リソースを使用すると、他のクラウドリソースと一緒に Aurora DSQL クラスターを宣言によりプロビジョニングできます。これにより、データインフラストラクチャがアプリケーションスタックの他の部分と一緒にデプロイおよび管理できるようになります。

Amazon Aurora DSQL AWS SDK とサンプルコード

AWS ソフトウェア開発キット (SDK) は、多くの一般的なプログラミング言語で使用できます。各 SDK には、デベロッパーが好みの言語でアプリケーションを簡単に構築できるようにする API、コード例、およびドキュメントが提供されています。

SDK ドキュメントとコード例の例	GitHub レーショナルマッピング (ORM)
AWS SDK for C++	AWS samples/aurora-dsql-samples/cpp
Go	GitHub samples/aurora-

SDK Hub サーバーレスの例 オブジェクトリレーショナルマッピング(ORM)	dsql-samples/go
JDBC JDBC JDBC	aws-samples/aurora-dsql-samples/java

SDK Hub	
データベースの例	リレーショナルマッピング (ORM)
JavaScript	aws-samples/aurora-dsql-samples/javascript

SDK Hub	
言語	
サービス	
例の説明	
リレーショナルマッピング (ORM)	
.NET	aws-samples/aurora-dsql-samples/dotnet

SDK Hub	
ドライバーの例	
レーショナルマッピング (ORM)	
Boto3	psycopg2
Boto3	Django
Boto3	Sequelize
Boto3	Prisma
Boto3	Drizzle
Boto3	Ecto
Boto3	Diesel
Boto3	Sqlx
Boto3	Gorm
Boto3	Sqlc
Boto3	Bun
Boto3	Ruby on Rails
Boto3	Ruby pg
Boto3	Ruby Sequel
Boto3	Ruby ActiveRecord
Boto3	Ruby Ecto
Boto3	Ruby Diesel
Boto3	Ruby Sqlx
Boto3	Ruby Gorm
Boto3	Ruby Sqlc
Boto3	Ruby Bun

SDK Hub	
ドライバーを使用した例	
レーショナルマッピング (ORM)	
Rust	aws-samples/aurora-dsql-samples/rust
TypeScript	aws-samples/aurora-dsql-samples/typescript
Type Size	
type-orm	

AWS CLI を使用した Aurora DSQL

AWS CLI を使用してクラスターを管理する方法については、以下のセクションを参照してください。

CreateCluster

クラスターを作成するには、`create-cluster` コマンドを使用します。

> **Note**
> クラスターの作成は非同期的に行われます。ステータスが `ACTIVE` になるまで `GetCluster` API を呼び出します。クラスターは、`ACTIVE` になったら接続できます。

サンプルコマンド

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

> **Note**
> 作成時に削除保護を無効にする場合は、`--no-deletion-protection-enabled` フラグを含めます。

レスポンス例

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

GetCluster

クラスターを記述するには、`get-cluster` コマンドを使用します。

サンプルコマンド

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

レスポンス例

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

UpdateCluster

既存のクラスターを更新するには、`update-cluster` コマンドを使用します。

> **Note**
>
> 更新は非同期的に行われます。ステータスが `ACTIVE` になるまで `GetCluster` API を呼び出し、変更が表示されます。

サンプルコマンド

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

```
  --identifier your_cluster_id
```

レスポンス例

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

DeleteCluster

既存のクラスターを削除するには、`delete-cluster` コマンドを使用します。

> **Note**
> 削除保護が無効になっているクラスターのみを削除できます。新しいクラスターを作成する場合は、デフォルトで削除保護が有効になっています。

サンプルコマンド

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

レスポンス例

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

ListClusters

クラスターの a を取得するには、`list-clusters` コマンドを使用します。

サンプルコマンド

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

レスポンス例

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

マルチリージョンクラスターの GetCluster

マルチリージョンクラスターに関する情報を取得するには、`get-cluster` コマンドを使用します。マルチリージョンクラスターの場合、レスポンスにはリンクされたクラスター ARN が含まれます。

サンプルコマンド

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

レスポンス例

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2025-05-22T13:56:18.716000-07:00",
    "deletionProtectionEnabled": true,
```

```
    "multiRegionProperties": {
        "witnessRegion": "us-west-2",
        "clusters": [
            "arn:aws:dsql:us-east-1:842685632318:cluster/fuabuc7d3szkr37uqd5znkjynu"
        ]
    },
    "tags": {},
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    }
}
```

Aurora DSQL クラスターの作成、読み取り、更新、削除

作成、読み取り、更新、削除 (CRUD) の例は、単一リージョンとマルチリージョンの両方のデプロイに関して提供されています。これらのキー管理タスクを示す各プログラミング言語の専用 `cluster_management` セクションが含まれています。

単一リージョンデプロイは、特定の地理的エリアのユーザーにサービスを提供するアプリケーションに最適です。これにより、管理が簡素化され、レイテンシーが低くなります。マルチリージョンデプロイは、データベースを複数の AWS リージョンに分散することで、より高い可用性とディザスタリカバリ機能を実現するのに役立ちます。

可用性、パフォーマンス、地理的分散に関するアプリケーションの要件に合ったデプロイタイプを選択します。

トピック

- クラスターを作成する
- クラスターを取得する
- クラスターを更新する
- クラスターを削除

クラスターを作成する

Aurora DSQL で単一リージョンクラスターとマルチリージョンクラスターを作成する方法については、次の情報を参照してください。

Python

単一の AWS リージョンにクラスターを作成するには、次の例を使用します。

```
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()
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

```
import boto3


def create_multi_region_clusters(region_1, region_2, witness_region):
    try:
        client_1 = boto3.client("dsql", region_name=region_1)
        client_2 = boto3.client("dsql", region_name=region_2)

        # We can only set the witness region for the first cluster
        cluster_1 = client_1.create_cluster(
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region},
            tags={"Name": "Python multi region cluster"}
        )
        print(f"Created {cluster_1["arn"]}")

        # For the second cluster we can set witness region and designate cluster_1
 as a peer
        cluster_2 = client_2.create_cluster(
            deletionProtectionEnabled=True,
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_1["arn"]]},
            tags={"Name": "Python multi region cluster"}
        )

        print(f"Created {cluster_2["arn"]}")
        # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
        client_1.update_cluster(
            identifier=cluster_1["identifier"],
            multiRegionProperties={"witnessRegion": witness_region, "clusters":
 [cluster_2["arn"]]}
        )
        print(f"Added {cluster_2["arn"]} as a peer of {cluster_1["arn"]}")

        # Now that multiRegionProperties is fully defined for both clusters
        # they'll begin the transition to ACTIVE
        print(f"Waiting for {cluster_1["arn"]} to become ACTIVE")
        client_1.get_waiter("cluster_active").wait(
            identifier=cluster_1["identifier"],
            WaiterConfig={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        print(f"Waiting for {cluster_2["arn"]} to become ACTIVE")
        client_2.get_waiter("cluster_active").wait(
            identifier=cluster_2["identifier"],
            WaiterConfig={
```

C++

次の例では、単一の AWS リージョンにクラスターを作成できます。

```
#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;
}
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

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

using namespace Aws;
```

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

/**
 * Creates multi-region clusters in Amazon Aurora DSQL
 */
std::pair<CreateClusterResult, CreateClusterResult> CreateMultiRegionClusters(
    const Aws::String& region1,
    const Aws::String& region2,
    const Aws::String& witnessRegion) {

    // Create clients for each region
    DSQL::DSQLClientConfiguration clientConfig1;
    clientConfig1.region = region1;
    DSQL::DSQLClient client1(clientConfig1);

    DSQL::DSQLClientConfiguration clientConfig2;
    clientConfig2.region = region2;
    DSQL::DSQLClient client2(clientConfig2);

    // We can only set the witness region for the first cluster
    std::cout << "Creating cluster in " << region1 << std::endl;

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region
    MultiRegionProperties multiRegionProps1;
    multiRegionProps1.SetWitnessRegion(witnessRegion);
    createClusterRequest1.SetMultiRegionProperties(multiRegionProps1);

    // Add tags
    Aws::Map<Aws::String, Aws::String> tags;
    tags["Name"] = "cpp multi region cluster 1";
    createClusterRequest1.SetTags(tags);
    createClusterRequest1.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome1 = client1.CreateCluster(createClusterRequest1);
    if (!createOutcome1.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region1 << ": "
                  << createOutcome1.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }
```

```
    auto cluster1 = createOutcome1.GetResult();
    std::cout << "Created " << cluster1.GetArn() << std::endl;

    // For the second cluster we can set witness region and designate cluster1 as a
peer
    std::cout << "Creating cluster in " << region2 << std::endl;

    CreateClusterRequest createClusterRequest2;
    createClusterRequest2.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region and cluster1 as peer
    MultiRegionProperties multiRegionProps2;
    multiRegionProps2.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> clusters;
    clusters.push_back(cluster1.GetArn());
    multiRegionProps2.SetClusters(clusters);

    tags["Name"] = "cpp multi region cluster 2";
    createClusterRequest2.SetMultiRegionProperties(multiRegionProps2);
    createClusterRequest2.SetTags(tags);
    createClusterRequest2.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto createOutcome2 = client2.CreateCluster(createClusterRequest2);
    if (!createOutcome2.IsSuccess()) {
        std::cerr << "Failed to create cluster in " << region2 << ": "
                  << createOutcome2.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to create multi-region clusters");
    }

    auto cluster2 = createOutcome2.GetResult();
    std::cout << "Created " << cluster2.GetArn() << std::endl;

    // Now that we know the cluster2 arn we can set it as a peer of cluster1
    UpdateClusterRequest updateClusterRequest;
    updateClusterRequest.SetIdentifier(cluster1.GetIdentifier());

    MultiRegionProperties updatedProps;
    updatedProps.SetWitnessRegion(witnessRegion);

    Aws::Vector<Aws::String> updatedClusters;
    updatedClusters.push_back(cluster2.GetArn());
    updatedProps.SetClusters(updatedClusters);
```

```
    updateClusterRequest.SetMultiRegionProperties(updatedProps);
    updateClusterRequest.SetClientToken(Aws::Utils::UUID::RandomUUID());

    auto updateOutcome = client1.UpdateCluster(updateClusterRequest);
    if (!updateOutcome.IsSuccess()) {
        std::cerr << "Failed to update cluster in " << region1 << ": "
                  << updateOutcome.GetError().GetMessage() << std::endl;
        throw std::runtime_error("Failed to update multi-region clusters");
    }

    std::cout << "Added " << cluster2.GetArn() << " as a peer of " <<
 cluster1.GetArn() << std::endl;

    return std::make_pair(cluster1, cluster2);
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    {
        try {
            // Define regions for the multi-region setup
            Aws::String region1 = "us-east-1";
            Aws::String region2 = "us-east-2";
            Aws::String witnessRegion = "us-west-2";

            auto [cluster1, cluster2] = CreateMultiRegionClusters(region1, region2,
 witnessRegion);

            std::cout << "Created multi region clusters:" << std::endl;
            std::cout << "Cluster 1 ARN: " << cluster1.GetArn() << std::endl;
            std::cout << "Cluster 2 ARN: " << cluster2.GetArn() << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

単一の AWS リージョンにクラスターを作成するには、次の例を使用します。

```
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();
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

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

async function createMultiRegionCluster(region1, region2, witnessRegion) {

    const client1 = new DSQLClient({ region: region1 });
    const client2 = new DSQLClient({ region: region2 });

    try {
        // We can only set the witness region for the first cluster
        console.log(`Creating cluster in ${region1}`);
        const createClusterCommand1 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 1"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion
            }
        });

        const response1 = await client1.send(createClusterCommand1);
        console.log(`Created ${response1.arn}`);

        // For the second cluster we can set witness region and designate the first
 cluster as a peer
        console.log(`Creating cluster in ${region2}`);
        const createClusterCommand2 = new CreateClusterCommand({
            deletionProtectionEnabled: true,
            tags: {
                Name: "javascript multi region cluster 2"
            },
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response1.arn]
            }
        });

        const response2 = await client2.send(createClusterCommand2);
        console.log(`Created ${response2.arn}`);
```

```
        // Now that we know the second cluster arn we can set it as a peer of the
first cluster
        const updateClusterCommand1 = new UpdateClusterCommand(
            {
                identifier: response1.identifier,
                multiRegionProperties: {
                    witnessRegion: witnessRegion,
                    clusters: [response2.arn]
                }
            }
        );

        await client1.send(updateClusterCommand1);
        console.log(`Added ${response2.arn} as a peer of ${response1.arn}`);

        // Now that multiRegionProperties is fully defined for both clusters
        // they'll begin the transition to ACTIVE
        console.log(`Waiting for cluster 1 ${response1.identifier} to become
ACTIVE`);

        await waitUntilClusterActive(
            {
                client: client1,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response1.identifier
            }
        );
        console.log(`Cluster 1 is now active`);

        console.log(`Waiting for cluster 2 ${response2.identifier} to become
ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client2,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response2.identifier
            }
        );
        console.log(`Cluster 2 is now active`);
```

```
        console.log("The multi region clusters are now active");
        return;
    } catch (error) {
        console.error("Failed to create cluster: ", error.message);
        throw error;
    }
}

async function main() {
    const region1 = "us-east-1";
    const region2 = "us-east-2";
    const witnessRegion = "us-west-2";

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();
```

Java

次の例を使用して、単一の AWS リージョンにクラスターを作成します。

```
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);
        }
    }
}
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

```
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

次の例を使用して、単一の AWS リージョンにクラスターを作成します。

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

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

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

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

    Client::from_conf(config)
}

/// Create a cluster without delete protection and a name
pub async fn create_cluster(region: &'static str) -> GetClusterOutput {
    let client = dsql_client(region).await;
    let tags = HashMap::from([(
        String::from("Name"),
        String::from("rust single region cluster"),
    )]);

    let create_cluster_output = client
        .create_cluster()
        .set_tags(Some(tags))
        .deletion_protection_enabled(true)
```

```
        .send()
        .await
        .unwrap();
    println!("Created {}", create_cluster_output.arn);

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

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

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

```
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

次の例を使用して、単一の AWS リージョンにクラスターを作成します。

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

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

```
  )
  puts "Created #{cluster.arn}"

  # The DSQL SDK offers built-in waiters to poll for a cluster's
  # transition to ACTIVE.
  puts "Waiting for cluster to become ACTIVE"
  client.wait_until(:cluster_active, identifier: cluster.identifier) do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end
rescue Aws::Errors::ServiceError => e
  abort "Unable to create cluster in #{region}: #{e.message}"
end

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

main if $PROGRAM_NAME == __FILE__
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

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

def create_multi_region_clusters(region_1, region_2, witness_region)
  client_1 = Aws::DSQL::Client.new(region: region_1)
  client_2 = Aws::DSQL::Client.new(region: region_2)

  # We can only set the witness region for the first cluster
  puts "Creating cluster in #{region_1}"
  cluster_1 = client_1.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region
    },
    tags: {
```

```
      Name: "ruby multi region cluster"
    }
  )
  puts "Created #{cluster_1.arn}"

  # For the second cluster we can set witness region and designate cluster_1 as a
peer
  puts "Creating cluster in #{region_2}"
  cluster_2 = client_2.create_cluster(
    deletion_protection_enabled: true,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_1.arn ]
    },
    tags: {
      Name: "ruby multi region cluster"
    }
  )
  puts "Created #{cluster_2.arn}"

  # Now that we know the cluster_2 arn we can set it as a peer of cluster_1
  client_1.update_cluster(
    identifier: cluster_1.identifier,
    multi_region_properties: {
      witness_region: witness_region,
      clusters: [ cluster_2.arn ]
    }
  )
  puts "Added #{cluster_2.arn} as a peer of #{cluster_1.arn}"

  # Now that multi_region_properties is fully defined for both clusters
  # they'll begin the transition to ACTIVE
  puts "Waiting for #{cluster_1.arn} to become ACTIVE"
  cluster_1 = client_1.wait_until(:cluster_active, identifier: cluster_1.identifier)
do |w|
    # Wait for 5 minutes
    w.max_attempts = 30
    w.delay = 10
  end

  puts "Waiting for #{cluster_2.arn} to become ACTIVE"
  cluster_2 = client_2.wait_until(:cluster_active, identifier: cluster_2.identifier)
do |w|
    w.max_attempts = 30
```

```
    w.delay = 10
  end

  [ cluster_1, cluster_2 ]
rescue Aws::Errors::ServiceError => e
  abort "Failed to create multi-region clusters: #{e.message}"
end

def main
  region_1 = "us-east-1"
  region_2 = "us-east-2"
  witness_region = "us-west-2"

  cluster_1, cluster_2 = create_multi_region_clusters(region_1, region_2,
 witness_region)

  puts "Created multi region clusters:"
  pp cluster_1
  pp cluster_2
end

main if $PROGRAM_NAME == __FILE__
```

.NET

次の例を使用して、単一の AWS リージョンにクラスターを作成します。

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

namespace DSQLExamples.examples
{
    public class CreateSingleRegionCluster
    {
        /// <summary>
        /// Create a client. We will use this later for performing operations on the
 cluster.
        /// </summary>
```

```
        private static async Task<AmazonDSQLClient> CreateDSQLClient(RegionEndpoint
region)
        {
            var awsCredentials = await
DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

        /// <summary>
        /// Create a cluster without delete protection and a name.
        /// </summary>
        public static async Task<CreateClusterResponse> Create(RegionEndpoint
region)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp single region cluster" }
                };

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

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

                return response;
            }
        }

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

            await Create(region);
```

```
        }
    }
}
```

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

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

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

        /// <summary>
        /// Create multi-region clusters with a witness region.
        /// </summary>
        public static async Task<(CreateClusterResponse, CreateClusterResponse)>
 Create(
            RegionEndpoint region1,
            RegionEndpoint region2,
```

```
            RegionEndpoint witnessRegion)
        {
            using (var client1 = await CreateDSQLClient(region1))
            using (var client2 = await CreateDSQLClient(region2))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp multi region cluster" }
                };

                // We can only set the witness region for the first cluster
                var createClusterRequest1 = new CreateClusterRequest
                {
                    DeletionProtectionEnabled = true,
                    Tags = tags,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName
                    }
                };

                var cluster1 = await
client1.CreateClusterAsync(createClusterRequest1);
                var cluster1Arn = cluster1.Arn;
                Console.WriteLine($"Initiated creation of {cluster1Arn}");

                // For the second cluster we can set witness region and designate
cluster1 as a peer
                var createClusterRequest2 = new CreateClusterRequest
                {
                    DeletionProtectionEnabled = true,
                    Tags = tags,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName,
                        Clusters = new List<string> { cluster1.Arn }
                    }
                };

                var cluster2 = await
client2.CreateClusterAsync(createClusterRequest2);
                var cluster2Arn = cluster2.Arn;
                Console.WriteLine($"Initiated creation of {cluster2Arn}");
```

```
                // Now that we know the cluster2 arn we can set it as a peer of
 cluster1
                var updateClusterRequest = new UpdateClusterRequest
                {
                    Identifier = cluster1.Identifier,
                    MultiRegionProperties = new MultiRegionProperties
                    {
                        WitnessRegion = witnessRegion.SystemName,
                        Clusters = new List<string> { cluster2.Arn }
                    }
                };

                await client1.UpdateClusterAsync(updateClusterRequest);
                Console.WriteLine($"Added {cluster2Arn} as a peer of
 {cluster1Arn}");

                return (cluster1, cluster2);
            }
        }

        private static async Task Main()
        {
            var region1 = RegionEndpoint.USEast1;
            var region2 = RegionEndpoint.USEast2;
            var witnessRegion = RegionEndpoint.USWest2;

            var (cluster1, cluster2) = await Create(region1, region2,
 witnessRegion);

            Console.WriteLine("Created multi region clusters:");
            Console.WriteLine($"Cluster 1: {cluster1.Arn}");
            Console.WriteLine($"Cluster 2: {cluster2.Arn}");
        }
    }
}
```

Golang

次の例を使用して、単一の AWS リージョンにクラスターを作成します。

```
package main

import (
```

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

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

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

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

 // Create a DSQL client
 client := dsql.NewFromConfig(cfg)

 deleteProtect := true

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

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

 if err != nil {
  return fmt.Errorf("error creating cluster: %w", err)
 }

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

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

```
 id := clusterProperties.Identifier

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

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

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

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

 region := "us-east-1"

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

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

マルチリージョンクラスターを作成するには、次の例を使用します。マルチリージョンクラスターの作成には時間がかかる場合があります。

```
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)
 }

}
```

クラスターを取得する

Aurora DSQL でクラスターの情報を返す方法についての詳細は、次の情報を参照してください。

Python

単一またはマルチリージョンクラスターに関する情報を取得するには、次の例を使用します。

```
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++

次の例を使用して、単一またはマルチリージョンクラスターに関する情報を取得します。

```
#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

単一またはマルチリージョンクラスターに関する情報を取得するには、次の例を使用します。

```
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

次の例では、単一リージョンクラスターまたはマルチリージョンクラスターに関する情報を取得できます。

```
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

次の例では、単一リージョンクラスターまたはマルチリージョンクラスターに関する情報を取得できます。

```
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

次の例では、単一リージョンクラスターまたはマルチリージョンクラスターに関する情報を取得できます。

```
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

次の例では、単一リージョンクラスターまたはマルチリージョンクラスターに関する情報を取得できます。

```
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

次の例では、単一リージョンクラスターまたはマルチリージョンクラスターに関する情報を取得できます。

```
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)
 }
```

```
}
```

クラスターを更新する

Aurora DSQL でクラスターを更新する方法については、次の情報を参照してください。クラスターの更新には 1 ~ 2 分かかる場合があります。しばらく待ってから get cluster を実行してクラスターのステータスを取得することをお勧めします。

Python

単一またはマルチリージョンクラスターを更新するには、次の例を使用します。

```
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++

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
#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

単一またはマルチリージョンクラスターを更新するには、次の例を使用します。

```
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

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
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

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
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

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
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

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
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

次の例を使用して、単一またはマルチリージョンクラスターを更新します。

```
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)
 }
}
```

クラスターを削除

Aurora DSQL でクラスターを削除する方法については、次の情報を参照してください。

Python

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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()
```

マルチリージョンクラスターを削除するには、次の例を使用します。

```
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++

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
#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;
}
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
#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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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();
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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);
        }
    }
}
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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(())
}
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
equire "aws-sdk-dsql"

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

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

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

main if $PROGRAM_NAME == __FILE__
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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);
        }
    }
}
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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

単一の AWS リージョンでクラスターを削除するには、次の例を使用します。

```
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)
 }

}
```

マルチリージョンクラスターを削除するには、次の例を使用します。マルチリージョンクラスターの削除には時間がかかる場合があります。

```
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)
 }
}
```

チュートリアル

GitHub の以下のチュートリアルとサンプルコードは、Aurora DSQL で一般的なタスクを実行するのに役立ちます。

- Aurora DSQL で Benchbase を使用する – Aurora DSQL と連携することが検証された Benchbase オープンソースベンチマークユーティリティのブランチ。
- Aurora DSQL ローダー – このオープンソースの Python スクリプトを使用すると、テスト用のテーブルの入力や Aurora DSQL へのデータ転送など、ユースケースに合わせてデータを Aurora DSQL に容易にロードできます。
- Aurora DSQL サンプル – GitHub の `aws-samples/aurora-dsql-samples` リポジトリには、AWS SDK、オブジェクトリレーショナルマッパー (ORM)、ウェブフレームワークを使用して、さまざまなプログラミング言語で Aurora DSQL を接続して使用する方法のコード例が含まれています。この例では、クライアントのインストール、認証の処理、CRUD オペレーションの実行などの一般的なタスクを実行する方法を示します。

Amazon Aurora DSQL での AWS Lambda の使用

次のチュートリアルでは、Aurora DSQL で Lambda を使用する方法について説明します。

前提条件

- Lambda 関数を作成する承認。詳細については「Getting started with Lambda」を参照してください。
- Lambda によって作成された IAM ポリシーを作成または変更する承認。アクセス許可 `iam:CreatePolicy` と `iam:AttachRolePolicy` が必要です。詳細については、「Actions, resources, and condition keys for IAM」を参照してください。
- npm v8.5.3 以降がインストールされている必要があります。
- zip v3.0 以降がインストールされている必要があります。

AWS Lambda で新しい関数を作成する

1. AWS Management Console にサインインして AWS Lambda コンソール (https://console.aws.amazon.com/lambda/) を開きます。
2. [Create function] (関数の作成) を選択します。
3. `dsql-sample` などの名前を指定します。
4. Lambda が基本的な Lambda アクセス許可を持つ新しいロールを作成するようにデフォルト設定を編集しないでください。
5. [関数の作成] を選択してください。

Lambda 実行ロールにクラスターへの接続を許可する

1. Lambda 関数で、[設定]、[アクセス許可] の順に選択します。
2. [ロール名] を選択して、IAM コンソールで実行ロールを開きます。
3. [アクセス許可の追加]、[インラインポリシーの作成] の順に選択し、JSON エディタを使用します。
4. アクションに以下のアクションを貼り付け、IAM ID が管理データベースロールを使用して接続できるように権限を付与します。

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

> **Note**
>
> 開始するための前提条件のステップを最小限に抑えるために、管理者ロールを使用しています。本番稼働用アプリケーションには管理者データベースロールを使用しないでください。データベースへのアクセス許可が最も少ない認可でカスタムデータベースロールを作成する方法については、「データベースロールと IAM 認証の使用」を参照してください。

5. [リソース] で、クラスターの Amazon リソースネーム (ARN) を追加します。ワイルドカードを使用することもできます。

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

6. [次へ] を選択します。
7. ポリシーの名前を入力します (例: `dsql-sample-dbconnect`)。

8. [Create policy] (ポリシーの作成) を選択します。

Lambda にアップロードするためのパッケージを作成します。

1. `myfunction` という名前のフォルダを作成します。
2. そのフォルダ内に、以下の内容を含む `package.json` という名前の新しいファイルを作成します。

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

3. フォルダ内のディレクトリに以下の内容で `index.mjs` というファイルを作成します。

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


async function dsql_sample(clusterEndpoint, region) {
  let client;
  try {
    // The token expiration time is optional, and the default value 900 seconds
    const signer = new DsqlSigner({
      hostname: clusterEndpoint,
      region,
    });
    const token = await signer.getDbConnectAdminAuthToken();
    // <https://node-postgres.com/apis/client>
    // By default `rejectUnauthorized` is true in TLS options
    // <https://nodejs.org/api/tls.html#tls_tls_connect_options_callback>
    // The config does not offer any specific parameter to set sslmode to verify-
full
    // Settings are controlled either via connection string or by setting
    // rejectUnauthorized to false in ssl options
    client = new Client({
      host: clusterEndpoint,
```

```
      user: "admin",
      password: token,
      database: "postgres",
      port: 5432,
      // <https://node-postgres.com/announcements> for version 8.0
      ssl: true,
      rejectUnauthorized: false
    });

    // Connect
    await client.connect();

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

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

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

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

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

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

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

4. 次のコマンドを使用してパッケージを作成します。

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

コードパッケージをアップロードし、Lambda 関数をテストする

1. Lambda 関数の [Code] タブで、[Upload from]、[zip ファイル] の順に選択します。
2. 作成した `pkg.zip` をアップロードします。詳細については、「.zip ファイルアーカイブで Node.js Lambda 関数をデプロイする」を参照してください。
3. Lambda 関数の [テスト] タブで、次の JSON ペイロードを貼り付け、クラスター ID を使用するように変更します。
4. Lambda 関数の [テスト] タブで、次のイベント JSON を変更してクラスターのエンドポイントを指定します。

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

5. イベント名を入力します (例: `dsql-sample-test`)。[保存] を選択します。
6. [Test] を選択します。
7. [詳細] を選択して、実行レスポンスとログ出力を展開します。
8. 成功した場合、Lambda 関数の実行レスポンスは 200 のステータスコードを返します。

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

データベースがエラーを返した場合、またはデータベースへの接続が失敗した場合、Lambda 関数の実行レスポンスは 500 ステータスコードを返します。

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

Amazon Aurora DSQL のバックアップと復元

Amazon Aurora DSQL は、フルマネージドのデータ保護サービスである AWS Backup との統合を通じて、規制コンプライアンスとビジネス継続性の要件を満たすのに役立ちます。このサービスにより、AWS サービス間、クラウド内、およびオンプレミスでのバックアップの一元化と自動化が容易になります。このサービスは、単一リージョンとマルチリージョンの両方の Aurora DSQL クラスターのバックアップの作成、管理、復元を合理化します。

主な機能は以下のとおりです。

- AWS Management Console、SDK、または AWS CLI による一元的なバックアップ管理
- クラスターのフルバックアップ
- バックアップスケジュールと保持ポリシーの自動化
- クロスリージョンおよびクロスアカウント機能
- 保存するすべてのバックアップの WORM (write-once, read-many) 構成

AWS Backup Vault Lock の機能および Aurora DSQL で使用できる AWS Backup 機能の広範なリストの詳細については、「AWS Backup デベロッパーガイド」の「Vault lock benefits」および「AWS Backup feature availability」を参照してください。

AWS Backup の開始方法

AWS Backup は Aurora DSQL クラスターの完全なコピーを作成します。Aurora DSQL に対応した AWS Backup の使用を開始するには、「Getting started with AWS Backup」のステップに従います。

1. 即時保護のためにオンデマンドバックアップを作成します。
2. スケジュールされた自動バックアップのバックアッププランを確立します。
3. 保持期間とクロスリージョンコピーを設定します。
4. バックアップアクティビティのモニタリングと通知を設定します。

バックアップの復元

Aurora DSQL クラスターを復元する場合、AWS Backup は常にソースデータを保存するための新しいクラスターを作成します。Aurora DSQL シングルリージョンクラスターを復元するには、https://

console.aws.amazon.com/backup または CLI を使用して、復元する復旧ポイント (バックアップ) を選択します。バックアップで作成する新しいクラスターの設定を構成します。

Aurora DSQL マルチリージョンクラスターの復元は、AWS CLI を通じてのみサポートされます。Aurora DSQL マルチリージョンクラスターを復元するには、AWS Backup と Aurora DSQL CLI の両方を使用する必要があります。

Aurora DSQL マルチリージョンクラスターを復元する方法。

1. マルチリージョンクラスターの復旧ポイントを選択します。
2. 復旧ポイントを、マルチリージョンクラスターをサポートする別の AWS リージョンにコピーします。

> **Note**
> マルチリージョンクラスターをサポートしていないリージョンでは、復元オペレーションが失敗します。

3. AWS Backup CLI を使用して、クラスターごとに復元ジョブを開始します。
4. マルチリージョンクラスターの設定 ドキュメントを使用して、新しく作成した Aurora DSQL クラスターをピアリングします。

これらの手順の詳細については、「Amazon Aurora DSQL restore」ドキュメントを参照してください。

マルチリージョン Aurora DSQL クラスターに復元するには、単一の AWS リージョンで取得したバックアップを使用できます。ただし、復元プロセスを開始する前に、まずマルチリージョンクラスターをサポートする別の AWS リージョンにバックアップをコピーする必要があります。このステップにより、復元オペレーションが正常に完了します。堅牢なディザスタリカバリオプションを有効にし、コンプライアンス要件を満たすために、米国東部 (バージニア北部)、米国東部 (オハイオ)、または米国西部 (オレゴン) などの主要な AWS リージョンにバックアップコピーを作成することをお勧めします。

モニタリングとコンプライアンス

AWS Backup は、次のリソースを使用して、バックアップおよび復元オペレーションを包括的に可視化します。

- バックアップジョブと復元ジョブを追跡するための一元化されたダッシュボード
- CloudWatch および CloudTrail との統合
- コンプライアンスレポートと監査用の AWS Backup Audit Manager

Aurora DSQL の使用中にユーザー、ロール、または AWS のサービスによって実行されたアクションのログ記録の詳細については、「AWS CloudTrail を使用した Aurora DSQL オペレーションのログ記録」を参照してください。

追加リソース

AWS Backup 機能と、それを Aurora DSQL と組み合わせて使用する方法の詳細については、次のリソースを参照してください。

- Managed policies for AWS Backup
- Amazon Aurora DSQL restore
- Supported services by AWS リージョン
- Encryption for backups in AWS Backup

Aurora DSQL に対応した AWS Backup を使用すると、管理オーバーヘッドを最小限に抑えながら重要なデータベースリソースを保護する、堅牢でコンプライアンスに準拠した自動バックアップ戦略を実装できます。1 つのクラスターを管理する場合でも、複雑なマルチリージョンデプロイを管理する場合でも、AWS Backup はデータの安全性と回復性を確保するために必要なツールを提供します。

Aurora DSQL のモニタリングとログ記録

モニタリングとログ記録は、Amazon Aurora DSQL リソースの信頼性、可用性、パフォーマンスを維持する上で重要な部分です。マルチポイント障害を簡単にデバッグできるように、Aurora DSQL リソースのすべての部分をモニタリングし、ログ記録データを収集する必要があります。

- Amazon CloudWatch は、AWS のリソースおよび AWS で実行しているアプリケーションをリアルタイムでモニタリングします。メトリクスの収集と追跡、カスタマイズしたダッシュボードの作成、および指定したメトリクスが指定したしきい値に達したときに通知またはアクションを実行するアラームの設定を行うことができます。例えば、CloudWatch で Amazon EC2 インスタンスの CPU 使用率などのメトリクスを追跡し、必要に応じて新しいインスタンスを自動的に起動できます。詳細については、「Amazon CloudWatch ユーザーガイド」を参照してください。
- AWS CloudTrail は、AWS アカウントにより、またはそのアカウントに代わって行われた API コールや関連イベントを取得し、指定した Amazon S3 バケットにログファイルを配信します。AWS を呼び出したユーザーとアカウント、呼び出し元の IP アドレス、および呼び出しの発生日時を特定できます。詳細については、AWS CloudTrailユーザーガイドを参照してください。

Aurora DSQL クラスターのステータスの表示

Aurora DSQL クラスターのステータスは、クラスターのヘルスと接続に関する重要な情報を提供します。AWS Management Console、AWS CLI、または Aurora DSQL API を使用して、クラスターとクラスターインスタンスのステータスを表示できます。

Aurora DSQL クラスターのステータスと定義

次の表に、Aurora DSQL クラスターで利用可能な各ステータスと、各ステータスの意味を示します。

ステータス	説明
[作成中]	Aurora DSQL はクラスターのリソースを作成または設定しようとしています。クラスターがこの状態にある間は、接続の試行は失敗します。
アクティブ	クラスターは稼働しており、すぐに使用できます。
アイドル状態	Aurora DSQL がそのために設定したリソースを再利用するのに十分な時間アイドル状態になると、クラスターはアイドル状態になります。アイドル状態のク

ステータス	説明
	ラスターに接続すると、Aurora DSQL はクラスターを [アクティブ] 状態に戻します。
非アクティブ	長期にわたってクラスターにアクティビティがない場合、クラスターは非アクティブになります。非アクティブのクラスターに接続しようとすると、Aurora DSQL はクラスターを自動的に [アクティブ] 状態に戻します。
[更新中]	クラスター設定を変更すると、クラスターは [更新中] ステータスに移行します。
削除	クラスターを削除するリクエストを送信すると、クラスターは [削除] ステータスに移行します。
[Deleted] (削除済み)	クラスターは正常に削除されました。
失敗	エラーが発生したため、Aurora DSQL はクラスターを作成できませんでした。
セットアップ保留中	マルチリージョンクラスターのみ。ウィットネスリージョンがある最初のリージョンにマルチリージョンクラスターを作成すると、マルチリージョンクラスターは [セットアップ保留中] ステータスになります。クラスターの作成は、セカンダリリージョンに別のクラスターを作成し、2 つのクラスターをピアリングするまで一時停止します。
削除を保留中	マルチリージョンクラスターのみ。クラスターを削除すると、マルチリージョンクラスターは [削除を保留中] ステータスになります。最後のピアクラスターを削除すると、クラスターは [削除中] 状態に移行します。

Aurora DSQL クラスターのステータスの表示

クラスターのステータスを表示するには、AWS Management Console、AWS CLI、または Aurora DSQL API を使用します。

コンソール

AWS Management Consoleでクラスターのステータスを表示するには、次の手順に従います。

コンソールでクラスターのステータスを表示するには

1. https://console.aws.amazon.com/dsql で Aurora DSQL コンソールを開きます。
2. ナビゲーションペインで [クラスター] を選択します。
3. ダッシュボードで各クラスターのステータスを表示します。

AWS CLI

次の AWS CLI コマンドを実行して、単一のクラスターのステータスを確認します。

```
aws dsql get-cluster --identifier cluster-id --query status --output text
```

すべてのクラスターのステータスを一覧表示するには、次のコマンドを実行します。

```
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
```

このサンプル出力は、2 つのアクティブなクラスターと、削除中の 1 つのクラスターを示しています。

```
aaabbb2bkx555xa7p42qd5cdef    ACTIVE
abcde123efghi77t35abcdefgh    ACTIVE
12abc6lqasc5bbbbbbbbbbbbbb    DELETING
```

Amazon CloudWatch を使用した Aurora DSQL のモニタリング

raw データを収集し、読み取り可能なほぼリアルタイムのメトリクスを処理する CloudWatch を使用して Aurora DSQL をモニタリングします。CloudWatch はこれらの統計を 15 か月間保持するため、ウェブアプリケーションやサービスパフォーマンスをより適切に把握できます。アラームを設定して特定のしきい値を監視し、通知を送信したり、しきい値に達したときにアクションを実行したりします。Aurora DSQL で利用できる使用状況とオブザーバビリティのメトリクスを以下で確認してください。

詳細については、「Amazon CloudWatch ユーザーガイド」を参照してください。

オブザーバビリティとパフォーマンス

この表は、Aurora DSQL のオブザーバビリティメトリクスの概要を示しています。これには、読み取り専用トランザクションと合計トランザクションを追跡するためのメトリクスが含まれており、全体的なワークロードの特性を提供します。クエリタイムアウトや OCC 競合率などの実用的なメトリクスは、パフォーマンスの問題や同時実行の競合を特定するのに役立ちます。セッション関連のメトリクスは、アクティブと合計の両方で、システムの現在の負荷に関するインサイトを提供します。

CloudWatch メトリクス名	メトリクス	単位	説明
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 メトリクス名	メトリクス	単位	説明
BytesRead	Bytes Read	bytes	Bytes read from storage
ComputeTime	QP compute time	milliseconds	QP wall clock time
ClusterStorageSize	Cluster Storage Size	bytes	Cluster size

使用状況メトリクス

Aurora DSQL は、分散処理ユニット (DPU) と呼ばれる単一の正規化された請求ユニットを使用して、クエリ処理、読み取り、書き込みなど、すべてのリクエストベースのアクティビティを測定します。

CloudWatch メトリクス名	メトリクス	ディメンション: ResourceId	単位	説明
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.

CloudWatch メトリクス名	メトリクス	ディメンション: ResourceId	単位	説明
ReadDPU	Read Units	<cluster-id>	DPU	Approximates the read active-use component of your Aurora DSQL cluster DPU usage.
ComputeDPU	Compute Units	<cluster-id>	DPU	Approximates the compute active-use component of your Aurora DSQL cluster DPU usage.
TotalDPU	Total Units	<cluster-id>	DPU	Approximates the total active-use component of your Aurora DSQL cluster DPU usage.

AWS CloudTrail を使用した Aurora DSQL オペレーションのログ記録

Amazon Aurora DSQL は、ユーザー、ロール、または AWS のサービス が実行したアクションの記録を提供するサービスである AWS CloudTrail と統合されています。CloudTrail には、2 種類のイベント (管理イベント、データイベント) があります。管理イベントは、AWS リソース設定の変更を監査するために出力されます。データイベントは、通常、サービスデータプレーンで AWS リソースの使用状況をキャプチャします。

CloudTrail は、Aurora DSQL のすべての API コールをイベントとしてキャプチャします。Aurora DSQL はコンソールアクティビティを管理イベントとして記録します。また、クラスターへの認証された接続試行をデータイベントとしてキャプチャします。

CloudTrail で収集された情報を使用すると、Aurora DSQL に対して発行されたリクエスト、リクエスト元の IP アドレス、リクエスト作成日時、リクエスト作成者のユーザー ID、その他の詳細情報などを確認できます。

アカウントを作成すると、AWS アカウントで CloudTrail がデフォルトで有効になり、CloudTrail の[イベント履歴] にアクセスできるようになります。CloudTrail の [イベント履歴] では、AWS リージョン で過去 90 日間に記録された 管理イベントの表示、検索、およびダウンロードが可能で、変更不可能な記録を確認できます。詳細については、「AWS CloudTrail ユーザーガイド」の「CloudTrail イベント履歴の使用」を参照してください。[イベント履歴] の記録には CloudTrail の料金はかかりません。

Aurora DSQL のイベントなど、AWS アカウント内のイベントの継続的な記録を作成するには、証跡または AWS CloudTrail Lake イベントデータストア (AWS CloudTrail イベントの一元化されたストレージおよび分析ソリューション) を作成します。証跡の作成に関する詳細については、「CloudTrail 証跡の使用」を参照してください。イベントデータストアのセットアップと管理の詳細については、「CloudTrail Lake イベントデータストア」を参照してください。

CloudTrail での Aurora DSQL 管理イベント

CloudTrail 管理イベントでは、AWS アカウントのリソースで実行される管理オペレーションについての情報が得られます。これらのイベントは、コントロールプレーンオペレーションとも呼ばれます。デフォルトでは、CloudTrail は管理イベントをイベント履歴にキャプチャします。

Amazon Aurora DSQL は、すべての Aurora DSQL コントロールプレーンオペレーションを管理イベントとして記録します。Amazon Aurora DSQL が CloudTrail にログ記録する Aurora DSQL コントロールプレーンオペレーションのリストについては、「Aurora DSQL API リファレンス」を参照してください。

コントロールプレーンのログ

Amazon Aurora DSQL は、次の Aurora DSQL コントロールプレーンオペレーションを管理イベントとして CloudTrail に記録します。

- CreateCluster
- DeleteCluster

- GetCluster
- GetVpcEndpointServiceName
- ListClusters
- ListTagsForResource
- TagResource
- UntagResource
- UpdateCluster

バックアップと復元ログ

Amazon Aurora DSQL は、次の Aurora DSQL のバックアップおよび復元オペレーションを管理イベントとして CloudTrail に記録します。

- `StartBackupJob`
- `StopBackupJob`
- `GetBackupJob`
- `StartRestoreJob`
- `StopRestoreJob`
- `GetRestoreJob`

AWS Backup を使用して Aurora DSQL クラスターを保護する方法の詳細については、「Amazon Aurora DSQL のバックアップと復元」を参照してください。

AWS KMS ログ

Amazon Aurora DSQL は、次の AWS KMS オペレーションを管理イベントとして CloudTrail に記録します。

- `GenerateDataKey`
- `Decrypt`

Aurora DSQL がユーザーに代わって AWS KMS に送信するリクエストを CloudTrail がどのように口グに記録するかの詳細については、「Aurora DSQL と AWS KMS のインタラクションのモニタリング」を参照してください。

CloudTrail での Aurora DSQL データイベント

CloudTrail データイベントでは通常、リソース上またはリソース内で実行されたリソースオペレーションについての情報が得られます。これらは、サービスのデータプレーンオペレーションをキャプチャするためにも使用されます。データイベントは、多くの場合、高ボリュームのアクティビティです。デフォルトでは、CloudTrail はデータイベントをログ記録しません。CloudTrail [イベント履歴] にはデータイベントは記録されません。

データイベントをログに記録する方法の詳細については、「AWS CloudTrail ユーザーガイド」の「AWS Management Console を使用したデータイベントのログ記録」および「AWS Command Line Interface を使用したデータイベントのログ記録」を参照してください。

追加の変更がイベントデータに適用されます。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

Aurora DSQL の場合、CloudTrail は Aurora DSQL クラスターへの接続試行をデータイベントとしてキャプチャします。次の表に、データイベントをログに記録できる Aurora DSQL リソースタイプを示します。リソースタイプ (コンソール) 列には、CloudTrail コンソールの [リソースタイプ] リストから選択する値が表示されます。resources.type 値列には、AWS CLI または CloudTrail API を使用して高度なイベントセレクタを設定するときに指定する `resources.type` 値が表示されます。CloudTrail に記録されたデータ API 列には、リソース タイプの CloudTrail にログ記録された API コールが表示されます。

リソースタイプ (コンソール)	resources.type 値	CloudTrail にログ記録されたデータ API
Amazon Aurora DSQL	`AWS::DSQL::Cluster`	• `DbConnect` • `DbConnectAdmin`

高度なイベントセレクタを設定して、`eventName`、および `resources.ARN` フィールドをフィルタリングし、フィルタリングされたイベントのみをログ記録することができます。オブジェクトの詳細については、「AWS CloudTrail API リファレンス」の「AdvancedFieldSelector」を参照してください。

次の例は、AWS CLI を使用して、Aurora DSQL のデータイベントを受信するように `dsql-data-events-trail` を設定する方法を示しています。

```
aws cloudtrail put-event-selectors \
```

```
--region us-east-1 \
--trail-name dsql-data-events-trail \
--advanced-event-selectors '[{
"Name": "Log DSQL Data Events",
    "FieldSelectors": [
       { "Field": "eventCategory", "Equals": ["Data"] },
       { "Field": "resources.type", "Equals": ["AWS::DSQL::Cluster"] } ]}]'
```

Amazon Aurora DSQL のセキュリティ

AWS でのクラウドセキュリティは最優先事項です。AWS のユーザーは、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャを利用できます。

セキュリティは、AWS とユーザーの間の責任共有です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ - AWS は、AWS クラウド で AWS のサービスを実行するインフラストラクチャを保護する責任を担います。また、AWS は、ユーザーが安全に使用できるサービスも提供します。AWS コンプライアンスプログラムの一環として、サードパーティーの監査が定期的にセキュリティの有効性をテストおよび検証しています。Amazon Aurora DSQL に適用されるコンプライアンスプログラムの詳細については、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」を参照してください。
- クラウド内のセキュリティ - ユーザーの責任は、使用する 「AWS」 のサービスに応じて異なります。また、ユーザーは、データの機密性、会社の要件、適用される法律や規制など、その他の要因についても責任を負います。

このドキュメントは、Aurora DSQL を使用する際に責任共有モデルを適用する方法を理解するのに役立ちます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するために Aurora DSQL を設定する方法を示します。また、Aurora DSQL リソースのモニタリングや保護に役立つ他の AWS のサービスの使用方法についても説明します。

トピック

- Amazon Aurora DSQL の AWS マネージドポリシー
- Amazon Aurora DSQL でのデータ保護
- Amazon Aurora DSQL のデータ暗号化
- Aurora DSQL での Identity and Access Management
- Aurora DSQL でのサービスリンクロールの使用
- Amazon Aurora DSQL での IAM 条件キーの使用
- Amazon Aurora DSQL でのインシデント対応
- Amazon Aurora DSQL のコンプライアンス検証
- Amazon Aurora DSQL の耐障害性

- Amazon Aurora DSQL でのインフラストラクチャセキュリティ
- Amazon Aurora DSQL での設定と脆弱性の分析
- サービス間の混乱した代理の防止
- Aurora DSQL のセキュリティのベストプラクティス

Amazon Aurora DSQL の AWS マネージドポリシー

AWS マネージドポリシーは、AWS が作成および管理するスタンドアロンポリシーです。AWS マネージドポリシーは、多くの一般的なユースケースに対してアクセス許可を提供するように設計されているため、ユーザー、グループ、ロールへのアクセス権の割り当てを開始できます。

AWS マネージドポリシーは、ご利用の特定のユースケースに対して最小特権のアクセス許可を付与しない場合があることにご注意ください。これは、すべての AWS ユーザーが使用できるようになるのを避けるためです。ユースケースに固有のカスタマー管理ポリシーを定義して、アクセス許可を絞り込むことをお勧めします。

AWS マネージドポリシーで定義されたアクセス許可は変更できません。AWS が AWS マネージドポリシーに定義されている権限を更新すると、更新はポリシーがアタッチされているすべてのプリンシパルアイデンティティ (ユーザー、グループ、ロール) に影響します。新しい AWS のサービス を起動するか、既存のサービスで新しい API オペレーションが使用可能になると、AWS が AWS マネージドポリシーを更新する可能性が最も高くなります。

詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」を参照してください。

AWS マネージドポリシー: AmazonAuroraDSQLFullAccess

ユーザー、グループおよびロールに `AmazonAuroraDSQLFullAccess` をアタッチできます。

このポリシーは、Aurora DSQL への完全な管理アクセスを許可する許可を付与します。これらのアクセス許可を持つプリンシパルは、マルチリージョンクラスターを含む Aurora DSQL クラスターを作成、削除、更新できます。クラスターにタグを追加または削除できます。クラスターを一覧表示し、個々のクラスターに関する情報を表示できます。Aurora DSQL クラスターにアタッチされたタグを表示できます。管理者を含む任意のユーザーとしてデータベースに接続できます。バックアップおよび復元ジョブの開始、停止、およびモニタリングを含む、Aurora DSQL クラスターのバックアップおよび復元操作を実行できます。このポリシーには、クラスターの暗号化に使用され

るカスタマーマネージドキーに対する操作を許可する AWS KMS アクセス許可が含まれています。アカウントの CloudWatch から任意のメトリクスを表示できます。また、クラスターの作成に必要な、`dsql.amazonaws.com` サービスにリンクされたロールを作成するアクセス許可も付与されます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- `dsql` – プリンシパルに Aurora DSQL へのフルアクセスを付与します。
- `cloudwatch` – メトリクスデータポイントを Amazon CloudWatch に発行する許可を付与します。
- `iam` – サービスリンクロールを作成するアクセス許可を付与します。
- `backup and restore` – Aurora DSQL クラスターのバックアップジョブと復元ジョブを開始、停止、およびモニタリングするアクセス許可を付与します。
- `kms` – クラスターの作成、更新、または接続時に Aurora DSQL クラスターの暗号化に使用されるカスタマーマネージドキーへのアクセスを検証するために必要なアクセス許可を付与します。

`AmazonAuroraDSQLFullAccess` ポリシーは IAM コンソールに、AmazonAuroraDSQLFullAccess は AWS マネージドポリシーリファレンスガイドに記載されています。

AWS マネージドポリシー: AmazonAuroraDSQLReadOnlyAccess

ユーザー、グループおよびロールに `AmazonAuroraDSQLReadOnlyAccess` をアタッチできます。

Aurora DSQL への読み取りアクセスを許可します。これらのアクセス許可を持つプリンシパルは、クラスターを一覧表示し、個々のクラスターに関する情報を表示できます。Aurora DSQL クラスターにアタッチされたタグを確認できます。アカウントの CloudWatch からメトリクスを取得して表示できます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- `dsql` – Aurora DSQL のすべてのリソースに読み取り専用アクセス許可を付与します。

- `cloudwatch` – CloudWatch メトリクスデータのバッチ容量を取得し、取得したデータで Metric Math を行うアクセス許可を付与します。

`AmazonAuroraDSQLReadOnlyAccess` ポリシーは IAM コンソールに、AmazonAuroraDSQLReadOnlyAccess は AWS マネージドポリシーリファレンスガイドに記載されています。

AWS マネージドポリシー: AmazonAuroraDSQLConsoleFullAccess

ユーザー、グループおよびロールに `AmazonAuroraDSQLConsoleFullAccess` をアタッチできます。

AWS Management Console経由で Amazon Aurora DSQL への完全な管理アクセスを許可します。これらのアクセス許可を持つプリンシパルは、コンソールを使用して、マルチリージョンクラスターを含む Aurora DSQL クラスターを作成、削除、更新できます。クラスターを一覧表示し、個々のクラスターに関する情報を表示できます。アカウントの任意のリソースのタグを表示できます。管理者を含む任意のユーザーとしてデータベースに接続できます。バックアップおよび復元ジョブの開始、停止、およびモニタリングを含む、Aurora DSQL クラスターのバックアップおよび復元操作を実行できます。このポリシーには、クラスターの暗号化に使用されるカスタマーマネージドキーに対する操作を許可する AWS KMS アクセス許可が含まれています。AWS Management Consoleから AWS CloudShell を起動できます。アカウントの CloudWatch から任意のメトリクスを表示できます。また、クラスターの作成に必要な、`dsql.amazonaws.com` サービスにリンクされたロールを作成するアクセス許可も付与されます。

`AmazonAuroraDSQLConsoleFullAccess` ポリシーは IAM コンソールに、AmazonAuroraDSQLConsoleFullAccess は AWS マネージドポリシーリファレンスガイドに記載されています。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- `dsql` – AWS Management Consoleを介して Aurora DSQL のすべてのリソースに完全な管理アクセス許可を付与します。
- `cloudwatch` – CloudWatch メトリクスデータのバッチ容量を取得し、取得したデータで Metric Math を行うアクセス許可を付与します。

- `tag` – 呼び出し元アカウントの指定された AWS リージョンで現在使用されているタグキーと値を返す許可を付与します。
- `backup and restore` – Aurora DSQL クラスターのバックアップジョブと復元ジョブを開始、停止、およびモニタリングするアクセス許可を付与します。
- `kms` – クラスターの作成、更新、または接続時に Aurora DSQL クラスターの暗号化に使用されるカスタマーマネージドキーへのアクセスを検証するために必要なアクセス許可を付与します。
- `cloudshell` – Aurora DSQL とやり取りするための AWS CloudShell を起動するアクセス許可を付与します。
- `ec2` – Aurora DSQL 接続に必要な Amazon VPC エンドポイント情報を表示するアクセス許可を付与します。

`AmazonAuroraDSQLReadOnlyAccess` ポリシーは IAM コンソールに、AmazonAuroraDSQLReadOnlyAccess は AWS マネージドポリシーリファレンスガイドに記載されています。

AWS マネージドポリシー: AuroraDSQLServiceRolePolicy

AuroraDSQLServiceRolePolicy を IAM エンティティにアタッチすることはできません。このポリシーは、Aurora DSQL がアカウントリソースにアクセスすることを許可する、サービスにリンクされたロールにアタッチされます。

`AuroraDSQLServiceRolePolicy` ポリシーは IAM コンソールに、AuroraDSQLServiceRolePolicy は AWS マネージドポリシーリファレンスガイドに記載されています。

AWS マネージドポリシーに対する Aurora DSQL の更新

Aurora DSQL の AWS マネージドポリシーの更新に関する詳細を、このサービスがこれらの変更の追跡を開始した以降の分について表示します。このページの変更に関する自動通知については、Aurora DSQL ドキュメントの履歴ページの RSS フィードをサブスクライブしてください。

変更	説明	日付
AmazonAuroraDSQLFullAccess の更新	ジョブの開始、停止、モニタリングなど、Aurora DSQL ク	2025 年 5 月 21 日

変更	説明	日付
	ラスターのバックアップおよび復元オペレーションを実行する機能が追加されました。また、クラスター暗号化にカスタマーマネージド KMS キーを使用する機能も追加されています。 詳細については、「AmazonAuroraDSQLFullAccess」および「Aurora DSQL でのサービスリンクロールの使用」を参照してください。	
AmazonAuroraDSQLConsoleFullAccess の更新	AWS Console Home を介して Aurora DSQL クラスターのバックアップおよび復元オペレーションを実行する機能を追加しました。これには、ジョブの開始、停止、モニタリングが含まれます。また、クラスターの暗号化と AWS CloudShell の起動にカスタマーマネージド KMS キーを使用することもサポートしています。 詳細については、「AmazonAuroraDSQLConsoleFullAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 21 日

変更	説明	日付
AmazonAuroraDSQLFullAccess の更新	このポリシーで、複数の AWS リージョンにまたがるデータベースクラスターを作成および管理するための 4 つの新しいアクセス許可 (`PutMultiRegionProperties` 、`PutWitnessRegion` 、`AddPeerCluster` 、`RemovePeerCluster`) が追加されます。これらのアクセス許可には、リソースレベルのコントロールと条件キーが含まれ、どのクラスターユーザーを変更できるかを制御できます。 このポリシーで、AWS PrivateLink を介して Aurora DSQL クラスターに接続するのに役立つ `GetVpcEndpointServiceName` アクセス許可も追加されます。 詳細については、「AmazonAuroraDSQLFullAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 13 日

変更	説明	日付
AmazonAuroraDSQLReadOnlyAccess の更新	AWS PrivateLink を介して Aurora DSQL クラスターに接続するときに正しい VPC エンドポイントサービス名を決定する機能が含まれます。Aurora DSQL はセルごとに一意のエンドポイントを作成するため、この API はクラスターの適切なエンドポイントを特定し、接続エラーを回避できます。 詳細については、「AmazonAuroraDSQLReadOnlyAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 13 日

変更	説明	日付
AmazonAuroraDSQLConsoleFullAccess の更新	Aurora DSQL に新しいアクセス許可を追加し、マルチリージョンクラスター管理と VPC エンドポイント接続をサポートします。新しいアクセス許可には以下が含まれます。`PutMultiRegionProperties` `PutWitnessRegion` `AddPeerCluster` `RemovePeerCluster` `GetVpcEndpointServiceName` 詳細については、「AmazonAuroraDSQLConsoleFullAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 13 日

変更	説明	日付
AuroraDsqlServiceLinkedRolePolicy の更新	ポリシーの `AWS/AuroraDSQL` および `AWS/Usage` `CloudWatch` 名前空間にメトリクスを発行する機能を追加します。これにより、関連付けられたサービスまたはロールは、より包括的な使用状況とパフォーマンスデータを CloudWatch 環境に出力できます。 詳細については、「AuroraDsqlServiceLinkedRolePolicy」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 8 日
ページの作成	Amazon Aurora DSQL に関連する AWS マネージドポリシーの追跡を開始しました。	2024 年 12 月 3 日

Amazon Aurora DSQL でのデータ保護

責任共有モデルは、データ保護に適用されます。このモデルで説明されているように、AWS クラウドのすべてを実行するグローバルインフラストラクチャを保護する責任があります。ユーザーは、このインフラストラクチャでホストされるコンテンツに対する管理を維持する責任があります。また、使用する「」のセキュリティ設定と管理タスクもユーザーの責任となります。データプライバシーの詳細については、データプライバシーに関するよくある質問を参照してください。欧州でのデータ保護の詳細については、 セキュリティブログに投稿された 責任共有モデルおよび GDPR のブログ記事を参照してください。

データを保護するため、認証情報を保護し、AWS IAM Identity Center または AWS Identity and Access Management を使用して個々のユーザーをセットアップすることをお勧めします。この方法

により、それぞれのジョブを遂行するために必要な権限のみが各ユーザーに付与されます。また、次の方法でデータを保護することもお勧めします:

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して 「」 リソースと通信します。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- AWS CloudTrail で API とユーザーアクティビティロギングを設定します。証跡を使用してアクティビティをキャプチャする方法については、「ユーザーガイド」の「Working with trails」を参照してください。
- AWS のサービス内のすべてのデフォルトセキュリティコントロールに加え、暗号化ソリューションを使用します。
- Amazon Macie などの高度な管理されたセキュリティサービスを使用します。これらは、Amazon S3 に保存されている機密データの検出と保護を支援します。

お客様の E メールアドレスなどの極秘または機密情報を、タグ、または [名前] フィールドなどの自由形式のテキストフィールドに含めないことを強くお勧めします。これは、コンソール、API、AWS CLI、または AWS SDK を使用して操作する場合も同様です。タグ、または名前に使用される自由形式のテキストフィールドに入力されるデータは、請求または診断ログに使用される場合があります。外部サーバーに URL を提供する場合、そのサーバーへのリクエストを検証できるように、認証情報を URL に含めないことを強くお勧めします。

データ暗号化

Amazon Aurora DSQL は、ミッションクリティカルで重要なデータストレージのために設計された、高い耐久性を備えたストレージインフラストラクチャです。データは、冗長性を確保するため、同一の Aurora DSQL リージョン内の複数施設に分散した複数のデバイスに保存されます。

転送中の暗号化

デフォルトでは、転送中の暗号化が設定されます。Aurora DSQL は TLS を使用して、SQL クライアントと Aurora DSQL 間のすべてのトラフィックを暗号化します。

AWS CLI、SDK、または API クライアントと Aurora DSQL エンドポイントの間で転送中のデータの暗号化と署名

- Aurora DSQL には、転送中のデータを暗号化するための HTTP エンドポイントが用意されています。

- Aurora DSQL への API 要求の整合性を保護するには、API コールが呼び出し元によって署名されている必要があります。呼び出しは、署名バージョン 4 の署名プロセス (Sigv4) に従って、X.509 証明書またはお客様の AWS シークレットアクセスキーを使用して署名される必要があります。詳細については、https://docs.aws.amazon.com/general/latest/gr/signature-version-4.htmlの「AWS 全般のリファレンス署名バージョン 4 の署名プロセス」を参照してください。
- AWS CLI または AWS SDK の 1 つを使用して、 AWSに要求を送信します。これらのツールで、設定時に指定されたアクセスキーを使用すると、自動的に要求に署名されます。

保管中の暗号化については、「Aurora DSQL での保管中の暗号化」を参照してください。

ネットワーク間トラフィックのプライバシー

接続は、Aurora DSQL とオンプレミスのアプリケーション間、および Aurora DSQL と、同じ AWS リージョン内の他の AWS リソース間で保護されます。

プライベートネットワークと AWS との間には 2 つの接続オプションがあります

- AWS Site-to-Site VPN 接続。詳細については、「AWS Site-to-Site VPN とは」を参照してください。
- AWS Direct Connect 接続。詳細については、「AWS Direct Connect とは?」を参照してください。

AWS が公開する API オペレーションを使用することにより、ネットワークを通じて、Aurora DSQL へのアクセスを取得できます。クライアントは以下をサポートする必要があります。

- Transport Layer Security (TLS)。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードはJava 7 以降など、ほとんどの最新システムでサポートされています。

Aurora DSQL 接続用の SSL/TLS 証明書の設定

Aurora DSQL では、すべての接続で Transport Layer Security (TLS) 暗号化を使用する必要があります。安全な接続を確立するには、クライアントシステムが Amazon ルート認証機関 (Amazon ルート CA 1) を信頼する必要があります。この証明書は、多くのオペレーティングシステムにプリインストールされています。このセクションでは、さまざまなオペレーティングシステムにプリインストー

ルされている Amazon ルート CA 1 証明書を検証する手順と、証明書が存在しない場合は手動でインストールする手順について説明します。

PostgreSQL バージョン 17 を使用することをお勧めします。

> **Important**
>
> 本番環境では、最高レベルの接続セキュリティを確保するために `verify-full` SSL モードを使用することをお勧めします。このモードでは、サーバー証明書が信頼できる認証機関によって署名されていること、およびサーバーのホスト名が証明書と一致することを確認します。

プリインストールされた証明書の検証

ほとんどのオペレーティングシステムでは、Amazon ルート CA 1 は既にプリインストールされています。これを検証するには、以下の手順に従います。

Linux (RedHat/CentOS/Fedora)

ターミナルで以下のコマンドを実行します。

```
trust list | grep "Amazon Root CA 1"
```

証明書がインストールされている場合、次の出力が表示されます。

```
label: Amazon Root CA 1
```

macOS

1. Spotlight 検索 (Command + スペース) を開く
2. [キーチェーンアクセス] を検索する
3. [システムキーチェーン] で [システムルート] を選択する
4. 証明書リストで [Amazon ルート CA 1] を検索する

Windows

> **Note**
>
> psql Windows クライアントの既知の問題により、システムルート証明書 (`sslrootcert=system`) を使用すると、`SSL error: unregistered scheme` エラーが返されることがあります。SSL を使用してクラスターに接続する代替方法として「Windows からの接続」を使用できます。

Amazon ルート CA 1 がオペレーティングシステムにインストールされていない場合は、以下の手順に従います。

証明書のインストール

`Amazon Root CA 1` 証明書がオペレーティングシステムにプリインストールされていない場合は、Aurora DSQL クラスターへの安全な接続を確立するために、証明書を手動でインストールする必要があります。

Linux 証明書のインストール

Linux システムに Amazon ルート CA 証明書をインストールするには、次の手順に従います。

1. ルート証明書をダウンロードします。

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 証明書をコピーしてトラストストアに追加します。

```
sudo cp ./AmazonRootCA1.pem /etc/pki/ca-trust/source/anchors/
```

3. CA トラストストアを更新します。

```
sudo update-ca-trust
```

4. インストールを確認します。

```
trust list | grep "Amazon Root CA 1"
```

macOS 証明書のインストール

これらの証明書のインストール手順はオプションです。「Linux 証明書のインストール」も macOS で機能します。

1. ルート証明書をダウンロードします。

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 証明書をシステムキーチェーンに追加します。

```
sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain
 AmazonRootCA1.pem
```

3. インストールを確認します。

```
security find-certificate -a -c "Amazon Root CA 1" -p /Library/Keychains/
System.keychain
```

SSL/TLS 検証を使用した接続

Aurora DSQL クラスターへの安全な接続のために SSL/TLS 証明書を設定する前に、次の前提条件を満たしていることを確認してください。

- PostgreSQL バージョン 17 がインストールされている
- AWS CLI が適切な認証情報で設定されている
- Aurora DSQL クラスターエンドポイント情報

Linux からの接続

1. 認証トークンを生成および設定します。

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --hostname your-cluster-endpoint)
```

2. システム証明書を使用して接続します (プリインストールされている場合)。

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
```

```
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. または、ダウンロードした証明書を使用して接続します。

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

> **Note**
>
> PGSSLMODE 設定の詳細については、PostgreSQL 17「Database Connection Control Functions」ドキュメントの「sslmode」を参照してください。

macOS からの接続

1. 認証トークンを生成および設定します。

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --hostname your-cluster-endpoint)
```

2. システム証明書を使用して接続します (プリインストールされている場合)。

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. または、ルート証明書をダウンロードして `root.pem` として保存します (証明書がプリインストールされていない場合)。

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
```

```
--host your_cluster_endpoint
```

4. psql を使用して接続します。

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

Windows からの接続

コマンドプロンプトを使用する

1. 認証トークンを生成する

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

2. パスワード環境変数を設定する

```
set "PGPASSWORD=token-from-above"
```

3. SSL を設定します。

```
set PGSSLROOTCERT=C:\full\path\to\root.pem
set PGSSLMODE=verify-full
```

4. データベースに接続します。

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres ^
--username admin ^
--host your-cluster-endpoint
```

PowerShell を使用する

1. 認証トークンを生成および設定します。

```
$env:PGPASSWORD = (aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --expires-in=3600 --hostname=your-cluster-endpoint)
```

2. SSL を設定します。

```
$env:PGSSLROOTCERT='C:\full\path\to\root.pem'
$env:PGSSLMODE='verify-full'
```

3. データベースに接続します。

```
 "C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres `
--username admin `
--host your-cluster-endpoint
```

追加リソース

- PostgreSQL SSL ドキュメント
- Amazon Trust Services

Amazon Aurora DSQL のデータ暗号化

Amazon Aurora DSQL は、保管中のすべてのユーザーデータを暗号化します。セキュリティを強化するために、この暗号化では AWS Key Management Service (AWS KMS) を使用します。この機能は、機密データの保護における負担と複雑な作業を減らすのに役立ちます。保管中の暗号化は、以下に役立ちます。

- 機密データを保護する運用上の負担を軽減する
- セキュリティを重視したアプリケーションを構築して、暗号化のコンプライアンスと規制の厳格な要件を満たす
- 暗号化されたクラスター内のデータを常に保護することで、追加のデータ保護レイヤーを提供する
- 組織のポリシー、業界または政府の規制、コンプライアンス要件に準拠する

Aurora DSQL では、セキュリティを重視したアプリケーションを構築して、暗号化のコンプライアンスと規制の厳格な要件を満たすことができます。以下のセクションでは、新規および既存の Aurora DSQL データベースの暗号化を設定し、暗号化キーを管理する方法について説明します。

トピック

- Aurora DSQL の KMS キーの種類
- Aurora DSQL での保管中の暗号化
- Aurora DSQL での AWS KMS およびデータキーの使用
- Aurora DSQL で AWS KMS keyの使用を許可する
- Aurora DSQL 暗号化コンテキスト
- Aurora DSQL と AWS KMS のインタラクションのモニタリング
- 暗号化された Aurora DSQL クラスターの作成
- Aurora DSQL クラスターのキーの削除または更新
- Aurora DSQL による暗号化に関する考慮事項

Aurora DSQL の KMS キーの種類

Aurora DSQL は AWS KMS と統合して、クラスターの暗号化キーを管理します。キーの種類と詳細については、「AWS Key Management Service デベロッパーガイド」の「AWS Key Management Service concepts」を参照してください。新しいクラスターを作成するときは、次の KMS キータイプから選択してクラスターを暗号化できます。

AWS 所有のキー

デフォルトの暗号化タイプ。Aurora DSQL は、追加料金なしでキーを所有します。Amazon Aurora DSQL は、暗号化されたクラスターにアクセスするときにクラスターデータを透過的に復号します。暗号化されたクラスターを使用または管理するためにコードやアプリケーションを変更する必要はありません。すべての Aurora DSQL クエリは暗号化されたデータで動作します。

カスタマーマネージドキー

お客様の AWS アカウントでキーを作成、所有、および管理できます。ユーザーには KMS キーに対するフルコントロールの権限があります (AWS KMS の料金が適用されます)。

AWS 所有のキー を使用した保管中の暗号化に追加の料金はかかりません。ただし、カスタマーマネージドキーには AWS KMS の料金がかかります。詳細については、AWS KMS 料金表ページを参照してください。

キーの種類は、いつでも切り替えることができます。キータイプの詳細については、「AWS Key Management Service デベロッパーガイド」の「カスタマーマネージドキー」および「AWS 所有のキー」を参照してください。

> **Note**
>
> Aurora DSQL の保管中の暗号化は、Aurora DSQL が利用可能なすべての AWS リージョンでご利用いただけます。

Aurora DSQL での保管中の暗号化

Amazon Aurora DSQL では、256 ビットの Advanced Encryption Standard (AES-256) を使用して保管中のデータを暗号化します。この暗号化は、基盤となるストレージへの不正アクセスからデータを保護するのに役立ちます。AWS KMS は、クラスターの暗号化キーを管理します。デフォルトの AWS 所有のキー を使用できます。または、独自の AWS KMS カスタマーマネージドキー を使用することを選択できます。Aurora DSQL クラスターのキーの指定と管理の詳細については、「暗号化された Aurora DSQL クラスターの作成」および「Aurora DSQL クラスターのキーの削除または更新」を参照してください。

トピック

- AWS 所有のキー
- カスタマーマネージドキー

AWS 所有のキー

Aurora DSQL は、デフォルトですべてのクラスターを AWS 所有のキー で暗号化します。これらのキーは、アカウントリソースを保護するために無料で使用でき、毎年ローテーションします。キーを表示、管理、使用、または監査する必要がないため、データ保護のためのアクションは必要ありません。AWS 所有のキー の詳細については、「AWS Key Management Service Developer Guide」の「AWS 所有のキー 」を参照してください。

カスタマーマネージドキー

AWS アカウントでカスタマーマネージドキーを作成、所有、管理できます。ポリシー、暗号化マテリアル、タグ、エイリアスなど、これらの KMS キーを完全に制御できます。アクセス許可の管理の詳細については、「AWS Key Management Service デベロッパーガイド」の「カスタマーマネージドキー」を参照してください。

クラスターレベルの暗号化にカスタマーマネージドキーを指定すると、Aurora DSQL はそのキーを使用して、クラスターとそのすべてのリージョンのデータを暗号化します。データ損失

を防ぎ、クラスターアクセスを維持するために、Aurora DSQL は暗号化キーにアクセスする必要があります。カスタマーマネージドキーを無効にした場合、キーの削除をスケジュールした場合、またはサービスアクセスを制限するポリシーがある場合、クラスターの暗号化ステータスは KMS_KEY_INACCESSIBLE に変わります。Aurora DSQL がキーにアクセスできない場合、ユーザーはクラスターに接続できず、クラスターの暗号化ステータスが KMS_KEY_INACCESSIBLE に変わります。また、サービスはクラスターデータにアクセスできなくなります。

マルチリージョンクラスターの場合、お客様は各リージョンの AWS KMS 暗号化キーを個別に設定でき、各リージョンのクラスターは独自のクラスターレベルの暗号化キーを使用します。Aurora DSQL がマルチリージョンクラスター内のピアの暗号化キーにアクセスできない場合、そのピアのステータスは KMS_KEY_INACCESSIBLE になり、読み取りおよび書き込み操作で使用できなくなります。他のピアは通常の操作を続行します。

> **Note**
>
> Aurora DSQL がカスタマーマネージドキーにアクセスできない場合、クラスターの暗号化ステータスは KMS_KEY_INACCESSIBLE に変わります。キーアクセスを復元すると、サービスは 15 分以内に復元を自動的に検出します。詳細については、「クラスターのアイドリング」を参照してください。
> マルチリージョンクラスターの場合、キーへのアクセスが長期間失われると、クラスターの復元時間は、キーにアクセスできない間に書き込まれたデータの量によって異なります。

Aurora DSQL での AWS KMS およびデータキーの使用

Aurora DSQL の保管中の暗号化機能は、AWS KMS keyおよびデータキーの階層を使用してクラスターのデータを保護します。

Aurora DSQL にクラスターを実装する前に、暗号化戦略を計画することをお勧めします。機密データまたは秘密データを Aurora DSQL に保存する場合は、クライアント側の暗号化をプランに含めることを検討してください。これにより、データをできるだけ送信元に近い状態で暗号化し、ライフサイクル全体にわたってデータを確実に保護できます。

トピック

- Aurora DSQL での AWS KMS keyの使用
- Aurora DSQL でのクラスターキーの使用
- クラスターキーのキャッシュ

Aurora DSQL での AWS KMS keyの使用

保管中の暗号化は、AWS KMS keyで Aurora DSQL クラスターを保護します。デフォルトでは、Aurora DSQL は AWS 所有のキー (Aurora DSQL サービスアカウントで作成および管理されるマルチテナント暗号化キー) を使用します。ただし、Aurora DSQL クラスターは、AWS アカウントのカスタマーマネージドキーで暗号化できます。マルチリージョン設定に参加している場合でも、クラスターごとに異なる KMS キーを選択できます。

クラスターを作成または更新するときは、クラスター用の KMS キーを選択します。クラスター用 KMS キーは、Aurora DSQL コンソールで、または `UpdateCluster` オペレーションを使用していつでも変更できます。キーの切り替えプロセスでは、ダウンタイムやサービスの低下が求められません。

> **Important**
>
> Aurora DSQL は、対称 KMS キーのみをサポートします。非対称 KMS キーを使用して Aurora DSQL クラスターを暗号化することはできません。

カスタマーマネージドキーには、次の利点があります。

- KMS キーを作成および管理します。これには、キーポリシーおよび IAM ポリシーの設定、KMS キーへのアクセスを制御する設定が含まれます。KMS キーの有効化または無効化、自動キーローテーションの有効化または無効化、使用しなくなった KMS キーの削除を実行できます。
- インポートされたキーマテリアルを持つカスタマーマネージドキー、またはユーザーが所有して管理するカスタムキーストアで、カスタマーマネージドキーを使用できます。
- Aurora DSQL クラスターの暗号化と復号を監査するには、AWS CloudTrail ログで AWS KMS への Aurora DSQL API コールを調べます。

ただし、AWS 所有のキー は無料で、その使用は AWS KMS リソースクォータまたはリクエストクォータに対してカウントされません。カスタマーマネージドキーには API コールごとに料金が発生し、これらのキーには AWS KMS クォータが適用されます。

Aurora DSQL でのクラスターキーの使用

Aurora DSQL は、クラスターの AWS KMS keyを使用して、クラスターキーと呼ばれるクラスターの一意のデータキーを生成および暗号化します。

クラスターキーは、キー暗号化キーとして使用されます。Aurora DSQL はこのクラスターキーを使用して、クラスターデータの暗号化に使用されるデータ暗号化キーを保護します。Aurora DSQL はクラスターの基本構造ごとに一意のデータ暗号化キーを生成しますが、複数のクラスター項目は同じデータ暗号化キーによって保護されていることがあります。

クラスターキーを復号するために、Aurora DSQL は暗号化されたクラスターに初めてアクセスするときに AWS KMS にリクエストを送信します。クラスターを使用可能にするために、Aurora DSQL はクラスターにアクティブにアクセスしていない場合でも、KMS キーへの復号アクセスを定期的に検証します。

Aurora DSQL は、クラスターキーとデータ暗号化キーを AWS KMS の外部に保存して使用します。これによって、Advanced Encryption Standard (AES) 暗号化および 256 ビット暗号化キーのすべてのキーが保護されます。続いて、暗号化されたキーを暗号化されたデータと一緒に保存します。これらのキーおよびデータは、必要なときにクラスターデータの復号に使用できます。

クラスターの KMS キーを変更すると、Aurora DSQL は既存のクラスターキーを新しい KMS キーで再暗号化します。

クラスターキーのキャッシュ

Aurora DSQL オペレーションごとに AWS KMS を呼び出さないように、Aurora DSQL は各呼び出しのプレーンテキストのクラスターキーをメモリにキャッシュします。Aurora DSQL では、キャッシュしたクラスターキーが 15 分間非アクティブ状態であった後にリクエストを取得すると、AWS KMS に新しいリクエストを送信してクラスターキーを復号します。この呼び出しは、クラスターキーの復号を求める前回のリクエスト以降に AWS KMS または AWS Identity and Access Management (IAM) で AWS KMS keyのアクセスポリシーに加えられた変更をすべてキャプチャします。

Aurora DSQL で AWS KMS keyの使用を許可する

Aurora DSQL クラスターを保護するために、アカウントでカスタマーマネージドキーを使用する場合は、そのキーのポリシーが Aurora DSQL に、ユーザーに代わってキーを使用する許可を付与する必要があります。

カスタマーマネージドキーに対するポリシーのフルコントロールはお客様にあります。Aurora DSQL では、デフォルトの AWS 所有のキー を使用して AWS アカウントの Aurora DSQL クラスターを保護するために追加の承認は必要ありません。

カスタマーマネージドキーのキーポリシー

Aurora DSQL クラスターを保護するためにカスタマーマネージドキーを選択する場合、Aurora DSQL には、選択を行うプリンシパルに代わって AWS KMS keyを使用するアクセス許可が必要です。そのプリンシパル (ユーザーまたはロール) は、Aurora DSQL に必要な AWS KMS keyのアクセス許可を持つ必要があります。それらのアクセス許可は、キーポリシーまたは IAM ポリシーにより付与できます。

Aurora DSQL には、少なくとも、カスタマーマネージドキーに対する次のアクセス許可が必要です。

- `kms:Encrypt`
- `kms:Decrypt`
- `kms:ReEncrypt*` (kms:ReEncryptFrom および kms:ReEncryptTo の場合)
- `kms:GenerateDataKey`
- `kms:DescribeKey`

例えば、次のキーポリシーの例では、必要なアクセス許可のみを提供します。このポリシーには、以下の影響があります。

- Aurora DSQL で AWS KMS keyを暗号化オペレーションで使用します。ただし、Aurora DSQL の使用許可を持つアカウント内のプリンシパルの代理として動作している場合に限ります。ポリシーステートメントで指定されたプリンシパルに Aurora DSQL を使用するアクセス許可がない場合、Aurora DSQL サービスから呼び出された場合でも、呼び出しは失敗します。
- `kms:ViaService` 条件キーは、ポリシーステートメントにリストされているプリンシパルの代わりに Aurora DSQL からリクエストが来た場合にのみアクセス許可を許可します。これらのプリンシパルは、これらのオペレーションを直接呼び出すことはできません。
- AWS KMS key管理者 (`db-team` ロールを引き受けることができるユーザー) に AWS KMS keyへの読み取り専用アクセスを付与します。

サンプルキーポリシーを使用する前に、サンプルプリンシパルを AWS アカウント の実際のプリンシパルに置き換えます。

```
{
  "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 暗号化コンテキスト

暗号化コンテキスト は、一連のキー値のペアおよび任意非シークレットデータを含みます。データを暗号化するリクエストに暗号化コンテキストを組み込むと、AWS KMS は暗号化コンテキストを暗号化されたデータに暗号化してバインドします。データを復号するには、同じ暗号化コンテキストに渡す必要があります。

Aurora DSQL は、すべての AWS KMS 暗号化オペレーションで同じ暗号化コンテキストを使用します。カスタマーマネージドキーを使用して Aurora DSQL クラスターを保護している場合は、暗号化

コンテキストを使って監査レコードやログの中で AWS KMS keyの使用を特定することができます。また、AWS CloudTrail などのログにもプレーンテキストで表示されます。

また、暗号化コンテクストはポリシーの認可用の条件としても使用できます。

Aurora DSQL は AWS KMS へのリクエストで、キーバリューペアを持つ暗号化コンテキストを使用します。

```
"encryptionContext": {
  "aws:dsql:ClusterId": "w4abucpbwuxx"
},
```

キーと値のペアは、Aurora DSQL が暗号化しているクラスターを識別します。キーは、`aws:dsql:ClusterId` です。値はクラスターの識別子です。

Aurora DSQL と AWS KMS のインタラクションのモニタリング

カスタマーマネージドキーを使って Aurora DSQL クラスターを保護している場合、AWS CloudTrail ログを使って、Aurora DSQL がお客様に代わって AWS KMS に送信したリクエストを追跡できます。

以下のセクションを展開して、Aurora DSQL が AWS KMS オペレーション `GenerateDataKey` と `Decrypt` を使用する方法を確認してください。

`GenerateDataKey`

クラスターで保管中の暗号化を有効にすると、Aurora DSQL は一意のクラスターキーを作成します。クラスターの AWS KMS keyを指定する `GenerateDataKey` リクエストを AWS KMS に送信します。

`GenerateDataKey` 演算を記録するイベントは、次のようなサンプルイベントになります。ユーザーは、Aurora DSQL サービスアカウントです。パラメータには、AWS KMS keyの Amazon リソースネーム (ARN)、256 ビットキーを必要とするキー指定子、クラスターを識別する暗号化コンテキストが含まれます。

```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "type": "AWSService",
        "invokedBy": "dsql.amazonaws.com"
```

```
    },
    "eventTime": "2025-05-16T18:41:24Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "dsql.amazonaws.com",
    "userAgent": "dsql.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "aws:dsql:ClusterId": "w4abucpbwuxx"
        },
        "keySpec": "AES_256",
        "keyId": "arn:aws:kms:us-east-1:982127530226:key/8b60dd9f-2ff8-4b1f-8a9c-
bf570cbfdb5e"
    },
    "responseElements": null,
    "requestID": "2da2dc32-d3f4-4d6c-8a41-aff27cd9a733",
    "eventID": "426df0a6-ba56-3244-9337-438411f826f4",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:982127530226:key/8b60dd9f-2ff8-4b1f-8a9c-
bf570cbfdb5e"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "sharedEventID": "f88e0dd8-6057-4ce0-b77d-800448426d4e",
    "vpcEndpointId": "AWS Internal",
    "vpcEndpointAccountId": "vpce-1a2b3c4d5e6f1a2b3",
    "eventCategory": "Management"
}
```


Decrypt

暗号化された Aurora DSQL クラスターにアクセスすると、Aurora DSQL はクラスターキーの復号化が必要となるため、階層の下でキーを復号できます。次に、クラスター内のデータを復号化します。クラスターキーを復号するために、Aurora DSQL はクラスターの AWS KMS keyを指定する `Decrypt` リクエストを AWS KMS に送信します。

Decrypt 演算を記録するイベントは、次のようなサンプルイベントになります。ユーザーは、クラスターにアクセスしている AWS アカウントのプリンシパルです。パラメータには、暗号化されたクラスターキー (暗号文の blob として)、およびクラスターを識別する暗号化コンテキストが含まれます。AWS KMS は、暗号文から AWS KMS keyの ID を取得します。

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AWSService",
    "invokedBy": "dsql.amazonaws.com"
  },
  "eventTime": "2018-02-14T16:42:39Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "Decrypt",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "dsql.amazonaws.com",
  "userAgent": "dsql.amazonaws.com",
  "requestParameters": {
    "keyId": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    "encryptionContext": {
      "aws:dsql:ClusterId": "w4abucpbwuxx"
    },
    "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
  },
  "responseElements": null,
  "requestID": "11cab293-11a6-11e8-8386-13160d3e5db5",
  "eventID": "b7d16574-e887-4b5b-a064-bf92f8ec9ad3",
  "readOnly": true,
  "resources": [
    {
      "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
      "accountId": "AWS Internal",
      "type": "AWS::KMS::Key"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
  "sharedEventID": "d99f2dc5-b576-45b6-aa1d-3a3822edbeeb",
  "vpcEndpointId": "AWS Internal",
  "vpcEndpointAccountId": "vpce-1a2b3c4d5e6f1a2b3",
```

```
  "eventCategory": "Management"
}
```

暗号化された Aurora DSQL クラスターの作成

すべての Aurora DSQL クラスターは保管中に暗号化されます。デフォルトでは、クラスターは AWS 所有のキーを無料で使用するか、カスタム AWS KMS キーを指定できます。AWS Management Consoleまたは AWS CLI から暗号化されたクラスターを作成するには、次の手順に従います。

Console

AWS Management Consoleで暗号化されたクラスターを作成するには

1. AWS マネジメントコンソールにサインインして Aurora DSQL コンソール (https://console.aws.amazon.com/dsql/) を開きます。
2. コンソールの左側のナビゲーションペインで、Clusters を選択します。
3. 右上で [クラスターを作成] を選択し、[シングルリージョン] を選択します。
4. [クラスターの暗号化設定] で、次のいずれかのオプションを選択します。
   - デフォルト設定を受け入れて、追加料金なしで AWS 所有のキー を使用して暗号化します。
   - [暗号化の設定をカスタマイズ (詳細設定)] を選択して、カスタム KMS キーを指定します。次に、KMS キーの ID またはエイリアスを検索または入力します。または、AWS KMS コンソールで新しいキーを作成する場合は、[AWS KMS キーを作成する] を選択します。
5. [クラスターを作成] を選択します。

クラスターの暗号化タイプを確認するには、[クラスター] ページに移動し、クラスターの ID を選択してクラスターの詳細を表示します。[クラスター設定] タブを確認します。[クラスター KMS キー] 設定には、AWS 所有キーを使用するクラスターの場合は Aurora DSQL デフォルトキー、その他の暗号化タイプの場合はキー ID が表示されます。

> **Note**
>
> 独自のキーを所有および管理することを選択した場合は、KMS キーポリシーが適切に設定されていることを確認してください。詳細情報と例については、「the section called "カスタマーマネージドキーのキーポリシー"」を参照してください。

CLI

デフォルトの AWS 所有のキーを使用して暗号化されたクラスターを作成するには

- 次のコマンドを使用して、Aurora DSQL クラスターを作成します。

```
aws dsql create-cluster
```

次の暗号化の詳細に示すように、クラスターの暗号化ステータスはデフォルトで有効になっており、デフォルトの暗号化タイプは AWS 所有キーです。クラスターが Aurora DSQL サービスアカウントのデフォルトの AWS 所有キーで暗号化されるようになりました。

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

カスタマーマネージドキーで暗号化されたクラスターを作成するには

- 次のコマンドを使用して Aurora DSQL クラスターを作成し、赤いテキストのキー ID をカスタマーマネージドキーの ID に置き換えます。

```
aws dsql create-cluster \
--kms-encryption-key d41d8cd98f00b204e9800998ecf8427e
```

次の暗号化の詳細に示すように、クラスターの暗号化ステータスはデフォルトで有効になっており、暗号化タイプはカスタマーマネージド KMS キーです。これで、クラスターがユーザーのキーで暗号化されるようになりました。

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
```

```
  "kmsKeyArn" : "arn:aws:kms:us-east-1:111122223333:key/
d41d8cd98f00b204e9800998ecf8427e",
  "encryptionStatus" : "ENABLED"
}
```

Aurora DSQL クラスターのキーの削除または更新

AWS Management Consoleまたは AWS CLI を使用して、Amazon Aurora DSQL の既存のクラスターの暗号化キーを更新または削除できます。キーを置き換えずに削除すると、Aurora DSQL はデフォルトの AWS 所有のキー を使用します。Aurora DSQL コンソールまたは AWS CLI から既存のクラスターの暗号化キーを更新するには、次の手順に従います。

Console

AWS Management Consoleで暗号化キーを更新または削除するには

1. AWS マネジメントコンソールにサインインして Aurora DSQL コンソール (https://console.aws.amazon.com/dsql/) を開きます。
2. コンソールの左側のナビゲーションペインで、Clusters を選択します。
3. リストビューから、更新するクラスターの行を見つけて選択します。
4. [アクション] メニューを選択し、次に [変更] を選択します。
5. [クラスターの暗号化設定] で、次のいずれかのオプションを選択して暗号化設定を変更します。
   - カスタムキーから AWS 所有のキー に切り替える場合は、[暗号化の設定をカスタマイズ (詳細設定)] オプションの選択を解除します。デフォルト設定が適用され、クラスターは AWS 所有のキーで無料で暗号化されます。
   - あるカスタム KMS キーから別のキーに切り替える場合、または AWS 所有のキーから KMS キーに切り替える場合は、[暗号化の設定をカスタマイズ (詳細設定)] オプションを選択します。次に、使用するキーの ID またはエイリアスを検索して選択します。または、AWS KMS コンソールで新しいキーを作成する場合は、[AWS KMS キーを作成する] を選択します。
6. [保存] を選択します。

CLI

次の例は、AWS CLI を使用して暗号化クラスターを更新する方法を示しています。

デフォルトの AWS 所有のキーで暗号化クラスターを更新するには

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key "AWS_OWNED_KMS_KEY"
```

クラスターの説明の `EncryptionStatus` は `ENABLED` に設定され、`EncryptionType` は `AWS_OWNED_KMS_KEY` になります。

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

このクラスターは、Aurora DSQL サービスアカウントでデフォルトの AWS 所有のキー を使用して暗号化されています。

Aurora DSQL のカスタマーマネージドキーを使用して暗号化されたクラスターを更新するには

暗号化されたクラスターを次の例のとおりに更新します。

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234a1b234
```

クラスターの説明の `EncryptionStatus` は `UPDATING` に移行し、`EncryptionType` は `CUSTOMER_MANAGED_KMS_KEY` になります。Aurora DSQL がプラットフォームを介した新しいキーのプロパゲートを完了すると、暗号化ステータスは `ENABLED` に移行します。

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:us-east-1:kms:key/abcd1234-abcd-1234-a123-ab1234a1b234",
  "encryptionStatus" : "ENABLED"
}
```

> **Note**
>
> 独自のキーを所有および管理することを選択した場合は、KMS キーポリシーが適切に設定されていることを確認してください。詳細情報と例については、「the section called “カスタマーマネージドキーのキーポリシー”」を参照してください。

Aurora DSQL による暗号化に関する考慮事項

- Aurora DSQL は、保管中のすべてのクラスターデータを暗号化します。この暗号化を無効にしたり、クラスター内の一部の項目のみを暗号化したりすることはできません。
- AWS Backup は、バックアップと、これらのバックアップから復元されたクラスターを暗号化します。AWS Backup でバックアップデータを暗号化するには、AWS 所有キーまたはカスタマーマネージドキーを使用します。
- Aurora DSQL では、次のデータ保護状態が有効になっています。
  - 保管中のデータ - Aurora DSQL は永続的ストレージメディア上のすべての静的データを暗号化します。
  - 転送中のデータ - Aurora DSQL は、デフォルトで Transport Layer Security (TLS) を使用してすべての通信を暗号化します。
- 別のキーに移行する場合は、移行が完了するまで元のキーを有効にしておくことをお勧めします。AWS は、新しいキーでデータを暗号化する前に、データを復号するために元のキーを必要とします。クラスターの `encryptionStatus` が `ENABLED` で、新しいカスタマーマネージドキーの `kmsKeyArn` が表示されると、このプロセスは完了です。
- カスタマーマネージドキーを無効にするか、Aurora DSQL がキーを使用するためのアクセスを取り消すと、クラスターは `IDLE` 状態になります。
- AWS Management Consoleと Amazon Aurora DSQL API は、暗号化タイプに異なる用語を使用します。
  - AWS コンソール – コンソールでは、カスタマーマネージドキーを使用する場合は `KMS`、AWS 所有のキーを使用する場合は `DEFAULT` が表示されます。
  - API – Amazon Aurora DSQL API は、カスタマーマネージドキーには `CUSTOMER_MANAGED_KMS_KEY` を使用し、AWS 所有のキーには `AWS_OWNED_KMS_KEY` を使用します。

- クラスターの作成時に暗号化キーを指定しない場合、Aurora DSQL は AWS 所有のキーを使用してデータを自動的に暗号化します。
- AWS 所有のキーとカスタマーマネージドキーはいつでも切り替えることができます。この変更は、AWS Management Console、AWS CLI、または Amazon Aurora DSQL API を使用して行います。

Aurora DSQL での Identity and Access Management

AWS Identity and Access Management (IAM) は管理者が AWS リソースへのアクセスを安全に制御するために役立つ AWS のサービスです。IAM 管理者は、誰が認証 (サインイン) され、Aurora DSQL リソースを使用する認可を受ける (許可がある) ことができるかを制御します。IAM は、追加費用なしで使用できる AWS のサービスです。

トピック

- 対象者
- アイデンティティによる認証
- ポリシーを使用したアクセス権の管理
- Amazon Aurora DSQL と IAM の連携
- Amazon Aurora DSQL のアイデンティティベースのポリシー例
- Amazon Aurora DSQL のアイデンティティおよびアクセスのトラブルシューティング

対象者

AWS Identity and Access Management (IAM) の使用方法は、Aurora DSQL で行う作業に応じて異なります。

サービスユーザー – ジョブを実行するために Aurora DSQL サービスを使用する場合は、管理者から必要な認証情報と許可が与えられます。作業を実行するために使用する Aurora DSQL 機能が増えるにつれて、追加のアクセス許可が必要になる場合があります。アクセスの管理方法を理解すると、管理者に適切なアクセス許可をリクエストするのに役に立ちます。Aurora DSQL の機能にアクセスできない場合は、「Amazon Aurora DSQL のアイデンティティおよびアクセスのトラブルシューティング」を参照してください。

サービス管理者 - 社内の Aurora DSQL リソースを担当している場合は、通常、Aurora DSQL へのフルアクセスがあります。サービスのユーザーがどの Aurora DSQL 機能やリソースにアクセスするか

を決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。お客様の会社が Aurora DSQL で IAM を使用する方法の詳細については、「Amazon Aurora DSQL と IAM の連携」を参照してください。

IAM 管理者 – IAM 管理者は、Aurora DSQL へのアクセスを管理するポリシーの作成方法の詳細について確認する場合があります。IAM で使用できる Aurora DSQL のアイデンティティベースのポリシーの例を表示するには、「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照してください。

アイデンティティによる認証

認証とは、アイデンティティ認証情報を使用して AWSにサインインする方法です。ユーザーは、AWS アカウントのルートユーザー、IAM ユーザーとして、または IAM ロールを引き受けることによって、認証される (AWSにサインインする) 必要があります。

ID ソースから提供された認証情報を使用して、フェデレーティッドアイデンティティとして AWS にサインインできます。AWS IAM Identity Centerフェデレーティッドアイデンティティの例としては、(IAM アイデンティティセンター) ユーザー、貴社のシングルサインオン認証、Google または Facebook の認証情報などがあります。フェデレーティッド ID としてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーションを使用して AWSにアクセスする場合、間接的にロールを引き受けることになります。

ユーザーのタイプに応じて、AWS Management Consoleまたは AWS アクセスポータルにサインインできます。AWS へのサインインの詳細については、「AWS サインイン ユーザーガイド」の「AWS アカウント にサインインする方法」を参照してください。

プログラムを使用して AWS にアクセスする場合、AWSは Software Development Kit (SDK) とコマンドラインインターフェイス (CLI) を提供し、認証情報を使用してリクエストに暗号で署名します。AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。リクエストに自分で署名する推奨方法の使用については、「IAM ユーザーガイド」の「API リクエストに対する AWS Signature Version 4」を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。例えば、AWS は、アカウントのセキュリティを強化するために多要素認証 (MFA) を使用することをお勧めします。詳細については、「AWS IAM Identity Center ユーザーガイド」の「多要素認証」および「IAM ユーザーガイド」の「IAM の AWS 多要素認証」を参照してください。

AWS アカウント のルートユーザー

AWS アカウント を作成する場合は、このアカウントのすべての AWS のサービス とリソースに対して完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。このアイデンティティは AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることによってアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、「IAM ユーザーガイド」の「ルートユーザー認証情報が必要なタスク」を参照してください。

フェデレーティッドアイデンティティ

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに対し、ID プロバイダーとのフェデレーションを使用して、一時的な認証情報の使用により、AWS のサービスにアクセスすることを要求します。

フェデレーティッド ID は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、Identity Center ディレクトリのユーザーか、または ID ソースから提供された認証情報を使用して AWS のサービス にアクセスするユーザーです。フェデレーティッド ID が AWS アカウントにアクセスすると、ロールが継承され、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、AWS IAM Identity Centerを使用することをお勧めします。IAM アイデンティティセンター でユーザーとグループを作成するか、すべての AWS アカウント とアプリケーションで使用するために、独自の ID ソースで一連のユーザーとグループに接続して同期することもできます。IAM Identity Center の詳細については、「AWS IAM Identity Centerユーザーガイド」の「What is IAM Identity Center?」(IAM Identity Center とは) を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、1 人のユーザーまたは 1 つのアプリケーションに対して特定の許可を持つ AWS アカウント 内のアイデンティティです。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する許可を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時認証情報が提供されます。詳細については、「IAM ユーザーガイド」の「IAM ユーザーに関するユースケース」を参照してください。

IAM ロール

IAM ロールは、特定の許可を持つ、AWS アカウント内のアイデンティティです。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。AWS Management Console で IAM ロールを一時的に引き受けるには、ユーザーから IAM ロール に切り替える (コンソール) ことができます。ロールを引き受けるには、AWS CLI または AWS API オペレーションを呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「ロールを引き受けるための各種方法」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッド ID に許可を割り当てるには、ロールを作成してそのロールの許可を定義します。フェデレーティッド ID が認証されると、その ID はロールに関連付けられ、ロールで定義されている許可が付与されます。フェデレーションのロールについては、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。IAM Identity Center を使用する場合は、許可セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。アクセス許可セットの詳細については、「AWS IAM Identity Center User Guide」の「Permission sets」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の AWS のサービス では、(ロールをプロキシとして使用する代わりに) リソースにポリシーを直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、

「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

- クロスサービスアクセス権 - 一部の AWS のサービス では、他の AWS のサービス の機能を使用します。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの許可、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。
  - 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用して AWS でアクションを実行するユーザーは、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、AWS のサービスを呼び出すプリンシパルの権限を、AWS のサービスのリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストは、サービスが、完了するために他の AWS のサービス またはリソースとのやりとりを必要とするリクエストを受け取ったときにのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。
  - サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除することができます。詳細については、「IAM ユーザーガイド」の「AWS のサービス に許可を委任するロールを作成する」を参照してください。
  - サービスにリンクされたロール - サービスにリンクされたロールは、AWS のサービス にリンクされたサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスにリンクされたロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスリンクロールのアクセス許可を表示できますが、編集することはできません。
- Amazon EC2 で実行されているアプリケーション - EC2 インスタンスで実行され、AWS CLI または AWS API リクエストを行っているアプリケーションの一時的な認証情報を管理するには、IAM ロールを使用できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。AWS ロールを EC2 インスタンスに割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスに添付されたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して許可を付与する」を参照してください。

ポリシーを使用したアクセス権の管理

AWS でアクセスを制御するには、ポリシーを作成して AWS ID またはリソースにアタッチします。ポリシーは AWS のオブジェクトであり、アイデンティティやリソースに関連付けて、これらのアクセス許可を定義します。AWS は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うと、これらのポリシーを評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。大半のポリシーは JSON ドキュメントとして AWS に保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き受けることができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの許可を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。このポリシーがあるユーザーは、AWS Management Console、AWS CLI、または AWS API からロール情報を取得できます。

アイデンティティベースのポリシー

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

アイデンティティベースのポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれます。マネージドポリシーは、AWS アカウント内の複数のユーザー、グループ、およびロールにアタッチできるスタンドアロンポリシーです。マネージドポリシーには、AWS マネージドポリシーとカスタマーマネージドポリシーがあります。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービス を含めることができます。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは IAM の AWSマネージドポリシーは使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、AWS WAF、および Amazon VPC は、ACL をサポートするサービスの例です。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS では、他の一般的ではないポリシータイプをサポートしています。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- アクセス許可の境界 - アクセス許可の境界は、アイデンティティベースポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとそのアクセス許可の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、アクセス許可の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。アクセス許可の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティのアクセス許可の境界」を参照してください。
- サービスコントロールポリシー (SCP) - SCP は、AWS Organizations で組織や組織単位 (OU) の最大許可を指定する JSON ポリシーです。AWS Organizations は、お客様が所有する複数の AWS

アカウントをグループ化し、一元的に管理するサービスです。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP はメンバーアカウントのエンティティに対する権限を制限します (各 AWS アカウントのルートユーザー など)。Organizations と SCP の詳細については、「AWS Organizations ユーザーガイド」の「サービスコントロールポリシー (SCP)」を参照してください。

- リソースコントロールポリシー (RCP) – RCP は、所有する各リソースにアタッチされた IAM ポリシーを更新することなく、アカウント内のリソースに利用可能な最大数のアクセス許可を設定するために使用できる JSON ポリシーです。RCP は、メンバーアカウントのリソースの許可を制限し、組織に属するかどうかにかかわらず、AWS アカウントのルートユーザー を含む ID のための有効な許可に影響を及ぼす可能性があります。RCP をサポートする AWS のサービス のリストを含む Organizations と RCP の詳細については、「AWS Organizations ユーザーガイド」の「リソースコントロールポリシー (RCP)」を参照してください。
- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もあります。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。複数のポリシータイプが関連するとき、リクエストを許可するかどうかを AWS が決定する方法の詳細については、「IAM ユーザーガイド」の「ポリシーの評価ロジック」を参照してください。

Amazon Aurora DSQL と IAM の連携

IAM を使用して Aurora DSQL へのアクセスを管理する前に、Aurora DSQL で利用できる IAM の機能について学びます。

Amazon Aurora DSQL で使用できる IAM の機能

IAM の機能	Aurora DSQL のサポート
アイデンティティベースポリシー	はい

IAM の機能	Aurora DSQL のサポート
リソースベースのポリシー	いいえ
ポリシーアクション	はい
ポリシーリソース	あり
ポリシー条件キー	Yes
ACL	いいえ
ABAC (ポリシー内のタグ)	あり
一時的な認証情報	はい
プリンシパル権限	はい
サービスロール	はい
サービスリンクロール	はい

Aurora DSQL およびその他の AWS サービスと多くの IAM 機能の連携についての概要は、「IAM ユーザーガイド」の「IAM と連携する AWS のサービス」を参照してください。

Aurora DSQL のアイデンティティベースのポリシー

アイデンティティベースのポリシーのサポート: あり

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。ID ベースのポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。プリンシパルは、それが添付されているユーザーまたはロールに適用されるため、アイデンティティベースのポリシーでは指定できません。JSON ポリシーで使用できるすべての要素について学ぶには、「IAM ユーザーガイド」の「IAM JSON ポリシーの要素のリファレンス」を参照してください。

Aurora DSQL のアイデンティティベースのポリシー例

Aurora DSQL でのアイデンティティベースのポリシーの例は、「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照してください。

Aurora DSQL 内のリソースベースのポリシー

リソースベースのポリシーのサポート: なし

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービス を含めることができます。

クロスアカウントアクセスを有効にするには、全体のアカウント、または別のアカウントの IAM エンティティを、リソースベースのポリシーのプリンシパルとして指定します。リソースベースのポリシーにクロスアカウントのプリンシパルを追加しても、信頼関係は半分しか確立されない点に注意してください。プリンシパルとリソースが異なる AWS アカウント にある場合、信頼できるアカウントの IAM 管理者は、リソースにアクセスするための権限をプリンシパルエンティティ (ユーザーまたはロール) に付与する必要もあります。IAM 管理者は、アイデンティティベースのポリシーをエンティティにアタッチすることで権限を付与します。ただし、リソースベースのポリシーで、同じアカウントのプリンシパルへのアクセス権が付与されている場合は、アイデンティティベースのポリシーをさらに付与する必要はありません。詳細については、「IAM ユーザーガイド」の「IAM でのクロスアカウントリソースアクセス」を参照してください。

Aurora DSQL のポリシーアクション

ポリシーアクションのサポート:あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対して、どのような条件下でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連する AWS API オペレーション

と同じです。一致する API オペレーションのない許可のみのアクションなど、いくつかの例外があります。また、ポリシーに複数のアクションが必要なオペレーションもあります。これらの追加アクションは依存アクションと呼ばれます。

このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

Aurora DSQL アクションのリストを確認するには、「サービス認可リファレンス」の「Actions Defined by Amazon Aurora DSQL」を参照してください。

Aurora DSQL のポリシーアクションは、アクションの前に次のプレフィックスを使用します。

```
dsql
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

Aurora DSQL でのアイデンティティベースのポリシーの例は、「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照してください。

Aurora DSQL のポリシーリソース

ポリシーリソースのサポート: あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

`Resource` JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ステートメントには`Resource` または `NotResource` 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの許可と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの権限をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

Aurora DSQL のリソースタイプとその ARN のリストを確認するには、「サービス認可リファレンス」の「Resources Defined by Amazon Aurora DSQL」を参照してください。どのアクションで各リソースの ARN を指定できるかについては、「Actions Defined by Amazon Aurora DSQL」を参照してください。

Aurora DSQL でのアイデンティティベースのポリシーの例は、「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照してください。

Aurora DSQL のポリシー条件キー

サービス固有のポリシー条件キーのサポート: あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

`Condition` 要素 (または `Condition` ブロック) を使用すると、ステートメントが有効な条件を指定できます。`Condition` 要素はオプションです。イコールや未満などの 条件演算子 を使用して条件式を作成して、ポリシーの条件とリクエスト内の値を一致させることができます。

1 つのステートメントに複数の `Condition` 要素を指定する場合、または 1 つの `Condition` 要素に複数のキーを指定する場合、AWS では `AND` 論理演算子を使用してそれらを評価します。単一の条件キーに複数の値を指定する場合、AWS では `OR` 論理演算子を使用して条件を評価します。ステートメントの権限が付与される前にすべての条件が満たされる必要があります。

条件を指定する際にプレースホルダー変数も使用できます。例えば IAM ユーザーに、IAM ユーザー名がタグ付けされている場合のみリソースにアクセスできる権限を付与することができます。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの要素: 変数およびタグ」を参照してください。

AWS はグローバル条件キーとサービス固有の条件キーをサポートしています。すべての AWS グローバル条件キーを確認するには、IAM ユーザーガイドのAWS グローバル条件コンテキストキーを参照してください。

Aurora DSQL の条件キーのリストを確認するには、「サービス認可リファレンス」の「Condition keys for Amazon Aurora DSQL」を参照してください。条件キーを使用できるアクションとリソースについては、「Actions defined by Amazon Aurora DSQL」を参照してください。

Aurora DSQL でのアイデンティティベースのポリシーの例は、「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照してください。

Aurora DSQL の ACL

ACL のサポート: なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Aurora DSQL での ABAC

ABAC (ポリシー内のタグ) のサポート: あり

属性ベースのアクセス制御 (ABAC) は、属性に基づいてアクセス許可を定義する認可戦略です。AWS では、属性はタグと呼ばれます。タグは、IAM エンティティ (ユーザーまたはロール)、および多数の AWS リソースにアタッチできます。エンティティとリソースのタグ付けは、ABAC の最初の手順です。その後、プリンシパルのタグがアクセスしようとしているリソースのタグと一致した場合にオペレーションを許可するように ABAC ポリシーをします。

ABAC は、急成長する環境やポリシー管理が煩雑になる状況で役立ちます。

タグに基づいてアクセスを管理するには、`aws:ResourceTag/key-name`、`aws:RequestTag/key-name`、または `aws:TagKeys` の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。

サービスがすべてのリソースタイプに対して 3 つの条件キーすべてをサポートする場合、そのサービスの値はありです。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、「IAM ユーザーガイド」の「ABAC 認可でアクセス許可を定義する」を参照してください。ABAC をセットアップする手順を説明するチュートリアルについては、「IAM ユーザーガイド」の「属性ベースのアクセスコントロール (ABAC) を使用する」を参照してください。

Aurora DSQLでの一時的な認証情報の使用

一時的な認証情報のサポート: あり

AWS のサービスには、一時的な認証情報を使用してサインインしても機能しないものがあります。一時的な認証情報を利用できる AWS のサービス を含めた詳細情報については、「IAM ユーザーガイド」の「IAM と連携する AWS のサービス」を参照してください。

ユーザー名とパスワード以外の方法で AWS Management Console にサインインする場合は、一時的な認証情報を使用していることになります。例えば、会社のシングルサインオン (SSO) リンクを使用して AWS にアクセスすると、そのプロセスは自動的に一時認証情報を作成します。また、ユーザーとしてコンソールにサインインしてからロールを切り替える場合も、一時的な認証情報が自動的に作成されます。ロールの切り替えに関する詳細については、「IAM ユーザーガイド」の「ユーザーから IAM ロールに切り替える (コンソール)」を参照してください。

一時認証情報は、AWS CLI または AWS API を使用して手動で作成できます。作成後、一時認証情報を使用して AWS にアクセスできるようになります。AWSは、長期的なアクセスキーを使用する代わりに、一時認証情報を動的に生成することをお勧めします。詳細については、「IAM の一時的セキュリティ認証情報」を参照してください。

Aurora DSQL のクロスサービスプリンシパル許可

転送アクセスセッション (FAS) のサポート: あり

IAM ユーザーまたはロールを使用して AWS でアクションを実行するユーザーは、プリンシパルとみなされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、AWS のサービスを呼び出すプリンシパルの権限を、AWS のサービスのリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストは、サービスが、完了するために他の AWS のサービス またはリソースとのやりとりを必要とするリクエストを受け取ったときにのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

Aurora DSQL のサービスロール

サービスロールのサポート: あり

サービスロールとは、サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービス に許可を委任するロールを作成する」を参照してください。

> **Warning**
> サービスロールのアクセス許可を変更すると、Aurora DSQL の機能が破損する可能性があります。Aurora DSQL が指示する場合以外は、サービスロールを編集しないでください。

Aurora DSQL のサービスリンクロール

サービスリンクロールのサポート: あり

サービスにリンクされたロールは、AWS のサービスにリンクされているサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスにリンクされたロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスリンクロールのアクセス許可を表示できますが、編集することはできません。

Aurora DSQL でのサービスリンクロールの作成または管理の詳細については、「Aurora DSQL でのサービスリンクロールの使用」を参照してください。

Amazon Aurora DSQL のアイデンティティベースのポリシー例

デフォルトでは、ユーザーとロールには Aurora DSQL リソースを作成または変更するアクセス許可がありません。また、AWS Management Console、AWS Command Line Interface (AWS CLI)、または AWS API を使用してタスクを実行することもできません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き継ぐことができます。

これらサンプルの JSON ポリシードキュメントを使用して、IAM アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーを作成する (コンソール)」を参照してください。

Aurora DSQL が定義するアクションとリソースタイプ (リソースタイプごとの ARN の形式を含む) の詳細については、「サービス認証リファレンス」の「Actions, Resources, and Condition Keys for Amazon Aurora DSQL 」を参照してください。

トピック

- ポリシーに関するベストプラクティス
- Aurora DSQL コンソールの使用

- 自分の権限の表示をユーザーに許可する

ポリシーに関するベストプラクティス

アイデンティティベースのポリシーは、誰がユーザーのアカウントの Aurora DSQL リソースを作成、アクセス、削除できるどうかを決定します。これらのアクションを実行すると、AWS アカウントに料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS マネージドポリシーを使用して開始し、最小特権の許可に移行する – ユーザーとワークロードへの許可の付与を開始するには、多くの一般的なユースケースのために許可を付与する AWS マネージドポリシーを使用します。これらは AWS アカウントで使用できます。ユースケース別に AWS カスタマーマネージドポリシーを定義して、マネージドポリシーを絞り込むことをお勧めします。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「ジョブ機能の AWS マネージドポリシー」を参照してください。
- 最小特権を適用する – IAM ポリシーで許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーとアクセス許可」を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する - ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。また、AWS CloudFormation などの特定の AWS のサービス を介して使用する場合、条件を使用してサービスアクションへのアクセスを許可することもできます。詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素:条件」を参照してください。
- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer でポリシーを検証する」を参照してください。
- 多要素認証 (MFA) を要求する – AWS アカウントで IAM ユーザーまたはルートユーザーを要求するシナリオがある場合は、セキュリティを強化するために MFA をオンにします。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細

については、「IAM ユーザーガイド」の「MFA を使用した安全な API アクセス」を参照してください。

IAM でのベストプラクティスの詳細については、「IAM ユーザーガイド」の「IAM でのセキュリティのベストプラクティス」を参照してください。

Aurora DSQL コンソールの使用

Amazon Aurora DSQL コンソールにアクセスするには、アクセス許可の最小限のセットが必要です。これらのアクセス許可により、AWS アカウント の Aurora DSQL リソースの詳細をリストおよび表示できます。最小限必要なアクセス許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) ではコンソールが意図したとおりに機能しません。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソール権限を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスを許可します。

ユーザーとロールが引き続き Aurora DSQL コンソールを使用できるようにするには、エンティティに Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess` または `AmazonAuroraDSQLReadOnlyAccess` AWS マネージドポリシーもアタッチします。詳細については、「IAM ユーザーガイド」の「ユーザーへのアクセス許可の追加」を参照してください。

自分の権限の表示をユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI か AWS API を使用してプログラム的に、このアクションを完了する権限が含まれています。

```
{
    "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": "*"
        }
    ]
}
```

Amazon Aurora DSQL のアイデンティティおよびアクセスのトラブルシューティング

以下の情報を使用して、Aurora DSQL と IAM の使用時に発生する可能性がある一般的な問題の診断と修正に役立てます。

トピック

- Aurora DSQL でアクションを実行する権限がない
- iam:PassRole を実行する権限がありません
- 自分の AWS アカウント以外のユーザーに Aurora DSQL リソースへのアクセスを許可したい

Aurora DSQL でアクションを実行する権限がない

アクションを実行する権限がないというエラーが表示された場合は、そのアクションを実行できるようにポリシーを更新する必要があります。

次の例のエラーは、mateojackson がコンソールを使用して *my-dsql-cluster* リソースの詳細を表示する際に *GetCluster* アクセス許可を持っていないときに発生します。

```
User: iam:::user/mateojackson is not authorized to perform: GetCluster on resource: my-dsql-cluster
```

この場合、*GetCluster* アクションを使用して *my-dsql-cluster*リソースへのアクセスを許可するように、mateojackson ユーザーのポリシーを更新する必要があります。

サポートが必要な場合は、 管理者に問い合わせてください。サインイン認証情報を提供した担当者が管理者です。

iam:PassRole を実行する権限がありません

iam:PassRole アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して Aurora DSQL にロールを渡せるようにする必要があります。

一部の AWS のサービスでは、新しいサービスロールまたはサービスリンクロールを作成せずに、既存のロールをサービスに渡すことが許可されています。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して Aurora DSQL でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。メアリーには、ロールをサービスに渡す許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに iam:PassRole アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者に問い合わせてください。サインイン認証情報を提供した担当者が管理者です。

自分の AWS アカウント以外のユーザーに Aurora DSQL リソースへのアクセスを許可したい

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまた

はアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください。

- Aurora DSQL がこれらの機能をサポートしているか確認するには、「Amazon Aurora DSQL と IAM の連携」を参照してください。
- 所有している AWS アカウント 全体のリソースへのアクセス権を提供する方法については、「IAM ユーザーガイド」の「所有している別の AWS アカウント へのアクセス権を IAM ユーザーに提供」を参照してください。
- サードパーティーの AWS アカウント にリソースへのアクセス権を提供する方法については、「IAM ユーザーガイド」の「サードパーティーが所有する AWS アカウント へのアクセス権を付与する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、「IAM ユーザーガイド」の「外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可」を参照してください。
- クロスアカウントアクセスにおけるロールとリソースベースのポリシーの使用方法の違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

Aurora DSQL でのサービスリンクロールの使用

Aurora DSQL は AWS Identity and Access Management (IAM) サービスにリンクされたロールを使用します。サービスリンクロールは、Aurora DSQL に直接リンクされた一意のタイプの IAM ロールです。サービスにリンクされたロールは、Aurora DSQL によって事前定義されており、サービスがお客様の Aurora DSQL クラスターの代わりに AWS のサービスを呼び出すために必要なアクセス許可がすべて含まれています。

サービスにリンクされたロールは、必要なアクセス許可を手動で追加する必要がないため、セットアッププロセスが簡素化されます。クラスターを作成すると、サービスにリンクされたロールが Aurora DSQL により自動的に作成されます。このサービスにリンクされたロールを削除するには、クラスターをすべて削除する必要があります。これにより、Aurora DSQL リソースへの必要なアクセス許可を不注意に削除することがなくなり、リソースが保護されます。

サービスにリンクされたロールをサポートする他のサービスについては、「IAM と連携する AWS のサービス」の「サービスにリンクされたロール」列が「はい」になっているサービスを検索してください。サービスリンクロールに関するドキュメントをサービスで表示するには、リンクで [はい] を選択します。

サービスにリンクされたロールは、サポートされているすべての Aurora DSQL リージョンで使用できます。

Aurora DSQL のサービスにリンクされたロールのアクセス許可

Aurora DSQL は、`AWSServiceRoleForAuroraDsql` という名前のサービスにリンクされたロールを使用します。Amazon Aurora DSQL がユーザーに代わって AWS リソースを作成および管理できるようにします。このサービスにリンクされたロールは、AuroraDsqlServiceLinkedRolePolicy マネージドポリシーにアタッチされます。

> **Note**
>
> サービスリンク役割の作成、編集、削除を IAM エンティティ (ユーザー、グループ、役割など) に許可するにはアクセス許可を設定する必要があります。`You don't have the permissions to create an Amazon Aurora DSQL service-linked role` というエラーメッセージが返される場合があります。このメッセージが表示された場合は、次のアクセス許可が有効であることを確認します。
>
> ```
> {
> "Sid" : "CreateDsqlServiceLinkedRole",
>  "Effect" : "Allow",
>  "Action" : "iam:CreateServiceLinkedRole",
>  "Resource" : "*",
>  "Condition" : {
>  "StringEquals" : {
>  "iam:AWSServiceName" : "dsql.amazonaws.com"
>  }
>  }
>  }
> ```
>
> 詳細については、「サービスにリンクされたロールのアクセス許可」を参照してください。

サービスにリンクされたロールの作成

AuroraDSQLServiceLinkedRolePolicy サービスにリンクされたロールを手動で作成する必要はありません。Aurora DSQL は、サービスにリンクされたロールを自動的に作成します。AuroraDSQLServiceLinkedRolePolicy サービスにリンクされたロールがアカウントから削除されている場合、新しい Aurora DSQL クラスターを作成すると、Aurora DSQL がロールを作成します。

サービスにリンクされたロールの編集

Aurora DSQL では、AuroraDSQLServiceLinkedRolePolicy サービスにリンクされたロールを編集することはできません。サービスリンクロールを作成すると、多くのエンティティによってロールが参照される可能性があるため、ロール名を変更することはできません。ただし、ロールの説明は IAM コンソール、AWS Command Line Interface (AWS CLI)、または IAM API を使用して編集することができます。

サービスにリンクされたロールを削除

サービスリンクロールを必要とする機能やサービスが不要になった場合は、ロールを削除することをお勧めします。そうすることで、使用していないエンティティがアクティブにモニタリングされたり、メンテナンスされたりすることがなくなります。

アカウントのサービスにリンクされたロールを削除するには、アカウントのクラスターを削除する必要があります。

サービスにリンクされたロールは、IAM コンソール、AWS CLI、 IAM API を使用して削除することができます。詳細については、「IAM ユーザーガイド」の「サービスにリンクされたロールの作成」を参照してください。

Aurora DSQL のサービスにリンクされたロールをサポートするリージョン

Aurora DSQL は、このサービスが利用可能なすべてのリージョンで、サービスにリンクされたロールの使用をサポートしています。詳細については、「AWS リージョンとエンドポイント」を参照してください。

Amazon Aurora DSQL での IAM 条件キーの使用

Aurora DSQL でアクセス許可を付与するときは、アクセス許可ポリシーを有効にする方法を決める条件を指定できます。以下に示しているのは、Aurora DSQL アクセス許可ポリシーでの条件キーの使用例です。

例 1: 特定の AWS リージョンにクラスターを作成するアクセス許可を付与する

次のポリシーは、米国東部 (バージニア北部) および米国東部 (オハイオ) リージョンにクラスターを作成するアクセス許可を付与します。このポリシーはリソース ARN を使用して許可されたリージョ

ンを制限するため、Aurora DSQL はポリシーの `Resource` セクションでその ARN が指定されている場合にのみクラスターを作成できます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            # Control where clusters can be created
            "Action": ["CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

例 2: 特定の AWS リージョンにマルチリージョンクラスターを作成するアクセス許可を付与する

次のポリシーは、米国東部 (バージニア北部) および米国東部 (オハイオ) リージョンにマルチリージョンクラスターを作成するアクセス許可を付与します。このポリシーはリソース ARN を使用して許可されたリージョンを制限するため、Aurora DSQL はポリシーの `Resource` セクションでこの ARN が指定されている場合にのみマルチリージョンクラスターを作成できます。マルチリージョンクラスターを作成するには、指定された各リージョンで `PutMultiRegionProperties`、`PutWitnessRegion`、および `AddPeerCluster` のアクセス許可も必要であることに注意してください。

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "dsql:CreateCluster",
          "dsql:PutMultiRegionProperties",
          "dsql:PutWitnessRegion",
          "dsql:AddPeerCluster"
        ],
        "Resource": [
```

```
            "arn:aws:dsql:us-east-1:123456789012:cluster/*",
            "arn:aws:dsql:us-east-2:123456789012:cluster/*"
        ]
      }
    ]
}
```

例 3: 特定の監視リージョンを持つマルチリージョンクラスターを作成するアクセス許可を付与する

次のポリシーでは、Aurora DSQL `dsql:WitnessRegion` 条件キーを使用し、米国西部 (オレゴン) に監視リージョンを持つマルチリージョンクラスターをユーザーが作成できるようにします。`dsql:WitnessRegion` 条件を指定しない場合は、任意のリージョンを監視リージョンとして使用できます。

```
{
    "Version": "2012-10-17",
      "Statement": [
          {
            "Effect": "Allow",
            "Action": [
                "dsql:CreateCluster",
                "dsql:PutMultiRegionProperties",
                "dsql:AddPeerCluster"
             ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
          },
          {
            "Effect": "Allow",
            "Action": [
                "dsql:PutWitnessRegion"
            ]
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": [
                        "us-west-2"
                     ]
                }

            }
          }
```

```
        ]
}
```

Amazon Aurora DSQL でのインシデント対応

AWS では、セキュリティが最優先事項です。AWS クラウドの責任共有モデルの一環として、AWS はセキュリティを最も重視する組織の要件を満たすデータセンターとネットワーク、ソフトウェアアーキテクチャを管理します。AWS は、Amazon Aurora DSQL サービス自体に関連するインシデント対応に責任を負います。また、AWS 顧客は、クラウドのセキュリティを維持する責任を共有します。つまり、アクセスできる AWS ツールや機能から、実装するセキュリティを制御できるということです。また、ユーザーは責任共有モデルのユーザー側でインシデント対応を行う責任があります。

クラウド上で稼働するアプリケーションの目標を満たすセキュリティベースラインを確立することで、対応可能な逸脱を検出できます。インシデント対応と選択が企業目標に与える影響を理解するために、次のリソースをご確認ください。

- AWS セキュリティインシデント対応ガイド
- AWS のセキュリティ、アイデンティティ、コンプライアンスのベストプラクティス
- AWS クラウド導入フレームワーク (CAF) のセキュリティのパースペクティブ ホワイトペーパー

Amazon GuardDuty は、悪意のある動作や不正な動作を継続的にモニタリングするマネージド脅威検出サービスです。これにより、顧客が AWS アカウントとワークロードを保護し、インシデントにエスカレートする前に疑わしいアクティビティを特定することができます。異常な API コールや、アカウントやリソースが侵害された可能性、悪意のある人物による偵察の可能性を示す不正なデプロイの可能性などのアクティビティをモニタリングします。例えば、Amazon GuardDuty は、新しい場所からログインして新しいクラスターを作成するユーザーなど、Amazon Aurora DSQL API での疑わしいアクティビティを検出できます。

Amazon Aurora DSQL のコンプライアンス検証

AWS のサービス が特定のコンプライアンスプログラムの対象であるかどうかを確認するには、コンプライアンスプログラムによる対象範囲内の AWS のサービス のサービスをご覧いただき、関心のあるコンプライアンスプログラムを選択してください。一般的な情報については、「AWSコンプライアンスプログラム」を参照してください。

AWS Artifact を使用して、サードパーティーの監査レポートをダウンロードできます。詳細については、「AWS Artifact でレポートをダウンロードする」を参照してください。

AWS のサービス を使用する際のユーザーのコンプライアンス責任は、ユーザーのデータの機密性や貴社のコンプライアンス目的、適用される法律および規制によって決まります。AWS では、コンプライアンスに役立つ次のリソースを提供しています。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスの機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべての AWS のサービスが HIPAA 適格であるわけではありません。
- 「AWS コンプライアンスのリソース」 – このワークブックおよびガイドのコレクションは、顧客の業界と拠点に適用されるものである場合があります。
- AWS Customer Compliance Guide - コンプライアンスの観点から見た責任共有モデルを理解できます。このガイドは、AWS のサービスを保護するためのベストプラクティスを要約したものであり、複数のフレームワーク (米国標準技術研究所 (NIST)、ペイメントカード業界セキュリティ標準評議会 (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティ統制へのガイダンスがまとめられています。
- 「AWS Config デベロッパーガイド」の「ルールでのリソースの評価」 - AWS Config サービスは、自社のプラクティス、業界ガイドライン、および規制に対するリソースの設定の準拠状態を評価します。
- AWS Security Hub – この AWS のサービス は、AWS 内のセキュリティ状態の包括的なビューを提供します。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。
- Amazon GuardDuty – この AWS のサービスは、環境をモニタリングして、疑わしいアクティビティや悪意のあるアクティビティがないか調べることで、AWS アカウント、ワークロード、コンテナ、データに対する潜在的な脅威を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager - この AWS のサービスは、AWS の使用状況を継続的に監査して、リスクの管理方法や、規制および業界標準へのコンプライアンスの管理方法を簡素化するために役立ちます。

Amazon Aurora DSQL の耐障害性

AWS グローバルインフラストラクチャは AWS リージョンおよびアベイラビリティーゾーン (AZ) を中心に構築されています。AWS リージョンは、低レイテンシー、高スループット、そして高度な冗長ネットワークで接続される物理的に独立、隔離された複数のアベイラビリティーゾーンを提供します。アベイラビリティーゾーンでは、ゾーン間で中断することなく自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、および拡張性が優れています。Aurora DSQL は、データベースの可用性を最大限に高めながら、AWS リージョンインフラストラクチャを活用できるように設計されています。デフォルトでは、Aurora DSQL の単一リージョンクラスターにはマルチ AZ の可用性があり、完全な AZ へのアクセスに影響を与える可能性のある主要なコンポーネントの障害やインフラストラクチャの中断に対する耐性を提供します。マルチリージョンクラスターは、AWS リージョンがアプリケーションクライアントにアクセスできない場合でも、マルチ AZ の耐障害性のすべての利点を提供しながら、強力な整合性のあるデータベースの可用性を提供します。

AWS リージョンとアベイラビリティーゾーンの詳細については、「AWS グローバルインフラストラクチャ」を参照してください。

AWS グローバルインフラストラクチャに加えて、Aurora DSQL は、データの耐障害性とバックアップのニーズに対応できるように複数の機能を提供しています。

バックアップと復元

Aurora DSQL は、AWS Backup コンソールによるバックアップと復元をサポートしています。単一リージョンクラスターとマルチリージョンクラスターに対してフルバックアップと復元を実行できます。詳細については、「Amazon Aurora DSQL のバックアップと復元」を参照してください。

レプリケーション

設計上、Aurora DSQL はすべての書き込みトランザクションを分散トランザクションログにコミットし、コミットされたすべてのログデータを 3 つの AZ のユーザーストレージレプリカに同期的にレプリケートします。マルチリージョンクラスターは、読み取りリージョンと書き込みリージョン間の完全なクロスリージョンレプリケーション機能を提供します。

指定されたウィットネスリージョンは、トランザクションログのみの書き込みをサポートし、ストレージを使用しません。監視リージョンにはエンドポイントがありません。つまり、監視リージョンは暗号化されたトランザクションログのみを保存し、管理や設定を不要で、ユーザーがアクセスできないということです。

Aurora DSQL トランザクションログとユーザーストレージは分散され、すべてのデータが単一の論理ボリュームとして Aurora DSQL クエリプロセッサに提示されます。Aurora DSQL は、データベースのプライマリキーの範囲とアクセスパターンに基づいて、データを自動的に分割、マージ、レプリケートします。Aurora DSQL は、読み取りアクセス頻度に基づいて、リードレプリカのスケールアップとスケールダウンを自動的に行います。

クラスターストレージレプリカは、マルチテナントストレージフリートに分散されます。コンポーネントまたは AZ に障害が発生した場合、Aurora DSQL は自動的にアクセスを正常なコンポーネントへリダイレクトし、欠落しているレプリカを非同期的に修復します。Aurora DSQL が障害のあるレプリカを修正すると、Aurora DSQL は自動的にそれらをストレージクォーラムに追加し、クラスターで使用できるようにします。

高可用性

デフォルトでは、Aurora DSQL の単一リージョンクラスターとマルチリージョンクラスターはアクティブ/アクティブであり、クラスターを手動でプロビジョニング、設定、または再設定する必要はありません。Aurora DSQL はクラスター復元を完全に自動化するため、従来のプライマリ/セカンダリフェイルオーバーオペレーションが不要になります。レプリケーションは常に同期され、複数の AZ で実行されるため、レプリケーションの遅延や障害復旧中の非同期セカンダリデータベースへのフェイルオーバーによるデータ損失のリスクはありません。

単一リージョンクラスターは、3 つの AZ 間で強力なデータ整合性を持つ同時アクセスを自動的に有効にするマルチ AZ 冗長エンドポイントを提供します。つまり、これら 3 つの AZ のいずれかにあるユーザーストレージレプリカは、常に同じ結果を 1 つ以上のリーダーに返し、常に書き込みを受信できます。この強力な整合性とマルチ AZ の耐障害性は、Aurora DSQL マルチリージョンクラスターのすべてのリージョンで利用できます。つまり、マルチリージョンクラスターは強力な整合性を備えたリージョンエンドポイントを 2 つ提供するため、クライアントはいずれのリージョンに対してもコミット時のレプリケーションラグなしで区別なく読み書きできます。

Aurora DSQL は、単一リージョンクラスターでは 99.99%、マルチリージョンクラスターでは 99.999% の可用性を提供します。

Amazon Aurora DSQL でのインフラストラクチャセキュリティ

マネージドサービスとして、Amazon Aurora DSQL は、「セキュリティ、アイデンティティ、コンプライアンスに関するベストプラクティス」で説明されている AWS グローバルネットワークセキュリティ手順によって保護されます。

ネットワーク経由で Aurora DSQL にアクセスするには、AWS が公開している API コールを使用します。クライアントは、Transport Layer Security (TLS) 1.2 以降をサポートする必要があります。また、DHE (Ephemeral Diffie-Hellman) や ECDHE (Elliptic Curve Ephemeral Diffie-Hellman) などの Perfect Forward Secrecy (PFS) を使用した暗号スイートもクライアントでサポートされている必要があります。これらのモードはJava 7 以降など、ほとんどの最新システムでサポートされています。

また、リクエストにはアクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。またはAWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

AWS PrivateLink を使用した Amazon Aurora DSQL クラスターの管理と接続

Amazon Aurora DSQL 用の AWS PrivateLink を使用すると、Amazon Virtual Private Cloud でインターフェイス Amazon VPC エンドポイント (インターフェイスエンドポイント) をプロビジョニングできます。これらのエンドポイントには、オンプレミスにあるアプリケーションから、Amazon VPC および AWS Direct Connect 経由で、または別の AWS リージョンにあるアプリケーションから Amazon VPC ピアリング経由で直接アクセスできます。AWS PrivateLink とインターフェイスエンドポイントを使用することで、アプリケーションから、Aurora DSQL へのプライベートネットワーク接続を簡素化できます。

Amazon VPC 内のアプリケーションは、パブリック IP アドレスを必要とせずに、Amazon VPC インターフェイスエンドポイントを使用して Aurora DSQL にアクセスできます。

インターフェイスエンドポイントは、Amazon VPC 内のサブネットからプライベート IP アドレスが割り当てられた 1 つ以上の Elastic Network Interface (ENI) で表されます。インターフェイスエンドポイントを介した Aurora DSQL へのリクエストは、AWS ネットワークに残ります。Amazon VPC をオンプレミスネットワークに接続する方法の詳細については、「AWS Direct Connect ユーザーガイド」および「AWS Site-to-Site VPN VPN ユーザーガイド」を参照してください。

インターフェイスエンドポイントに関する一般的な情報については、「AWS PrivateLink ユーザーガイド」の「Access an AWS service using an interface Amazon VPC endpoint」を参照してください。

Aurora DSQL で使用される Amazon VPC エンドポイントのタイプ

Aurora DSQL には 2 つの異なるタイプの AWS PrivateLink エンドポイントが必要です。

1. 管理エンドポイント — このエンドポイントは、Aurora DSQL クラスターの `get`、`create`、`update`、`delete`、`list` などの管理オペレーションに使用されます。「AWS PrivateLink を使用した Aurora DSQL クラスターの管理」を参照してください。
2. 接続エンドポイント — このエンドポイントは、PostgreSQL クライアントを介して Aurora DSQL クラスターに接続するために使用されます。「AWS PrivateLink を使用した Aurora DSQL クラスターへの接続」を参照してください。

Aurora DSQL 用 AWS PrivateLink を使用する際の考慮事項

Amazon VPC に関する考慮事項は、Aurora DSQL 用 AWS PrivateLink に適用されます。詳細については、「AWS PrivateLink ガイド」の「Access an AWS service using an interface VPC endpoint」と「AWS PrivateLink quotas」を参照してください。

AWS PrivateLink を使用した Aurora DSQL クラスターの管理

AWS Command Line Interface または AWS Software Development Kit (SDK) を使用して、Aurora DSQL インターフェイスエンドポイントを介して Aurora DSQL クラスターを管理できます。

Amazon VPC エンドポイントの作成

Amazon VPC インターフェイスエンドポイントを作成するには、「AWS PrivateLink ガイド」の「Create an Amazon VPC endpoint」を参照してください。

```
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 \
```

Aurora DSQL API リクエストにデフォルトのリージョン DNS 名を使用するには、Aurora DSQL インターフェイスエンドポイントの作成時にプライベート DNS を無効にしないでください。プライベート DNS を有効にすると、Amazon VPC 内から行われた Aurora DSQL サービスへのリクエストは、パブリック DNS 名ではなく、Amazon VPC エンドポイントのプライベート IP アドレスに自動的に解決されます。プライベート DNS を有効にすると、Amazon VPC 内で行われた Aurora DSQL リクエストは Amazon VPC エンドポイントに自動的に解決されます。

プライベート DNS が有効になっていない場合は、AWS CLI コマンドで `--region` および `--endpoint-url` パラメータを使用して、Aurora DSQL インターフェイスエンドポイントを介して Aurora DSQL クラスターを管理します。

エンドポイント URL を使用したクラスターの一覧表示

次の例では、AWS リージョン `us-east-1` と Amazon VPC エンドポイント ID の DNS 名 `vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` をユーザー自身の情報に置き換えます。

```
aws dsql --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsql.us-east-1.vpce.amazonaws.com list-clusters
```

API オペレーション

Aurora DSQL でのリソースの管理に関するドキュメントについては、「Aurora DSQL API reference」を参照してください。

エンドポイントポリシーの管理

Amazon VPC エンドポイントポリシーを徹底的にテストして設定することで、Aurora DSQL クラスターの安全性、コンプライアンス、組織固有のアクセスコントロールやガバナンスの要件との整合性を確保できます。

例: 完全な Aurora DSQL アクセスポリシー

次のポリシーは、指定された Amazon VPC エンドポイントを介して、すべての Aurora DSQL アクションとリソースへのフルアクセスを許可します。

```
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": "*"
        }
      ]
```

```
    }'
```

例: 制限付き Aurora DSQL アクセスポリシー

次のポリシーでは、これらの Aurora DSQL アクションのみを許可します。

- `CreateCluster`
- `GetCluster`
- `ListClusters`

その他の Aurora DSQL アクションはすべて拒否されます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "*"
    }
  ]
}
```

AWS PrivateLink を使用した Aurora DSQL クラスターへの接続

AWS PrivateLink エンドポイントが設定され、アクティブになったら、PostgreSQL クライアントを使用して Aurora DSQL クラスターに接続できます。以下の接続手順では、AWS PrivateLink エンドポイント経由で接続するための適切なホスト名を作成する手順の概要を説明します。

AWS PrivateLink 接続エンドポイントの設定

ステップ 1: クラスターのサービス名を取得する

クラスターに接続するための AWS PrivateLink エンドポイントを作成するときは、まずクラスター固有のサービス名を取得する必要があります。

AWS CLI

```
aws dsql get-vpc-endpoint-service-name \
--region us-east-1 \
--identifier your-cluster-id
```

レスポンスの例

```
{
    "serviceName": "com.amazonaws.us-east-1.dsql-fnh4"
}
```

サービス名には、例にある `dsql-fnh4` のように識別子が含まれます。この識別子は、クラスターに接続するためのホスト名を作成するときにも必要です。

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);
```

ステップ 2: Amazon VPC エンドポイントを作成する

前の手順で取得したサービス名を使用して、Amazon VPC エンドポイントを作成します。

> **Important**
>
> 以下の接続手順は、プライベート DNS が有効になっている場合のクラスターへの接続にのみ動作します。エンドポイントの作成時に `--no-private-dns-enabled` フラグを使用しないでください。使用すると、以下の接続手順が正しく動作しなくなります。プライベート DNS を無効にする場合は、作成したエンドポイントを指すワイルドカードプライベート DNS レコードを独自に作成する必要があります。

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
```

レスポンスの例

```
{
    "VpcEndpoint": {
        "VpcEndpointId": "vpce-0123456789abcdef0",
        "VpcEndpointType": "Interface",
        "VpcId": "vpc-0123456789abcdef0",
        "ServiceName": "com.amazonaws.us-east-1.dsql-fnh4",
```

```
        "State": "pending",
        "RouteTableIds": [],
        "SubnetIds": [
            "subnet-0123456789abcdef0",
            "subnet-0123456789abcdef1"
        ],
        "Groups": [
            {
                "GroupId": "sg-0123456789abcdef0",
                "GroupName": "default"
            }
        ],
        "PrivateDnsEnabled": true,
        "RequesterManaged": false,
        "NetworkInterfaceIds": [
            "eni-0123456789abcdef0",
            "eni-0123456789abcdef1"
        ],
        "DnsEntries": [
            {
                "DnsName": "*.dsql-fnh4.us-east-1.vpce.amazonaws.com",
                "HostedZoneId": "Z7HUB22UULQXV"
            }
        ],
        "CreationTimestamp": "2025-01-01T00:00:00.000Z"
    }
}
```

SDK for Python

```
import boto3

ec2_client = boto3.client('ec2', region_name='us-east-1')
response = ec2_client.create_vpc_endpoint(
    VpcEndpointType='Interface',
    VpcId='your-vpc-id',
    ServiceName='com.amazonaws.us-east-1.dsql-fnh4',  # Use the service name from
 previous step
    SubnetIds=[
        'subnet-id-1',
        'subnet-id-2'
    ],
    SecurityGroupIds=[
```

```
        'security-group-id'
    ]
)

vpc_endpoint_id = response['VpcEndpoint']['VpcEndpointId']
print(f"VPC Endpoint created with ID: {vpc_endpoint_id}")
```

SDK for Java 2.x

Aurora DSQL API のエンドポイント URL を使用する

```
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);
```

AWS PrivateLink 接続エンドポイントを使用した Aurora DSQL クラスターへの接続

AWS PrivateLink エンドポイントが設定され、アクティブになったら (`State` が `available` になっていることを確認します)、PostgreSQL クライアントを使用して Aurora DSQL クラスターに接続できます。AWS SDK の使用方法については、「Programming with Aurora DSQL」のガイドを参照してください。クラスターエンドポイントをホスト名の形式に合わせて変更する必要があります。

ホスト名の作成

AWS PrivateLink を介して接続するためのホスト名は、パブリック DNS ホスト名とは異なります。次のコンポーネントを使用して作成する必要があります。

1. `Your-cluster-id`
2. サービス名のサービス識別子。例: `dsql-fnh4`
3. AWS リージョン

次の形式を使用します。*`cluster-id`*`.`*`service-identifier`*`.`*`region`*`.on.aws`

例: PostgreSQL を使用した接続

```
# 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
```

AWS PrivateLink で問題のトラブルシューティング

一般的な問題と解決策

次の表に、Aurora DSQL と AWS PrivateLink に関連する一般的な問題と解決策を示します。

問題	考えられる原因	ソリューション
接続タイムアウト	セキュリティグループが正しく設定されていない	Amazon VPC Reachability Analyzer を使用して、ネットワーク設定でポート 5432 のトラフィックが許可されていることを確認します。
DNS 解決の失敗	プライベート DNS が有効になっていない	Amazon VPC エンドポイントがプライベート DNS を有効にして作成されたことを確認します。
認証の失敗	認証情報が正しくないか、トークンの有効期限が切れている	新しい認証トークンを生成し、ユーザー名を確認します。
サービス名が見つからない	クラスター ID が正しくない	サービス名を取得するときに、クラスター ID と AWS リージョンを再確認します。

関連リソース

詳細については、以下のリソースを参照してください。

- Amazon Aurora DSQL ユーザーガイド
- AWS PrivateLink ドキュメント
- Access AWS services through AWS PrivateLink

Amazon Aurora DSQL での設定と脆弱性の分析

AWS は、ゲストオペレーティングシステム (OS) やデータベースへのパッチ適用、ファイアウォール設定、災害対策などの基本的なセキュリティタスクを処理します。これらの手順は適切な第三者によって確認され、証明されています。詳細については、以下のリソース を参照してください。

- 責任共有モデル
- アマゾン ウェブ サービス: セキュリティプロセスの概要 (ホワイトペーパー)

サービス間の混乱した代理の防止

混乱した代理問題とは、アクションを実行する許可を持たないエンティティが、より高い特権を持つエンティティにそのアクションの実行を強制できるというセキュリティ問題です。AWS では、サービス間でのなりすましによって、混乱した代理問題が発生する場合があります。サービス間でのなりすましは、1 つのサービス (呼び出し元サービス) が、別のサービス (呼び出し対象サービス) を呼び出すときに発生する可能性があります。呼び出し元サービスは、本来ならアクセスすることが許可されるべきではない方法でその許可を使用して、別のお客様のリソースに対する処理を実行するように操作される場合があります。これを防ぐため、AWS では、アカウントのリソースへのアクセス権が付与されたサービスプリンシパルで、すべてのサービスのデータを保護するために役立つツールを提供しています。

リソースポリシー内では `aws:SourceArn` および `aws:SourceAccount` のグローバル条件コンテキストキーを使用して、Amazon Aurora DSQL が別のサービスに付与する、リソースへのアクセス許可を制限することをお勧めします。クロスサービスアクセスにリソースを 1 つだけ関連付けたい場合は、`aws:SourceArn` を使用します。そのアカウント内のリソースをクロスサービスの使用に関連付けることを許可する場合は、`aws:SourceAccount` を使用します。

混乱した代理問題から保護するための最も効果的な方法は、リソースの完全な ARN を指定して、`aws:SourceArn` グローバル条件コンテキストキーを使用することです。リソースの完全な ARN が不明な場合や、複数のリソースを指定する場合には、グローバルコンテキスト条件キー `aws:SourceArn` で、ARN の未知部分を示すためにワイルドカード文字 (*) を使用します。例えば、`arn:aws:servicename:*:123456789012:*`。

`aws:SourceArn` の値に Amazon S3 バケット ARN などのアカウント ID が含まれていない場合は、両方のグローバル条件コンテキストキーを使用して、アクセス許可を制限する必要があります。

`aws:SourceArn` の値は ResourceDescription である必要があります。

次の例では、Aurora DSQL で `aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーを使用して、混乱した代理問題を回避する方法を示します。

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "servicename.amazonaws.com"
    },
```

```
    "Action": "servicename:ActionName",
    "Resource": [
      "arn:aws:servicename:::ResourceName/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:servicename:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

Aurora DSQL のセキュリティのベストプラクティス

Aurora DSQL には、独自のセキュリティポリシーを開発および実装する際に考慮する必要のあるいくつかのセキュリティ機能が用意されています。以下のベストプラクティスは一般的なガイドラインであり、完全なセキュリティソリューションを説明するものではありません。これらのベストプラクティスはお客様の環境に適切ではないか、十分ではない場合があるため、これらは指示ではなく、有用な考慮事項と見なしてください。

トピック

- Aurora DSQL のセキュリティのベストプラクティス
- Aurora DSQL の予防的セキュリティのベストプラクティス

Aurora DSQL のセキュリティのベストプラクティス

Aurora DSQL を安全に使用する以下の方法に加えて、クラウドテクノロジーがセキュリティをどのように向上させるかについては、AWS Well-Architected Tool の「セキュリティ」を参照してください。

Amazon CloudWatch アラーム

Amazon CloudWatch アラームを使用して、指定した期間中、1 つのメトリクスをモニタリングします。メトリクスが特定の閾値を超えると、Amazon SNS トピックまたは AWS Auto Scaling ポリシーに通知が送信されます。CloudWatch アラームは、特定の状態にあるという理由ではアクションを呼び出しません。状態が変わり、それが指定した期間だけ維持される必要があります。

識別とオートメーションのために Aurora DSQL リソースにタグを付ける

AWS のリソースにメタデータをタグ付け形式で割り当てることができます。各タグは、カスタマー定義のキーと値 (オプション) で構成されるシンプルなラベルです。タグを使用すると、リソースの管理、検索、フィルターが容易になります。

タグ付けを行うと、グループ化されたコントロールを実装できます。タグには固有のタイプはありませんが、 用途、所有者、環境などの基準でリソースを分類できます。次に例をいくつか示します。

- セキュリティ — 暗号化などの要件を決定するために使用されます。
- 機密性 — リソースがサポートするデータ機密性レベルの識別子。
- 環境 — 開発、テスト、本番稼働用インフラストラクチャを区別するために使用されます。

AWS のリソースにメタデータをタグ付け形式で割り当てることができます。各タグは、カスタマー定義のキーと値 (オプション) で構成されるシンプルなラベルです。タグを使用すると、リソースの管理、検索、フィルターが容易になります。

タグ付けを行うと、グループ化されたコントロールを実装できます。タグには、固有なタイプはありませんが、 リソースを用途、所有者、環境などの基準で分類できます。の例を以下に示します。

- セキュリティ — 暗号化などの要件を決定するために使用されます。
- 機密性 — リソースがサポートするデータ機密性レベルの識別子。
- 環境 — 開発、テスト、本番稼働用インフラストラクチャを区別するために使用されます。

詳細については、「Best Practices for Tagging AWS Resources」を参照してください。

Aurora DSQL の予防的セキュリティのベストプラクティス

Aurora DSQL を安全に使用する以下の方法に加えて、クラウドテクノロジーがセキュリティをどのように向上させるかについては、AWS Well-Architected Tool の「セキュリティ」を参照してください。

IAM ロールを使用した Aurora DSQL へのアクセス認証

Aurora DSQL にアクセスするユーザー、アプリケーション、およびその他の AWS のサービスは、AWS API および AWS CLI リクエストに有効な AWS 認証情報を含める必要があります。アプリケーションまたは EC2 インスタンスに AWS 認証情報を直接保存しないでください。これらは、自動的にローテーションされない長期的な認証情報です。これらの認証情報が漏えいした場

合、ビジネスに重大な影響を与えます。IAM ロールにより、AWS のサービスおよびリソースへのアクセスに使用できる一時的なアクセスキーを取得することができます。

詳細については、「Aurora DSQL の認証および認可」を参照してください。

Aurora DSQL ベース認可に IAM ポリシーを使用する

許可を付与する場合、許可を取得するユーザー、取得する許可の対象となる Aurora DSQL API オペレーション、およびそれらのリソースに対して許可される特定のアクションを決定します。最小限の特権の実装は、セキュリティリスクはもちろん、エラーや悪意ある行動によってもたらされる可能性のある影響を減らす上での鍵となります。

アクセス許可ポリシーを IAM ロールにアタッチし、Aurora DSQL リソースでオペレーションを実行するアクセス許可を付与します。また、IAM エンティティのアクセス許可の境界も利用できます。これにより、アイデンティティベースのポリシーが IAM エンティティに付与できるアクセス許可の上限を設定できます。

「AWS アカウントのルートユーザーのベストプラクティス」と同様に、Aurora DSQL の `admin` ロールを使用して日常的なオペレーションを実行しないでください。代わりに、クラスターを管理して接続するためのカスタムデータベースロールを作成することをお勧めします。詳細については、「Accessing Aurora DSQL」および「Understanding authentication and authorization for Aurora DSQL」を参照してください。

実稼働環境で **`verify-full`** を使用する

この設定では、サーバー証明書が信頼できる認証機関によって署名されていること、およびサーバーのホスト名が証明書と一致することを確認します。

PostgreSQL クライアントを更新する

セキュリティ強化の恩恵を受けるために、PostgreSQL クライアントを定期的に最新バージョンに更新してください。PostgreSQL バージョン 17 を使用することをお勧めします。

Aurora DSQL でのリソースのタグ付け

AWS では、タグはユーザー定義のキーと値のペアであり、クラスターなどの Aurora DSQL リソースを定義して関連付けます。タグはオプションです。キーを指定する場合、値はオプションです。

AWS Management Console、AWS CLI、または AWS SDK を使用して、Aurora DSQL クラスターのタグを追加、一覧表示、削除できます。AWS コンソールを使用して、クラスターの作成中および作成後にタグを追加できます。AWS CLI を使用して作成後にクラスターにタグを付けるには、`TagResource` オペレーションを使用します。

名前でクラスターにタグ付けする

Aurora DSQL は、Amazon リソースネーム (ARN) としてグローバルに一意の識別子が割り当てられたクラスターを作成します。クラスターにわかりやすい名前を割り当てる場合は、タグを使用することをお勧めします。

Aurora DSQL コンソールでコンソールを作成すると、Aurora DSQL は自動的にタグを作成します。このタグには、Name というキーと、クラスターの名前を表す自動生成された値があります。この値は設定可能であるため、クラスターにわかりやすい名前を割り当てることができます。クラスターに関連付けられた値を持つ Name タグがある場合は、Aurora DSQL コンソール全体で値を確認できます。

タグ付け要件

タグには、次の要件があります。

- キーにプレフィックス `aws:` を付けることはできません。
- キーはタグセットごとに一意であることが必要です。
- キーに使用できる文字数は 1 ~ 128 文字です。
- 値に使用できる文字数は 0 ~ 256 文字です。
- 値は、タグセットごとに一意にする必要はありません。
- キーと値に使用できる文字は、数字、空白、および _ . : / = + - @ の記号です。
- キーと値は大文字と小文字が区別されます。

タグ付けの使用に関する注意事項

Aurora DSQL でタグを使用する場合は、次の点を考慮してください。

- AWS CLI または Aurora DSQL API オペレーションを使用するときは、操作する Aurora DSQL リソースの Amazon リソースネーム (ARN) を指定する必要があります。詳細については、「Amazon Resource Name (ARNs) format for Aurora DSQL resources」を参照してください。
- 各リソースには、リソースに割り当てられた 1 つ以上のタグの集合であるタグセットが 1 つあります。
- リソースごとに、タグセットあたり最大 50 個のタグを含めることができます。
- リソースを削除した場合、関連付けられたタグが削除されます。
- リソースの作成時にタグを追加できます。また、`TagResource`、`UntagResource` および `ListTagsForResource` の API オペレーションを使用してタグを表示および変更できます。
- IAM ポリシーでタグを使用できます。これらを使用して、Aurora DSQL クラスターへのアクセスを管理したり、それらのリソースに適用できるアクションを制御したりできます。詳細については、「タグを使用した AWS リソースへのアクセスの制御」を参照してください。
- タグは、AWS 全体で他のさまざまなアクティビティに使用できます。詳細については、「一般的なタグ付け戦略」を参照してください。

Amazon Aurora DSQL の使用に関する考慮事項

Amazon Aurora DSQL を使用する場合は、以下の動作を考慮してください。PostgreSQL の互換性とサポートの詳細については、「Aurora DSQL での SQL 機能の互換性」を参照してください。クォータと制限については、「Amazon Aurora DSQL のクラスタークォータとデータベース制限」を参照してください。

- Aurora DSQL は、大きなテーブルの場合、トランザクションタイムアウト前に `COUNT(*)` オペレーションを完了しません。システムカタログからテーブル行数を取得するには、「Using systems tables and commands in Aurora DSQL」を参照してください。
- `PG_PREPARED_STATEMENTS` を呼び出すドライバーは、クラスターのキャッシュされたプリペアドステートメントの表示に一貫性がない場合があります。同じクラスターと IAM ロールに対して、接続ごとに予想される数よりも多くのプリペアドステートメントが表示される場合があります。Aurora DSQL は、プリペアドステートメントの名前を保持しません。
- まれに、マルチリージョンにまたがるリンククラスターに障害が発生した場合、トランザクションコミットの可用性が回復するまでに予想よりも時間がかかることがあります。一般的に、自動クラスター復旧オペレーションでは、一時的な同時実行制御または接続エラーが発生する可能性があります。ほとんどの場合、影響はワークロードの一部にのみ現れます。これらのトランジットエラーが発生した場合は、トランザクションを再試行するか、クライアントに再接続してください。
- Datagrip などの一部の SQL クライアントは、システムメタデータに対して拡張呼び出しを行い、スキーマ情報を入力します。Aurora DSQL はこれらの情報をすべてサポートしていないため、エラーを返します。この問題は SQL クエリ機能には影響しませんが、スキーマの表示に影響する可能性があります。
- 管理者ロールには、データベース管理タスクに関連する一連のアクセス許可があります。デフォルトでは、これらのアクセス許可は他のユーザーが作成したオブジェクトには適用されません。管理者ロールは、これらのユーザーが作成したオブジェクトに対するアクセス許可を他のユーザーに付与したり、取り消したりすることはできません。管理者ユーザーは、これらのオブジェクトに対する必要なアクセス許可を取得するために、自身に他のロールを付与することができます。

Amazon Aurora DSQL のクラスタークォータとデータベース制限

以下のセクションでは、Aurora DSQL のクラスタークォータとデータベース制限について説明します。

クラスタークォータ

AWS アカウントには、Aurora DSQL で次のクラスタークォータがあります。特定の AWS リージョン内の単一リージョンクラスターとマルチリージョンクラスターのサービスクォータの増加をリクエストするには、Service Quotas コンソールページを使用します。その他のクォータの増加については、AWS サポート にお問い合わせください。

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード
AWS アカウントあたりの単一リージョンクラスターの最大数。	20 クラスター	はい	API エラーコード `ServiceQuotaExceededExcept`
AWS アカウントあたりのマルチリージョンクラスターの最大数	5 クラスター	はい	API エラーコード `ServiceQuotaExceededExcept`
クラスターあたりの最大ストレージ	10 TiB のデフォルト制限、承認済みの制限引き上げで最大 128 TiB	はい	`DISK_FULL(53100)`

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード
クラスターあたりの最大接続数	10,000 接続	はい	`TOO_MANY_CONNECTIONS(53300)`
クラスターあたりの最大接続レート	100 接続/秒	いいえ	`CONFIGURED_LIMIT_EXCEEDED(53400)`
クラスターあたりの最大接続バーストキャパシティ	1,000 接続	いいえ	エラーコードなし
同時復元ジョブの最大数	4	いいえ	エラーコードなし
接続のリフィルレート	100 接続/秒	いいえ	エラーコードなし

Aurora DSQL のデータベース制限

次の表に、Aurora DSQL のデータベース制限を示します。

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード	エラーメッセージ
プライマリキーで使用される列の最大合計サイズ	1 KiB	いいえ	`54000`	`ERROR: key size too large`

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード	エラーメッセージ
セカンダリインデックスの列の最大合計サイズ	1 KiB	いいえ	54000	ERROR: key size too large
テーブル内の行の最大サイズ	2 MiB	いいえ	54000	ERROR: maximum row size excee
インデックスの一部ではない列の最大サイズ	1 MiB	いいえ	54000	ERROR: maximum column size ex
プライマリキーまたはセカンダリインデックスの列の最大数	8	いいえ	54011	ERROR: more than 8 column key are not supported
テーブル内の列の最大数	255	いいえ	54011	ERROR: tables can have at mos
テーブル内のインデックスの最大数	24	いいえ	54000	ERROR: more than 24 indexes p allowed
書き込みトランザクションで変更されたすべてのデータの最大サイズ	10 MiB	いいえ	54000	ERROR: transaction size limit DETAIL: Current transaction s 10mb

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード	エラーメッセージ
トランザクションブロックで変更できるテーブル行とインデックス行の最大数	トランザクションあたり 3,000 行。「PostgreSQL 互換性に関する Aurora DSQL の考慮事項」を参照してください。	いいえ	54000	ERROR: transaction row limit
クエリオペレーションで使用できるメモリの基本最大量	トランザクションあたり 128 MiB	いいえ	53200	ERROR: query requires too muc out of memory.
データベースで定義されたスキーマの最大数	10	いいえ	54000	ERROR: more than 10 schemas n
データベース内のテーブルの最大数	1,000 テーブル	いいえ	54000	ERROR: creating more than 100 allowed
クラスター内のデータベースの最大数	1	いいえ	エラーコードなし	ERROR: unsupported statement
最大トランザクション時間	5 分	いいえ	54000	ERROR: transaction age limit exceeded

説明	デフォルトの制限	設定可能?	Aurora DSQL エラーコード	エラーメッセージ
最大接続時間	60 分	いいえ	エラーコードなし	エラーメッセージなし
データベースのビューの最大数	5,000	いいえ	54000	ERROR: creating more than 500 allowed
ビュー定義の最大サイズ	2 MiB	いいえ	54000	ERROR: view definition too la

Aurora DSQL に固有のデータ型の制限については、「Aurora DSQL でサポートされているデータ型」を参照してください。

Aurora DSQL API のリファレンス

AWS Management Consoleおよび AWS Command Line Interface (AWS CLI) に加えて、Aurora DSQL は API インターフェイスも提供します。API オペレーションを使用して、Aurora DSQL でリソースを管理できます。

API オペレーションのアルファベット順リストについては、「アクション」を参照してください。

データ型のアルファベット順リストについては、「データ型」を参照してください。

共通クエリパラメータのリストについては、「共通パラメータ」を参照してください。

エラーコードの説明については、「共通エラー」を参照してください。

AWS CLI の詳細については、「AWS Command Line Interface reference for Aurora DSQL」を参照してください。

Aurora DSQL の問題のトラブルシューティング

> **Note**
>
> 以下のトピックでは、Aurora DSQL の使用時に発生する可能性のあるエラーや問題のトラブルシューティングに関するアドバイスを提供します。こちらに記載されていない問題が見つかった場合は、AWS サポートにお問い合わせください。

トピック

- 接続エラーのトラブルシューティング
- 認証エラーのトラブルシューティング
- 認可エラーのトラブルシューティング
- SQL エラーのトラブルシューティング
- OCC エラーのトラブルシューティング
- SSL/TLS 接続のトラブルシューティング

接続エラーのトラブルシューティング

error: unrecognized SSL error code: 6

原因: Server Name Indication (SNI) をサポートしていないバージョン 14 より前の psql バージョンを使用している可能性があります。SNI は Aurora DSQL に接続するときに必要です。

クライアントのバージョンは `psql --version` で確認できます。

error: NetworkUnreachable

接続試行中の `NetworkUnreachable` エラーは、実際のネットワーク問題を示すのではなく、クライアントが IPv6 接続をサポートしていないことを示している可能性があります。このエラーは、PostgreSQL クライアントがデュアルスタック接続を処理する方法が原因で、一般的に IPv4 のみのインスタンスで発生します。サーバーがデュアルスタックモードをサポートしている場合、これらのクライアントはまずホスト名を IPv4 アドレスと IPv6 アドレスの両方に解決します。最初に IPv4 接続を試み、最初の接続が失敗した場合は IPv6 を試みます。システムが IPv6 をサポートし

ていない場合は、明確な「IPv6 はサポートされていません」というメッセージの代わりに一般的な `NetworkUnreachable` エラーが表示されます。

認証エラーのトラブルシューティング

IAM authentication failed for user "..."

Aurora DSQL IAM 認証トークンを生成する場合、設定できる最大期間は 1 週間です。1 週間後、そのトークンで認証することはできません。

さらに、引き受けたロールの有効期限が切れている場合、Aurora DSQL は接続リクエストを拒否します。例えば、認証トークンの有効期限が切れていない場合でも、一時的な IAM ロールに接続しようとすると、Aurora DSQL は接続リクエストを拒否します。

IAM と Aurora DSQL の連携の詳細については、「Understanding authentication and authorization for Aurora DSQL」および「AWS Identity and Access Management in Aurora DSQL」を参照してください。

An error occurred (InvalidAccessKeyId) when calling the GetObject operation: The AWS Access Key ID you provided does not exist in our records

IAM がリクエストを拒否しました。詳細については、「リクエストに署名する理由」を参照してください。

IAM role <role> does not exist

Aurora DSQL は IAM ロールを見つけることができませんでした。詳細については、「IAM ロール」を参照してください。

IAM role must look like an IAM ARN

詳細については、「IAM ID - IAM ARN」を参照してください。

認可エラーのトラブルシューティング

Role <role> not supported

Aurora DSQL は `GRANT` オペレーションをサポートしていません。「Supported subsets of PostgreSQL commands in Aurora DSQL」を参照してください。

Cannot establish trust with role <role>

Aurora DSQL は GRANT オペレーションをサポートしていません。「Supported subsets of PostgreSQL commands in Aurora DSQL」を参照してください。

Role <role> does not exist

Aurora DSQL は、指定されたデータベースユーザーを見つけることができませんでした。「Authorize custom database roles to connect to a cluster」を参照してください。

ERROR: permission denied to grant IAM trust with role <role>

データベースロールへのアクセスを許可するには、管理者ロールを使用してクラスターに接続している必要があります。詳細については、「Authorize database roles to use SQL in a database」を参照してください。

ERROR: role <role> must have the LOGIN attribute

作成するデータベースロールには、LOGIN アクセス許可が必要です。

このエラーに対処するには、LOGIN アクセス許可を持つ PostgreSQL ロールが作成されていることを確認してください。詳細については、「PostgreSQL のドキュメント」の「CREATE ROLE」および「ALTER ROLE」を参照してください。

ERROR: role <role> cannot be dropped because some objects depend on it

AWS IAM REVOKE を使用して関係を取り消すまで、IAM 関係を持つデータベースロールを削除すると、Aurora DSQL はエラーを返します。詳細については、「Revoking authorization」を参照してください。

SQL エラーのトラブルシューティング

Error: Not supported

Aurora DSQL は、すべての PostgreSQL ベースのダイアレクトをサポートしているわけではありません。サポートされている機能については、「Supported PostgreSQL features in Aurora DSQL」を参照してください。

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

読み取り専用トランザクションで許可されていないオペレーションを試行しています。詳細については、「Understanding concurrency control in Aurora DSQL」を参照してください。

Error: use **`CREATE INDEX ASYNC`** instead

既存の行を持つテーブルにインデックスを作成するには、`CREATE INDEX ASYNC` コマンドを使用する必要があります。詳細については、「Creating indexes asynchronously in Aurora DSQL」を参照してください。

OCC エラーのトラブルシューティング

OC000 “ERROR: mutation conflicts with another transaction, retry as needed”

OC001 “ERROR: schema has been updated by another transaction, retry as needed”

PostgreSQL セッションにスキーマカタログのキャッシュされたコピーがありました。このキャッシュされたコピーは、ロード時に有効でした。時刻 T1 とバージョン V1 を呼び出すことにします。

別のトランザクションが時刻 T2 にカタログを更新します。この V2 を呼び出すことにします。

元のセッションが時刻 T2 でストレージからの読み取りを試みても、カタログバージョン V1 は引き続き使用されています。T2 の最新のカタログバージョンが V2 であるため、Aurora DSQL のストレージレイヤーはリクエストを拒否します。

元のセッションから時刻 T3 に再試行すると、Aurora DSQL はカタログキャッシュを更新します。T3 のトランザクションはカタログ V2 を使用しています。Aurora DSQL は、時刻 T2 以降に他のカタログ変更が行われない限り、トランザクションを完了します。

SSL/TLS 接続のトラブルシューティング

SSL エラー: 証明書の検証に失敗しました

このエラーは、クライアントがサーバーの証明書を検証できないことを示します。以下を確認してください。

1. Amazon ルート CA 1 証明書が正しくインストールされていること。この証明書を検証してインストールする方法については、「Aurora DSQL 接続用の SSL/TLS 証明書の設定」を参照してください。
2. `PGSSLROOTCERT` 環境変数は、正しい証明書ファイルを指します。
3. 証明書ファイルには正しいアクセス許可があります。

認識されない SSL エラーコード: 6

このエラーは、バージョン 14 未満の PostgreSQL クライアントで発生します。この問題を解決するには、PostgreSQL クライアントをバージョン 17 にアップグレードします。

SSL エラー: 未登録スキーム (Windows)

これは、システム証明書を使用する場合の Windows psql クライアントの既知の問題です。「Windows からの接続」の手順で説明されているダウンロードした証明書ファイルメソッドを使用します。

Amazon Aurora DSQL ユーザーガイドのドキュメント履歴

以下の表は、Aurora DSQL のドキュメントリリースの内容をまとめたものです。

変更	説明	日付
Amazon Aurora DSQL の一般提供 (GA)	Amazon Aurora DSQL が一般公開され、CloudWatch モニタリング、強化されたデータ保護機能、AWS Backup 統合のサポートが追加されました。詳細については、「CloudWatch を使用した Aurora DSQL のモニタリング」、「Amazon Aurora DSQL のバックアップと復元」、および「Amazon Aurora DSQL のデータ暗号化」を参照してください。	2025 年 5 月 27 日
AmazonAuroraDSQLFullAccess の更新	ジョブの開始、停止、モニタリングなど、Aurora DSQL クラスターのバックアップおよび復元オペレーションを実行する機能が追加されました。また、クラスター暗号化にカスタマーマネージド KMS キーを使用する機能も追加されています。詳細については、「AmazonAuroraDSQLFullAccess」および「Aurora DSQL でのサービスリンクロールの使用」を参照してください。	2025 年 5 月 21 日
AmazonAuroraDSQLConsoleFullAccess の更新	AWS Console Home を介して Aurora DSQL クラス	2025 年 5 月 21 日

	ターのバックアップおよび復元オペレーションを実行する機能を追加しました。これには、ジョブの開始、停止、モニタリングが含まれます。また、クラスターの暗号化と AWS CloudShell の起動にカスタマーマネージド KMS キーを使用することもサポートしています。詳細については、「AmazonAuroraDSQLConsoleFullAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	
AmazonAuroraDSQLReadOnlyAccess の更新	AWS PrivateLink を介して Aurora DSQL クラスターに接続するときに正しい VPC エンドポイントサービス名を決定する機能が含まれます。Aurora DSQL はセルごとに一意のエンドポイントを作成するため、この API はクラスターの適切なエンドポイントを特定し、接続エラーを回避できます。詳細については、「AmazonAuroraDSQLReadOnlyAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 13 日

AmazonAuroraDSQLFullAccess の更新	このポリシーで、複数の AWS リージョンにまたがるデータベースクラスターを作成および管理するための 4 つの新しいアクセス許可 (`PutMultiRegionProperties` 、`PutWitnessRegion` 、`AddPeerCluster` 、`RemovePeerCluster`) が追加されます。これらのアクセス許可には、リソースレベルのコントロールと条件キーが含まれ、どのクラスターユーザーを変更できるかを制御できます。このポリシーは、AWS PrivateLink を介して Aurora DSQL クラスターに接続するのに役立つ `GetVpcEndpointServiceName` アクセス許可も追加します。詳細については、「AmazonAuroraDSQLConsoleFullAccess」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 13 日

AmazonAuroraDSQLConsoleFullAccess の更新	マルチリージョンクラスター管理と VPC エンドポイント接続をサポートするために、Aurora DSQL に新しいアクセス許可を追加します。新しいアクセス許可には、`PutMultiRegionProperties` 、`PutWitnessRegion` 、`AddPeerCluster` 、`RemovePeerCluster` 、`GetVpcEndpointServiceName` が含まれます。「AmazonAuroraDSQLConsoleFullAccess」および「Aurora DSQL でのサービスリンクロールの使用」を参照してください。	2025 年 5 月 13 日
AuroraDsqlServiceLinkedRolePolicy の更新	ポリシーの `AWS/AuroraDSQL` および `AWS/Usage` `CloudWatch` 名前空間にメトリクスを発行する機能を追加します。これにより、関連付けられたサービスまたはロールは、より包括的な使用状況とパフォーマンスデータを CloudWatch 環境に出力できます。詳細については、「AuroraDsqlServiceLinkedRolePolicy」および「Using service-linked roles in Aurora DSQL」を参照してください。	2025 年 5 月 8 日

Amazon Aurora DSQL の AWS PrivateLink	Aurora DSQL が AWS PrivateLink をサポートするようになりました。AWS PrivateLink では、インターフェイス Amazon VPC エンドポイントおよびプライベート IP アドレスを使用して、仮想プライベートクラウド (VPC)、Aurora DSQL、およびオンプレミスデータセンター間のプライベートネットワーク接続を簡素化できます。詳細については、「Managing and connecting to Amazon Aurora DSQL clusters using AWS PrivateLink」を参照してください。	2025 年 5 月 8 日
初回リリース	Amazon Aurora DSQL ユーザーガイドの初版リリース	2024 年 12 月 3 日