aws

ユーザーガイド

Amazon Aurora DSQL

Amazon Aurora DSQL: ユーザーガイド

Table of Contents

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

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 のフォールトトレランス」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 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 の幅広い機能、式、およびデータ型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。

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 (オハイオ)	us-east-2	dsql.us-east-2.api.aws dsql-fips.us-east-2.api.aws	HTTPS HTTPS
米国東部 (バージニア北部)	us-east-1	dsql.us-east-1.api.aws dsql-fips.us-east-1.api.aws	HTTPS HTTPS
米国西部 (オレゴン)	us-west-2	dsql.us-west-2.api.aws dsql-fips.us-west-2.api.aws	HTTPS HTTPS
アジアパシフィック (香港)	ap-east-1	dsql.ap-east-1.api.aws	HTTPS
アジアパシフィック (メルボルン)	ap-southeast-4	dsql.ap-southeast-4.api.aws	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル	
アジアパシフィック (ムンバイ)	ap-south-1	dsql.ap-south-1.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 (ソウル)	ap-northeast-2	dsql.ap-northeast-2.api.aws	HTTPS	
アジアパシフィック (シンガポール)	ap-southeast-1	dsql.ap-southeast-1.api.aws	HTTPS	
アジアパシフィック (シドニー)	ap-southeast-2	dsql.ap-southeast-2.api.aws	HTTPS	
アジアパシフィック (東京)	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS	
カナダ (中部)	ca-central-1	dsql.ca-central-1.api.aws dsql-fips.ca-central-1.api.aws	HTTPS HTTPS	
カナダ西部 (カルガリー)	ca-west-1	dsql.ca-west-1.api.aws dsql-fips.ca-west-1.api.aws	HTTPS HTTPS	

リージョン名	リージョン	エンドポイント	プロトコル	
欧州 (フランクフルト)	eu-central-1	dsql.eu-central-1.api.aws	HTTPS	
欧州 (アイルランド)	eu-west-1	dsql.eu-west-1.api.aws	HTTPS	
欧州 (ロンドン)	eu-west-2	dsql.eu-west-2.api.aws	HTTPS	
欧州 (パリ)	eu-west-3	dsql.eu-west-3.api.aws	HTTPS	
欧州 (ストックホルム)	eu-north-1	dsql.eu-north-1.api.aws	HTTPS	
南米 (サンパウロ)	sa-east-1	dsql.sa-east-1.api.aws	HTTPS	

Aurora DSQL のマルチリージョンクラスターの可用性

特定の AWS リージョンセット内に Aurora DSQL マルチリージョンクラスターを作成できます。各リージョンセットは、マルチリージョンクラスター内で連携可能な、地理的に関連するリージョンをグループ化します。

米国リージョン

- 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)

アジアパシフィックリージョン

- アジアパシフィック (大阪)
- アジアパシフィック (ソウル)
- アジアパシフィック (東京)

欧州リージョン

- 欧州 (フランクフルト)
- 欧州 (アイルランド)
- 欧州 (ロンドン)
- 欧州 (パリ)

重要な制限事項

マルチリージョンクラスターは、単一のリージョンセット内に作成する必要があります。例えば、米国東部 (バージニア北部) リージョンと欧州 (アイルランド) リージョンの両方をまたぐクラスターを作成することはできません。

> **Important**
>
> Aurora DSQL は現在、大陸間をまたぐマルチリージョンクラスターをサポートしていません。

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

- PostgreSQL 互換クライアントを使用した Aurora DSQL へのアクセス
- Aurora DSQL と 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でフルマネージド型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 コンソールを使用してクラスターにアクセスし、必要に応じて他の PostgreSQL クライアントを使用してデータベースとやり取りします。最終的に、テストまたは本番ワークロードに使用可能な、動作中の Aurora DSQL クラスターが設定されます。

トピック

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、コンソール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
  - `AmazonAuroraDSQLConsoleFullAccess` AWS マネージドポリシーがアタッチされています。

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 マネジメントコンソールにサインインし、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) デフォルトの Name タグの値を変更します。
4. (オプション) このクラスターにタグを追加します。
5. (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。カスタマーマネージドキーを使用する場合は、キーポリシーが、Aurora DSQL に必要なアクセス許可を付与することを確認します。詳細については、「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。
   - 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。
   - [リソースベースのポリシー (詳細)] を選択して、このクラスターのアクセスコントロールポリシーを指定します。
6. [クラスターを作成] を選択します。
7. コンソールでは、[ロール] ページが返されます。クラスターが作成されていることを示す通知バナーが表示されます。[クラスター ID] を選択して、クラスターの詳細ビューを開きます。

ステップ 2: Aurora DSQL クラスターに接続する

Aurora DSQL は、DSQL クエリエディタ、AWS CloudShell、ローカル psql クライアント、その他の PostgreSQL 互換ツールなど、クラスターに接続する複数の方法をサポートしています。このステップでは、Aurora DSQL クエリエディタを使用して接続します。これにより、新しいクラスターの操作をすばやく開始できます。

クエリエディタを使用して接続するには

1. Aurora DSQL コンソール (https://console.aws.amazon.com/dsql) で、[クラスター] ページを開き、クラスターの作成が完了し、そのステータスがアクティブであることを確認します。
2. リストからクラスターを選択、または [クラスター ID] を選択してクラスターの詳細ページを開きます。
3. [クエリエディタで接続]を選択します。

4. 先ほど作成したクラスターの [管理者] として接続を選択します。

   - 必要に応じて、カスタムロールを使用して接続することもできます。「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。

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。クエリエディタでクラスターを開いたら、各サンプルクエリをステップバイステップで選択して実行します。

Aurora DSQL でサンプル SQL コマンドを実行する

1. `test` という名前のスキーマを作成します。

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

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 hello_world テーブルを作成します。

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

3. サンプル行を挿入します。

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

4. 挿入された値を読み取ります。

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

5. オプションでクリーンアップします。

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

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。

次の手順を使用して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. Aurora DSQL コンソールにサインインします。
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) デフォルトの Name タグの値を変更します。
5. (オプション) このクラスターにタグを追加します。
6. [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。

7. (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。カスタマーマネージドキーを使用する場合は、キーポリシーが、Aurora DSQL に必要なアクセス許可を付与することを確認します。詳細については、「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。
   - 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。
   - [リソースベースのポリシー (詳細)] を選択して、このクラスターのアクセスコントロールポリシーを指定します。

8. 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。

9. [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。

10. 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。

11. [***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。

12. (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。カスタマーマネージドキーを使用する場合は、キーポリシーが、Aurora DSQL に必要なアクセス許可を付与することを確認します。詳細については、「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。

13. 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。

14. [***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。

15. 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. [クエリエディタで接続] を選択します。
5. [管理者として接続] を選択します。
6. 「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、サンプルスキーマおよびテーブルを作成して、データを挿入します。

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. [クエリエディタで接続] を選択します。
5. [管理者として接続] を選択します。
6. 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 * FROM test.hello_world;
   ```

トラブルシューティング

Aurora DSQL ドキュメントの「トラブルシューティング」セクションを参照してください。

Aurora DSQL での請求の仕組み

Amazon Aurora DSQL では、前払いコストなしで、使用した分のみお支払いいただきます。このセクションでは、Aurora DSQL でデータベースアクティビティを計測し、それを AWS 請求書の料金に変換する方法について説明します。リージョン別の現在の料金については、Aurora DSQL の料金ページを参照してください。

トピック

計測の仕組み

プロビジョニングした容量に対して課金する従来のデータベースとは異なり、Aurora DSQL では実際に実行した作業に対してのみ課金されます。Aurora DSQL は、2 つの主要コンポーネントを計測します。分散処理ユニット (DPU) で測定するデータベースアクティビティと、GiB-月で測定するストレージです。

DPU は、SQL ワークロードを実行するためにシステムが費やす作業量を測定します。シングルリージョンクラスターの場合、これは 3 つのコンポーネント (コンピューティング DPU、読み取り DPU、書き込み DPU) で構成されます。マルチリージョンクラスターの場合は、追加のマルチリージョン書き込み DPU コンポーネントが発生します。詳細については、「マルチリージョン請求」を参照してください。

次の表は、Aurora DSQL がデータベースアクティビティを計測するために使用するコンポーネントをまとめたものです。請求書には 2 つの明細項目のみが表示されます。1 つはストレージ用です。もう 1 つは DPU 用で、個々のコンポーネントの合計です。

計測単位	アクティビティタイプ	測定
コンピューティング DPU	クエリ処理	CPU 時間
読み取り DPU	データベースからのデータの読み取り	ストレージから読み取られたバイト数
書き込み DPU	データベースへのデータの書き込み	ストレージに書き込まれたバイト数
Storage	テーブルストレージ	GiB-月

DPU コンポーネントの計測の説明

トランザクションごとに、Aurora DSQL は合計 DPU を、コンピューティング DPU、読み取り DPU、書き込み DPU の 3 つのコンポーネントの合計として計算します。以下のセクションでは、Aurora DSQL で各コンポーネントを計測する方法について説明します。

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

コンピューティング DPU

コンピューティング DPU は、結合、関数、集計、ソート、クエリ計画など、クエリの実行に費やされた合計処理時間を使用して測定されます。クエリの一部は並行して処理される場合があるため、コンピューティング DPU はクエリのウォールクロック時間ではなく、すべての処理時間の合計を反映します。

次の式は、コンピューティング DPU の計算方法を示しています。

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

書き込み DPU

トランザクションごとに、Aurora DSQL はストレージに書き込まれた合計バイト数で書き込み DPU を測定します。書き込み DPU には、ベーステーブルに書き込まれた合計データとセカンダリインデックスが含まれます。Aurora DSQLは、ベーステーブルおよびセカンダリインデックスに書き込ま

れた 128 バイト未満の各行を、128 バイトとして計算して請求します。Aurora DSQL は、1,024 バイト未満の書き込みトランザクションを、1,024 バイトを書き込んだものとして請求します。

> **Note**
>
> Aurora DSQL はプライマリキーインデックスを読み取って一意性を検証してから書き込みを行うため、書き込みオペレーションにも読み取り DPU 料金が発生します。

次の式は、書き込み DPU を計算するステップを示しています。

ステップ 1: 書き込まれたバイト数を計算する

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

ステップ 2: 書き込み DPU を計算する

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

読み取り DPU

トランザクションごとに、Aurora DSQL は、ストレージから読み取った合計バイト数で読み取り DPU を測定します。読み取り DPU には、ベーステーブルから読み取られたデータとセカンダリインデックスが含まれます。

パーティションあたりの最小値: Aurora DSQL は、行ごとではなく、ストレージパーティションごとに読み取られたバイト数を測定します。ストレージパーティションへの読み取りリクエストが 128 バイト未満を返す場合、Aurora DSQL はそれを 128 バイトに切り上げます。例えば、クエリが 4 つのパーティションから読み取るとします。1 つのパーティションからは 200 バイト、他の 3 つの各パーティションからは 50 バイトを読み取る場合、3 つの 50 バイトの読み取りはそれぞれ 128 バイトに切り上げられ、合計 200 + 128 + 128 + 128 = 584 バイトとして請求されます。

トランザクションの最小値: Aurora DSQL は、合計 2,048 バイト未満の読み取りトランザクションを 2,048 バイトの読み取りとして請求します。

次の式は、読み取り DPU を計算するステップを示しています。

ステップ 1: 読み取られたバイト数を計算する

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

> **Note**
>
> 実際に読み取られたバイト数は、ストレージパーティション間でのデータの分散方法によって異なります。パーティションあたりの最小値である 128 バイトが適用されるためです。すべての行サイズが 128 バイトを超える場合は、読み取られた行数に各行のサイズを掛けるだけです。

ステップ 2: 読み取り DPU を計算する

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

請求例

以下の例は、Aurora DSQL が一般的なオペレーションの DPU を計算する方法を示しています。これらの例のコスト値では、us-east-1 リージョン料金を使用しています。他のリージョンの料金については、Aurora DSQL の料金ページを参照してください。

例: シンプルなポイントルックアップ (読み取り)

この例では、トランザクションの最小値が適用されるポイントルックアップの読み取り DPU 計算を示します。

スキーマ:

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

クエリ:

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

シナリオ: クエリは 5 行 (各行約 100 バイト) を返します。すべての行が 1 つのストレージパーティションに存在すると仮定すると、読み取られる総バイト数は 5 × 100 = 500 バイトです。500 バイトは、パーティションあたりの最小値である 128 バイトを超えているため、パーティションごとの最小値は適用されません。

読み取り DPU を計算する:

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

500 バイトは 2,048 バイト未満であるため、トランザクションの最小値である 2,048 バイトが適用されます。

合計トランザクションコスト:

クエリ実行時間を 3 ミリ秒 (0.003 秒) と仮定します。

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

例: フィルタリングされた範囲スキャン (読み取り)

スキーマ:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id VARCHAR(50) NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(20)
);
-- Average row size: ~100 bytes
-- Table contains 100 orders for customer 'cust-12345'
```

クエリ:

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

```
  AND total_amount > 500.00;
```

シナリオ: クエリはお客様「cust-12345」について 100 行をスキャンしますが、total_amount > 500.00 フィルターは結果を減らして 10 行のみを返します。Aurora DSQL は、スキャンされた 100 行すべてに対して請求します。すべての行が 1 つのストレージパーティションに存在すると仮定すると、読み取られる総バイト数は 100 × 100 = 10,000 バイトです。

読み取り DPU を計算する:

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

10,000 バイトはトランザクションの最小値である 2,048 バイトを超えているため、実際に読み取られたバイト数が使用されます。

合計トランザクションコスト:

クエリ実行時間を 8 ミリ秒 (0.008 秒) と仮定します。

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

> **Important**
>
> 読み取り DPU コストを最小限に抑えるには、必要な行のみをスキャンするようにクエリとインデックスを設計します。この例では、(customer_id, total_amount) にインデックスを追加することで、クエリがスキャンする行数を減少できるようにします。

例: 単一挿入 (読み取りと書き込み)

スキーマ:

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

```
    status VARCHAR(20)
);
-- Average row size: ~100 bytes
```

クエリ:

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

シナリオ: 約 100 バイトの 1 行を挿入します。

書き込み DPU の計算:

ステップ 1 - 書き込まれたバイト数を計算する:

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

ステップ 2 - 書き込み DPU を計算する:

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

128 バイトは 1,024 バイト未満であるため、トランザクションの最小値である 1,024 バイトが適用されます。

読み取り DPU (プライマリキーチェック):

Aurora DSQL はプライマリキーインデックスを読み取って一意性を検証してから書き込みます。これにより、トランザクションの最小読み取り料金が発生します。

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

合計トランザクションコスト:

クエリ実行時間を 8 ミリ秒 (0.008 秒) と仮定します。

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

例: 一括挿入 (読み取りと書き込み)

スキーマ:

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

クエリ:

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

シナリオ: 100 行 (各行約 100 バイト) を挿入します。

書き込み DPU の計算:

ステップ 1 - 書き込まれたバイト数を計算する:

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

ステップ 2 - 書き込み DPU を計算する:

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

読み取り DPU (プライマリキーチェック):

Aurora DSQL は、各行のプライマリキーインデックスを読み取り、一意性を検証します。100 個のキールックアップがすべて 1 つのストレージパーティションに存在すると仮定すると、読み取られる総バイト数は 100 × 16 バイト (UUID) = 1,600 バイトです。

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

1,600 バイトは 2,048 バイト未満であるため、トランザクションの最小値である 2,048 バイトが適用されます。

合計トランザクションコスト:

クエリ実行時間を 80 ミリ秒 (0.08 秒) と仮定します。

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

マルチリージョン請求

マルチリージョンクラスターでは、標準のコンピューティング DPU、読み取り DPU、書き込み DPU に加えて、追加のマルチリージョン書き込み DPU コンポーネントが発生します。このセクションは、マルチリージョンクラスターにのみ適用されます。シングルリージョンクラスターの場合、この料金は発生しません。

マルチリージョン書き込み DPU は、ピアリングされたリージョンに書き込まれた合計バイト数を測定します。Aurora DSQL はピアリングされたリージョンに書き込むデータを同期的にレプリケートするため、マルチリージョン書き込み DPU 値は書き込み DPU と同等です。Aurora DSQL は、ピアリングされたリージョンではなく、書き込みが発生したリージョンでこの DPU に課金します。

```
MultiRegionWriteDPU = WriteDPU
```

CloudWatch による DPU 使用状況のモニタリング

Aurora DSQL は Amazon CloudWatch に使用状況メトリクスを発行するため、消費状況をほぼリアルタイムでモニタリングできます。

使用可能な DPU メトリクス

DPU メトリクス

CloudWatch メトリクス	説明	ディメンション
`WriteDPU`	書き込み使用量コンポーネント	ClusterId

CloudWatch メトリクス	説明	ディメンション
ReadDPU	読み取り使用量コンポーネント	ClusterId
ComputeDPU	クエリ処理コンポーネント	ClusterId
MultiRegionWriteDPU	マルチリージョンレプリケーション (マルチリージョンクラスターのみ)	ClusterId
TotalDPU	すべての DPU コンポーネントの合計	ClusterId

DPU メトリクスの表示

CloudWatch で DPU メトリクスを表示するには

1. CloudWatch コンソールを開きます。
2. [メトリクス]、[AuroraDSQL]、[ClusterId] の順に移動します。
3. モニタリングするクラスターと DPU メトリクスを選択します。

> **Tip**
>
> DPU メトリクスの [合計] 統計を使用して、一定期間の合計使用量を確認します。LAST ラベルを追加して、最新の値を表示します。

追加のオブザーバビリティメトリクス

Aurora DSQL のメトリクスとモニタリング機能の詳細なリストについては、「モニタリングとログ記録」を参照してください。

オブザーバビリティメトリクス

メトリクス	説明
ClusterStorageSize	現在のストレージサイズ (バイト単位)
TotalTransactions	実行されたトランザクションの合計

メトリクス	説明
ReadOnlyTransactions	実行された読み取り専用トランザクション
QueryTimeouts	制限時間を超えたクエリ
OccConflicts	OCC の競合により中止されたトランザクション
BytesWritten	ストレージに書き込まれた生のバイト数
BytesRead	ストレージから読み取られた生のバイト数

EXPLAIN ANALYZE VERBOSE を使用したコスト認識

Aurora DSQL は EXPLAIN ANALYZE VERBOSE を拡張して、ステートメントレベルの DPU 使用量の見積もりを出力の最後に含めます。これにより、クエリコストがすぐに可視化され、ワークロードのコスト要因の特定、クエリのパフォーマンスの調整、リソース使用量の予測に役立ちます。

> **Note**
>
> DPU の見積もりを表示するには、EXPLAIN ANALYZE VERBOSE (VERBOSE 付き) を使用する必要があります。VERBOSE を付けずに EXPLAIN ANALYZE だけを使用すると、DPU 情報は表示されません。

例 1: SELECT クエリ

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

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

```
Query Identifier: qymgw1m77maoe
Planning Time: 11.415 ms
Execution Time: 4.528 ms
Statement DPU Estimate:
  Compute: 0.01607 DPU
  Read: 0.04312 DPU
  Write: 0.00000 DPU
  Total: 0.05919 DPU
```

この例では、SELECT ステートメントはインデックスのみのスキャンを実行するため、ほとんどのコストは、ストレージから取得されたデータを表す読み取り DPU (0.04312) と、結果を処理して返すために使用されるコンピューティングリソースを反映したコンピューティング DPU (0.01607) から発生します。クエリはデータを変更しないため、書き込み DPU はありません。合計 DPU (0.05919) は、コンピューティング + 読み取り + 書き込みの合計です。

例 2: INSERT クエリ

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

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

このステートメントは主に書き込みを実行するため、ほとんどのコストは書き込み DPU に関連付けられます。コンピューティング DPU (0.01550) は、値を処理して挿入するために行われた作業を表します。読み取り DPU (0.00307) は、マイナーシステム読み取り (カタログルックアップまたはインデックスチェックの場合) を反映します。

読み取り DPU と書き込み DPU の横に表示される括弧内のトランザクションの最小値に注意してください。これらの最小値はトランザクションレベルで適用されます。つまり、トランザクション全体の読み取り DPU または書き込み DPU の合計がこれらの値を下回ることはありません。EXPLAIN ANALYZE VERBOSE を使用してコストを予測しており、これがトランザクションの唯一のステートメントである場合は、raw ステートメントの見積もりではなく、トランザクションの最小値を使用します。トランザクションに複数のステートメントが含まれている場合、最小値はすべてのステートメントの集計に適用されます。EXPLAIN ANALYZE VERBOSE はステートメントレベルの見積もりを報告し、請求はトランザクションレベルの最小値を適用するため、これらの値は CloudWatch メトリクスや請求データと正確に一致しない場合があります。

最適化のための DPU 情報の使用

ステートメントごとの DPU 見積もりは、実行時間を超えてクエリを最適化する強力な方法を提供します。一般的ユースケースには以下が含まれます。

- コスト認識: クエリが他のクエリと比較してどのくらいのコストがかかるかを把握します。
- スキーマの最適化: インデックスまたはスキーマの変更がパフォーマンスとリソース効率の両方に与える影響を比較します。
- 予算計画: 観測された DPU 使用量に基づいてワークロードコストを見積もります。
- クエリ比較: 代替クエリアプローチを相対的な DPU 消費量で評価します。

DPU 情報の解釈

EXPLAIN ANALYZE VERBOSE の DPU データを使用する場合は、次のベストプラクティスに注意してください。

- 方向的に使用する: 報告されたDPUを、CloudWatchメトリクスや請求データとの厳密な一致ではなく、クエリの相対的なコストを理解する手段として扱います。EXPLAIN ANALYZE VERBOSE はステートメントレベルのコストを報告し、CloudWatch はトランザクションレベルのアクティビティを集計するため、差異が想定されます。CloudWatch には、EXPLAIN ANALYZE VERBOSE が意図的に除外するバックグラウンドオペレーション (非同期 ANALYZE や圧縮など) とトランザクションオーバーヘッド (BEGIN/COMMIT) も含まれます。
- 概念実証の代表的なデータを使用してテストする: 概念実証を実行してコストを評価するときは、予想される本番ワークロードと同様のデータボリュームとディストリビューションがテーブルに含まれていることを確認します。DPU の見積もり (EXPLAIN ANALYZE VERBOSE によるか CloudWatch メトリクスによるかを問わず) は、空のテーブルに基づいているか、データがまばらなテーブルに基づいている場合、実際のコストを反映しません。

- 実行間の DPU の変動は分散システムでは正常であり、エラーを示すものではありません。キャッシュ、実行計画の変更、同時実行、バックグラウンドオペレーション (非同期 ANALYZE など)、データ分散のシフトなどの要因により、同じクエリでも実行するたびに異なるリソースを消費する可能性があります。
- 小規模なオペレーションのバッチ化: ワークロードで多数の小さなステートメントを発行する場合は、より大きな書き込みオペレーションにバッチ化して 1 つのトランザクションにまとめることを検討してください (変更はトランザクションあたり 10 MB を超えることはできませんが、読み取りの制限は 5 分間のトランザクションタイムアウトのみです)。これにより、作業量を増やしてトランザクションの最小値が適用されないようにすることで、より意味のあるコスト見積もりが生成されます。
- 請求ではなく調整に使用する: `EXPLAIN ANALYZE VERBOSE` の DPU データは、コスト認識、クエリ調整、最適化のために設計されています。請求グレードのメトリクスではありません。信頼できるコストと使用状況データについては、常に CloudWatch メトリクスまたは毎月の請求レポートに依存します。

コスト見積もりのベストプラクティス

- 最適化前にモニタリングする: 最適化を決定する前に、CloudWatch メトリクスを使用して現在の使用状況パターンを理解します。詳細については、「the section called “DPU 使用状況のモニタリング”」を参照してください。
- トランザクション効率を重視する: 最小値はトランザクションレベルで適用されるため、関連するオペレーションをバッチにまとめることで最低料金が適用されないようにします。
- 開発中に EXPLAIN ANALYZE VERBOSE を使用する: 開発中に重要なクエリで `EXPLAIN ANALYZE VERBOSE` を実行して、コスト特性を理解します。概念実証を実行してコストを評価する場合は、代表的なデータボリュームとディストリビューションを持つテーブルに対してテストを行います。空のテーブルまたはデータのまばらなテーブルに基づく見積もりは、本稼働コストを反映しません。詳細については、「the section called “EXPLAIN ANALYZE VERBOSE によるコスト認識”」を参照してください。
- CloudWatch アラームの設定: DPU メトリクスにアラームを作成して、予期しない使用量の急増の通知を受け取ります。

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 のメリットと 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 マネジメントコンソール、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、クラスターを記述するには、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:GetCluster` のアクセス許可があることを確認します。

```
{
  "Effect": "Allow",
  "Action": "dsql:GetCluster",
  "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。

JSON

```
{
"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。

JSON

```
{
```

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

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、AWS CLI、または SDK 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 15 分で自動的に期限切れになります。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 マネジメントコンソールにサインインし、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、Aurora DSQL クラスターを作成してください。

AWS CloudShell を使用して認証トークンを生成するには

1. AWS マネジメントコンソールにサインインし、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. 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。

4. 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
```

5. 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:
```

6. [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。

> **Note**
>
> トークン生成は、現在の IAM 認証情報を使用してリクエストに署名するローカルオペレーションです。認証情報を検証するために AWS に問い合わせることはありません。認証情報の有効期限が切れているか、無効である場合、トークンの生成は成功しても、接続の試行は失敗します。トークンを生成する前に、IAM 認証情報が有効であることを確認します。

- 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

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Python 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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。

```
import boto3

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

C++ SDK

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) {
    DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQLClient client{clientConfig};
    std::string token = "";

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

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

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

JavaScript SDK

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Node.js 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Java JDBC 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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.builder().build())
                .build();

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

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

Rust SDK

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Rust SQLx 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Ruby pg 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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::CredentialProviderChain.new.resolve

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

```
    :credentials => credentials
  })

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

PHP SDK

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する PHP PDO_PGSQL 用 Aurora DSQL コネクタ を使用することをお勧めします。

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、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。

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

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

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

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

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

.NET

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する .NET Npgsql 用 Aurora DSQL コネクタを使用することをお勧めします。

> **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;

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

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

Console.WriteLine(token);
```

Go

> **Tip**
>
> AWS では、トークンの生成を自動的に処理する Go pgx 用 Aurora DSQL コネクタ を使用することをお勧めします。

AWS SDK for Go v2 には、github.com/aws/aws-sdk-go-v2/feature/dsql/auth パッケージで認証トークンを生成するための組み込みメソッドが用意されています。

- admin ロールに接続する場合は、auth.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、auth.GenerateDbConnectAuthToken を使用します。

```
package main

import (
 "context"
 "fmt"

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

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

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

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

 fmt.Println(token)
}
```

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 マネジメントコンソール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 からデータベースへのロールマッピングの表示

IAM ロールとデータベースロール間のマッピングを表示するには、sys.iam_pg_role_mappings システムテーブルをクエリします。

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

出力例:

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

この表は、IAM ロール (ARN で識別) と PostgreSQL データベースロール間のすべてのマッピングを示しています。

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 と 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、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の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。主なハイライトは次のとおりです。

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 構文

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。

分散アーキテクチャのメリット

Aurora DSQL の分散型で共有なしの設計により、従来の単一ノードデータベースを超えるパフォーマンスとスケーラビリティのメリットが得られます。主な機能は次のとおりです。

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。

PostgreSQL で EXPLAIN コマンドを使用する方法の詳細については、「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

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 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

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
`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

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
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。[1] 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。	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。

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	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
		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	解像度	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 ]`		時刻 (タイムゾーンなし)	00:00:00 – 24:00:00	1 マイクロ秒	8 バイト	はい
`time [ (p) ] with time zone`	`timet`	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

名前	エイリアス	説明	Range	解像度	ストレージサイズ	インデックスのサポート
`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`	`timestz`	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
`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

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
UUID		Universally Unique Identifier		16 バイト	はい
json		JSON データ	1 MiB[2]	最大 1 MiB の変数制限。[2]	いいえ

[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。

[2] – Aurora DSQL は json 列に自動的に圧縮を適用し、デフォルトでは INSERT および UPDATE オペレーション中に大きな json 値を圧縮します。1 MiB の制限は圧縮後のサイズに適用されるため、1 MiB を大幅に超える json 値でも、圧縮後のサイズが制限値未満であれば保存できます。

圧縮を無効にするには、STORAGE キーワードを使用します。詳細については、「CREATE TABLE」および「ALTER TABLE」を参照してください。

JSON 関数および演算子

Aurora DSQL は、「section 9.16 JSON Functions and Operators」のすべての PostgreSQL JSON 関数とオペレーションを同じ動作でサポートしています。

> **Note**
>
> json_populate_record と json_populate_recordset 関数はテーブルタイプとビュー行タイプで動作しますが、Aurora DSQL は現在 CREATE TYPE をサポートしていないため、カスタム複合タイプでは動作しません。

次の例は、テーブル行タイプで json_populate_record および json_populate_recordset を使用した例を示しています。

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

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

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

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

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。

JSONB タイプ

Aurora DSQL は、クエリ処理のランタイムデータ型として JSONB をサポートしています。JSON データを保存するには、`json` 型を使用します。

Aurora DSQL は、セクション 9.16 JSON 関数および演算子のすべての PostgreSQL JSONB 関数を同じ動作でサポートしています。「JSON 関数および演算子」で説明されているのと同じ複合型の制限が `jsonb_populate_record` と `jsonb_populate_recordset` に適用されます。

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`	

プライマリ句	サポートされている句
USING	
WITH (テーブル共通表現)	
INNER JOIN	ON
OUTER JOIN	LEFT, RIGHT, FULL, ON
CROSS JOIN	ON
UNION	ALL
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR UPDATE	すべてのプライマリキー列に等価述語を指定します (例: WHERE pk = value)。範囲、IN、OR、またはその他の非等価述語を使用した場合、クエリは ERROR: locking clause such as FOR UPDATE can be applied only on tables with equality predicates on the key を返します。 単一テーブルに対するクエリで指定します。結合または複数のテーブルを使用する場合、クエリは ERROR: locking clause such as FOR UPDATE can be applied on a single table を返します。

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	
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	SEQUENCE	CREATE SEQUENCE コマンドでサポートされている構文については、「CREATE SEQUENCE」を参照してください。

コマンド	プライマリ句	サポートされている句
ALTER	SEQUENCE	ALTER SEQUENCE コマンドでサポートされている構文については、「ALTER SEQUENCE」を参照してください。
DROP	SEQUENCE	DROP SEQUENCE コマンドでサポートされている構文については、「DROP SEQUENCE」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 [ON CONFLICT]
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	エイリアス
COMMIT	[WORK | TRANSACTION] [AND NO CHAIN]	END
BEGIN	[WORK | TRANSACTION] [ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
START TRANSACTION	[ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
ROLLBACK	[WORK | TRANSACTION] [AND NO CHAIN]	ABORT

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

このセクションでは、サポートされている SQL コマンド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 には多くの句とパラメータがあり、そのサブセットは Aurora DSQL でサポートされています。このセクションでは、Aurora DSQL がサポートする使い慣れた PostgreSQL 構文要素を使用した、一般的な SQL コマンドでサポートされているサブセットについて説明します。

トピック

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 [ STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } ]
 [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression )|
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
```

```
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY ( sequence_options ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] index_parameters |
  PRIMARY KEY index_parameters |

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 [, ... ] ) ]
```

アイデンティティ列

> **Note**
>
> ID 列を使用する場合は、キャッシュ値を慎重に検討する必要があります。詳細については、「CREATE SEQUENCE」ページの重要なコールアウトを参照してください。
> ワークロードパターンに基づいて ID 列を使用する最適な方法のガイダンスについては、「シーケンスと ID 列の使用」を参照してください。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY (*sequence_options*) 句は、列を ID 列として作成します。暗黙的なシーケンスがアタッチされ、新しく挿入された行の列には、割り当てられたシーケンスの値が自動的に設定されます。このような列は暗黙的に NOT NULL です。

ALWAYS 句と BY DEFAULT 句は、INSERT コマンドと UPDATE コマンドでユーザーが明示的に指定した値がどのように処理されるかを決定します。

INSERT コマンドで ALWAYS を選択した場合、ユーザー指定の値は、INSERT ステートメントで OVERRIDING SYSTEM VALUE が指定されている場合にのみ受け入れられます。BY DEFAULT が選択されている場合、ユーザー指定の値が優先されます。

UPDATE コマンドで ALWAYS を選択した場合、列を DEFAULT 以外の値に更新することは拒否されます。BY DEFAULT を選択した場合、列は通常どおり更新できます。(UPDATE コマンドには OVERRIDING 句はありません)。

sequence_options 句を使用して、シーケンスのパラメータを上書きできます。使用可能なオプションには、CREATE SEQUENCE に表示されているオプションに加えて、SEQUENCE NAME *name* が含まれます。SEQUENCE NAME がない場合、システムはシーケンスに未使用の名前を選択します。

ストレージモード

オプションの STORAGE 句は、列のストレージモードを設定します。これらのオプションを使用して、JSON などの可変長データ型の圧縮動作を制御します。

Amazon Aurora DSQL は、特定のサイズを超えると、一部のデータ型を圧縮します。この動作を無効にするには、PLAIN または EXTERNAL オプションを使用します。

PLAIN

Aurora DSQL は、データを圧縮せずにインラインで保存します。これは、integer などの固定長データ型の唯一のオプションです。このオプションを使用して、一部の可変長タイプの圧縮を無効にできます。

MAIN | **EXTENDED** | **DEFAULT**

MAIN と EXTENDED は、基になるデータ型が圧縮をサポートしている場合に、列の圧縮をオプションで許可します。DEFAULT は、列のデータ型のストレージモードをデフォルトモードに設定します。

EXTERNAL

Aurora DSQL は現在 TOAST テーブルをサポートしていません。ただし、EXTERNAL を指定すると、圧縮をサポートするデータ型の圧縮が無効になります。

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 [ STORAGE { PLAIN | EXTERNAL
 | EXTENDED | MAIN | DEFAULT } ]
    ADD table_constraint_using_index
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN |
 DEFAULT }
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET
 sequence_option | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

and table_constraint_using_index is:

    [ CONSTRAINT constraint_name ]
    UNIQUE USING INDEX index_name
```

ストレージモード

SET STORAGE

このフォームは、列のストレージモードを設定します。使用可能なストレージモードの詳細については、CREATE TABLE ページの「ストレージモード」を参照してください。

ID 列のアクション

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

これらのフォームは、列が ID 列であるかどうかを変更したり、既存の ID 列の生成属性を変更したりします。詳細については、「CREATE TABLE」を参照してください。SET DEFAULT と同様に、これらの形式は後続の INSERT コマンドと UPDATE コマンドの動作にのみ影響します。テーブル内の既存の行が変更されることはありません。

sequence_option は、INCREMENT BY などの ALTER SEQUENCE でサポートされるオプションです。これらのフォームは、既存の ID 列の基盤となるシーケンスを変更します。

DROP IDENTITY [IF EXISTS]

このフォームは、列から ID プロパティを削除します。DROP IDENTITY IF EXISTS が指定されていて、列が ID 列でない場合、エラーはスローされません。この場合、代わりに通知が発行されます。

制約アクションを追加する

ADD *table_constraint_using_index*

このフォームは、既存の一意のインデックスに基づいてテーブルに新しい UNIQUE 制約を追加します。インデックスのすべての列が制約に含まれます。

インデックスは VALID 状態である必要があります。インデックスが現在構築中である場合、そのインデックスを使用して一意の制約を追加することはサポートされていません。

制約名を指定すると、制約名と一致するようにインデックス名が変更されます。指定しない場合、制約はインデックスと同じ名前になります。

このコマンドを実行すると、インデックスは通常の CREATE UNIQUE INDEX ASYNC コマンドで構築した場合と同じ方法で、制約によって「所有」されます。特に、制約を削除すると、インデックスも消えます。

CREATE SEQUENCE

CREATE SEQUENCE – 新しいシーケンスジェネレーターを定義します。

> **Important**
>
> PostgreSQL では、CACHE の指定はオプションで、デフォルトは 1 です。Amazon Aurora DSQL などの分散システムでは、シーケンスオペレーションには調整が必要であり、キャッシュサイズが 1 の場合、同時実行数が多いと調整オーバーヘッドが増加する可能性があります。キャッシュ値を大きくすると、ローカルに事前割り当てられた範囲からシーケンス番号を提供できるようになり、スループットが向上しますが、未使用の予約値が失われ、ギャップと順序付けのずれがより顕著になる可能性があります。アプリケーションによって割り当て順序とスループットに対する感度が異なるため、Amazon Aurora DSQL では CACHE を明示的に指定する必要があり、現在は CACHE = 1 または CACHE >= 65536 をサポートしており、厳密にシーケンシャルな生成に近い割り当て動作と、高度な同時実行ワークロード向けに最適化された割り当て動作を明確に区別しています。

CACHE >= 65536 の場合、シーケンス値は一意であることが保証されますが、セッション間で厳密に増加順で生成されない可能性があり、特にキャッシュされた値が完全に消費されていない場合にギャップが発生する可能性があります。これらの特性は、同時使用時のキャッシュされたシーケンスの PostgreSQL セマンティクスと一致しています。どちらのシステムも個別の値が保証されますが、セッション間での厳密な連続順序は保証されません。単一のクライアントセッション内では、特に明示的なトランザクションの外部では、シーケンス値が常に厳密に増加するとは限りません。この動作は、接続プーリングを使用する PostgreSQL デプロイに似ています。CACHE = 1 を使用するか、明示的なトランザクション内でシーケンス値を取得することで、シングルセッションの PostgreSQL 環境に近い割り当て動作を実現できます。
CACHE = 1 の場合、シーケンスの割り当ては PostgreSQL のキャッシュされていないシーケンス動作に従います。
ワークロードパターンに基づいてシーケンスを使用する最適な方法のガイダンスについては、「シーケンスと ID 列の使用」を参照してください。

サポートされている構文

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

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

説明

CREATE SEQUENCE は新しいシーケンス番号ジェネレーターを作成します。これには、*name* という名前の新しい特殊な単一行テーブルの作成と初期化が含まれます。ジェネレーターは、コマンドを発行するユーザーによって所有されます。

スキーマ名が指定されている場合は、指定されたスキーマ内にシーケンスが作成されます。指定しない場合、現在のスキーマに作成されます。シーケンスの名前は、同じスキーマ内の他のリレーション (テーブル、シーケンス、インデックス、ビュー、マテリアライズドビュー、または外部テーブル) の名前とは異なる必要があります。

シーケンスを作成したら、関数 nextval、currval、および setval を使用してシーケンスを操作します。これらの関数については、「シーケンス操作関数」で文書化されています。

シーケンスを直接更新することはできませんが、次のようなクエリを使用できます。

```
SELECT * FROM name;
```

シーケンスの一部のパラメータと現在の状態を調べます。特に、シーケンスの last_value フィールドには、任意のセッションによって割り当てられた最後の値が表示されます。(もちろん、他のセッションがアクティブに nextval 呼び出しを実行している場合、この値は出力される時点までに古くなっている可能性があります。) *increment* や *maxvalue* などの他のパラメータは、pg_sequences ビューで確認できます。

パラメータ

IF NOT EXISTS

同じ名前のリレーションが既に存在する場合は、エラーをスローしません。この場合、通知が発行されます。既存のリレーションが、作成されたシーケンスと似ている保証はありません。シーケンスではない可能性があることに注意してください。

.name

作成するシーケンスの名前 (オプションでスキーマ修飾)。

data_type

オプションの AS *data_type* 句は、シーケンスのデータ型を指定します。有効な値は bigint です。bigint がデフォルトです。データ型によって、シーケンスのデフォルトの最小値と最大値が決まります。

##

オプションの INCREMENT BY *increment* 句では、新しい値を作成するために現在のシーケンス値に追加する値を指定します。正の値を指定すると昇順のシーケンスになり、負の値を指定すると降順のシーケンスになります。デフォルト値は 1 です。

minvalue / **NO MINVALUE**

オプションの MINVALUE *minvalue* 句は、シーケンスが生成できる最小値を決定します。この句が指定されていない場合、または NO MINVALUE が指定されている場合は、デフォルトが使用されます。昇順シーケンスのデフォルトは 1 です。降順シーケンスのデフォルトは、データ型の最小値です。

***maxvalue* / NO MAXVALUE**

オプションの MAXVALUE *maxvalue* 句は、シーケンスの最大値を決定します。この句が指定されていない場合、または NO MAXVALUE が指定されている場合は、デフォルト値が使用されます。昇順シーケンスのデフォルトは、データ型の最大値です。降順シーケンスのデフォルトは -1 です。

CYCLE / NO CYCLE

CYCLE オプションを使用すると、昇順または降順のシーケンスがそれぞれ *maxvalue* または *minvalue* に達したときにシーケンスをラップアラウンドすることができます。制限に達すると、生成される次の数値はそれぞれ *minvalue* または *maxvalue* になります。

NO CYCLE が指定された場合、シーケンスが最大値に達した後の nextval の呼び出しではエラーが返されます。CYCLE または NO CYCLE のどちらも指定されていない場合は、NO CYCLE がデフォルトです。

##

オプションの START WITH *start* 句を使用すると、シーケンスを任意の場所から開始できます。デフォルトの開始値は、昇順シーケンスの場合は *minvalue*、降順シーケンスの場合は *maxvalue* です。

#####

CACHE *cache* 句は、アクセスを高速化するために事前に割り当ててメモリに保存するシーケンス番号の数を指定します。Aurora DSQL の CACHE の許容値は 1 または 65536 以上の任意の数値です。最小値は 1 です (一度に生成できる値は 1 つだけです。つまり、キャッシュはありません)。

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

OWNED BY オプションを使用すると、シーケンスが特定のテーブル列に関連付けられるため、その列 (またはそのテーブル全体) が削除されると、シーケンスも自動的に削除されます。指定されたテーブルは、シーケンスと同じ所有者を持ち、同じスキーマ内にある必要があります。デフォルトの OWNED BY NONE は、そのような関連付けがないことを指定します。

注意事項

DROP SEQUENCE を使用してシーケンスを削除します。

シーケンスは bigint 算術に基づいているため、範囲は 8 バイト整数の範囲 (-9223372036854775808 ~ 9223372036854775807) を超えることはできません。

nextval および setval 呼び出しはロールバックされないため、シーケンス番号の「ギャップレス」割り当てが必要な場合は、シーケンスオブジェクトを使用できません。

各セッションは、シーケンスオブジェクトへの 1 回のアクセス中に連続するシーケンス値を割り当てキャッシュし、それに応じてシーケンスオブジェクトの last_value を増やします。その後、そのセッション内での次の *cache*-1 の nextval の使用は、シーケンスオブジェクトにアクセスせずに、単純に事前に割り当てられた値を返します。したがって、セッション内で割り当てられたが使用されなかった番号は、そのセッションが終了すると失われ、シーケンスに「ホール」が発生します。

さらに、複数のセッションに個別のシーケンス値が割り当てられることが保証されていますが、すべてのセッションを考慮すると、値が順序どおりに生成されない可能性があります。例えば、*cache* 設定が 10 の場合、セッション A は値 1..10 を予約して nextval=1 を返し、セッション A が nextval=2 を生成する前に、セッション B が値 11..20 を予約して nextval=11 を返す場合があります。したがって、*cache* 設定が 1 の場合、nextval 値が順番に生成されると想定しても問題ありません。*cache* 設定が 1 より大きい場合は、nextval 値がすべて異なる値であることのみを想定する必要があり、純粋に順番に生成されると想定するべきではありません。また、last_value は、nextval によって返されたかどうかにかかわらず、任意のセッションによって予約された最新の値を反映します。

もう 1 つの考慮事項は、このようなシーケンスで実行された setval は、キャッシュされた事前割り当て値を使い切るまで、他のセッションによって認識されないことです。

例

101 から始まる serial という昇順のシーケンスを作成します。

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

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');

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

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');

 nextval
```

```
---------
     102
```

INSERT コマンドでこのシーケンスを使用します。

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

setval を使用してシーケンスを特定の値にリセットします。

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

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

互換性

CREATE SEQUENCE は SQL 標準に準拠していますが、以下の例外があります。

- 次の値の取得は、標準の NEXT VALUE FOR 式ではなく nextval() 関数を使用して行われます。
- OWNED BY 句は PostgreSQL 拡張機能です。

ALTER SEQUENCE

ALTER SEQUENCE – シーケンスジェネレーターの定義を変更します。

> **Important**
>
> シーケンスを使用する場合は、キャッシュ値を慎重に検討する必要があります。詳細については、「CREATE SEQUENCE」ページの重要なコールアウトを参照してください。
> ワークロードパターンに基づいてシーケンスを使用する最適な方法のガイダンスについては、「シーケンスと ID 列の使用」を参照してください。

サポートされている構文

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
```

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

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

説明

ALTER SEQUENCE は既存のシーケンスジェネレーターのパラメータを変更します。ALTER SEQUENCE コマンドで特に設定されていないパラメータでは、以前の設定が保持されます。

ALTER SEQUENCE を使用するには、シーケンスを所有している必要があります。シーケンス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 ロールにはシーケンスのスキーマに対する CREATE 権限が必要です。(これらの制限により、所有者を変更しても、シーケンスを削除して再作成することでできること以上のことは何も行われません。ただし、スーパーユーザーは、どのシーケンスの所有権も変更できます。)

パラメータ

.name

変更するシーケンスの名前 (オプションでスキーマ修飾)。

IF EXISTS

シーケンス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。

##

INCREMENT BY *increment* 句はオプションです。正の値を指定すると昇順のシーケンスになり、負の値を指定すると降順のシーケンスになります。指定しない場合は、古い増分値は維持されます。

minvalue / **NO MINVALUE**

オプションの MINVALUE *minvalue* 句は、シーケンスが生成できる最小値を決定します。NO MINVALUE を指定した場合、デフォルトとして昇順シーケンスでは 1、降順シーケンスではデー

タ型の最小値がそれぞれ使用されます。どちらのオプションも指定しない場合、現在の最小値が維持されます。

maxvalue / **NO MAXVALUE**

オプションの MAXVALUE *maxvalue* 句は、シーケンスの最大値を決定します。NO MAXVALUE を指定した場合、デフォルトとして昇順シーケンスではデータ型の最大値、降順シーケンスでは -1 がそれぞれ使用されます。どちらのオプションも指定しない場合、現在の最大値が維持されます。

CYCLE

オプションの CYCLE キーワードを使用すると、昇順または降順のシーケンスがそれぞれ *maxvalue* または *minvalue* に達したときに、シーケンスをラップアラウンドすることができます。制限に達すると、生成される次の数値はそれぞれ *minvalue* または *maxvalue* になります。

NO CYCLE

オプションの NO CYCLE キーワードを指定した場合、シーケンスが最大値に達した後に nextval を呼び出すと、エラーが返されます。CYCLE または NO CYCLE が指定されていない場合、古いサイクル動作が維持されます。

##

オプションの START WITH *start* 句は、シーケンスの記録された開始値を変更します。これは現在のシーケンス値には影響しません。将来の ALTER SEQUENCE RESTART コマンドが使用する値を設定するだけです。

###

オプションの RESTART [WITH *restart*] 句は、シーケンスの現在の値を変更します。これは、is_called = false で setval 関数を呼び出す場合と似ています。指定された値は、次の nextval 呼び出しによって返されます。*restart* 値を指定せずに RESTART を書き込むことは、CREATE SEQUENCE によって記録された開始値、または ALTER SEQUENCE START WITH によって最後に設定された開始値を指定することと同じです。

setval 呼び出しとは対照的に、シーケンスに対する RESTART オペレーションはトランザクションであり、同時トランザクションが同じシーケンスから番号を取得するのをブロックします。これが目的のオペレーションモードではない場合は、setval を使用する必要があります。

#####

CACHE *cache* 句を使用すると、シーケンス番号を事前に割り当ててメモリに保存し、アクセスを高速化できます。値は 1 または 65536 以上の値のいずれかである必要があります。指定しない場合、古いキャッシュ値が維持されます。キャッシュ動作の詳細については、「CREATE SEQUENCE」のガイダンスを参照してください。

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

OWNED BY オプションを使用すると、シーケンスが特定のテーブル列に関連付けられるため、その列 (またはそのテーブル全体) が削除されると、シーケンスも自動的に削除されます。指定した場合、この関連付けは、シーケンスに対して以前に指定された関連付けを置き換えます。指定されたテーブルは、シーケンスと同じ所有者を持ち、同じスキーマにある必要があります。OWNED BY NONE を指定すると、既存の関連付けが削除され、シーケンスが「フリースタンディング」になります。

new_owner

シーケンスの新しい所有者のユーザー名です。

new_name

シーケンスの新しい名前。

new_schema

シーケンスの新しいスキーマ。

注意事項

ALTER SEQUENCE は、現在のバックエンド以外の、事前に割り当てられた (キャッシュされた) シーケンス値を持つバックエンドの nextval の結果にすぐには影響しません。変更されたシーケンス生成パラメータが認識される前に、キャッシュされたすべての値を使い果たします。現在のバックエンドはすぐに影響を受けます。

ALTER SEQUENCE はシーケンスの currval ステータスには影響しません。

ALTER SEQUENCE は、他のトランザクションに OCC を引き起こす可能性があります。

歴史上の理由から、ALTER TABLE はシーケンスでも使用できますが、シーケンスで許可される ALTER TABLE のバリエーションは、上記の形式と同等のものだけです。

例

serial というシーケンスを 105 で再開します。

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

互換性

ALTER SEQUENCE は、PostgreSQL 拡張機能である AS、START WITH、OWNED BY、OWNER TO、RENAME TO、および SET SCHEMA 句を除き、SQL 標準に準拠しています。

DROP SEQUENCE

DROP SEQUENCE – シーケンスを削除します。

サポートされている構文

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

説明

DROP SEQUENCE はシーケンス番号ジェネレーターを削除します。シーケンスを削除できるのは、その所有者またはスーパーユーザーのみです。

パラメータ

IF EXISTS

シーケンス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

シーケンスの名前 (オプションでスキーマ修飾)。

CASCADE

シーケンスに依存するオブジェクト、および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が自動的に削除されます。

RESTRICT

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

例

シーケンス seq を削除するには:

```
DROP SEQUENCE seq;
```

互換性

DROP SEQUENCE 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 つのシーケンスのみを削除できます。また、PostgreSQL 拡張である IF EXISTS オプションは除きます。

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。

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。

注意事項

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。
- 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。
- 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。

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';
```

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;
```

再帰的ビューを作成します。

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

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、ビューを所有している必要があります。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。

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

既存の列の名前、または既存の列の新しい名前。

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。

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] [, ...])

ビューのオプションを設定します。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。
- security_invoker (boolean) – ビューのセキュリティ呼び出しプロパティを変更します。

RESET (view_option_name [, ...])

ビューのオプションをデフォルト値にリセットします。

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。

PostgreSQL から Aurora DSQL への移行

Aurora DSQL は PostgreSQL と互換性があるように設計されており、ACID トランザクション、セカンダリインデックス、結合、標準 DML オペレーションなどのコアリレーショナル機能をサポートしています。既存の PostgreSQL アプリケーションのほとんどは、最小限の変更で Aurora DSQL に移行できます。

このセクションでは、フレームワークの互換性、移行パターン、アーキテクチャ上の考慮事項など、アプリケーションを Aurora DSQL に移行するための実用的なガイダンスを提供します。

フレームワークと ORM の互換性

Aurora DSQL は標準の PostgreSQL ワイヤプロトコルを使用して、PostgreSQL ドライバーおよびフレームワークとの互換性を確保します。最も一般的な ORM は、最小限の変更または変更なしで Aurora DSQL で動作します。リファレンス実装と利用可能な ORM 統合については、「the section called “Aurora DSQL アダプター”」を参照してください。

一般的な移行パターン

PostgreSQL から Aurora DSQL に移行する場合、一部の機能の動作が異なったり、代替構文が使用されたりすることがあります。このセクションでは、一般的な移行シナリオに関するガイダンスを提供します。

DDL オペレーションの代替方法

Aurora DSQL は、従来の PostgreSQL DDL オペレーションに代わる最新の操作を提供します。

インデックスの作成

ノンブロッキングインデックスの作成には、`CREATE INDEX` の代わりに `CREATE INDEX ASYNC` を使用します。

ベネフィット: 大規模なテーブルでのダウンタイムなしでインデックスを作成できます。

データの削除

`TRUNCATE` ではなく `DELETE FROM table_name` を使用します。

代替方法: テーブルの完全な再作成には、DROP TABLE に続いて CREATE TABLE を使用します。

システム設定

Aurora DSQL は完全に管理されているため、設定はワークロードパターンに基づいて自動的に処理されます。AWS マネジメントコンソールまたは API を使用して、クラスター設定を管理します。

ベネフィット: データベースのチューニングやパラメータ管理は必要ありません。

スキーマ設計パターン

Aurora DSQL との互換性のために、以下の一般的な PostgreSQL パターンを適応させます。

参照整合性パターン

Aurora DSQL はテーブルの関係と JOIN オペレーションをサポートしています。参照整合性を保つには、アプリケーションレイヤーに検証を実装します。この設計は、アプリケーションレイヤーの検証により柔軟性が向上し、カスケードオペレーションによるパフォーマンスのボトルネックを回避する最新の分散データベースパターンと一致しています。

パターン: 一貫した命名規則、検証ロジック、トランザクション境界を使用して、アプリケーションレイヤーに参照整合性チェックを実装します。多くの大規模なアプリケーションでは、エラー処理とパフォーマンスをより適切に制御するためにこのアプローチが好まれます。

一時的なデータ処理

一時テーブルの代わりに、CTE、サブクエリ、またはクリーンアップロジックを備えた通常のテーブルを使用します。

代替方法: セッション固有の名前を持つテーブルを作成し、アプリケーション内でクリーンアップします。

アーキテクチャの違いを理解する

Aurora DSQL の分散サーバーレスアーキテクチャは、いくつかの点で従来の PostgreSQL と意図的に異なります。これらの違いにより、Aurora DSQL の主なベネフィットであるシンプルさとスケールが実現されます。

簡素化されたデータベースモデル

クラスターごとに単一のデータベース

Aurora DSQL には、クラスターごとに `postgres` という名前の組み込みデータベースが 1 つ用意されています。

移行のヒント: アプリケーションで複数のデータベースを使用する場合は、論理的に分離するために個別の Aurora DSQL クラスターを作成するか、単一のクラスター内でスキーマを使用します。

一時テーブルはありません

一時的なデータ処理には、共通テーブル式 (CTE) とサブクエリを使用する必要があります。これにより、複雑なクエリの柔軟な代替手段が提供されます。

代替方法: 一時的な結果セットには `WITH` 句を使用した CTE を使用するか、セッション固有のデータ用に一意の名前を持つ通常のテーブルを使用します。

Automatic Storage Management

Aurora DSQL により、テーブルスペースと手動のストレージ管理が不要になります。ストレージは、データパターンに基づいて自動的にスケーリングおよび最適化されます。

ベネフィット: ディスク容量のモニタリング、ストレージ割り当ての計画、テーブルスペース設定の管理を行う必要がありません。

最新のアプリケーションパターン

Aurora DSQL は、メンテナンス性とパフォーマンスを向上させる最新のアプリケーション開発パターンを奨励します。

データベーストリガーではなくアプリケーションレベルのロジック

トリガーのような機能を実現するには、アプリケーションレイヤーにイベント駆動型ロジックを実装します。

移行戦略: トリガーロジックをアプリケーションコードに移動し、EventBridge などの AWS のサービスでイベント駆動型アーキテクチャを使用するか、アプリケーションのログ記録を使用して監査証跡を実装します。

データ処理用の SQL 関数

Aurora DSQL は SQL ベースの関数をサポートしていますが、PL/pgSQL などの手続き型言語はサポートしていません。

代替方法: データ変換に SQL 関数を使用するか、複雑なロジックをアプリケーションレイヤーまたは AWS Lambda 関数に移動します。

悲観的ロックではなく最適化されたオプティミスティック同時実行制御

Aurora DSQL は、従来のデータベースロックメカニズムとは異なるロックフリーアプローチであるオプティミスティック同時実行制御 (OCC) を使用します。Aurora DSQL では、他のトランザクションをブロックするロックを取得する代わりに、トランザクションをブロックせずに続行し、コミット時に競合を検出できます。これにより、デッドロックが解消され、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ことがなくなります。

主な違い: 競合が発生すると、Aurora DSQL はトランザクションをロックまで待機させるのではなく、シリアル化エラーを返します。これには、従来のデータベースでのロックタイムアウトの処理と同様に、アプリケーションで再試行ロジックを実装する必要がありますが、ブロッキング待機が発生するのではなく、競合はすぐに解決されます。

設計パターン: 再試行メカニズムを使用してべき等トランザクションロジックを実装します。ランダムなプライマリキーを使用し、キー範囲全体に更新を分散することで、競合を最小限に抑えるスキーマを設計します。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。

運用の簡素化

Aurora DSQL は、従来のデータベースメンテナンスタスクの多くを排除し、運用上のオーバーヘッドを削減します。

手動メンテナンスは不要

Aurora DSQL は、ストレージの最適化、統計収集、パフォーマンス調整を自動的に管理します。`VACUUM` のような従来のメンテナンスコマンドはシステムによって処理されます。

ベネフィット: データベースのメンテナンスウィンドウ、バキュームスケジュール、システムパラメータの調整が不要になります。

自動パーティショニングとスケーリング

Aurora DSQL は、アクセスパターンに基づいてデータを自動的にパーティション化して分散します。最適なディストリビューションのために、UUID またはアプリケーション生成 ID を使用します。

移行のヒント: 手動パーティショニングロジックを削除し、Aurora DSQL でデータ分散を処理できるようにします。最適なディストリビューションのために、UUID またはアプリケーション生成 ID を使用します。アプリケーションで連続した識別子が必要な場合は、「シーケンスと ID 列」を参照してください。

AI ツールによるエージェント移行

AI コーディングエージェントは、スキーマを分析し、コードを変換し、組み込みの安全チェックを使用して DDL 移行を実行することで、Aurora DSQL への移行を加速できます。

Kiro を使用した移行

Kiro などのコーディングエージェントは、PostgreSQL コードを分析して Aurora DSQL に移行するのに役立ちます。

- スキーマ分析: 既存のスキーマファイルをアップロードし、Kiro に潜在的な互換性の問題を特定して代替案を提案してもらいます
- コード変換: アプリケーションコードを提供し、トリガーロジックのリファクタリング、シーケンスの UUID への置き換え、トランザクションパターンの変更などを Kiro に依頼します
- 移行計画: Kiro に特定のアプリケーションアーキテクチャに基づいたステップバイステップの移行計画の作成を依頼します
- DDL 移行: 安全チェックとユーザー検証が組み込まれたテーブル再作成パターンを使用してスキーマ変更を実行します

プロンプトの例:

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

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

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

"Drop the legacy_status column from the orders table"

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

テーブルの再作成による DDL 移行

Aurora DSQL MCP サーバーで AI エージェントを使用する場合、特定の ALTER TABLE オペレーションでは、データを安全に移行するテーブル再作成パターンが使用されます。エージェントは複雑な処理を引き受けながら、各ステップで常に情報を提供します。

次のオペレーションでは、テーブルの再作成パターンが使用されます。

Operation	アプローチ
DROP COLUMN	新しいテーブルから列を除外する
ALTER COLUMN TYPE	移行中にデータ型をキャストする
ALTER COLUMN SET/DROP NOT NULL	新しいテーブル定義の制約を変更する
ALTER COLUMN SET/DROP DEFAULT	新しいテーブル定義でデフォルトを定義する
ADD/DROP CONSTRAINT	新しいテーブルに制約を追加または削除する
MODIFY PRIMARY KEY	一意性検証を備えた新しい PK を定義する
列の分割/結合	SPLIT_PART、SUBSTRING、または CONCAT を使用する

次の ALTER TABLE オペレーションは、テーブルの再作成なしで直接サポートされています。

- ALTER TABLE ... RENAME COLUMN – 列名の変更
- ALTER TABLE ... RENAME TO – テーブル名の変更
- ALTER TABLE ... ADD COLUMN – 新しい列を追加する

安全機能: DDL 移行を実行するときに、AI エージェントは移行計画を提示し、データの互換性を検証し、行数を確認し、DROP TABLE などの破壊的な操作を行う前に明示的な承認をリクエストします。

バッチ移行: 3,000 行を超えるテーブルの場合、エージェントはトランザクション制限内に収まるように 500 ~ 1,000 行単位で移行を自動的にバッチ処理します。

Aurora DSQL MCP サーバー

Aurora DSQL モデルコンテキストプロトコル (MCP) サーバーを使用すると、AI アシスタントが Aurora DSQL クラスターに直接接続し、Aurora DSQL ドキュメントを検索できるようになります。これにより、AI は次のことが可能になります。

- 既存のスキーマを分析し、移行の変更を提案します
- テーブル再作成パターンを使用して DDL 移行を実行する
- 移行中にクエリをテストし、互換性を検証します
- 最新の Aurora DSQL ドキュメントに基づいて、正確で最新のガイダンスを提供します

AI アシスタントで Aurora DSQL MCP サーバーを使用するには、Aurora DSQL MCP サーバーのセットアップ手順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 には、分散アーキテクチャ、サーバーレスオペレーション、自動スケーリングを可能にするセルフマネージド PostgreSQL とは異なる機能サポートがあります。ほとんどのアプリケーションは、これらの違いを変更せずにそのまま動作します。

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、クラスターごとに `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。論理的に分離するには、個別の Aurora DSQL クラスターを作成するか、単一のクラスター内でスキーマを使用します。
- `postgres` データベースは UTF-8 文字エンコードを使用しており、幅広い国際文字サポートを提供します。
- 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。Postgres は、タイムゾーンに対応したすべての日付と時刻を内部的に UTC に保存します。TimeZone 設定パラメータを設定して、クライアントへの表示方法を変換し、サーバーが内部的に UTC に変換するために使用するクライアント入力のデフォルトとして機能させることができます。
- 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。
  - DDL オペレーションと DML オペレーションには個別のトランザクションが必要です
  - 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 は、スキーマレベルの許可を通じてアクセス許可を管理します。管理者ユーザーは CREATE SCHEMA を使用してスキーマを作成し、GRANT USAGE ON SCHEMA を使用してアクセス権を付与します。管理者ユーザーはパブリックスキーマ内のオブジェクトを管理しますが、管理者以外のユーザーはユーザーが作成したスキーマ内にオブジェクトを作成して、所有権の境界を明確にします。詳細については、「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」を参照してください。

移行に関してサポートが必要ですか。

移行に不可欠であるにもかかわらず、Aurora DSQL で現在サポートされていない機能に遭遇した場合、「Amazon Aurora DSQL に関するフィードバックの提供」を参照して AWS とフィードバックを共有する方法を確認してください。

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 の主なメリットは、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、パフォーマンスと拡張性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。

同時実行制御レスポンス

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。Aurora DSQL は、競合を検出すると、PostgreSQL シリアル化の失敗を示す SQLSTATE コード `40001` を返します。レスポンスメッセージには、競合のタイプを識別する OCC コードが含まれます。

OC000 — データ競合

2 つのトランザクションが同じ行を変更しようとしました。コミット時間が最も早いトランザクションは成功し、競合するトランザクションは OC000 レスポンスを受け取ります。

```
ERROR: change conflicts with another transaction (OC000) (SQLSTATE 40001)
```

OC001 — スキーマ競合

セッションのキャッシュされたスキーマカタログが古くなっています。セッションがキャッシュをロードしてからカタログバージョンが変更されたことを Aurora DSQL が検出し、トランザクションが現在のバージョンに安全にリベースできない場合、トランザクションは OC001 レスポンスを受け取ります。

```
ERROR: schema has been updated by another transaction (OC001) (SQLSTATE 40001)
```

スキーマカタログを変更するオペレーションでは、DDL ステートメント (`CREATE TABLE` や `ALTER TABLE` など)、`GRANT` ステートメント、`REVOKE` ステートメントなど、OC001 レスポンスが発生する可能性があります。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。

これらのレスポンスを処理するには、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。

同時実行制御レスポンス

データベースカタログは分散されているため、Aurora DSQL は、カタログバージョンを更新する分散トランザクションとして DDL スキーマの変更を管理します。古いバージョンのカタログのキャッシュされたコピーを持つセッションは、次にストレージとやり取りするときに、SQLSTATE コード `40001` や OCC コード `OC001` の同時実行制御レスポンスを受け取ることがあります。

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。これにより、カタログバージョンが更新されます。
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。このセッションには、以前のカタログバージョンがまだキャッシュされています。

   Aurora DSQL は `SQL Error [40001]: ERROR: schema has been updated by another transaction (OC001)` を返します。

> **Note**
>
> OC001 レスポンスは、影響を受けるトランザクションが開始される前にスキーマの変更が既に完了している場合にも発生する可能性があります。Aurora DSQL クエリプロセッサは、クエリの実行中にカタログの変更を事後的に検出するため、アイドル状態のセッションがまだ古いカタログバージョンで動作している可能性があります。再試行すると、セッションはカタログキャッシュを更新し、通常、トランザクションは成功します。

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 ロックすると、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インパクト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。ライト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の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。

シーケンスと ID 列

シーケンスと ID 列は整数値を生成し、コンパクトな識別子や人間が読み取れる識別子が必要な場合に役立ちます。これらの値には、`CREATE SEQUENCE` ドキュメントで説明されている割り当てとキャッシュの動作が含まれます。

トピック

- シーケンス操作関数
- アイデンティティ列
- シーケンスと ID 列の使用

シーケンス操作関数

このセクションでは、シーケンスジェネレーターまたは単にシーケンスとも呼ばれるシーケンスオブジェクトを操作する関数について説明します。シーケンスオブジェクトは、`CREATE SEQUENCE` で作成された特殊な単一行のテーブルです。シーケンスオブジェクトは、通常、テーブルの行の一意の識別子を生成するために使用されます。シーケンス関数は、シーケンスオブジェクトから連続するシーケンス値を取得するためのシンプルでマルチユーザーセーフなメソッドを提供します。

> **Important**
>
> シーケンスを使用する場合は、キャッシュ値を慎重に検討する必要があります。詳細については、「CREATE SEQUENCE」ページの重要なコールアウトを参照してください。
> ワークロードパターンに基づいてシーケンスを使用する最適な方法のガイダンスについては、「シーケンスと ID 列の使用」を参照してください。

関数	説明
`nextval ( regclass ) # bigint`	シーケンスオブジェクトを次の値に進め、その値を返します。これはアトミックに実行されます。複数のセッションが同時に nextval を実行しても、各セッションは個別のシーケンス値を安全に受け取ります。シーケンスオブジェクトがデフォルトのパラメータで作成されている場合、連続する nextval 呼び出しでは 1 から始まる増加する値が返されます。その他の動作は、CREATE SEQUENCE コマンドで適切なパラメータを使用して実現できます。この関数には、シーケンスに対する USAGE または UPDATE 権限が必要です。
`setval ( regclass, bigint [, boolean ] ) # bigint`	シーケンスオブジェクトの現在の値を設定し、オプションでその is_called フラグを設定します。2 つのパラメータフォームでは、シーケンスの last_value フィールドが指定された値に設定され、その is_called フィールドが true に設定されます。つまり、次の nextval は値を返す前にシーケンスを進めます。currval によって報告される値も、指定された値に設定されます。3 つのパラメータフォームでは、is_called を true または false のいずれかに設定できます。true は 2 つのパラメータフォームと同じ効果があります。false に設定すると、次の nextval は指定された値を正確に返し、シーケンスの進行は次の nextval から開始されます。なお、currval によって報告された値はここでは変更されません。例えば、次のようになります。

関数	説明

	`SELECT setval('myseq', 42);  -- Next nextval will return 43`
	`SELECT setval('myseq', 42, true);  -- Same as above`
	`SELECT setval('myseq', 42, false);  -- Next nextval will return 42`
	
	setval によって返される結果は、2 番目の引数の値のみです。この関数には、シーケンスに対する UPDATE 権限が必要です。

currval (regclass) # bigint	現在のセッションで、このシーケンスに対して nextval が最後に取得した値を返します。(このセッションでこのシーケンスに対して nextval が一度も呼び出されなかった場合は、エラーが報告されます。) これはセッション内のローカル値を返すため、現在のセッションが nextval を実行している限り、他のセッションが実行したかどうかに関係なく、予測可能な回答が得られます。この関数には、シーケンスに対する USAGE または SELECT 権限が必要です。
lastval () # bigint	現在のトランザクションで nextval によって最後に返された値を返します。この関数は currval と同じですが、シーケンス名を引数として取る代わりに、現在のトランザクションで最後に適用されたシーケンス nextval を参照する点が異なります。nextval が現在のトランザクションでまだ呼び出されていない場合、lastval を呼び出すとはエラーになります。この関数では、最後に使用されたシーケンスに対する USAGE または SELECT 権限が必要です。

> **Warning**
>
> 呼び出し元のトランザクションが後で中止された場合、nextval によって取得された値は再利用のために回収されません。つまり、トランザクションが中止されたり、データベー

スがクラッシュしたりすると、割り当てられた値のシーケンスにギャップが生じる可能性があります。これは、トランザクションが中止されなくても発生する可能性があります。例えば、ON CONFLICT 句を含む INSERT は、ON CONFLICT ルールに従う原因となる競合を検出する前に、必要な nextval 呼び出しの実行を含め、挿入されるタプルを計算します。したがって、Aurora DSQL のシーケンスオブジェクトは、「ギャップのない」シーケンスを取得するために使用できません。
同様に、setval によって行われたシーケンス状態の変更は、他のトランザクションにすぐに表示され、呼び出し元のトランザクションがロールバックされても元に戻されません。

シーケンス関数によって操作されるシーケンスは、regclass 引数によって指定されます。これは単に pg_class システムカタログ内のシーケンスの OID です。ただし、regclass データ型の入力コンバーターが処理を行うため、OID を手動で検索する必要はありません。詳細については、オブジェクト識別子型に関する PostgreSQL ドキュメントを参照してください。

アイデンティティ列

Important

ID 列を使用する場合は、キャッシュ値を慎重に検討する必要があります。詳細については、「CREATE SEQUENCE」ページの重要なコールアウトを参照してください。
ワークロードパターンに基づいて ID 列を使用する最適な方法のガイダンスについては、「シーケンスと ID 列の使用」を参照してください。

ID 列は、暗黙的なシーケンスから自動的に生成される特殊な列です。キー値の生成に使用できます。ID 列を作成するには、CREATE TABLE で GENERATED ... AS IDENTITY 句を使用します。次に例を示します。

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

または、以下を実行します。

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

```
    ...
);
```

詳細については、「CREATE TABLE」を参照してください。

ID 列を持つテーブルに対して INSERT コマンドが実行され、ID 列に値が明示的に指定されていない場合、暗黙的なシーケンスによって生成された値が挿入されます。例えば、前述の定義で、追加の適切な列を想定して、次のように記述します。

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

id 列の値が 1 から始まり、次のテーブルデータが生成されます。

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

または、値の代わりにキーワード DEFAULT を指定して、シーケンス生成値を明示的にリクエストすることもできます。

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

同様に、キーワード DEFAULT は UPDATE コマンドでも使用できます。

したがって、多くの点で、ID 列はデフォルト値を持つ列のように動作します。

列定義の ALWAYS 句と BY DEFAULT 句は、INSERT コマンドと UPDATE コマンドでユーザーが明示的に指定した値がどのように処理されるかを決定します。INSERT コマンドで ALWAYS を選択した場合、ユーザー指定の値は、INSERT ステートメントで OVERRIDING SYSTEM VALUE が指定されている場合にのみ受け入れられます。BY DEFAULT が選択されている場合、ユーザー指定の値が優先されます。したがって、BY DEFAULT を使用すると、デフォルト値に似た動作になります。デフォルト値は明示的な値で上書きできますが、ALWAYS を使用すると、明示的な値が誤って挿入されることに対する保護が強化されます。

ID 列のデータ型は、シーケンスでサポートされているデータ型のいずれかである必要があります。(「CREATE SEQUENCE」を参照してください。) 関連付けられたシーケンスのプロパティは、ID 列の作成時に指定することも (「CREATE TABLE」を参照)、後で変更することもできます (「ALTER TABLE」を参照)。

ID 列は自動的に NOT NULL とマークされます。ただし、ID 列では一意性が保証されません。(シーケンスは通常一意の値を返しますが、前述のように、シーケンスをリセットしたり、ID 列に値を手動で挿入したりすることもできます)。一意性は、PRIMARY KEY または UNIQUE 制約を使用して強制する必要があります。

シーケンスと ID 列の使用

このセクションでは、ワークロードパターンに基づいてシーケンスと ID 列を使用する最適な方法を理解するのに役立ちます。

> **Important**
>
> 割り当てとキャッシュの動作の詳細については、「CREATE SEQUENCE」ページの重要なコールアウトを参照してください。

識別子タイプの選択

Amazon Aurora DSQL は、UUID ベースの識別子と、シーケンスまたは ID 列を使用して生成された整数値の両方をサポートします。これらのオプションは、値の割り当て方法と負荷時のスケーリング方法が異なります。

UUID 値は調整なしで生成でき、識別子が頻繁に作成されるか、多数のセッションにまたがって作成されるワークロードに適しています。Amazon Aurora DSQL は分散オペレーション用に設計されているため、調整を避けることは多くの場合有益です。このため、特にスケーラビリティが重要で、識別子の厳密な順序付けを必要としないワークロードのプライマリキーの場合、UUID がデフォルトの識別子タイプとしてに推奨されます。

シーケンスと ID 列は、人間が読み取れる識別子、レポート、外部インターフェイスに便利なコンパクトな整数値を生成します。ユーザビリティまたは統合の理由で数値識別子が優先される場合は、UUID ベースの識別子と組み合わせてシーケンスまたは ID 列を使用することを検討してください。整数シーケンスまたは ID 値が必要な場合は、適切なキャッシュサイズを選択することがワークロード設計の重要な部分になります。キャッシュサイズの選択に関するガイダンスについては、次のセクションを参照してください。

キャッシュサイズの選択

適切なキャッシュ値を選択することは、シーケンスと ID 列を効果的に使用する上で重要な部分です。キャッシュ設定は、負荷がかかった状態での識別子の割り当ての動作を決定し、システムスループットと値が割り当て順序をどの程度反映するかに影響を与えます。

CACHE >= 65536 の大きなキャッシュサイズは、次の場合に適しています。

- 識別子が高頻度で生成される
- 多くのセッションが同時に挿入される
- ワークロードがギャップや目に見える順序付け効果を許容できる

例えば、大量のイベント取り込みワークロード (IoT やテレメトリなど) や、ジョブ実行 ID、サポートケースの参照、内部注文番号などの運用識別子は、通常、識別子が頻繁に生成され、厳密な順序付けを必要としないため、キャッシュサイズを大きくするとメリットがあります。

次のような場合、キャッシュサイズ 1 の方が適しています。

- 配分レートが比較的低い
- 識別子が時間の経過とともに割り当て順序により厳密に従うことが予想される
- ギャップを最小限に抑えることがスループットの最大化よりも重要

アカウント番号や参照番号の割り当てなど、識別子の生成頻度が低く、順序を近づけるほうが望ましいワークロードでは、キャッシュサイズを 1 にするとより適切になります。

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**
>
> この更新中に同じネームスペース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ネームスペース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 where job_id = 'wqhu6ewifze5xitg3umt24h5ua';
```

Aurora DSQL は、次のようなレスポンスを返します。

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

[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。

ステータス	説明
submitted	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。
processing	Aurora DSQL はタスクを処理しています。
failed	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。
completed	Aurora DSQL はタスクを正常に完了しました。

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やカタログ、およびシステムに関する情報 (バージョンなど) を取得するための便利なクエリについては、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
pg_am	はい
pg_amop	いいえ

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

名前	Aurora DSQL に適用可能
pg_foreign_table	なし
pg_index	はい
pg_inherits	はい
pg_init_privs	いいえ
pg_language	いいえ
pg_largeobject	なし
pg_largeobject_metadata	はい
pg_namespace	はい
pg_opclass	いいえ
pg_operator	はい
pg_opfamily	いいえ
pg_parameter_acl	はい
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	なし
pg_range	はい
pg_replication_origin	いいえ

名前	Aurora DSQL に適用可能
pg_rewrite	いいえ
pg_seclabel	いいえ
pg_sequence	なし
pg_shdepend	はい
pg_shdescription	はい
pg_shseclabel	いいえ
pg_statistic	はい
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	なし
pg_ts_config	はい
pg_ts_config_map	はい
pg_ts_dict	はい
pg_ts_parser	はい
pg_ts_template	はい
pg_type	はい

名前	Aurora DSQL に適用可能
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	なし
pg_backend_memory_contexts	はい
pg_config	いいえ
pg_cursors	いいえ
pg_file_settings	なし
pg_group	はい
pg_hba_file_rules	いいえ
pg_ident_file_mappings	なし
pg_indexes	はい
pg_locks	いいえ
pg_matviews	いいえ
pg_policies	いいえ
pg_prepared_statements	いいえ
pg_prepared_xacts	いいえ

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

名前	Aurora DSQL に適用可能
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	なし
pg_stat_ssl	はい
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
pg_statio_all_sequences	いいえ
pg_stat_slru	いいえ

名前	Aurora DSQL に適用可能
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
pg_statio_user_indexes	いいえ

sys.jobs ビュー

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;
```

Aurora DSQL は、次のようなレスポンスを返します。

```
           job_id           |  status   | details |  job_type   | class_id | object_id
 |    object_name    |       start_time       |      update_time
----------------------------+-----------+---------+-------------+----------+-----------
+------------------+------------------------+------------------------
 wqhu6ewifze5xitg3umt24h5ua | completed |         | INDEX_BUILD |     1259 |     26433
 | public.nt2_c1_idx | 2025-09-25 22:07:31+00 | 2025-09-25 22:07:46+00
 kkngzf33dndl3daacxehpx5eba | completed |         | ANALYZE     |     1259 |     26419
 | public.nt         | 2025-09-25 21:57:05+00 | 2025-09-25 21:57:27+00
 fyopxjb6ovdn7po6lrkj63cyea | completed |         | DROP        |     1259 |     26422
 |                   | 2025-09-25 22:05:57+00 | 2025-09-25 22:06:03+00
```

次の表では、sys.jobs ビューの列について説明します。

sys.jobs ビューの列

列	型	説明
job_id	text	ジョブを表す base-32 の UUID。
status	text	ジョブの現在のステータス。指定できる値は submitted 、processing 、completed 、failed です。詳細については、「sys.jobs ステータスの値」を参照してください。
details	text	ジョブに関連する詳細。ジョブが失敗した場合、詳細な理由が表示されます。
job_type	text	非同期ジョブのタイプ。指定できる値は次のとおりです。INDEX_BUILD - 非同期インデックスビルド。ANALYZE - システムによって送信された自動分析ジョブ。DROP - DROP TABLE オペレーションま

列	型	説明
		たは DROP INDEX オペレーションの後に物理データを削除します。
class_id	oid	オブジェクトを含むカタログテーブルの OID。
object_id	oid	オブジェクトの OID。
object_name	text	オブジェクトの完全修飾名。DROP ジョブは、既に削除されたオブジェクトを参照できません。参照したオブジェクトが既に削除されている場合、object_name は NULL である可能性があります。
start_time	timestamp with time zone	ジョブが送信されたタイムスタンプ。
update_time	timestamp with time zone	ジョブの行が最後に更新されたタイムスタンプ。

sys.jobs ステータスの値

ステータス	説明
submitted	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。
processing	Aurora DSQL はタスクを処理しています。
failed	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。
completed	Aurora DSQL はタスクを正常に完了しました。

sys.iam_pg_role_mappings ビュー

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;
```

役立つシステムメタデータクエリ

これらのクエリを使用して、テーブルのフルスキャンなどの高コストのオペレーションを実行せずに、テーブル統計とシステムメタデータを取得します。

テーブルの推定行数を取得する

テーブルのフルスキャンを実行せずにテーブル内の行数の概算数を取得するには、次のクエリを使用します。

```
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
```

このアプローチは、Aurora DSQL の大きなテーブルに対する SELECT COUNT(*) よりも効率的です。

現在の Aurora DSQL メジャーバージョンを取得する

Aurora DSQL クラスターの現在のメジャーバージョンを取得するには、次のクエリを使用します。

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

このコマンドにより、以下のような出力が返されます。

```
 dsql_major_version
--------------------
```

```
                    1
```

これにより、Aurora DSQL で SQL 接続がオンになっているメジャーバージョンが返されます。

現在の PostgreSQL バージョンを取得する

Aurora DSQL クラスターの現在の PostgreSQL バージョンを取得するには、次のクエリを実行します。

```
SHOW server_version;
```

このコマンドにより、以下のような出力が返されます。

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

これにより、Aurora DSQL で SQL 接続がオンになっている PostgreSQL バージョンが返されます。

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処理として 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 EXPLAIN プランの操作

Aurora DSQL は PostgreSQL と同様の EXPLAIN プラン構造を使用しますが、分散アーキテクチャと実行モデルを反映するキーが追加されています。

このドキュメントでは、Aurora DSQL EXPLAIN プランの概要を示し、PostgreSQL と比較した類似点と相違点を強調します。Aurora DSQL で使用できるさまざまなタイプのスキャンオペレーションについて説明し、クエリの実行コストを理解するのに役立ちます。

PostgreSQL と Aurora DSQL EXPLAIN プラン

Aurora DSQL は PostgreSQL データベース上に構築され、ほとんどの計画構造を PostgreSQL と共有しますが、クエリの実行と最適化に影響するアーキテクチャ上の主な違いがあります。

機能	PostgreSQL	Aurora DSQL
データストレージ	ヒープストレージ	ヒープなし。すべての行は一意の識別子によってインデックスが作成されます
プライマリキー	プライマリキーインデックスはテーブルデータとは異なります	プライマリキーインデックスは、すべての追加列を INCLUDE 列とするテーブルです。
セカンダリインデックス	標準セカンダリインデックス	PostgreSQL と同じように動作し、キー以外の列を含める機能があります。

機能	PostgreSQL	Aurora DSQL
フィルタリング機能	インデックス条件、ヒープフィルター	インデックス条件、ストレージフィルター、クエリプロセッサフィルター
スキャンタイプ	シーケンシャルスキャン、インデックススキャン、インデックスのみのスキャン	フルスキャン、インデックスのみのスキャン、インデックススキャン
クエリの実行	データベースのローカル	分散 (コンピューティングとストレージは別)

Aurora DSQL は、テーブルデータを個別のヒープではなく、プライマリキーの順序で直接保存します。各行は一意のキー、通常はプライマリキーによって識別されるため、データベースの検索をより効率的に最適化できます。アーキテクチャの違いは、PostgreSQL がシーケンシャルスキャンを選択する可能性がある場合に、Aurora DSQL がインデックスのみのスキャンを使用することが多い理由の説明になります。

もう 1 つの主な違いは、Aurora DSQL がコンピューティングをストレージから分離し、実行パスの早い段階でフィルターを適用してデータの移動を減らし、パフォーマンスを向上させることです。

PostgreSQL での EXPLAIN プランの使用の詳細については、「PostgreSQL EXPLAIN documentation」を参照してください。

Aurora DSQL EXPLAIN プランの主な要素

Aurora DSQL EXPLAIN プランは、フィルタリングが発生する場所やストレージから取得される列を含め、クエリの実行方法に関する詳細情報を提供します。この出力を理解することにより、クエリのパフォーマンスを最適化できます。

インデックス条件

インデックスのナビゲーションに使用される条件。スキャンされたデータを削減するのに最も効率的なフィルタリング。Aurora DSQL では、インデックス条件を実行プランの複数のレイヤーに適用できます。

計画

ストレージから取得した列。射影が少ないほど、パフォーマンスが向上します。

ストレージフィルター

ストレージレベルで適用される条件。クエリプロセッサフィルターよりも効率的です。

クエリプロセッサフィルター

クエリプロセッサレベルで適用される条件。フィルタリングの前にすべてのデータを転送する必要があるため、データの移動と処理のオーバーヘッドが増加します。

Aurora DSQL のフィルター

Aurora DSQL はストレージからコンピューティングを分離します。つまり、クエリの実行中にフィルターが適用されるポイントはパフォーマンスに大きな影響を与えます。大量のデータが転送される前に適用されるフィルターは、レイテンシーを低減し、効率を向上させます。フィルターの適用が早いほど、処理、移動、スキャンが必要なデータが少なくなり、クエリが高速化されます。

Aurora DSQL は、クエリパスの複数のステージでフィルターを適用できます。これらのステージを理解することは、クエリプランを解釈し、パフォーマンスを最適化する上で重要です。

レベル	フィルタータイプ	Description
1	インデックス条件	インデックスのスキャン中に適用されます。ストレージから読み取るデータの量を制限し、コンピューティングレイヤーに送信されるデータを削減します。
2	ストレージフィルター	データがストレージから読み取られた後、コンピューティングに送信される前に適用されます。インデックスの include 列のフィルターの例を以下に示します。データ転送は減少しますが、読み取り量は減少しません。
3	クエリプロセッサフィルター	データがコンピューティングレイヤーに到達した後に適用されます。すべてのデータを最初に転送する必要があるため、レイテンシーとコストが増加します。現在、Aurora DSQL はストレージ上ですべてのフィルタリングと射影オペレーションを実行できないため、一部のクエリはこのタイプのフィルタリ

レベル	フィルタータイプ	Description
		ングに強制的にフォールバックされる可能性があります。

Aurora DSQL EXPLAIN プランの読み取り

EXPLAIN プランの読み方を理解することは、クエリのパフォーマンスを最適化するうえで重要です。このセクションでは、Aurora DSQL クエリプランの実際の例を説明し、さまざまなスキャンタイプの動作を示し、フィルターが適用される場所を説明し、最適化の機会を強調します。

これらの例で使用するサンプルテーブル

以下の例では、`transaction` と `account` の 2 つのテーブルを参照します。

`transaction` テーブルにはプライマリキーがないため、Aurora DSQL はクエリ時にテーブルのフルスキャンを実行します。

`account` テーブルには `customer_id` のインデックスがあります。このインデックスには `balance` と `status` がカバーリング列として含まれているため、ベーステーブルから読み取ることなく、特定のクエリをインデックスから直接完結できます。ただし、インデックスには `created_at` が含まれていないため、この列を参照するクエリには追加のテーブルアクセスが必要です。

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

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

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

フルスキャンの例

Aurora DSQL には、PostgreSQL と機能的に同じシーケンシャルスキャンとフルスキャンの両方があります。これら 2 つの唯一の違いは、フルスキャンはストレージで追加のフィルタリングを利用できることです。このため、ほとんどの場合、上記のシーケンシャルスキャンが選択されます。類似しているため、より魅力的なフルスキャンの例のみを取り上げます。

フルスキャンは、主にプライマリキーのないテーブルで使用されます。Aurora DSQL プライマリキーはデフォルトでフルカバーインデックスであるため、PostgreSQL がシーケンシャルスキャンを使用する多くの状況では、Aurora DSQL はほとんどの場合、プライマリキーでインデックスのみのスキャンを使用します。他のほとんどのデータベースと同様に、インデックスのないテーブルは正しくスケールされません。

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

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------
 Full Scan (btree-table) on transaction  (cost=125100.05..177933.38 rows=33333
 width=16)
   Filter: (description ~~ '%external%'::text)
   -> Storage Scan on transaction (cost=12510.05..17793.38 rows=66666 width=16)
        Projections: account_id, description
        Filters: (transaction_date > '2025-01-01 00:00:00'::timestamp without time
 zone)
        -> B-Tree Scan on transaction (cost=12510.05..17793.38 rows=100000 width=30)
```

このプランには、異なるステージに適用される 2 つのフィルターが表示されます。`transaction_date > '2025-01-01'` 条件はストレージレイヤーに適用され、返されるデータの量を減らします。`description LIKE '%external%'` 条件は、データが転送された後、クエリプロセッサで後に適用されるため、効率が低下します。より選択的なフィルターをストレージまたはインデックスレイヤーにプッシュすると、一般的にパフォーマンスが向上します。

インデックスのみのスキャンの例

インデックスのみのスキャンは、ストレージレイヤーへのラウンドトリップが最も少なく、最もフィルタリングできるため、Aurora DSQL で最適なスキャンタイプです。ただし、インデックスのみのスキャンが表示されるからといって、最適なプランとは限りません。発生する可能性のあるさまざまなレベルのフィルタリングのため、フィルタリングが発生する可能性のあるさまざまな場所に引き続き注意することが重要です。

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

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

このプランでは、インデックス条件 customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb') がインデックススキャン中に最初に評価されます。これは、ストレージから読み取るデータの量を制限するため、最も効率的なステージです。ストレージフィルター status = 'pending' は、データの読み取り後、コンピューティングレイヤーに送信される前に適用され、転送されるデータ量を削減します。最後に、クエリプロセッサフィルター balance > 100 は、データが移動された後に最後に実行されるため、最も効率的ではありません。これらのうち、インデックス条件はスキャンされるデータ量を直接制御するため、最高のパフォーマンスを提供します。

インデックススキャンの例

インデックススキャンはインデックスのみのスキャンに似ていますが、ベーステーブルに呼び出す追加のステップがあります。Aurora DSQL はストレージフィルターを指定できるため、インデックス呼び出しとルックアップ呼び出しの両方で指定できます。

これを明確にするために、Aurora DSQL はプランを 2 つのノードとして示します。このように、インクルード列の追加がストレージから返される行に関してどの程度役立つかを明確に確認できます。

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

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

このプランでは、フィルタリングが複数のステージ間でどのように行われるかを示します。

- customer_id のインデックス条件は、データを早期にフィルタリングします。
- status のストレージフィルターは、コンピューティングに送信される前に結果をさらに絞り込みます。
- balance のクエリプロセッサフィルターは、転送後に適用されます。
- created_at のルックアップフィルターは、ベーステーブルから追加の列を取得するときに評価されます。

頻繁に使用される列を INCLUDE フィールドとして追加すると、多くの場合、このルックアップが解消され、パフォーマンスが向上します。

ベストプラクティス

- フィルターをインデックス化された列に合わせて、フィルタリングを早めにプッシュします。
- INCLUDE 列を使用して、インデックスのみのスキャンを許可し、ルックアップを回避します。
- パフォーマンスの問題を調査するときに、行数の見積もりを検証します。Aurora DSQL は、データ変更率に基づいてバックグラウンドで ANALYZE を実行することで、統計を自動的に管理します。見積りが不正確に見える場合は、ANALYZE を手動で実行して統計を即座に更新できます。
- 大きなテーブルに対するインデックス化されていないクエリを回避し、ワークロードの高いフルスキャンを防ぎます。

EXPLAIN ANALYZE の DPU について

Aurora DSQL は、EXPLAIN ANALYZE VERBOSE プラン出力にステートメントレベルの分散処理ユニット (DPU) 情報を提供し、開発中のクエリコストをより詳細に可視化します。このセクションでは、DPU とは何か、EXPLAIN ANALYZE VERBOSE 出力でそれらを解釈する方法について説明します。

DPU とは

分散処理ユニット (DPU) は、Aurora DSQL によって実行された作業の標準化された測定単位です。これは以下で構成されます。

- ComputeDPU – SQL クエリの実行にかかった時間
- ReadDPU – ストレージからデータを読み取るために使用されるリソース
- WriteDPU - ストレージへのデータの書き込みに使用されるリソース
- MultiRegionWriteDPU – マルチリージョン設定でピア接続されたクラスターへの書き込みをレプリケートするために使用されるリソース

EXPLAIN ANALYZE VERBOSE での DPU の使用

Aurora DSQL は EXPLAIN ANALYZE VERBOSE を拡張して、ステートメントレベルの DPU 使用量の見積もりを出力の最後に含めます。これにより、クエリコストがすぐに可視化され、ワークロードのコスト要因の特定、クエリのパフォーマンスの調整、リソース使用量の予測に役立ちます。

次の例は、EXPLAIN ANALYZE VERBOSE 出力に含まれるステートメントレベルの DPU 見積もりを解釈する方法を示しています。

例 1: SELECT クエリ

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

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

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

この例では、SELECT ステートメントはインデックスのみのスキャンを実行するため、ほとんどのコストは、ストレージから取得されたデータを表す読み取り DPU (0.04312) と、結果を処理して返すために使用されるコンピューティングリソースを反映したコンピューティング DPU (0.01607) から発生します。クエリはデータを変更しないため、書き込み DPU はありません。合計 DPU (0.05919) は、コンピューティング + 読み取り + 書き込みの合計です。

例 2: INSERT クエリ

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

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

このステートメントは主に書き込みを実行するため、ほとんどのコストは書き込み DPU に関連付けられます。コンピューティング DPU (0.01550) は、値を処理して挿入するために行われた作業を表

します。読み取り DPU (0.00307) は、マイナーシステム読み取り (カタログルックアップまたはインデックスチェックの場合) を反映します。

読み取りおよび書き込み DPU の横に表示されるトランザクションの最小値に注意してください。これらは、オペレーションに読み取りまたは書き込みが含まれている場合にのみ適用されるトランザクションあたりのベースラインコストを示します。すべてのトランザクションで 0.00375 読み取り DPU または 0.05 書き込み DPU 料金が自動的に発生するわけではありません。代わりに、これらの最小値は、コスト集計中に、そのトランザクション内で読み取りまたは書き込みが行われた場合にのみ、トランザクションレベルで適用されます。このスコープの違いにより、EXPLAIN ANALYZE VERBOSE のステートメントレベルの見積もりが、CloudWatch または請求データで報告されたトランザクションレベルのメトリクスと正確に一致しない場合があります。

最適化のための DPU 情報の使用

ステートメントごとの DPU 見積もりは、実行時間を超えてクエリを最適化する強力な方法を提供します。一般的ユースケースには以下が含まれます。

- コスト認識: クエリが他のクエリと比較してどのくらいのコストがかかるかを把握します。
- スキーマの最適化: インデックスまたはスキーマの変更がパフォーマンスとリソース効率の両方に与える影響を比較します。
- Budget Planning: 観測された DPU 使用量に基づいてワークロードコストを見積もります。
- クエリ比較: 代替クエリアプローチを相対的な DPU 消費量で評価します。

DPU 情報の解釈

EXPLAIN ANALYZE VERBOSE の DPU データを使用する場合は、次のベストプラクティスに注意してください。

- 方向的に使用する: 報告されたDPUを、CloudWatchメトリクスや請求データとの厳密な一致ではなく、クエリの相対的なコストを理解する手段として扱います。EXPLAIN ANALYZE VERBOSE はステートメントレベルのコストを報告し、CloudWatch はトランザクションレベルのアクティビティを集計するため、差異が想定されます。CloudWatch には、EXPLAIN ANALYZE VERBOSE が意図的に除外するバックグラウンドオペレーション (ANALYZE や圧縮など) とトランザクションオーバーヘッド (BEGIN/COMMIT) も含まれています。
- 実行間の DPU の変動は分散システムでは正常であり、エラーを示すものではありません。キャッシュ、実行計画の変更、同時実行、データ分散のシフトなどの要因により、同じクエリが 1 回の実行から次の実行まで異なるリソースを消費する可能性があります。

- バッチスモールオペレーション: ワークロードが多数のスモールステートメントを発行する場合は、それらをより大きなオペレーション (10MB を超えない) にバッチ処理することを検討してください。これにより、四捨五入オーバーヘッドが削減され、より意味のあるコスト見積もりが生成されます。
- 請求ではなく調整に使用する: EXPLAIN ANALYZE VERBOSE の DPU データは、コスト認識、クエリ調整、最適化のために設計されています。請求グレードのメトリクスではありません。信頼できるコストと使用状況データについては、常に CloudWatch メトリクスまたは毎月の請求レポートに依存します。

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

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

AWS CLI または Python、C++、JavaScript、Java、Rust、Ruby、.NET、Golang などの任意のプログラミング言語を使用して、AWS リージョン のクラスターを設定および管理します。AWS CLI はシェルコマンドによるクイックアクセスを提供し、AWS Software Development Kit (SDK) はネイティブ言語サポートを使用した制御をプログラムで行えるようにします。

トピック

AWS SDK の使用

AWS SDK は、任意のプログラミング言語でプログラムによる Aurora DSQL へのアクセスを提供します。次のセクションで、さまざまなプログラミング言語を使用して一般的なクラスター操作を実行する方法を示します。

クラスターを作成する

次の例は、さまざまなプログラミング言語を使用して単一リージョンクラスターを作成する方法を示しています。

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()
```

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;
}
```

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();
```

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);
        }
    }
}
```

Rust

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。

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

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

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

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

    Client::from_conf(config)
}

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

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

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

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

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

    cluster_output
}
```

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let region = "us-east-1";

    let cluster = create_cluster(region).await;

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

    Ok(())
}
```

Ruby

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。

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

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

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

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

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

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

  cluster = create_cluster(region)

  puts "Created single region cluster:"
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

.NET

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。

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

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

```
        /// <summary>
        /// Create a cluster with deletion protection enabled and a name tag.
        /// </summary>
        public static async Task<CreateClusterResponse> Create(RegionEndpoint
 region)
        {
            using (var client = await CreateDSQLClient(region))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp single region cluster" }
                };

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

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

                return cluster;
            }
        }

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

            var cluster = await Create(region);

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

Golang

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。

```
package main
```

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

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

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

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

 client := dsql.NewFromConfig(cfg)

 deleteProtect := true

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

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

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

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

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

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

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

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

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

}
```

クラスターを取得する

次の例は、さまざまなプログラミング言語を使用して単一リージョンクラスターの情報を取得する方法を示しています。

Python

単一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)
 }
}
```

クラスターの更新

次の例は、さまざまなプログラミング言語を使用して単一リージョンクラスターを更新する方法を示しています。

Python

単一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)
 }
}
```

クラスターの削除

次の例は、さまざまなプログラミング言語を使用して単一リージョンクラスターを削除する方法を示しています。

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()
```

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;
}
```

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();
```

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);
        }
    }
}
```

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(())
}
```

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。

```
require "aws-sdk-dsql"

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

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

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

main if $PROGRAM_NAME == __FILE__
```

.NET

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);
        }
    }
}
```

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)
 }

}
```

その他のコードサンプルと例については、「Aurora DSQL Samples GitHub repository」を参照してください。

AWS CLI を使用する

AWS CLI は、Aurora DSQL クラスターを管理するためのコマンドラインインターフェイスを提供します。次の例では、下記に示すような一般的なクラスター管理操作をデモンストレーションします。

クラスターを作成する

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応答

```
{
    "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応答

```
{
    "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応答

```
{
    "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応答

```
{
    "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応答

```
{
    "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

AWS CLI または Python、C++、JavaScript、Java、Rust、Ruby、.NET、Golang などの任意のプログラミング言語を使用して、複数の AWS リージョン 間でクラスターを設定および管理します。AWS CLI はシェルコマンドによるクイックアクセスを提供し、AWS SDK はネイティブ言語サポートを使用した制御をプログラムで行えるようにします。

トピック

- AWS SDK の使用
- AWS CLI を使用する

AWS SDK の使用

AWS SDK は、任意のプログラミング言語でプログラムによる Aurora DSQL へのアクセスを提供します。次のセクションで、さまざまなプログラミング言語を使用して一般的なクラスター操作を実行する方法を示します。

クラスターを作成する

次の例は、さまざまなプログラミング言語を使用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。

Python

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={
                'Delay': 10,
                'MaxAttempts': 30
            }
        )

        return (cluster_1, cluster_2)

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


def main():
    region_1 = "us-east-1"
    region_2 = "us-east-2"
    witness_region = "us-west-2"
    (cluster_1, cluster_2) = create_multi_region_clusters(region_1, region_2,
 witness_region)
    print("Created multi region clusters:")
    print("Cluster id: " + cluster_1['arn'])
    print("Cluster id: " + cluster_2['arn'])


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

C++

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);

    std::cout << "Creating cluster in " << region1 << std::endl;

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

    // Set multi-region properties with witness region
    MultiRegionProperties multiRegionProps1;
```

```
    multiRegionProps1.SetWitnessRegion(witnessRegion);
    createClusterRequest1.SetMultiRegionProperties(multiRegionProps1);

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

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

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

    // Create second cluster
    std::cout << "Creating cluster in " << region2 << std::endl;

    CreateClusterRequest createClusterRequest2;
    createClusterRequest2.SetDeletionProtectionEnabled(true);

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

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

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

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

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

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

    MultiRegionProperties updatedProps;
    updatedProps.SetWitnessRegion(witnessRegion);

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

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

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

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

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

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

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

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

JavaScript

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 updateClusterCommand = new UpdateClusterCommand({
            identifier: response1.identifier,
            multiRegionProperties: {
                witnessRegion: witnessRegion,
                clusters: [response2.arn]
            }
        });
        await client1.send(updateClusterCommand);
        console.log(`Added ${response2.arn} as a peer of ${response1.arn}`);

        // Now that multiRegionProperties is fully defined for both clusters they'll
begin the transition to ACTIVE
        console.log(`Waiting for cluster ${response1.identifier} to become ACTIVE`);
        await waitUntilClusterActive(
            {
                client: client1,
                maxWaitTime: 300 // Wait for 5 minutes
            },
            {
                identifier: response1.identifier
            }
        );
        console.log(`Cluster 1 is now active`);

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

```
            },
            {
                identifier: response2.identifier
            }
        );
        console.log(`Cluster 2 is now active`);
        console.log("The multi region clusters are now active");
        return;
    } catch (error) {
        console.error("Failed to create cluster: ", error.message);
        throw error;
    }
}

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

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();
```

Java

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

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

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__
```

Golang

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)
 }

}
```

.NET

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}");
        }
    }
}
```

クラスターを取得する

次の例は、さまざまなプログラミング言語を使用してマルチリージョンクラスターの情報を取得する方法を示しています。

Python

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、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

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、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、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、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、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、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)
 }
}
```

クラスターの更新

次の例は、さまざまなプログラミング言語を使用してマルチリージョンクラスターを更新する方法を示しています。

Python

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、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

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、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、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、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、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、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)
 }
}
```

クラスターの削除

次の例は、さまざまなプログラミング言語を使用してマルチリージョンクラスターを削除する方法を示しています。

Python

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 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++

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

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

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

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

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

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

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)
 }
}
```

その他のコードサンプルと例については、「Aurora DSQL Samples GitHub repository」を参照してください。

AWS CLI を使用する

AWS CLI は、マルチリージョン Aurora DSQL クラスターを管理するためのコマンドラインインターフェイスを提供します。次の例は、マルチリージョンクラスターを作成、設定、削除する方法を示しています。

マルチリージョンクラスターに接続する

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を作成する

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応答

```
{
    "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応答

```
{
    "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を表示する

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応答

```
{
    "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を削除する

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 Aurora DSQL クラスターの設定

同じ 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 クラスターのライフサイクル

Aurora DSQL クラスターのライフサイクルを理解することで、クラスターを効果的に管理できます。このセクションでは、クラスターのステータス定義と、コスト最適化のためのゼロにスケールする機能について説明します。

Aurora DSQL クラスターのステータスの定義

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 マネジメントコンソール、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。

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	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 が実行中のリソースをスケールダウンして容量とコストを削減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。アイドル状態のクラスターに接続すると、Aurora DSQL はクラスターを [アクティブ] ステータスに戻します。
非アクティブな	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、アイドルクラスターは非アクティブになります。この中断状態では、実行中のリソースは、データが保持されている間、ゼロにスケールされます。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] ステータスに戻します。復元する時間は、クラスターのサイズによって異なります。
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。

ステータス	説明
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] ステータスに移行します。

アイドルクラスターと非アクティブクラスターの使用

Aurora DSQL は、クラスターで一定期間接続アクティビティが検出されない場合、クラスターを [アイドル] 状態に移行し、実行中のリソースを減らして容量とコストを最小限に抑えます。接続アクティビティが長期間存在しない場合、[アイドル] クラスターは自動的に [非アクティブ] 状態に移行し、実行中のリソースはデータの保存中にゼロにスケールされます。

通常のオペレーションを再開するには、通常どおりクラスターに接続します。クラスターに正常に接続すると、Aurora DSQL は自動的にクラスターをアクティブ状態に移行します。

> **Note**
>
> [アイドル] または [非アクティブ] クラスターへの最初の接続試行は、通常よりも遅くなります。

アクティブクラスター状態を必要とするオペレーション

一部のオペレーションでは、クラスターがアクティブ状態である必要があります。これらのオペレーションを [アイドル] クラスターまたは [非アクティブ] クラスターで実行するには、クラスターに接続してクラスターをアクティブに戻す必要があります。

バックアップオペレーション

バックアップを取得するには、クラスターの状態アクティブであることが必要です。クラスターが [アイドル] または [非アクティブ] の場合、バックアップは次のエラーで失敗します。

```
"Error": {
    "Code": "FailedPrecondition",
    "Message": "Cluster 'cluster-id' is in state 'IDLE' and can't be backed up.
    In order to take a backup of your cluster, it must be in Active state. Please
    connect to your cluster to transition it to Active to perform the backup."
}
```

バックアップを続行するには:

1. 任意のデータベースクライアントまたは Aurora DSQL コンソールを使用してクラスターに接続し、起動します。
2. [アクティブ] 状態への自動移行を待ちます。
3. クラスターが完全に動作可能になった後、バックアップを開始します。

> **Note**
>
> クラスターがアイドルまたは非アクティブになる前に実行された既存のバックアップは、引き続き有効であり、影響を受けません。クラスターで新しいバックアップの試行は、クラスターが自動的に起動するために接続されるまで失敗します。

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 マネジメントコンソール、AWS CLI、または Aurora DSQL API を使用します。

コンソール

AWS マネジメントコンソール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
```

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。

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 CloudFormation リソースです。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。

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。

Aurora DSQL 用コネクタ

Aurora DSQL は、既存のデータベースドライバーを拡張してシームレスな IAM 認証と AWS サービスとの統合を可能にする特殊なコネクタを提供します。これらのコネクタは、既存の PostgreSQL ワークフローとの互換性を維持しながら、一般的なプログラミング言語やフレームワークで動作するように設計されています。

今後のリリースでは、追加のコネクタが予定されています。コネクタの高可用性に関する最新情報については、「Aurora DSQL samples repository」を参照してください。

Java JDBC 用 Aurora DSQL コネクタ

JDBC 用 Aurora DSQL コネクタは、PostgreSQL JDBC ドライバーの機能を拡張して、アプリケーションが IAM 認証情報を使用して Aurora DSQL で認証できるようにする認証プラグインとして設計されています。コネクタはデータベースに直接接続しませんが、基盤となる PostgreSQL JDBC ドライバー上でシームレスな IAM 認証を提供します。

JDBC 用 Aurora DSQL コネクタは PostgreSQL JDBC ドライバーで動作するように構築されており、Aurora DSQL の IAM 認証要件とシームレスに統合できます。

PostgreSQL JDBC ドライバーと組み合わせて、JDBC 用 Aurora DSQL コネクタは Aurora DSQL の IAM ベースの認証を有効にします。AWS Identity and Access Management (IAM) などの AWS 認証サービスとの密接な統合が導入されました。

コネクタについて

Aurora DSQL は、PostgreSQL 互換アプリケーションに高可用性とスケーラビリティを提供する分散 SQL データベースサービスです。Aurora DSQL には、既存の JDBC ドライバーがネイティブにサポートしていない時間制限付きトークンを使用した IAM ベースの認証が必要です。

JDBC 用 Aurora DSQL コネクタの主な考え方は、IAM トークン生成を処理する PostgreSQL JDBC ドライバーの上に認証レイヤーを追加することです。これにより、ユーザーは既存の JDBC ワークフローを変更せずに Aurora DSQL に接続できます。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます

- トークン生成: 認証トークンは AWS 認証情報を使用して生成され、設定可能な有効期間があります

JDBC 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成するように設計されています。

JDBC 用 Aurora DSQL コネクタの利点

Aurora DSQL は PostgreSQL 互換インターフェイスを提供しますが、既存の PostgreSQL ドライバーは現在、Aurora DSQL の IAM 認証要件をサポートしていません。JDBC 用 Aurora DSQL コネクタを使用すると、以下の方法で IAM 認証を有効にしながら、既存の PostgreSQL ワークフローを引き続き使用できます。

- 自動トークン生成: IAM トークンは AWS 認証情報を使用して自動的に生成されます
- シームレスな統合: 既存の JDBC 接続パターンで動作します
- AWS 認証情報のサポート: さまざまな AWS 認証情報プロバイダー (デフォルト、プロファイルベースなど) をサポートします。

接続プーリングでの JDBC 用 Aurora DSQL コネクタの使用

JDBC 用 Aurora DSQL コネクタは、HikariCP などの接続プーリングライブラリで動作します。コネクタは、接続の確立中に IAM トークンの生成を処理し、接続プールが正常に動作できるようにします。

主な特徴

自動トークンの生成

IAM トークンは AWS 認証情報を使用して自動的に生成されます。

シームレスな統合

ワークフローの変更を必要とせずに、既存の JDBC 接続パターンと連携します。

AWS 認証情報のサポート

さまざまな AWS 認証情報プロバイダー (デフォルト、プロファイルベースなど) をサポートします。

接続プーリングの互換性

HikariCP などの接続プーリングライブラリとシームレスに連携します。

前提条件

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。

- Aurora DSQL でクラスターを作成します。
- Java 開発キット (JDK) をインストールします。バージョンが 17 以上であることを確認します。
- アプリケーションが Aurora DSQL に接続できるように、適切な IAM アクセス許可を設定します。
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由) 。

基本的な接続での JDBC 用 Aurora DSQL コネクタの使用

Java アプリケーションで JDBC 用 Aurora DSQL コネクタを使用するには、次の手順に従います。

1. 以下の依存関係を Maven プロジェクトに追加します。

```
<dependencies>
    <!-- Aurora DSQL Connector for JDBC -->
    <dependency>
        <groupId>software.amazon.dsql</groupId>
        <artifactId>aurora-dsql-jdbc-connector</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

   Gradle プロジェクトの場合は、この依存関係を追加します。

```
implementation("software.amazon.dsql:aurora-dsql-jdbc-connector:1.0.0")
```

2. AWS DSQL PostgreSQL コネクタ形式を使用して、Aurora DSQL クラスターへの基本的な接続を作成します。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DsqlJdbcConnectorExample {
    public static void main(String[] args) {
        // Using AWS DSQL PostgreSQL Connector prefix
```

```
        String jdbcUrl = "jdbc:aws-dsql:postgresql://your-cluster.dsql.us-
east-1.on.aws/postgres?user=admin";

        try (Connection connection = DriverManager.getConnection(jdbcUrl)) {
            // Use the connection
            try (Statement statement = connection.createStatement()) {
                // Create a table
                statement.execute("CREATE TABLE IF NOT EXISTS test_table (id UUID
 PRIMARY KEY DEFAULT gen_random_uuid(), name VARCHAR(100))");

                // Insert data
                statement.execute("INSERT INTO test_table (name) VALUES ('Test
 Name')");

                // Query data
                try (ResultSet resultSet = statement.executeQuery("SELECT * FROM
 test_table")) {
                    while (resultSet.next()) {
                        System.out.println("ID: " + resultSet.getInt("id") + ",
 Name: " + resultSet.getString("name"));
                    }
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

設定プロパティ

JDBC 用 Aurora DSQL コネクタは、次の接続プロパティをサポートしています。

ユーザー

接続ユーザーと使用されるトークン生成方法を決定します。例:admin

token-duration-secs

トークン有効期間の秒単位での期間。トークンの制限の詳細については、「Amazon Aurora DSQL での認証トークンの生成」を参照してください。

profile

指定されたプロファイル名を使用したトークン生成用の ProfileCredentialsProvider をインスタンス化するために使用されます。

リージョン

Aurora DSQL 接続の AWS リージョン。これは省略可能です。指定された場合、URL から抽出されたリージョンは上書きされます。

データベース

接続するデータベース名。デフォルトは `postgres` です。

ログ記録

Aurora DSQL JDBC コネクタの使用中に発生する可能性のある問題をトラブルシューティングするためのログ記録を有効にします。

コネクタは、Java の組み込みログ記録システム (java.util.logging) を使用します。`logging.properties` ファイルを作成することにより、ログ記録レベルを設定できます。

```
# Set root logger level to INFO for clean output
.level = INFO

# Show Aurora DSQL Connector for JDBC FINE logs for detailed debugging
software.amazon.dsql.level = FINE

# Console handler configuration
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Detailed formatter pattern with timestamp and logger name
java.util.logging.SimpleFormatter.format = %1$tH:%1$tM:%1$tS.%1$tL [%4$s] %3$s - %5$s%n
```

例

より包括的な例とユースケースについては、以下を含む「JDBC 用 Aurora DSQL コネクタレポジトリ」を参照してください。

Python 用 Aurora DSQL コネクタ

Python 用 Aurora DSQL コネクタは、Python アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合します。内部的には、psycopg、psycopg2、および asyncpg クライアントライブラリを使用します。

Python 用 Aurora DSQL コネクタは psycopg、psycopg2、および asyncpg のクライアントライブラリ機能を拡張して、アプリケーションが IAM 認証情報を使用して Amazon Aurora DSQL で認証できるようにする認証プラグインとして設計されています。コネクタはデータベースに直接接続しませんが、基盤となるクライアントライブラリ上にシームレスな IAM 認証を提供します。

コネクタについて

Amazon Aurora DSQL は、PostgreSQL 互換アプリケーションに高可用性とスケーラビリティを提供する分散 SQL データベースサービスです。Aurora DSQL には、既存の Python ライブラリがネイティブにサポートしていない時間制限付きトークンを使用した IAM ベースの認証が必要です。

Python 用 Aurora DSQL コネクタの考え方は、IAM トークン生成を処理する psycopg、psycopg2、および asyncpg のクライアントライブラリの上に認証レイヤーを追加することです。これにより、ユーザーは既存のワークフローを変更せずに Aurora DSQL に接続できます。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: 認証トークンは AWS 認証情報を使用して生成され、設定可能な有効期間があります

Python 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成するように設計されています。

機能

- 自動 IAM 認証 - AWS 認証情報を使用して IAM トークンを自動的に生成
- psycopg、psycopg2、および asyncpg 上に構築 - psycopg、psycopg2、および asyncpg クライアントライブラリを活用
- シームレスな統合 - ワークフローの変更を必要とせずに、既存の psycopg、psycopg2、および asyncpg 接続パターンと連携

- リージョンの自動検出 - DSQL クラスターホスト名から AWS リージョンを抽出
- AWS 認証情報のサポート: さまざまな認証情報プロバイダー (デフォルト、プロファイルベース、カスタム) をサポート
- 接続プーリングの互換性 - psycopg、psycopg2、および asyncpg の組み込み接続プーリングで動作

クイックスタートガイド

要件

- Python 3.10 以降
- Aurora DSQL クラスターへのアクセス
- アプリケーションが Aurora DSQL に接続できるように、適切な IAM アクセス許可を設定します。
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)

インストール

```
pip install aurora-dsql-python-connector
```

psycopg、psycopg2 または asyncpg を個別にインストールする

Python 用 Aurora DSQL コネクタインストーラは、基盤となるライブラリをインストールしません。これらは、個別にインストールする必要があります。次に例を示します。

```
# Install psycopg and psycopg pool
pip install "psycopg[binary,pool]"
```

```
# Install psycopg2
pip install psycopg2-binary
```

```
# Install asyncpg
pip install asyncpg
```

注記:

必要なライブラリのみをインストールする必要があります。したがって、クライアントが psycopg を使用する場合は、psycopg のみをインストールする必要があります。クライアントが psycopg2 を

使用する場合は、psycopg2 のみをインストールする必要があります。クライアントが asyncpg を使用する場合は、asyncpg のみをインストールする必要があります。

クライアントに複数のライブラリが必要な場合は、必要なライブラリをすべてインストールする必要があります。

基本的な使用法

psycopg

```
    import aurora_dsql_psycopg as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
    }

    conn = dsql.connect(**config)
    with conn.cursor() as cur:
        cur.execute("SELECT 1")
        result = cur.fetchone()
        print(result)
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
    }

    conn = dsql.connect(**config)
    with conn.cursor() as cur:
        cur.execute("SELECT 1")
        result = cur.fetchone()
        print(result)
```

asyncpg

```
    import asyncio
```

```
    import aurora_dsql_asyncpg as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
    }

    conn = await dsql.connect(**config)
    result = await conn.fetchrow("SELECT 1")
    await conn.close()
    print(result)
```

just host の使用

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("your-cluster.dsql.us-east-1.on.aws")
```

クラスター ID のみを使用

psycopg

```
    import aurora_dsql_psycopg as dsql
```

```
    conn = dsql.connect("your-cluster")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("your-cluster")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("your-cluster")
```

注記:

「クラスター ID のみを使用」シナリオでは、マシンで以前に設定されたリージョンが使用されます。次に例を示します。

```
aws configure set region us-east-1
```

リージョンが設定されていない場合、または指定されたクラスター ID が別のリージョンにある場合、接続は失敗します。これを機能させるには、次の例のようにリージョンをパラメータとして指定します。

psycopg

```
    import aurora_dsql_psycopg as dsql

    config = {
            "region": "us-east-1",
    }

    conn = dsql.connect("your-cluster", **config)
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql
```

```
    config = {
            "region": "us-east-1",
    }

    conn = dsql.connect("your-cluster", **config)
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    config = {
            "region": "us-east-1",
    }

    conn = await dsql.connect("your-cluster", **config)
```

接続文字列

psycopg

```
    import aurora_dsql_psycopg as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    conn = dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/postgres?
user=admin&token_duration_secs=15")
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

    conn = await dsql.connect("postgresql://your-cluster.dsql.us-east-1.on.aws/
postgres?user=admin&token_duration_secs=15")
```

高度な設定

psycopg

```
    import aurora_dsql_psycopg as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
        "profile": "default",
        "token_duration_secs": "15",
    }

    conn = dsql.connect(**config)
    with conn.cursor() as cur:
        cur.execute("SELECT 1")
        result = cur.fetchone()
        print(result)
```

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
        "profile": "default",
        "token_duration_secs": "15",
    }

    conn = dsql.connect(**config)
    with conn.cursor() as cur:
        cur.execute("SELECT 1")
        result = cur.fetchone()
        print(result)
```

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql
```

```
    config = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'region': "us-east-1",
        'user': "admin",
        "profile": "default",
        "token_duration_secs": "15",
    }

    conn = await dsql.connect(**config)
    result = await conn.fetchrow("SELECT 1")
    await conn.close()
    print(result)
```

設定オプション

オプション	タイプ	必須	説明
host	string	はい	DSQL クラスターホスト名またはクラスター ID
user	string	いいえ	DSQL ユーザー名。デフォルト: admin
dbname	string	いいえ	データベース名。デフォルト: postgres
region	string	いいえ	AWS リージョン (指定されていない場合はホスト名から自動検出)
port	int	いいえ	デフォルトは 5432
custom_credentials_provider	CredentialProvider	いいえ	カスタム AWS 認証情報プロバイダー

オプション	タイプ	必須	説明
profile	string	いいえ	IAM プロファイル名。デフォルト: default。
token_duration_secs	int	いいえ	トークンの有効期限の秒数

基盤となる psycopg、psycopg2、および asyncpg ライブラリのすべての標準接続オプションもサポートされます。ただし、DSQL でサポートされていない asyncpg パラメータ krbsrvname と gsslib は除きます。

接続プーリングでの Python 用 Aurora DSQL コネクタの使用

Python 用 Aurora DSQL コネクタは、psycopg、psycopg2、および asyncpg の組み込み接続プーリングで動作します。コネクタは、接続の確立中に IAM トークンの生成を処理し、接続プールが正常に動作できるようにします。

psycopg

psycopg の場合、コネクタは psycopg_pool.ConnectionPool コンストラクタに直接渡すことができる DSQLConnection という名前の接続クラスを実装します。非同期オペレーションの場合、DSQLAsyncConnection という名前のクラスの非同期バージョンもあります。

```
    from psycopg_pool import ConnectionPool as PsycopgPool

    ...
    pool = PsycopgPool(
        "",
        connection_class=dsql.DSQLConnection,
        kwargs=conn_params,
        min_size=2,
        max_size=8,
        max_lifetime=3300
    )
```

注: Connection max_lifetime 設定

max_lifetime パラメータは、Aurora DSQL データベースで許可される最大接続時間であるため、3,600 秒 (1 時間) 未満に設定する必要があります。max_lifetime を低く設定すると、接続プールは接続リサイクルをプロアクティブに管理できます。これは、データベースからの接続タイムアウトエラーを処理するよりも効率的です。

psycopg2

psycopg2 の場合、コネクタは psycopg2.pool.ThreadedConnectionPool から継承する AuroraDSQLThreadedConnectionPool という名前のクラスを提供します。AuroraDSQLThreadedConnectionPool クラスは、内部 _connect メソッドのみを上書きします。残りの実装は、psycopg2.pool.ThreadedConnectionPool によって変更されずに提供されます。

```
    import aurora_dsql_psycopg2 as dsql

    pool = dsql.AuroraDSQLThreadedConnectionPool(
            minconn=2,
            maxconn=8,
            **conn_params,
    )
```

asyncpg

asyncpg の場合、コネクタは asyncpg.Pool のインスタンスを返す create_pool 関数を提供します。

```
    import asyncio
    import os

    import aurora_dsql_asyncpg as dsql

    pool_params = {
        'host': "your-cluster.dsql.us-east-1.on.aws",
        'user': "admin",
        "min_size": 2,
        "max_size": 5,
    }

    pool = await dsql.create_pool(**pool_params)
```

認証

コネクタは、DSQL クライアントトークンジェネレーターを使用してトークンを生成することで、DSQL 認証を自動的に処理します。AWS リージョンが指定されていない場合、指定されたホスト名から自動的に解析されます。

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

管理者ユーザーと通常のユーザー

- `"admin"` という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーが管理者以外の認証トークンを使用します。
- トークンは接続ごとに動的に生成されます。

例

完全なサンプルコードについては、以下のセクションで示す例を参照してください。例を実行する手順については、READMDE ファイルの例を参照してください。

psycopg

README の例

説明	例
基本的な接続での Python 用 Aurora DSQL コネクタの使用	基本的な接続の例
基本的な非同期接続での Python 用 Aurora DSQL コネクタの使用	基本的な非同期接続の例
接続プーリングでの Python 用 Aurora DSQL コネクタの使用	接続プールを使用した基本的な接続の例
	接続プールを使用した同時接続の例

説明	例
非同期接続プールでの Python 用 Aurora DSQL コネクタの使用	非同期接続プールを使用した基本的な接続の例

psycopg2

README の例

説明	例
基本的な接続での Python 用 Aurora DSQL コネクタの使用	基本的な接続の例
接続プーリングでの Python 用 Aurora DSQL コネクタの使用	接続プールを使用した基本的な接続の例
	接続プールを使用した同時接続の例

asyncpg

README の例

説明	例
基本的な接続での Python 用 Aurora DSQL コネクタの使用	基本的な接続の例
接続プーリングでの Python 用 Aurora DSQL コネクタの使用	接続プールを使用した基本的な接続の例
	接続プールを使用した同時接続の例

Go pgx 用 Aurora DSQL コネクタ

Go 用 Aurora DSQL コネクタは、自動 IAM 認証で pgx をラップします。コネクタはトークン生成、SSL 設定、および接続管理を処理するため、ユーザーはアプリケーションロジックに集中できます。

コネクタについて

Aurora DSQL には、既存の Go PostgreSQL ドライバーがネイティブにサポートしていない時間制限付きトークンを使用した IAM ベースの認証が必要です。Go 用 Aurora DSQL コネクタは、IAM トークン生成を処理する pgx ドライバーの上に認証レイヤーを追加し、既存の pgx ワークフローを変更せずに Aurora DSQL に接続できるようにします。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: コネクタは AWS 認証情報を使用して認証トークンを生成します。これらのトークンの有効期間は設定可能です。

Go 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成するように設計されています。

Go 用 Aurora DSQL コネクタの利点

Go 用 Aurora DSQL コネクタを使用すると、以下の方法で IAM 認証を有効にしながら、既存の pgx ワークフローを引き続き使用できます。

- 自動トークン生成: コネクタは、接続ごとに IAM トークンを自動的に生成します。
- 接続プーリング: 接続ごとにトークンを自動生成する `pgxpool` の組み込みサポート
- 柔軟な設定: リージョンの自動検出による完全なエンドポイントまたはクラスター ID のサポート
- AWS 認証情報のサポート: AWS プロファイルとカスタム認証情報プロバイダーをサポート

主な特徴

自動トークン管理

コネクタは、事前に解決された認証情報を使用して、新しい接続ごとに IAM トークンを自動生成します。

接続プーリング

接続ごとにトークンを自動生成する pgxpool を利用した接続プーリング。

柔軟なホスト設定

自動リージョン検出を備えた完全なクラスターエンドポイントとクラスター ID の両方をサポートします。

SSL セキュリティ

SSL は verify-full モードと直接 TLS ネゴシエーションで常に有効になります。

前提条件

- Go 1.24 以降
- AWS 認証情報が設定されている
- Aurora DSQL クラスター

コネクタは AWS SDK for Go v2 のデフォルトの認証情報チェーンを使用し、認証情報を次の順序で解決します。

1. 環境変数 (AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY)
2. 共有認証情報ファイル (~/.aws/credentials)
3. 共有設定ファイル (~/.aws/config)
4. Amazon EC2/ECS/Lambda の IAM ロール

インストール

Go モジュールを使用してコネクタをインストールします。

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql
```

クイックスタート

次の例は、接続プールを作成し、クエリを実行する方法を示しています。

```
package main

import (
    "context"
    "log"

    "github.com/awslabs/aurora-dsql-connectors/go/pgx/dsql"
)

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

    // Create a connection pool
    pool, err := dsql.NewPool(ctx, dsql.Config{
        Host: "your-cluster.dsql.us-east-1.on.aws",
    })
    if err != nil {
        log.Fatal(err)
    }
    defer pool.Close()

    // Execute a query
    var greeting string
    err = pool.QueryRow(ctx, "SELECT 'Hello, DSQL!'").Scan(&greeting)
    if err != nil {
        log.Fatal(err)
    }
    log.Println(greeting)
}
```

設定オプション

このコネクタでは以下の設定オプションがサポートされています。

フィールド	タイプ	デフォルト	説明
ホスト	string	(必須)	クラスターエンドポイントまたはクラスター ID

フィールド	タイプ	デフォルト	説明
リージョン	string	(自動検出)	AWS リージョン (Host がクラスター ID の場合は必須)
ユーザー	string	「admin」	データベースユーザー
データベース	string	「postgres」	データベース名
ポート	int	5432	データベースポート
プロファイル	string	""	認証情報の AWS プロファイル名
TokenDurationSecs	int	900 (15 分)	トークンの有効期間 (秒) (最大許容時間: 1 週間、デフォルト: 15 分)
MaxConns	int32	0	最大プール接続数 (0 = pgxpool のデフォルト)
MinConns	int32	0	最小プール接続数 (0 = pgxpool のデフォルト)
MaxConnLifetime	time.Duration	55 分	最大接続有効期間

接続文字列形式

コネクタは、PostgreSQL および DSQL 接続文字列形式をサポートしています。

```
postgres://[user@]host[:port]/[database][?param=value&...]
dsql://[user@]host[:port]/[database][?param=value&...]
```

サポートされているクエリパラメータ:

- `region` - AWS リージョン
- `profile` - AWS プロファイル名
- `tokenDurationSecs` - トークンの有効期間 (秒)

例:

```
// Full endpoint (region auto-detected)
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres")

// Using dsql:// scheme (also supported)
pool, _ := dsql.NewPool(ctx, "dsql://admin@cluster.dsql.us-east-1.on.aws/postgres")

// With explicit region
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/mydb?
region=us-east-1")

// With AWS profile
pool, _ := dsql.NewPool(ctx, "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?
profile=dev")
```

高度な使用法

ホスト設定

コネクタは次の 2 つのホスト形式をサポートしています。

フルエンドポイント (リージョンの自動検出)。

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
```

クラスター ID (リージョンが必要)。

```
pool, _ := dsql.NewPool(ctx, dsql.Config{
    Host:   "your-cluster-id",
    Region: "us-east-1",
})
```

プール設定の調整

ワークロードの接続プールを設定します。

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:              "your-cluster.dsql.us-east-1.on.aws",
    MaxConns:          20,
```

```
    MinConns:          5,
    MaxConnLifetime:   time.Hour,
    MaxConnIdleTime:   30 * time.Minute,
    HealthCheckPeriod: time.Minute,
})
```

単一接続の使用

シンプルなスクリプトの場合、または接続プーリングが必要ない場合。

```
conn, err := dsql.Connect(ctx, dsql.Config{
    Host: "your-cluster.dsql.us-east-1.on.aws",
})
if err != nil {
    log.Fatal(err)
}
defer conn.Close(ctx)

// Use the connection
rows, err := conn.Query(ctx, "SELECT * FROM users")
```

AWS プロファイルの使用

認証情報の AWS プロファイルを指定します。

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:    "your-cluster.dsql.us-east-1.on.aws",
    Profile: "production",
})
```

OCC 再試行

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。2 つのトランザクションが同じデータを変更する場合、最初のトランザクションはコミットに成功し、2 番目のトランザクションは OCC エラーを受け取ります。

`occretry` パッケージには、エクスポネンシャルバックオフとジッターによる自動再試行用のヘルパーが用意されています。以下を使用して、インストールします。

```
go get github.com/awslabs/aurora-dsql-connectors/go/pgx/occretry
```

トランザクション書き込みには、WithRetry を使用します。

```
err := occretry.WithRetry(ctx, pool, occretry.DefaultConfig(), func(tx pgx.Tx) error {
    _, err := tx.Exec(ctx, "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
 100, fromID)
    if err != nil {
        return err
    }
    _, err = tx.Exec(ctx, "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
 100, toID)
    return err
})
```

DDL または単一ステートメントには、ExecWithRetry を使用します。

```
err := occretry.ExecWithRetry(ctx, pool, occretry.DefaultConfig(),
    "CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY, name TEXT)")
```

> **Important**
>
> WithRetry は BEGIN/COMMIT/ROLLBACK を内部的に管理します。コールバックはトランザクションを受け取り、データベースオペレーションのみを含むものとし、再試行しても安全であることが必要です。

例

より包括的な例とユースケースについては、「Go 用 Aurora DSQL コネクタの例」を参照してください。

例	説明
example_preferred	推奨: 同時クエリによる接続プール
トランザクション	BEGIN/COMMIT/ROLLBACK によるトランザクション処理
occ_retry	エクスポネンシャルバックオフによる OCC 競合の処理
connection_string	設定に接続文字列を使用する

Node.js 用 Aurora DSQL コネクタ

node-postgres 用 Aurora DSQL コネクタおよび Postgres.js 用 Aurora DSQL コネクタは、node-postgres および Postgres.js クライアントの機能を拡張して、IAM 認証情報を使用してアプリケーションが Aurora DSQL で認証できるようにする認証プラグインです。

node-postgres 用の Aurora DSQL コネクタ

node-postgres 用 Aurora DSQL コネクタは、JavaScript/TypeScript アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合する node-postgres 上に構築された Node.js コネクタです。

Aurora DSQL コネクタは node-postgres のクライアントおよびプールの機能を拡張して、アプリケーションが IAM 認証情報を使用して Amazon Aurora DSQL で認証できるようにする認証プラグインとして設計されています。

コネクタについて

Amazon Aurora DSQL は、PostgreSQL と互換性があるクラウドネイティブの分散データベースです。IAM 認証と期限付きトークンが必要ですが、従来の Node.js データベースドライバーにはこの組み込みサポートがありません。

node-postgres 用 Aurora DSQL コネクタは、node-postgres とシームレスに連携する認証ミドルウェアを実装することで、このギャップを埋めます。このアプローチにより、デベロッパーは既存のノードポストグレスコードを維持し、自動トークン管理を通じて Aurora DSQL クラスターへの安全な IAM ベースのアクセスを取得できます。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: 認証トークンは AWS 認証情報を使用して生成され、設定可能な有効期間があります

node-postgres 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成するように設計されています。

機能

- 自動 IAM 認証 - DSQL トークンの生成と更新を処理

- node-postgres 上に構築 - Node.js 用の一般的な PostgreSQL クライアントを活用
- シームレスな統合 既存の node-postgres 接続パターンで動作
- リージョンの自動検出 - DSQL クラスターホスト名から AWS リージョンを抽出
- フル TypeScript サポート - フルタイプのセキュリティを提供
- AWS 認証情報のサポート: さまざまな認証情報プロバイダー (デフォルト、プロファイルベース、カスタム) をサポート
- 接続プーリングの互換性 - 組み込みの接続プーリングとシームレスに連携

サンプルアプリケーション

例には、node-postgres に Aurora DSQL コネクタを使用する方法を示すサンプルアプリケーションが含まれています。含まれている例を実行するには、README の例を参照してください。

クイックスタートガイド

要件

- Node.js 20 以降
- Aurora DSQL クラスターへのアクセス
- アプリケーションが Aurora DSQL に接続できるように、適切な IAM アクセス許可を設定します。
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)

インストール

```
npm install @aws/aurora-dsql-node-postgres-connector
```

ピアの依存関係

```
npm install @aws-sdk/credential-providers @aws-sdk/dsql-signer pg tsx
npm install --save-dev @types/pg
```

Usage

クライアント接続

```
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
```

```
});
await client.connect();
const result = await client.query("SELECT NOW()");
await client.end();
```

プール接続

```
import { AuroraDSQLPool } from "@aws/aurora-dsql-node-postgres-connector";

const pool = new AuroraDSQLPool({
  host: "<CLUSTER_ENDPOINT>",
  user: "admin",
  max: 3,
  idleTimeoutMillis: 60000,
});

const result = await pool.query("SELECT NOW()");
```

高度な使用法

```
import { fromNodeProviderChain } from "@aws-sdk/credential-providers";
import { AuroraDSQLClient } from "@aws/aurora-dsql-node-postgres-connector";

const client = new AuroraDSQLClient({
  host: "example.dsql.us-east-1.on.aws",
  user: "admin",
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
});

await client.connect();
const result = await client.query("SELECT NOW()");
await client.end();
```

設定オプション

オプション	タイプ	必須	説明
host	string	はい	DSQL クラスターホスト名

オプション	タイプ	必須	説明
username	string	はい	DSQL ユーザー名
database	string	いいえ	データベース名
region	string	いいえ	AWS リージョン (指定されていない場合はホスト名から自動検出)
port	number	いいえ	デフォルトは 5432
customCredentialsProvider	AwsCredentialIdentity / AwsCredentialIdentityProvider	いいえ	カスタム AWS 認証情報プロバイダー
profile	string	いいえ	IAM プロファイル名。デフォルトは「default」
tokenDurationSecs	number	いいえ	トークンの有効期限の秒数

クライアント/プールからの他のすべてのパラメータがサポートされています。

認証

コネクタは、DSQL クライアントトークンジェネレーターを使用してトークンを生成することで、DSQL 認証を自動的に処理します。AWS リージョンが指定されていない場合、指定されたホスト名から自動的に解析されます。

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

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーは通常の認証トークンを使用します。
- トークンは接続ごとに動的に生成されます。

Postgres.js 用 Aurora DSQL コネクタ

Postgres.js 用 Aurora DSQL コネクタは、JavaScript アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合する Postgres.js 上に構築された Node.js コネクタです。

Postgres.js 用 Aurora DSQL コネクタは Postgres.js のクライアント機能を拡張して、アプリケーションが IAM 認証情報を使用して Amazon Aurora DSQL で認証できるようにする認証プラグインとして設計されています。コネクタはデータベースに直接接続しませんが、基盤となる Postgres.js ドライバー上でシームレスな IAM 認証を提供します。

コネクタについて

Amazon Aurora DSQL は、PostgreSQL 互換アプリケーションに高可用性とスケーラビリティを提供する分散 SQL データベースサービスです。Aurora DSQL には、既存の Node.js ドライバーがネイティブにサポートしていない時間制限付きトークンを使用した IAM ベースの認証が必要です。

Postgres.js 用 Aurora DSQL コネクタの考え方は、IAM トークン生成を処理する Postgres.js クライアントの上に認証レイヤーを追加することです。これにより、ユーザーは既存の Postgres.js ワークフローを変更せずに Aurora DSQL に接続できます。

Postgres.js 用 Aurora DSQL コネクタは、ほとんどのバージョンの Postgres.js で動作します。ユーザーは Postgres.js を直接インストールして独自のバージョンを提供します。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: 認証トークンは AWS 認証情報を使用して生成され、設定可能な有効期間があります

Postgres.js 用 Aurora DSQL Connector は、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成するように設計されています。

機能

- 自動 IAM 認証 - DSQL トークンの生成と更新を処理
- Postgres.js 上に構築 - Node.js 用の高速 PostgreSQL クライアントを活用
- シームレスな統合 - 既存の Postgres.js 接続パターンで動作
- リージョンの自動検出 - DSQL クラスターホスト名から AWS リージョンを抽出
- フル TypeScript サポート - フルタイプのセキュリティを提供
- AWS 認証情報のサポート: さまざまな 認証情報プロバイダー (デフォルト、プロファイルベース、カスタム) をサポート
- 接続プーリングの互換性 - Postgres.js の組み込み接続プーリングとシームレスに連携

クイックスタートガイド

要件

- Node.js 20 以降
- Aurora DSQL クラスターへのアクセス
- アプリケーションが Aurora DSQL に接続できるように、適切な IAM アクセス許可を設定します。
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)

インストール

```
npm install @aws/aurora-dsql-postgresjs-connector
# Postgres.js is a peer-dependency, so users must install it themselves
npm install postgres
```

基本的な使用法

```
import { auroraDSQLPostgres } from '@aws/aurora-dsql-postgresjs-connector';

const sql = auroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  username: 'admin',

});

// Execute queries
const result = await sql`SELECT current_timestamp`;
```

```
console.log(result);

// Clean up
await sql.end();
```

ホストの代わりにクラスター ID を使用する

```
const sql = auroraDSQLPostgres({
  host: 'your-cluster-id',
  region: 'us-east-1',
  username: 'admin',

});
```

接続文字列

```
const sql = AuroraDSQLPostgres(
  'postgres://admin@your-cluster.dsql.us-east-1.on.aws'
);

const result = await sql`SELECT current_timestamp`;
```

高度な設定

```
import { fromNodeProviderChain } from '@aws-sdk/credential-providers';

const sql = AuroraDSQLPostgres({
  host: 'your-cluster.dsql.us-east-1.on.aws',
  database: 'postgres',
  username: 'admin',
  customCredentialsProvider: fromNodeProviderChain(), // Optionally provide custom
 credentials provider
  tokenDurationSecs: 3600,                            // Token expiration (seconds)

  // Standard Postgres.js options
  max: 20,                                // Connection pool size
  ssl: { rejectUnauthorized: false }    // SSL configuration
});
```

設定オプション

オプション	タイプ	必須	説明
host	string	はい	DSQL クラスターホスト名またはクラスター ID
database	string?	いいえ	データベース名
username	string?	いいえ	データベースユーザー名 (指定されていない場合は admin を使用)
region	string?	いいえ	AWS リージョン (指定されていない場合はホスト名から自動検出)
customCredentialsProvider	AwsCredentialIdentityProvider?	いいえ	カスタム AWS 認証情報プロバイダー
tokenDurationSecs	number?	いいえ	トークンの有効期限の秒数

標準の Postgres.js オプションもすべてサポートされています。

認証

コネクタは、DSQL クライアントトークンジェネレーターを使用してトークンを生成することで、DSQL 認証を自動的に処理します。AWS リージョンが指定されていない場合、指定されたホスト名から自動的に解析されます。

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

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーは通常の認証トークンを使用します。
- トークンは接続ごとに動的に生成されます。

使用例

Postgres.js 用 Aurora DSQL コネクタを使用した JavaScript の例は、GitHub で入手できます。例を実行する手順については、例のディレクトリを参照してください。

説明	例
複数のワーカー間でのテーブルの作成、挿入、読み取りなど、同時クエリによる接続プーリング	接続プーリングの例 (推奨)
接続プーリングなしの CRUD オペレーション (テーブルの作成、挿入、選択、削除)	接続プーリングなしの例

Ruby pg 用 Aurora DSQL コネクタ

Ruby 用 Aurora DSQL コネクタは、Ruby アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合した pg 上に構築された Ruby コネクタです。

このコネクタは、トークン生成、SSL 設定、接続プーリングを処理し、ユーザーがアプリケーションロジックに集中できるようにします。

コネクタについて

Amazon Aurora DSQL には、既存の Ruby PostgreSQL ドライバーがネイティブにサポートしていない時間制限付きトークンを使用する IAM 認証が必要です。Ruby 用 Aurora DSQL コネクタは、IAM トークン生成を処理する pg gem 上に認証レイヤーを追加し、既存の pg ワークフローを変更せずに Aurora DSQL に接続できるようにします。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: コネクタは AWS 認証情報を使用して認証トークンを生成します。これらのトークンの有効期間は設定可能です。

Ruby 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成します。

機能

- 自動 IAM 認証 - Aurora DSQL トークンの生成と更新を処理
- pg 上に構築 - Ruby 用の一般的な PostgreSQL Gem をラップ
- シームレスな統合 - 既存の pg gem ワークフローと連携
- 接続プーリング - max_lifetime の適用による `connection_pool` gem を介した組み込みサポート
- リージョンの自動検出 - Aurora DSQL クラスターホスト名から AWS リージョンを抽出
- AWS 認証情報のサポート: AWS プロファイルとカスタム認証情報プロバイダーをサポート
- OCC 再試行 - エクスポネンシャルバックオフを使用したオプトインのオプティミスティック同時実行制御の再試行

サンプルアプリケーション

詳細な例については、GitHub のサンプルアプリケーションを参照してください。

クイックスタートガイド

要件

- Ruby 3.1 以降
- Aurora DSQL クラスターへのアクセス
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)。

インストール

Gemfile に追加:

```
gem "aurora-dsql-ruby-pg"
```

または直接インストール:

```
gem install aurora-dsql-ruby-pg
```

Usage

プール接続

```
require "aurora_dsql_pg"

# Create a connection pool with OCC retry enabled
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

# Read
pool.with do |conn|
  result = conn.exec("SELECT 'Hello, DSQL!'")
  puts result[0]["?column?"]
end

# Write — you must wrap writes in a transaction
pool.with do |conn|
  conn.transaction do
    conn.exec_params("INSERT INTO users (id, name) VALUES (gen_random_uuid(), $1)",
 ["Alice"])
  end
end

pool.shutdown
```

シングル接続

シンプルなスクリプトの場合、または接続プーリングが必要ない場合。

```
conn = AuroraDsql::Pg.connect(host: "your-cluster.dsql.us-east-1.on.aws")
conn.exec("SELECT 1")
conn.close
```

高度な使用法

ホスト設定

コネクタは、完全なクラスターエンドポイント (リージョンの自動検出) とクラスター ID (リージョンが必要) の両方をサポートしています。

```
# Full endpoint (region auto-detected)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws"
)

# Cluster ID (region required)
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster-id",
  region: "us-east-1"
)
```

AWS プロファイル

認証情報の AWS プロファイルを指定します。

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  profile: "production"
)
```

接続文字列形式

コネクタは、PostgreSQL 接続文字列形式をサポートしています。

```
postgres://[user@]host[:port]/[database][?param=value&...]
postgresql://[user@]host[:port]/[database][?param=value&...]
```

サポートされているクエリパラメータ: `region`、`profile`、`tokenDurationSecs`。

```
# Full endpoint with profile
pool = AuroraDsql::Pg.create_pool(
  "postgres://admin@cluster.dsql.us-east-1.on.aws/postgres?profile=dev"
)
```

OCC 再試行

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。2 つのトランザクションが同じデータを変更する場合、最初のトランザクションはコミットに成功し、2 番目のトランザクションは OCC エラーを受け取ります。

OCC 再試行はオプトインです。プールの作成時に `occ_max_retries` を設定し、`pool.with` でエクスポネンシャルバックオフとジッターによる自動再試行を有効にします。

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
  occ_max_retries: 3
)

pool.with do |conn|
  conn.transaction do
    conn.exec_params("UPDATE accounts SET balance = balance - $1 WHERE id = $2", [100,
 from_id])
    conn.exec_params("UPDATE accounts SET balance = balance + $1 WHERE id = $2", [100,
 to_id])
  end
end
```

> **Warning**
>
> `pool.with` は、ブロックをトランザクションに自動的にラップしません。書き込みオペレーションでは、`conn.transaction` を自分自身で呼び出す必要があります。OCC の競合では、コネクタがブロック全体を再実行するため、データベースオペレーションのみが含まれ、安全に再試行できることが必要です。

個々の呼び出しの再試行をスキップするには、`retry_occ: false` を渡します。

```
pool.with(retry_occ: false) do |conn|
  conn.exec("SELECT 1")
end
```

設定オプション

フィールド	タイプ	デフォルト	説明
ホスト	String	(必須)	クラスターエンドポイントまたはクラスター ID
リージョン	String	(自動検出)	AWS リージョン (host がクラスター ID の場合は必須)
ユーザー	String	「admin」	データベースユーザー
データベース	String	「postgres」	データベース名
ポート	整数	5432	データベースポート
profile	String	nil	認証情報の AWS プロファイル名
token_duration	整数	900 (15 分)	トークンの有効期間 (秒) (最大許容時間: 1 週間、デフォルト: 15 分)
credentials_provider	Aws::Credentials	nil	カスタム認証情報プロバイダー
max_lifetime	整数	3300 (55 分)	最大接続有効期間 (秒単位)
application_name	String	nil	application_name の ORM プレフィックス
ロガー	ロガー	nil	OCC 再試行警告のロガー
occ_max_retries	整数	nil (無効)	`pool.with` での最大 OCC 再試行回数。設定すると再試行が有効化

`create_pool` は、`ConnectionPool.new` に直接渡すオプションのハッシュを含む `pool:` キーワードも受け入れます。`pool:` を省略すると、コネクタはデフォルトで `{size: 5, timeout: 5}` になります。指定したキーは、これらの特定のデフォルトのみを上書きします。

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsql.us-east-1.on.aws",
```

```
  pool: { size: 10, timeout: 10 }
)
```

認証

コネクタは、AWS 認証情報を使用してトークンを生成することで、Aurora DSQL 認証を自動的に処理します。AWS リージョンを指定しない場合、コネクタはホスト名からリージョンを解析します。

Aurora DSQL の認証の詳細については、「Aurora DSQL の認証および認可」を参照してください。

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーは通常の認証トークンを使用します。
- コネクタは、接続ごとにトークンを動的に生成します。

PHP **PDO_PGSQL** 用 Aurora DSQL コネクタ

PHP 用 Aurora DSQL コネクタは、PHP アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合した PDO_PGSQL 上に構築された PHP コネクタです。

コネクタはトークン生成、SSL 設定、および接続管理を処理するため、ユーザーはアプリケーションロジックに集中できます。

コネクタについて

PHP 用 Aurora DSQL コネクタは、IAM トークン生成を処理する PDO_PGSQL 上に認証レイヤーを追加し、既存の PDO ワークフローを使用して Aurora DSQL に接続できるようにします。Amazon Aurora DSQL には、時間制限付きトークンによる IAM 認証が必要であり、コネクタは自動的にトークン生成を管理します。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: コネクタは AWS 認証情報を使用して認証トークンを生成します。これらのトークンの有効期間は設定可能です。

PHP 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成します。

機能

- 自動 IAM 認証 - Aurora DSQL トークンの生成と更新を処理
- **PDO_PGSQL** 上に構築 - PHP 用の標準の PostgreSQL 拡張機能をラップ
- シームレスな統合 - 既存の PDO ワークフローと連携
- SSL 適用 - 常に `verify-full` モードと直接 TLS ネゴシエーションで SSL を使用
- リージョンの自動検出 - Aurora DSQL クラスターホスト名から AWS リージョンを抽出
- AWS 認証情報のサポート: AWS プロファイルとカスタム認証情報プロバイダーをサポート
- OCC 再試行 - エクスポネンシャルバックオフとジッターを使用したオプトインのオプティミスティック同時実行制御の再試行
- PSR-3 ログ記録 - 再試行診断用の互換性のあるログ記録

サンプルアプリケーション

詳細な例については、GitHub のサンプルアプリケーションを参照してください。

クイックスタートガイド

要件

- PHP 8.2 以降
- `ext-pdo_pgsql` エクステンション
- Aurora DSQL クラスターへのアクセス
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)。

インストール

パッケージをプロジェクトに追加します。

```
composer require awslabs/aurora-dsql-pdo-pgsql
```

Usage

設定ベースの接続

```
<?php

require_once 'vendor/autoload.php';

use Aws\AuroraDsql\PdoPgsql\AuroraDsql;
use Aws\AuroraDsql\PdoPgsql\DsqlConfig;

$config = new DsqlConfig(
    host: 'your-cluster.dsql.us-east-1.on.aws',
    occMaxRetries: 3
);
$pdo = AuroraDsql::connect($config);

// Read
$stmt = $pdo->query('SELECT 1 AS result');
$row = $stmt->fetch(PDO::FETCH_ASSOC);
echo "Connected: {$row['result']}\n";

// Transactional write with automatic OCC retry
$id = $pdo->transaction(function (PDO $conn): string {
    $stmt = $conn->prepare('INSERT INTO users (name) VALUES (?) RETURNING id');
    $stmt->execute(['Alice']);
    return $stmt->fetchColumn();
});
```

接続文字列形式

シンプルなスクリプトの場合、または接続文字列構文を使用する場合、コネクタは Aurora DSQL 固有のクエリパラメータを使用して `postgres://` および `postgresql://` 接続文字列をサポートします。

```
$pdo = AuroraDsql::connectFromDsn(
    'postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres?region=us-east-1'
);
```

サポートされているクエリパラメータ: `region`、`profile`、`tokenDurationSecs`、`ormPrefix`。

OCC 再試行

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。2 つのトランザクションが同じデータを変更する場合、最初のトランザクションはコミットに成功し、2 番目のトランザクションは OCC エラーを受け取ります。

OCC 再試行はオプトインです。エクスポネンシャルバックオフとジッターによる自動再試行を有効にするには、config で `occMaxRetries` を設定します。

```
$config = new DsqlConfig(
    host: 'your-cluster.dsql.us-east-1.on.aws',
    occMaxRetries: 3
);
$pdo = AuroraDsql::connect($config);

// Single statements are automatically retried via exec()
$pdo->exec("CREATE INDEX ASYNC ON users (email)");

// Multi-statement transactions are retried via transaction()
$pdo->transaction(function (PDO $conn) {
    $conn->exec("UPDATE accounts SET balance = balance - 100 WHERE id = 1");
    $conn->exec("UPDATE accounts SET balance = balance + 100 WHERE id = 2");
});
```

> **Important**
>
> `transaction()` は `beginTransaction()`/`commit()`/`rollBack()` を内部的に管理します。コールバックは、データベースオペレーションのみを含むものとし、再試行しても安全であることが必要です。

設定オプション

コネクタは、設定用のクエリパラメータを持つ `postgres://` および `postgresql://` 接続文字列も受け入れます。サポートされているクエリパラメータ: `region`、`profile`、`tokenDurationSecs`、`ormPrefix`。

フィールド	タイプ	デフォルト	説明
host	string	(必須)	クラスターエンドポイントまたは 26 文字のクラスター ID
region	?string	null (自動検出)	AWS リージョン (host がクラスター ID の場合は必須)
user	string	"admin"	データベースユーザー
database	string	"postgres"	データベース名
port	int	5432	データベースポート
profile	?string	null	認証情報の AWS プロファイル名
credentialsProvider	?\Closure	null	カスタム AWS 認証情報プロバイダー
tokenDurationSecs	int	900 (15 分)	トークンの有効期間 (秒)
occMaxRetries	?int	null (無効)	exec() および transaction() のデフォルトの最大 OCC 再試行回数
ormPrefix	?string	null	application_name の先頭に付加される ORM プレフィックス
logger	?LoggerInterface	null	再試行の警告と診断のための PSR-3 ロガー

認証

コネクタは、AWS 認証情報を使用してトークンを生成することで、Aurora DSQL 認証を自動的に処理します。AWS リージョンを指定しない場合、コネクタはホスト名からリージョンを解析します。

Aurora DSQL の認証の詳細については、「Aurora DSQL の認証および認可」を参照してください。

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーは通常の認証トークンを使用します。
- コネクタは、接続ごとにトークンを動的に生成します。

.NET Npgsql 用 Aurora DSQL コネクタ

.NET 用 Aurora DSQL コネクタは、.NET アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合した Npgsql 上に構築された .NET コネクタです。

このコネクタは、トークン生成、SSL 設定、接続プーリングを処理し、ユーザーがアプリケーションロジックに集中できるようにします。

コネクタについて

Amazon Aurora DSQL には、既存の .NET PostgreSQL ドライバーがネイティブにサポートしていない時限トークンを使用した IAM 認証が必要です。.NET 用 Aurora DSQL コネクタは、IAM トークン生成を処理する Npgsql 上に認証レイヤーを追加し、既存の Npgsql ワークフローを変更せずに Aurora DSQL に接続できるようにします。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: コネクタは AWS 認証情報を使用して認証トークンを生成します。これらのトークンの有効期間は設定可能です。

.NET 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成します。

機能

- 自動 IAM 認証 - Aurora DSQL トークンの生成と更新を処理
- Npgsql 上に構築 - .NET 用の一般的な PostgreSQL ドライバーをラップ
- シームレスな統合 - 既存の Npgsql ワークフローと連携
- 接続プーリング - 最大有効期間を強制する `NpgsqlDataSource` を利用した組み込みサポート

- リージョンの自動検出 - Aurora DSQL クラスターホスト名から AWS リージョンを抽出
- AWS 認証情報のサポート: AWS プロファイルとカスタム認証情報プロバイダーをサポート
- OCC 再試行 - エクスポネンシャルバックオフを使用したオプトインのオプティミスティック同時実行制御の再試行
- SSL 適用 - 常に `verify-full` モードと直接 TLS ネゴシエーションで SSL を使用

サンプルアプリケーション

詳細な例については、GitHub のサンプルアプリケーションを参照してください。

クイックスタートガイド

要件

- .NET 8.0 以降
- Aurora DSQL クラスターへのアクセス
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)。

インストール

パッケージをプロジェクトに追加します。

```
dotnet add package Amazon.AuroraDsql.Npgsql
```

Usage

プール接続

```
using Amazon.AuroraDsql.Npgsql;

// Create a connection pool
await using var ds = await AuroraDsql.CreateDataSourceAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws",
    OccMaxRetries = 3
});

// Read
await using (var conn = await ds.OpenConnectionAsync())
```

```
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT 'Hello, DSQL!'";
    var greeting = await cmd.ExecuteScalarAsync();
    Console.WriteLine(greeting);
}

// Transactional write with OCC retry
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "INSERT INTO users (id, name) VALUES (gen_random_uuid(), @name)";
    cmd.Parameters.AddWithValue("name", "Alice");
    await cmd.ExecuteNonQueryAsync();
});
```

シングル接続

シンプルなスクリプトの場合、または接続プーリングが必要ない場合。

```
await using var conn = await AuroraDsql.ConnectAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws"
});

await using var cmd = conn.CreateCommand("SELECT 1");
await cmd.ExecuteScalarAsync();
```

OCC 再試行

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。2 つのトランザクションが同じデータを変更する場合、最初のトランザクションはコミットに成功し、2 番目のトランザクションは OCC エラーを受け取ります。

OCC 再試行はオプトインです。エクスポネンシャルバックオフとジッターによる自動再試行を有効にするには、config で `OccMaxRetries` を設定します。トランザクション書き込みには、`WithTransactionRetryAsync` を使用します。

```
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();
```

```
    cmd.CommandText = "UPDATE accounts SET balance = balance - 100 WHERE id = @from";
    cmd.Parameters.AddWithValue("from", fromId);
    await cmd.ExecuteNonQueryAsync();

    cmd.CommandText = "UPDATE accounts SET balance = balance + 100 WHERE id = @to";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("to", toId);
    await cmd.ExecuteNonQueryAsync();
});
```

DDL または単一ステートメントには、ExecWithRetryAsync を使用します。

```
await ds.ExecWithRetryAsync("CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY,
 name TEXT)");
```

> **Important**
>
> WithTransactionRetryAsync は BEGIN/COMMIT/ROLLBACK を内部的に管理し、試行ごとに新しい接続を開きます。コールバックは、データベースオペレーションのみを含むものとし、再試行しても安全であることが必要です。

設定オプション

コネクタは、region および profile クエリパラメータを持つ postgres:// および postgresql:// 接続文字列も受け入れます。

フィールド	タイプ	デフォルト	説明
Host	string	(必須)	クラスターエンドポイントまたは 26 文字のクラスター ID
Region	string?	(自動検出)	AWS リージョン (Host がクラスター ID の場合は必須)
User	string	"admin"	データベースユーザー
Database	string	"postgres"	データベース名
Port	int	5432	データベースポート

フィールド	タイプ	デフォルト	説明
Profile	string?	null	認証情報の AWS プロファイル名
CustomCredentialsProvider	AWSCredentials?	null	カスタム AWS 認証情報プロバイダー
TokenDurationSecs	int?	null (SDK デフォルト、900 秒)	トークンの有効期間 (秒)
OccMaxRetries	int?	null (無効)	データソースに対する再試行メソッドのデフォルトの最大 OCC 再試行回数
OrmPrefix	string?	null	application_name の先頭に付加される ORM プレフィックス
LoggerFactory	ILoggerFactory?	null	再試行の警告と診断のためのロガーファクトリー
ConfigureConnectionString	Action<NpgsqlConnectionStringBuilder>?	null	プール設定を上書きしたり、追加の Npgsql 接続文字列プロパティを設定したりするためのコールバック。SSL と Enlist は必須のセキュリティ要件であり、上書きすることはできません。

認証

コネクタは、AWS 認証情報を使用してトークンを生成することで、Aurora DSQL 認証を自動的に処理します。AWS リージョンを指定しない場合、コネクタはホスト名からリージョンを解析します。

Aurora DSQL の認証の詳細については、「Aurora DSQL の認証および認可」を参照してください。

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。

- 他のすべてのユーザーは通常の認証トークンを使用します。
- コネクタは、接続ごとにトークンを動的に生成します。

Rust SQLx 用 Aurora DSQL コネクタ

Rust 用 Aurora DSQL コネクタは、Rust アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合した SQLx 上に構築された Rust コネクタです。

コネクタはトークン生成、SSL 設定、および接続管理を処理するため、ユーザーはアプリケーションロジックに集中できます。

コネクタについて

Rust 用 Aurora DSQL コネクタは、IAM トークン生成を処理する SQLx 上に認証レイヤーを追加し、既存の SQLx ワークフローを変更せずに Aurora DSQL に接続できるようにします。

Aurora DSQL 認証とは

Aurora DSQL では、認証に以下が含まれます。

- IAM 認証: すべての接続で、時間制限付きトークンによる IAM ベースの認証が使用されます
- トークン生成: コネクタは AWS 認証情報を使用して認証トークンを生成します。これらのトークンの有効期間は設定可能です。

Rust 用 Aurora DSQL コネクタは、これらの要件を理解し、接続の確立時に IAM 認証トークンを自動的に生成します。

機能

- 自動 IAM 認証 - Aurora DSQL トークンの生成と更新を処理
- SQLx 上に構築 - Rust 用の一般的な PostgreSQL ドライバーをラップ
- シームレスな統合 - 既存の SQLx ワークフローと連携
- 接続プーリング - `pool` 機能を介したバックグラウンドでのトークン更新によるオプトインのプールサポート
- リージョンの自動検出 - Aurora DSQL クラスターホスト名から AWS リージョンを抽出
- AWS 認証情報のサポート - AWS プロファイルとデフォルトの認証情報チェーンをサポート
- OCC 再試行 - エクスポネンシャルバックオフとジッターを使用したオプトインのオプティミスティック同時実行制御の再試行

サンプルアプリケーション

詳細な例については、GitHub のサンプルアプリケーションを参照してください。

クイックスタートガイド

要件

- Rust 1.80 以降
- Aurora DSQL の開始方法
- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)。

インストール

`Cargo.toml` に追加:

```
[dependencies]
aurora-dsql-sqlx-connector = "0.1.2"
```

ほとんどのアプリケーションでは、`pool` 機能と `occ` 機能の両方を有効にします。

```
[dependencies]
aurora-dsql-sqlx-connector = { version = "0.1.2", features = ["pool", "occ"] }
```

機能フラグ

機能	デフォルト	説明
`pool`	いいえ	バックグラウンドでのトークン更新機能付きの SQLx プールヘルパー
`occ`	いいえ	OCC 再試行ヘルパー (`retry_on_occ` 、`is_occ_error`)

Usage

プール接続

```
use sqlx::Row;
```

```
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let pool = aurora_dsql_sqlx_connector::pool::connect(
        "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
    ).await?;

    // Read
    let row = sqlx::query("SELECT 'Hello, DSQL!' as greeting")
        .fetch_one(&pool)
        .await?;
    let greeting: &str = row.get("greeting");
    println!("{}", greeting);

    // Write — you must wrap writes in a transaction
    let mut tx = pool.begin().await?;
    sqlx::query("INSERT INTO users (id, name) VALUES (gen_random_uuid(), $1)")
        .bind("Alice")
        .execute(&mut *tx)
        .await?;
    tx.commit().await?;

    pool.close().await;
    Ok(())
}
```

シングル接続

シンプルなスクリプトの場合、または接続プーリングが必要ない場合:

```
use sqlx::Row;

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let mut conn = aurora_dsql_sqlx_connector::connection::connect(
        "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
    ).await?;

    let row = sqlx::query("SELECT 1 as value")
        .fetch_one(&mut conn)
        .await?;
    let value: i32 = row.get("value");
    println!("Result: {}", value);
```

```
    Ok(())
}
```

connection::connect() を呼び出すたびに、新しい IAM トークンが生成されます。トークン期間より長いオペレーションの場合は、新しい接続を作成します。

高度な使用法

ホスト設定

コネクタは、完全なクラスターエンドポイント (リージョンの自動検出) とクラスター ID (リージョンが必要) の両方をサポートしています。

```
// Full endpoint (region auto-detected)
let opts = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
)?;

// Cluster ID (region required)
let opts = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster-id/postgres?region=us-east-1"
)?;
```

AWS プロファイル

認証情報の AWS プロファイルを指定します。

```
let pool = aurora_dsql_sqlx_connector::pool::connect(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres?profile=production"
).await?;
```

接続文字列形式

コネクタは、PostgreSQL 接続文字列形式をサポートしています。

```
postgres://[user@]host[:port]/[database][?param=value&...]
postgresql://[user@]host[:port]/[database][?param=value&...]
```

サポートされているクエリパラメータ: region、profile、tokenDurationSecs、ormPrefix。

プール設定

カスタムプール設定の場合は、PgPoolOptions を connect_with() に渡します。

```
use aurora_dsql_sqlx_connector::DsqlConnectOptions;
use sqlx::postgres::PgPoolOptions;

let config = DsqlConnectOptions::from_connection_string(
    "postgres://admin@your-cluster.dsql.us-east-1.on.aws/postgres"
)?;

let pool = aurora_dsql_sqlx_connector::pool::connect_with(
    &config,
    PgPoolOptions::new().max_connections(20),
).await?;
```

プログラムによる設定

プログラムで設定する場合は DsqlConnectOptionsBuilder を使用します。

```
use aurora_dsql_sqlx_connector::{DsqlConnectOptionsBuilder, Region};
use sqlx::postgres::PgConnectOptions;

let pg = PgConnectOptions::new()
    .host("your-cluster.dsql.us-east-1.on.aws")
    .username("admin")
    .database("postgres");

let opts = DsqlConnectOptionsBuilder::default()
    .pg_connect_options(pg)
    .region(Some(Region::new("us-east-1")))
    .build()?;

let mut conn = aurora_dsql_sqlx_connector::connection::connect_with(&opts).await?;
```

OCC 再試行

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。2 つのトランザクションが同じデータを変更する場合、最初のトランザクションはコミットに成功し、2 番目のトランザクションは OCC エラーを受け取ります。

OCC 再試行はオプトインです。occ 機能を有効にし、retry_on_occ を使用してエクスポネンシャルバックオフとジッターによる自動再試行を有効にします。

```
use aurora_dsql_sqlx_connector::{retry_on_occ, OCCRetryConfig};
```

```
let config = OCCRetryConfig::default(); // max_attempts: 3, exponential backoff

retry_on_occ(&config, || async {
    let mut tx = pool.begin().await?;

    sqlx::query("UPDATE accounts SET balance = balance - 100 WHERE id = $1")
        .bind(account_id)
        .execute(&mut *tx)
        .await?;

    tx.commit().await?;
    Ok(())
}).await?;
```

> **Warning**
>
> OCC の競合が発生した場合、retry_on_occ はクロージャ全体を再実行するため、クロージャ内にはデータベースオペレーションのみを含め、安全に再試行できるようにする必要があります。

設定オプション

フィールド	タイプ	デフォルト	説明
host	String	(必須)	クラスターエンドポイントまたはクラスター ID
region	Option<Region>	(自動検出)	AWS リージョン (host がクラスター ID の場合は必須)
user	String	"admin"	データベースユーザー
database	String	"postgres"	データベース名
port	u16	5432	データベースポート
profile	Option<String>	None	認証情報の AWS プロファイル名

フィールド	タイプ	デフォルト	説明
`tokenDurationSecs`	`u64`	900 (15 分)	トークンの有効期間 (秒)
`ormPrefix`	`Option<String>`	None	application_name の ORM プレフィックス (例えば、"diesel" は "diesel:aurora-dsql-rust-sqlx/{version}" を生成します)

認証

コネクタは、AWS 認証情報を使用してトークンを生成することで、Aurora DSQL 認証を自動的に処理します。AWS リージョンを指定しない場合、コネクタはホスト名からリージョンを解析します。

Aurora DSQL の認証の詳細については、「Aurora DSQL の認証および認可」を参照してください。

トークン生成

- 接続プール: バックグラウンドタスクは、トークン期間の 80% でトークンを更新します。更新タスクを停止してプールリソースを解放するには、`pool.close().await` を呼び出します。
- 単一接続: コネクタは接続時に新しいトークンを生成します。
- トークン生成は、ローカル SigV4 署名付きオペレーションであり、コストはごくわずかです。

管理者ユーザーと通常のユーザー

- 「admin」という名前のユーザーは、管理者認証トークンを自動的に使用します。
- 他のすべてのユーザーは通常の認証トークンを使用します。
- コネクタは、接続ごとにトークンを動的に生成します。

PostgreSQL 互換クライアントを使用した Aurora DSQL へのアクセス

Aurora DSQL は PostgreSQL ワイヤプロトコルを使用します。AWS CloudShell、psql、DBeaver、DataGrip など、さまざまなツールやクライアントを使用して

PostgreSQL に接続できます。次の表は、Aurora DSQL が一般的な PostgreSQL 接続パラメータをマッピングする方法をまとめたものです。

PostgreSQL	Aurora DSQL	注意事項
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、「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。
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。

接続する場合、Aurora DSQL は従来のパスワードの代わりに署名付き IAM 認証トークンを必要とします。これらの一時トークンは AWS 署名バージョン 4 を使用して生成され、接続確立中のみ使用されます。接続すると、セッションが終了するかクライアントが切断されるまで、セッションはアクティブのままになります。

期限切れのトークンを使用して新しいセッションを開こうとすると、接続リクエストは失敗し、新しいトークンを生成する必要があります。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。

SQL クライアントを使用した Aurora DSQL へのアクセス

Aurora DSQL は、クラスターに接続するための複数の PostgreSQL 互換クライアントをサポートしています。以下のセクションでは、PostgreSQL を使用して AWS CloudShell またはローカルコマンドラインに接続する方法と、DBeaver や JetBrains DataGrip などの GUI ベースのツールについて説明します。各クライアントには、前のセクションで説明した有効な認証トークンが必要です。

トピック

- DBeaver を使用して Aurora DSQL にアクセスする
- Aurora DSQL へのアクセスに JetBrains DataGrip を使用する
- psql (PostgreSQL インタラクティブターミナル) を使用して Aurora DSQL にアクセスする
- Aurora DSQL Driver for SQLTools を使用する
- トラブルシューティング

DBeaver を使用して Aurora DSQL にアクセスする

DBeaver は、JDBC ドライバーを持つデータベースを管理するために使用できるユニバーサル SQL クライアントです。堅牢なデータ表示、編集、管理の機能を備えているため、開発者やデータベース管理者の間で広く使用されています。DBeaver のクラウド接続オプションを使用すると、DBeaver を Aurora DSQL にネイティブ接続できます。

DBeaver Pro

DBeaver PRO 製品は、バージョン 25.3 より Aurora DSQL とのネイティブ統合を提供します。DBeaver ドキュメントの指示に従って、Aurora DSQL クラスターに接続します。

DBeaver Community Edition

DBeaver Community Edition は、無料のオープンソースバージョンです。インストール手順については、ダウンロードページを参照してください。DBeaver Community Edition から DSQL に接続するには、DBeaver 用の Aurora DSQL プラグインをインストールする必要があります。

DBeaver 用の Aurora DSQL プラグインは、JDBC 用 Aurora DSQL コネクタ上に構築されており、Aurora DSQL クラスターへの IAM 認証を有効にします。DBeaver UI を介して簡単にインストールできるため、トークン生成コードを記述したり、有効な IAM トークンを手動で提供したりする必要がなくなるため、認証が簡素化されるとともに、従来のユーザー生成パスワードに関連するセキュリティリスクが排除されます。

機能

- IAM 認証のサポート: AWS IAM 認証情報を使用して Aurora DSQL クラスターに接続し、安全でパスワードのない認証を実現します
- 自動ドライバー管理: JDBC 用 Aurora DSQL コネクタをシームレスにインストールして設定します
- 柔軟な接続オプション: ホストベースまたは JDBC URL ベースの接続設定を選択できます

DBeaver インストール用の Aurora DSQL プラグイン

1. DBeaver を開いた状態で、ドロップダウンメニューの [ヘルプ] → [新しいソフトウェアをインストール] に進みます
2. [追加] をクリックして新しいリポジトリを追加します
3. 次のとおりに入力します。
   - 名前: `Aurora DSQL Plugin`
   - 場所: `https://awslabs.github.io/aurora-dsql-dbeaver-plugin/update-site/`
4. [Aurora DSQL Connector for JDBC] をチェックします
5. [次へ] をクリックし、ライセンスに同意してインストールを完了します
6. プロンプトが表示されたら DBeaver を再起動します

Aurora DSQL 接続を作成する

1. [新しいデータベース接続] をクリックします
2. [Aurora DSQL] を選択します
3. [サーバー] で、[接続する] に次のいずれかを選択します
   - ホスト:
     - 次のフィールドのユーザーインターフェイスのテキスト入力を有効にします。

    - エンドポイント: DSQL クラスターエンドポイント
    - ユーザー名: DSQL ユーザー名 (例: admin)
    - AWS プロファイル: 例: default - 特定のプロファイルが指定されていない場合に使用される標準プロファイル
    - AWS リージョン (オプション): DSQL クラスターが存在するリージョンと一致する必要があります。一致しない場合は認証が失敗します
- の URL
    - 次の形式の JDBC URL:

```
jdbc:aws-dsql:postgresql://{cluster_endpoint}/{database}?
user=admin&profile=default&region=us-east-1
```

    - 注意: このモードでは、URL 入力のみが有効になります。JDBC 接続文字列にパラメータを追加するには、最初のパラメータとして ? で始まり、後続のパラメータとして & を追加する URL クエリパラメータ形式を使用します。
4. [テスト接続] をクリックして、Aurora DSQL 接続が機能することを確認します。
5. [完了] をクリックします。

トラブルシューティング

Windows Trust Store の問題

Windows ユーザーが Maven Central から Aurora DSQL Connector for JDBC ドライバーをダウンロードする際に問題が発生する場合があります。

原因: Windows Trust Store に Maven Central リポジトリへのアクセスに必要な証明書が含まれていない可能性があります。

解決策:

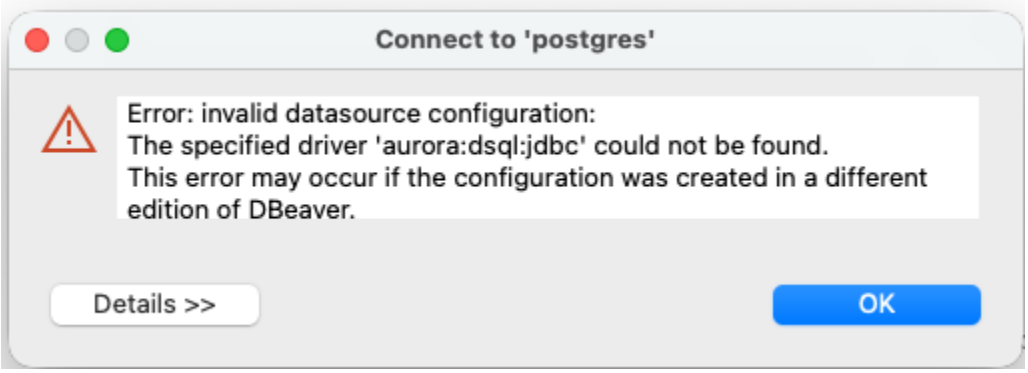
1. DBeaver を「管理者」として実行する
2. この設定のチェックを解除する - Windows > Preferences > Connections > "Use Windows Trust store"

ドライバー不足エラー

ドライバーアイコンが見つからない、または接続エラーが表示された場合は、現在の DBeaver バージョンに Aurora DSQL (コミュニティプラグイン) がインストールされていない可能性があります。エラーの例と修正方法については、以下を参照してください。

- 不足しているドライバーを使用して新しい接続を作成します。

- ドライバーなしで接続を試みます。



原因: 複数の DBeaver バージョンがインストールされている場合、接続設定は共有されますが、ドライバーはアプリケーションごとにインストールされます。

解決策: 上記のインストール手順に従って Aurora DSQL (コミュニティプラグイン) を再インストールします。

> **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、Aurora DSQL 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。

Aurora DSQL へのアクセスに JetBrains DataGrip を使用する

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 and PGSSLMODE=verify-full` 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。`verify-full` オプションでは、SSL 証明書をローカルにインストールする必要があります。詳細については、「SSL/TLS 証明書」を参照してください。

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、Aurora DSQL 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。

psql (PostgreSQL インタラクティブターミナル) を使用して Aurora DSQL にアクセスする

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

AWS CloudShell から PostgreSQL インタラクティブターミナルを使用して Aurora DSQL にアクセスするには、次の手順に従います。詳細については、「AWS CloudShell とは」を参照してください。

AWS CloudShell を使用して接続するには

1. Aurora DSQL コンソールにサインインします。
2. CloudShell で開くクラスターを選択します。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「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. [クエリエディタと接続] を選択し、[CloudShell で接続] 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. [CloudShell で起動] を選択し、次の CloudShell ダイアログで [実行] を選択します。

ローカル CLI を使用して PostgreSQL インタラクティブターミナル (psql) で Aurora DSQL にアクセスする

PostgreSQL ユーティリティへのターミナルベースのフロントエンドである `psql` を使用して、クエリをインタラクティブに入力し、PostgreSQL に発行して、クエリ結果を表示します。

> **Note**
>
> 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。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。

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、PostgreSQL ウェブサイトの「PostgreSQL クライアントアプリケーション」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。

```
# 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
```

Aurora DSQL Driver for SQLTools を使用する

Aurora DSQL Driver for SQLTools は、SQLTools と統合される Amazon Aurora DSQL の Visual Studio Code 拡張機能です。これにより、開発者は VS Code から直接 Aurora DSQL データベースに接続してクエリを実行できます。ドライバーは、Visual Studio Marketplace および Open VSX Registry からインストールできます。Kiro、Cursor、およびその他の VSCode ベースの IDE では、このページで説明されている標準インストール手順に従って、Open VSX Registry を使用してドライバーをインストールできます。

機能

- 自動 IAM 認証
- スキーマ、テーブルの参照、SQL クエリの実行などの標準的なデータベースオペレーション。

インストール

1. Extensions ビューを開きます。
2. 「Aurora DSQL Driver for SQLTools」を検索します。
3. [インストール] をクリックします。

注記:

SQLTools 拡張機能がまだ存在しない場合は、自動的にインストールされます。

認証

Aurora DSQL では、すべての接続で時間制限付きトークンによる IAM ベースの認証が使用されます。ドライバーは、Aurora DSQL Connector for node-postgres を使用して Aurora DSQL 認証を自動的に処理します。

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

Aurora DSQL 接続を作成する

前提条件

- AWS 認証情報の設定 (AWS CLI、環境変数、または IAM ロール経由)

Steps

1. 左側のサイドバーにある SQLTools アイコンをクリックします。
2. SQLTools ペインで、CONNECTIONS にカーソルを合わせ、[新しい接続を追加] アイコンをクリックします。
3. SQLTools 設定タブで、リストから Aurora DSQL Driver を選択します。
4. 接続パラメータを入力します。
   - AWS リージョン
     - オプション - リージョンは Aurora DSQL クラスターエンドポイントから解析されます。
     - DSQL クラスターフィールドにクラスター ID のみが指定されている場合は必須です。
   - AWS プロファイル
     - トークンの生成に使用されます。
     - 指定されていない場合はデフォルトのプロファイルを使用します。
5. [接続のテストボタン] をクリックして接続をテストします。
6. [接続を保存] をクリックします。

トラブルシューティング

SQL クライアント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、接続の [パスワード] フィールドに指定する必要があります。新しいセッションを開こうとすると (例えば、新しいテーブルを一覧表示したり、新しい SQL コンソールを開いたりする場合)、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。`expires-in` オプションで IAM 認証トークンの有効期間を選択する際は、デフォルトで 15 分に設定可能であり、最大 7 日間まで設定できる点に留意してください。

さらに、Aurora DSQL ドキュメントの「トラブルシューティング」セクションを参照してください。

Amazon Aurora DSQL クラスター接続ツール

Aurora DSQL は、多くのサードパーティーのデータベースドライバーや ORM ライブラリと互換性があります。AWS には、Aurora DSQL の操作を簡素化するための 2 種類のツールが用意されています。

- コネクタ – データベースドライバーを拡張して IAM トークンの生成を自動的に処理する認証プラグイン。データベースドライバーを直接操作する場合は、コネクタを使用します。
- アダプターとダイアレクト – IAM 認証と Aurora DSQL の互換性の向上を提供する特定の ORM フレームワークの拡張機能。サポートされている ORM フレームワークを使用する場合は、アダプターを使用します。

Aurora DSQL アダプターとダイアレクト

次の表は、Aurora DSQL で使用できるアダプターとダイアレクトを示しています。

プログラミング言語	ORM/Framework	リポジトリのリンク
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/java/hibernate

プログラミング言語	ORM/Framework	リポジトリのリンク
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm

データベースドライバーのサンプル

次の表は、サードパーティーのデータベースドライバーを使用して Aurora DSQL に接続するためのサンプルコードを示しています。

プログラミング言語	ドライバー	サンプルリポジトリのリンク
C++	libpq	https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/libpq
C# (.NET)	Npgsql	https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/npgsql
Go	pgx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/pgx
Java	HikariCP + pgJDBC	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc

プログラミング言語	ドライバー	サンプルリポジトリのリンク
JavaScript	node-postgres (AWS Lambda)	https://github.com/aws-samples/aurora-dsql-samples/tree/main/lambda
JavaScript	node-postgres	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/node-postgres
JavaScript	Postgres.js	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js
Python	asyncpg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/asyncpg
Python	Psycopg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg
Python	Psycopg2	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg2
Ruby	pg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/ruby-pg
Rust	SQLx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/sqlx

ORM とフレームワークのサンプル

次の表は、Aurora DSQL でサードパーティーの ORM ライブラリとフレームワークを使用するためのサンプルコードを示しています。

プログラミング言語	ORM/Framework	サンプルリポジトリのリンク
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/java/hibernate/examples/pet-clinic-app
Java	Liquibase	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/liquibase
Java	Spring Boot	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/spring_boot
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django/examples/pet-clinic-app
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy/examples/pet-clinic-app
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm/example
Ruby	Rails	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails

プログラミング言語	ORM/Framework	サンプルリポジトリのリンク
TypeScript	Prisma	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/prisma-multi-region
TypeScript	Sequelize	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize
TypeScript	TypeORM	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm

Aurora DSQL へのデータのロード

既存のデータベースから移行する場合も、Amazon Simple Storage Service からファイルをインポートする場合も、ローカルシステムからデータをロードする場合も、Aurora DSQL はデータを取得するための複数のアプローチを提供します。このセクションでは、ギガバイトから数百テラバイトまでのすべてのサイズのデータロードに推奨されるツールと手法について説明します。

ロードアプローチの選択

Aurora DSQL は標準の PostgreSQL データロードコマンドをサポートしていますが、データの大規模なロードを効率的に行うには、並列化、接続管理、エラー復旧を処理する必要があります。次の表に、オプションをまとめます。

アプローチ	次の用途に適しています	考慮事項
Aurora DSQL Loader - Aurora DSQL の使用時に挿入を簡単に並列化できるオープンソースユーティリティ	ほとんどのデータロードシナリオ (特に移行と一括インポート)	並列化、接続プーリング、競合解決、IAM 認証を自動的に処理します。ソースコードまたはバイナリとして使用できます。

アプローチ	次の用途に適しています	考慮事項
PostgreSQL **\copy** - クライアント側の psql メタコマンド	psql を介して既に接続している場合の単純なロード	クライアント上のファイルを読み取り、接続経由でデータをストリーミングします。並列化は自分で管理します。
INSERT トランザクション - 標準 SQL DML	小さなデータセットまたはアプリケーション駆動型挿入	最もシンプルなアプローチであるが、データの一括処理は最も遅い

ほとんどのデータロードタスクでは、Aurora DSQL Loader を使用します。複数の接続での並列実行や失敗したオペレーションの自動再試行など、分散データベースへのデータのロードに伴う運用上の複雑さを処理します。

Aurora DSQL Loader

Aurora DSQL Loader は、Aurora DSQL クラスターにデータを効率的にロードするように設計されたオープンソースのコマンドラインユーティリティです。接続プーリングを管理し、複数のワーカー間でデータ転送を並列化し、競合と再試行を自動的に処理します。

主な特徴

Aurora DSQL Loader には以下の機能があります。

並列ロード

設定可能なワーカースレッドにより、複数の接続間で同時データロードが可能になり、パフォーマンスが向上します。

接続プーリング

Aurora DSQL クラスターへの接続プールを管理し、IAM 認証と接続ライフサイクルを自動的に処理します。

複数のファイル形式のサポート

CSV (カンマ区切り値)、TSV (タブ区切り値)、および Apache Parquet 列形式をサポートします。ローダーは、ソース URI 拡張子に基づいてファイル形式を自動的に検出します。

自動スキーマ推論

`--if-not-exists` フラグとともに使用すると、ローダーはデータに基づいて適切な列タイプを持つテーブルを自動的に作成できます。

競合処理

ターゲットテーブルにユニーク制約がある場合、`--on-conflict` オプションを使用してローダーが競合を処理する方法 (重複をスキップする、レコードを upsert する、またはエラーを返す) を設定します。

耐障害性

自動再試行とジョブ再開機能により、中断されたロードを最初から開始するのではなく、停止ポイントから継続できます。

ローカルソースと S3 ソース

ローカルのファイルシステムパス、または S3 URI を使用して Amazon S3 バケットから直接、データをロードします。

前提条件

Aurora DSQL Loader を使用する前に、以下があることを確認してください。

- 有効なエンドポイントを持つアクティブな Aurora DSQL クラスター。
- AWS CLI (aws configure)、AWS シングルサインオン (aws sso login)、または IAM ロールを介して設定された AWS 認証情報。
- IAM アクセス許可: Aurora DSQL クラスターに対する `dsql:DbConnectAdmin` または `dsql:DbConnect`。
- S3 ソースの場合、ソースバケットから読み取るための適切なアクセス許可。

インストール

GitHub リリースページから最新リリースをダウンロードします。構築済みのバイナリは、一般的なプラットフォームで使用できます。ソースから構築する手順については、「Aurora DSQL Loader リポジトリ」を参照してください。

使用例

以下の例は、Aurora DSQL Loader の一般的なユースケースを示しています。

Exampleローカル CSV ファイルのロード

この例では、ローカルファイルシステムから既存のテーブルに CSV ファイルをロードします。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table
```

Example Amazon S3 からデータをロードする

この例では、Amazon S3 バケットから Parquet ファイルをロードします。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/data.parquet \
  --table my_table
```

Exampleテーブルの自動作成

この例では、データスキーマに基づいて新しいテーブルを自動的に作成します。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --if-not-exists
```

Exampleロード前の検証

この例では、データを実際にロードせずに設定を検証します。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --dry-run
```

Example中断されたロードの再開

ロードオペレーションが中断された場合は、前回の実行のジョブ ID を使用して再開できます。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --resume-job-id job-id \
  --manifest-dir ./loader-state
```

> **Note**
>
> 再開すると、ローダーは完了済みのほとんどの作業をスキップしますが、一部のレコードを再試行することがあります。ターゲットテーブルにユニーク制約がある場合は、`--on-conflict` オプションを使用して重複を処理します。例えば、`DO NOTHING` を使用してスキップしたり、`DO UPDATE` を使用して upsert したりします。

コマンドラインオプション

Aurora DSQL Loader は、以下のコマンドラインオプションをサポートしています。

`--endpoint`

(必須) Aurora DSQL クラスターエンドポイント。例:*cluster-id*.dsql.*region*.on.aws

`--source-uri`

(必須) データファイルへのパス。ローカルファイルパスまたは S3 URI (例: `s3://`*bucket-name*`/`*file.parquet*) を使用できます。

`--table`

(必須) Aurora DSQL データベース内のターゲットテーブルの名前。

`--if-not-exists`

(オプション) ターゲットテーブルが存在しない場合は、自動的に作成します。ローダーはデータからスキーマを推論します。

`--dry-run`

(オプション) データベースに実際にロードすることなく、設定とデータを検証します。

`--resume-job-id`

(オプション) 指定したジョブ ID を使用して、以前に中断されたロードオペレーションを再開します。

`--manifest-dir`

(オプション) ジョブの再開に使用されるジョブの状態とマニフェストを保存するディレクトリ。

`--on-conflict`

(オプション) ターゲットテーブルのユニーク制約に違反する行が挿入されたときに、競合を処理する方法を指定します。有効な値は、`error` (エラーを返す)、`do-nothing` (重複行をスキップする)、または `do-update` (既存の行を新しい値で更新する) です。

オプションと追加の設定パラメータの詳細なリストについては、以下を実行します。

```
aurora-dsql-loader load --help
```

ベストプラクティス

- 検証にドライランを使用する – 本番稼働用テーブルにデータをロードする前に、必ず `--dry-run` でロード設定をテストします。
- 再開のためのユニーク制約を定義する – 中断されたロードを再開する必要がある場合は、ターゲットテーブルにユニーク制約を定義し、`--on-conflict` オプションを使用して既にロードされたレコードを処理します。
- 大規模なデータセットに Parquet を使用する – Parquet の列指向形式は、通常、CSV や TSV と比較して、大規模なデータセットの圧縮とロードを高速化します。
- マニフェストディレクトリを保持する – ロードが正常に完了したことを確認するまで、ロードジョブのマニフェストディレクトリを保持し、必要に応じて再開できるようにします。
- 可能な場合はテーブルを事前作成する – データをロードする前に、明示的な列データ型とプライマリキーを使用してターゲットテーブルを定義します。スキーマを事前に作成すると、型の精度とインデックス作成を制御できるため、通常、自動推論スキーマと比較してクエリのパフォーマンスが向上します。

トラブルシューティング

認証エラー

AWS 認証情報が正しく設定されていること、およびターゲットクラスターに必要な `dsql:DbConnect` アクセス許可または `dsql:DbConnectAdmin` アクセス許可を IAM ID が持っていることを確認します。

S3 アクセスエラー

ソースバケットとオブジェクトに対する適切な S3 読み取りアクセス許可を IAM ID が持っていることを確認します。

スキーマ推論エラー

`--if-not-exists` を使用する場合は、データファイルに一貫した列タイプがあることを確認してください。複数の列タイプが混在していると、スキーマ推論が失敗する可能性があります。

再開時の重複キーエラー

ロードを再開するときに重複キーエラーが発生した場合は、ターゲットテーブルにユニーク制約を追加して、ローダーが `ON CONFLICT DO NOTHING` を使用してロード済みのレコードをスキップできるようにします。

トラブルシューティングの詳細については、「Aurora DSQL Loader GitHub リポジトリ」を参照してください。

移行パス

以下のセクションでは、一般的なソースシステムから Aurora DSQL にデータを移行する方法について説明します。

PostgreSQL からの移行

既存の PostgreSQL データベースから Aurora DSQL にデータを移行するには

1. PostgreSQL から CSV 形式または Parquet 形式にデータをエクスポートします。PostgreSQL `COPY` コマンドを使用して、各テーブルをエクスポートできます。

```
COPY my_table TO '/path/to/my_table.csv' WITH (FORMAT csv, HEADER true);
```

2. Aurora DSQL でターゲットテーブルを作成します。スキーマを手動で作成することも、ローダーの `--if-not-exists` フラグを使用してデータからスキーマを推論することもできます。
3. Aurora DSQL Loader を使用してエクスポートされたデータをロードします。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

> **Tip**
>
> 大規模な移行では、圧縮を改善し、ロードを高速化するために、Parquet 形式へのエクスポートを検討してください。DuckDB などのツールは、CSV ファイルを Parquet に効率的に変換できます。

MySQL からの移行

MySQL から Aurora DSQL にデータを移行するには

1. `SELECT INTO OUTFILE` を使用するか、`--tab` オプションで mysqldump などのツールを使用して、MySQL から CSV 形式にデータをエクスポートします。

```
SELECT * FROM my_table
INTO OUTFILE '/path/to/my_table.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

2. 適切な PostgreSQL 互換データ型を使用して、Aurora DSQL でターゲットテーブルを作成します。
3. Aurora DSQL Loader を使用してエクスポートされたデータをロードします。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

> **Note**
>
> MySQL と PostgreSQL のデータ型システムは異なります。Aurora DSQL でテーブルを作成するときに、スキーマを確認し、必要に応じてデータ型を調整します。

Amazon S3 からのロード

データが既に Amazon S3 にある場合は、ローカルシステムにダウンロードせずに直接ロードできます。Aurora DSQL Loader は S3 URI をネイティブにサポートしています。

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/path/to/data.parquet \
  --table my_table
```

ソースオブジェクトに対する `s3:GetObject` アクセス許可を IAM ID が持っていることを確認します。

PostgreSQL \copy の使用

IAM 認証を処理する `psql` セッションを通じて Aurora DSQL に既に接続している場合は、クライアント側の `\copy` メタコマンドを使用してローカルファイルシステムからデータをロードできます。サーバー側の `COPY` ステートメントとは異なり、`\copy` はクライアントマシン上のファイルを読み取り、既存の接続経由でデータをストリーミングするため、サーバー側のファイルにアクセスする必要はありません。このアプローチは、シンプルなシングルスレッドロードに適しています。

Example\copy を使用した CSV ファイルのロード

```
\copy my_table FROM '/path/to/data.csv' WITH (FORMAT csv, HEADER true);
```

`\copy` を直接使用する場合、ユーザーは以下を行う責任があります。

- 複数のファイルまたは大規模なデータセットをロードする場合の並列化の管理
- 接続管理と認証トークンの更新の処理
- 失敗したオペレーションの再試行ロジックの実装

INSERT トランザクションのベストプラクティス

INSERT ステートメントを使用して Aurora DSQL にデータをロードする場合は、スループットと信頼性を向上させるために以下のプラクティスに従ってください。

- 行をバッチ化して複数行 INSERT にする – 複数の行を 1 つの INSERT ステートメントにグループ化して、ラウンドトリップを減らします。例えば、`INSERT INTO my_table VALUES (1, 'a'), (2, 'b'), (3, 'c')` は 3 つの個別のステートメントよりも効率的です。
- パラメータ化されたクエリを使用する – 文字列連結ではなく、パラメータバインディングによるプリペアドステートメントを使用します。これにより、SQL インジェクションのリスクが回避され、データベースでクエリプランを再利用できます。
- トランザクションを小さくする – Aurora DSQL はオプティミスティック同時実行制御を使用するため、多くの行に影響を与える大規模なトランザクションでは競合が発生する可能性が高くなります。数千行ではなく数百行のトランザクションを対象とします。
- 再試行ロジックを実装する – オプティミスティック同時実行制御 (OCC) の競合などの一時的なエラーが分散システムで予期されます。失敗したトランザクションを再試行して エクスポネンシャルバックオフを実装します。
- 接続間で並列化する — 複数の接続を開き、接続間に挿入を分散します。接続ごとにデータの異なるサブセットを同時に処理できます。

ほとんどのユースケースの場合、Aurora DSQL Loader はデータロードに対する、よりシンプルで堅牢なアプローチを提供します。

その他のリソース

- GitHub の Aurora DSQL Loader – ソースコード、ドキュメント、問題の追跡
- Amazon Aurora DSQL での認証トークンの生成 – Aurora DSQL の IAM 認証トークンについて説明する
- PostgreSQL 互換クライアントを使用した Aurora DSQL へのアクセス – さまざまなクライアントやツールを使用して Aurora DSQL に接続する

Aurora DSQL の生成 AI

このセクションでは、Aurora DSQL で生成 AI ツールを使用する方法の詳細な手順について説明します。

AWS ラボ Aurora DSQL MCP サーバー

Aurora DSQL 用の AWS ラボモデルコンテキストプロトコル (MCP) サーバー

機能

- 人間が読める質問やコマンドを、構造化された Postgres 互換 SQL クエリに変換し、設定された Aurora DSQL データベースに対して実行します。
- デフォルトで読み取り専用、`--allow-writes` で有効になっているトランザクション
- パフォーマンスを向上させるためのリクエスト間の接続の再利用
- Aurora DSQL ドキュメント、検索、ベストプラクティスの推奨事項への組み込みアクセス

使用可能なツール

データベースのオペレーション

- readonly_query - DSQL クラスターに対して読み取り専用 SQL クエリを実行
- transact - トランザクションで書き込みオペレーションを実行 (`--allow-writes` が必要)
- get_schema - テーブルスキーマ情報を取得

ドキュメントと推奨事項

- dsql_search_documentation - Aurora DSQL ドキュメントの検索
  - パラメータ: `search_phrase` (必須)、`limit` (オプション)
- dsql_read_documentation - 特定の DSQL ドキュメントページを読み込み
  - パラメータ: `url` (必須)、`start_index` (オプション)、`max_length` (オプション)
- dsql_recommend - DSQL のベストプラクティスに関する推奨事項を取得
  - 必須パラメータ: `url` (必須)

前提条件

1. Aurora DSQL クラスターがある AWS アカウント
2. この MCP サーバーは、LLM クライアントと同じホストでのみローカルで実行できます。
3. AWS サービスにアクセスできる AWS 認証情報を設定する
   - 以下のアクセス許可を含むロールを持つ AWS アカウントが必要です。
     - `dsql:DbConnectAdmin` - 管理者ユーザーとして DSQL クラスターに接続

- `dsql:DbConnect` - カスタムデータベースロールを使用して DSQL クラスターに接続 (管理者以外のユーザーを使用する場合のみ必要)
- `aws configure` または環境変数を使用した AWS 認証情報の設定

インストール

ほとんどのツールでは、デフォルトのインストール手順に従って設定を更新すれば十分です。

Claude Code と Codex には個別の手順が概説されています。

デフォルトのインストール: 関連する MCP Config ファイルの更新

uvの使用

1. Astral または GitHub README から uv をインストールする
2. `uv python install 3.10` を使用して Python をインストールする

MCP クライアント設定で MCP サーバーを設定する (MCP Config ファイルの検索)

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "command": "uvx",
      "args": [
        "awslabs.aurora-dsql-mcp-server@latest",
        "--cluster_endpoint",
        "[your dsql cluster endpoint, e.g. abcdefghijklmnopqrst234567.dsql.us-
east-1.on.aws]",
        "--region",
        "[your dsql cluster region, e.g. us-east-1]",
        "--database_user",
        "[your dsql username, e.g. admin]",
        "--profile",
        "[your aws profile, e.g. default]"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR"
      },
      "disabled": false,
      "autoApprove": []
    }
```

```
  }
}
```

Windows のインストール

Windows ユーザーの場合、MCP サーバー設定形式は若干異なります。

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "disabled": false,
      "timeout": 60,
      "type": "stdio",
      "command": "uv",
      "args": [
        "tool",
        "run",
        "--from",
        "awslabs.aurora-dsql-mcp-server@latest",
        "awslabs.aurora-dsql-mcp-server.exe"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR",
        "AWS_PROFILE": "your-aws-profile",
        "AWS_REGION": "us-east-1"
      }
    }
  }
}
```

MCP クライアント設定ファイルの検索

最も一般的なエージェント開発ツールの一部では、MCP クライアント設定は次のファイルパスで確認できます。

- Kiro:
  - ユーザー設定: `~/.kiro/settings/mcp.json`
  - Workspace 設定: `/path/to/workspace/.kiro/settings/mcp.json`
- Claude Code: 詳細なセットアップヘルプについては、「Claude Code のインストール」を参照してください
  - ユーザー設定: `"mcpServers"` の `~/.claude.json`

  - プロジェクト設定: `/path/to/project/.mcp.json`
  - ローカル設定: `"projects" -> "path/to/project" -> "mcpServers"` の `~/.claude.json`
- Cursor:
  - グローバル: `~/.cursor/mcp.json`
  - プロジェクト: `/path/to/project/.cursor/mcp.json`
- Codex: `~/.codex/config.toml`
  - 各 MCP サーバーは、設定ファイル内の `[mcp_servers.<server-name>]` テーブルを使用して設定されます。「カスタムコーデックのインストール手順」を参照してください
- Warp:
  - ファイルの編集: `~/.warp/mcp_settings.json`
  - アプリケーションエディタ: `Settings > AI > Manage MCP Servers` を選択し JSON を貼り付ける
- Amazon Q Developer CLI: `~/.aws/amazonq/mcp.json`
- Cline: 通常、ネストされた VS Code パス - `~/.vscode-server/path/to/cline_mcp_settings.json`

Claude Code

前提条件

重要: MCP サーバー管理は、VS Code ネイティブパネルモードではなく Claude Code CLI ターミナルエクスペリエンスを通じてのみ使用できます。

最初に、Claude のネイティブインストール推奨プロセスに従って、Claude Code CLI をインストールします。

適切なスコープの選択

Claude Code は、ローカル (デフォルト)、プロジェクト、ユーザーという 3 つの異なるスコープを提供し、認証情報の機密性と共有の必要性に基づいてどのスコープを選択するかを詳細に説明しています。詳細については、MCP インストールスコープに関する Claude Code ドキュメントを参照してください。

1. ローカルスコープのサーバーはデフォルトの設定レベルを表し、プロジェクトのパスの下の `~/.claude.json` に保存されます。どちらもプライベートであり、現在のプロジェクトディレクトリ内でのみアクセスできます。これは MCP サーバーを作成するときのデフォルトの `scope` です。

2. プロジェクトスコープのサーバーを使用すると、プロジェクトディレクトリからのみアクセスでき、チームのコラボレーションが可能になります。プロジェクトスコープのサーバーは、プロジェクトのルートディレクトリに .mcp.json ファイルを追加します。このファイルはバージョン管理にチェックインされるように設計されており、すべてのチームメンバーが同じ MCP ツールとサービスにアクセスできるようになります。プロジェクトスコープのサーバーを追加すると、Claude Code は適切な設定構造でこのファイルを自動的に作成または更新します。
3. ユーザースコープのサーバーは ~/.claude.json に保存され、クロスプロジェクトアクセシビリティを提供するため、ユーザーアカウントに対してはプライベートなまま、マシン上のすべてのプロジェクトで利用できるようになります。

Claude CLI の使用 (推奨)

インタラクティブ claude CLI セッションを使用するとトラブルシューティングのエクスペリエンスが向上するため、これが推奨されるパスです。

```
claude mcp add amazon-aurora-dsql \
  --scope [one of local, project, or user] \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

トラブルシューティング: 別の AWS アカウントの Bedrock と Claude Code を使用する

Claude Code を、dsql クラスターへの接続に必要なプロファイルとは異なる Bedrock AWS アカウントまたはプロファイルで設定した場合は、追加の環境引数を指定する必要があります。

```
  --env AWS_PROFILE="[dsql profile, eg. default]" \
  --env AWS_REGION="[dsql cluster region, eg. us-east-1]" \
```

設定ファイルの直接変更

Claude Code では英数字の命名が必要なため、サーバーに aurora-dsql-mcp-server という名前を付けることをお勧めします。

ローカルスコープ

プロジェクト固有の mcpServers フィールド内の ~/.claude.json を更新します。

```
{
  "projects": {
    "/path/to/project": {
      "mcpServers": {}
    }
  }
}
```

プロジェクトスコープ

mcpServers フィールドの /path/to/project/root/.mcp.json を更新します。

```
{
  "mcpServers": {}
}
```

ユーザースコープ

プロジェクト固有の mcpServers フィールド内の ~/.claude.json を更新します。

```
{
  "mcpServers": {}
}
```

Codex

オプション 1: Codex CLI

Codex CLI がインストールされている場合は、codex mcp コマンドを使用して MCP サーバーを設定できます。

```
codex mcp add amazon-aurora-dsql \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
```

```
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

オプション 2: config.toml

MCP サーバーオプションをよりきめ細かく制御するには、~/.codex/config.toml 設定ファイルを手動で編集できます。各 MCP サーバーは、設定ファイル内の [mcp_servers.<server-name>] テーブルを使用して設定されます。

```
[mcp_servers.amazon-aurora-dsql]
command = "uvx"
args = [
  "awslabs.aurora-dsql-mcp-server@latest",
  "--cluster_endpoint", "<DSQL_CLUSTER_ID>.dsql.<AWS_REGION>.on.aws",
  "--region", "<AWS_REGION>",
  "--database_user", "<DATABASE_USERNAME>"
]

[mcp_servers.amazon-aurora-dsql.env]
FASTMCP_LOG_LEVEL = "ERROR"
```

インストールの検証

Amazon Q Developer CLI、Kiro CLI、Claude CLI/TUI、または Codex CLI/TUI の場合は、/mcp を実行して MCP サーバーのステータスを確認します。

Kiro IDE の場合、Kiro パネルの MCP SERVERS タブに移動して、設定されているすべての MCP サーバーとその接続ステータスインジケータを表示することもできます。

サーバー設定オプション

`--allow-writes`

デフォルトでは、dsql MCP サーバーは書き込みオペレーション (「読み取り専用モード」) を許可しません。トランザクションツールの呼び出しは、このモードでは失敗します。トランザクションツールを使用するには、--allow-writes パラメータを渡して書き込みを許可します。

DSQL に接続するときは、最小特権アクセスを使用することをお勧めします。例えば、ユーザーは可能な限り読み取り専用のロールを使用する必要があります。読み取り専用モードには、ミューテーションを拒否するためのベストエフォートなクライアント側の強制があります。

--cluster_endpoint

これは、接続するクラスターを指定するための必須パラメータです。これはクラスターの完全なエンドポイントである必要があります。例: `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws`

--database_user

これは、接続するユーザーを指定するための必須パラメータです。例えば、`admin` または `my_user` など。使用している AWS 認証情報には、そのユーザーとしてログインするためのアクセス許可が必要です。DSQL でのデータベースロールの設定と使用の詳細については、「IAM ロールでのデータベースロールの使用」を参照してください。

--profile

認証情報に使用する aws プロファイルを指定できます。これは docker のインストールではサポートされていないことに注意してください。

MCP 設定での `AWS_PROFILE` 環境変数の使用もサポートされています。

```
"env": {
  "AWS_PROFILE": "your-aws-profile"
}
```

どちらも指定されていない場合、MCP サーバーはデフォルトで AWS 設定ファイルの「デフォルト」プロファイルを使用します。

--region

これは、DSQL データベースのリージョンを指定するための必須パラメータです。

--knowledge-server

DSQL ナレッジツール (ドキュメント検索、読み取り、レコメンデーション) のリモート MCP サーバーエンドポイントを指定するオプションのパラメータ。デフォルトでは、事前設定されています。

例:

```
--knowledge-server https://custom-knowledge-server.example.com
```

注: セキュリティのため、信頼できるナレッジサーバーエンドポイントのみを使用してください。サーバーは HTTPS エンドポイントである必要があります。

`--knowledge-timeout`

ナレッジサーバーへのリクエストのタイムアウトを秒単位で指定するオプションのパラメータ。

デフォルト: `30.0`

例:

```
--knowledge-timeout 60.0
```

低速ネットワークでドキュメントにアクセスするときにタイムアウトが発生した場合は、この値を増やします。

Aurora DSQL ステアリング: Skills と Powers

このセクションでは、Skills と Powers を使用して Aurora DSQL の AI ステアリングを設定する方法について説明します。これらのマークダウンベースの設定ファイルは、エージェント開発の品質を向上させるためにコードを生成するときに AI アシスタントが自動的に適用するコンテキストとガイダンスを提供します。

概要

Skills と Powers は、Aurora DSQL の AI アシスタント機能を拡張するモジュラー機能です。これらは、AI アシスタントが Aurora DSQL データベースを操作するときに自動的に使用する手順、メタデータ、リソースをパッケージ化します。

Skills と Powers を使用する理由

Skills と Powers は、Aurora DSQL 開発にいくつかの重要な利点をもたらします。

- AI アシスタントの専門化 - ベストプラクティス、Postgres 互換 SQL パターン、分散データベースの最適化など、Aurora DSQL のドメイン固有の専門知識を提供します。
- 繰り返しを削減 - 一度作成すれば自動的に使用されます。複数の会話で同じガイダンスを繰り返し提供する必要がなくなります。

- コンテキスト効率性 - Skills は、コンテキストを事前に消費するのではなく、オンデマンドでロードします。AI は、必要に応じて情報を段階的にロードします。
- 継続的な学習 - Aurora DSQL の機能が進化するにつれて、Skills が更新されると、AI アシスタントは更新されたパターンに自動的にアクセスします。

推奨されるセットアップパス

開発環境に適したセットアップパスを選択します。

- the section called “Skills CLI” (エージェントに依存しない)
- the section called “Kiro Power”
- the section called “Claude Skill”
- the section called “Gemini Skill”
- the section called “Codex Skill”

DSQL Skill は、skill フォルダをツールの `rules` または `skills` ディレクトリにコピーすることで、他の AI コーディングエージェントでも使用できます。

Skills CLI

DSQL skill は Skills CLI を使用してインストールできます。このエージェントに依存しないセットアップ方法は、ほとんどの AI コーディングアシスタントで動作し、skill を複数のエージェントに一度にインストールできます。

セットアップ

次のコマンドを実行して、Aurora DSQL skill をインストールします。

```
npx skills add awslabs/mcp --skill dsql
```

CLI は次の手順を案内します。

- エージェントの選択 - インストールするエージェントを選択します (Kiro、Claude Code、Cursor、Copilot、Gemini、Codex、Roo、Cline、OpenCode、Windsurf など)
- インストールスコープ - 次のいずれかを選択します。
  - プロジェクト: 現在のディレクトリにインストールする (プロジェクトにコミット済み)
  - グローバル: ホームディレクトリにインストールする (すべてのプロジェクトで利用可能)
- インストール方法 - 次のいずれかを選択します。

- Symlink (推奨): 信頼できる単一のソース、簡単に更新可能
- すべてのエージェントにコピー: エージェントごとに個別のコピー

スキルの管理

以下を使用して、いつでも skills を確認し更新できます。

```
npx skills check
npx skills update
```

Kiro Power

Kiro Powers は、MCP ツールをフレームワークの専門知識とステアリング手順にバンドルする統合パッケージです。各 power には、使用可能な MCP ツールとアクティベーショントリガー、MCP サーバー設定、およびオンデマンドでロードされる追加のワークフロー固有のガイダンスを説明するエントリポイントドキュメントが含まれています。

Powers は、ユーザーのコンテキストに基づいて動的にアクティブ化されます。Powers は、すべてのツールを事前にロードするのではなく、関連するキーワードがアクティベーションをトリガーするまでほぼゼロのベースライン使用量を維持します。

セットアップ

Aurora DSQL の Kiro Power を設定するには:

1. Kiro Powers Registry から直接インストールする
2. IDE の Power にリダイレクトされたら、次のいずれかを実行します。
   - [Power を試す] ボタンを選択します。AI による MCP サーバーのセットアップガイドや、Aurora DSQL を使用したインタラクティブなオンボーディングエクスペリエンスで新しいクラスターを作成したりしたいユーザーに推奨されます。
   - 新しい Kiro チャットを開き、Aurora DSQL に関連する質問をします。必要に応じて、既存のクラスターの詳細で MCP Config を更新して MCP サーバー接続をテストし、power を使用してすぐに使用できるようにします。Kiro エージェントは、ユーザーのタスクを完了するために power が有益であると判断した場合、power を自動的にアクティブ化します。

Claude Skill

Claude skills は、Claude の機能を拡張するモジュラー機能です。各 skill は、Claude が必要に応じて自動的に使用する手順、メタデータ、およびオプションのリソースをパッケージ化します。Skills は

ファイルシステムベースで、コンテキストの使用を最小限に抑えるためにオンデマンドでロードされます。

Skills CLI を使用した簡単なセットアップ

skill は、the section called “Skills CLI” を使用して Claude Code にインストールできます。インストール先のエージェントとして Claude Code のみを指定するには、以下を使用します。

```
npx skills add awslabs/mcp --skill dsql --agent claude-code
```

代替方法: Git クローンを使用した直接セットアップ

代替セットアップでは、dsql-skill ディレクトリのスパースクローンを取得し、このクローンを `~/.claude/skills/` フォルダにシンボリックリンクします。これにより、skill を更新する必要があるときはいつでも、skill への変更をプルできるようになります。

前提条件

- インストール済みGit

セットアップステップ

1. ベースリポジトリディレクトリを作成する

```
mkdir -p .dsql_skill_repos
```

2. MCP リポジトリから skill をスパースクローンする

`dsql-skill` フォルダのみをクローンします (他のファイルはクローンしません)。

```
cd .dsql_skill_repos
git clone --filter=blob:none --no-checkout https://github.com/awslabs/mcp.git
cd mcp
git sparse-checkout init --cone
git sparse-checkout set src/aurora-dsql-mcp-server/skills/dsql-skill
git checkout
cd ../..
```

3. Skills ディレクトリに skill をシンボリックリンクする

skills ディレクトリを追加します (デフォルト: global/user-scoped):

```
mkdir -p ~/.claude/skills
```

> **Note**
>
> これをプロジェクトスコープの skill にする場合は、代わりにプロジェクトルートの `.claude/skills/` ディレクトリを使用します。

シンボリックリンクを追加します。

```
ln -s "$(pwd)/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/dsql-skill"
 ~/.claude/skills/dsql-skill
```

4. セットアップを確認する

```
# Should show SKILL.md and other skill files
ls -la ~/.claude/skills/dsql-skill/
```

5. Skill の使用を確認する

skill が設定されたら、新しい skill コマンドが作成されます。`/dsql`。skill を追加した後、skill が検出されるように Claude Code を再起動する必要がある場合があります。このコマンドは、必要に応じて Claude Code CLI またはパネルから使用できます。

Skill の更新

リポジトリから最新の変更をプルするには:

```
cd .dsql_skill_repos/mcp
git pull
```

ディレクトリ構造

グローバル skill を設定すると、次のディレクトリが表示されます。

```
.dsql_skill_repos/
### mcp/                              # Sparse git checkout
    ### src/
        ### aurora-dsql-mcp-server/
            ### skills/
```

```
                ### dsql-skill/
                    ### SKILL.md
                    ### ...

~/.claude/
### skills/
    ### dsql-skill -> /path/to/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/
dsql-skill
```

> **Note**
>
> 追跡しない場合は、.gitignore に .dsql_skill_repos/ を追加します。スパースチェックアウトでは skill フォルダのみが保持されるため、ディスク使用量が最小限に抑えられます。

Gemini Skill

Aurora DSQL skill を Gemini に直接追加するには、スコープ workspace (プロジェクトに含まれる) または user (デフォルト、グローバル) を決定し、skills インストーラを使用します。

セットアップ

```
gemini skills install https://github.com/awslabs/mcp.git --path src/aurora-dsql-mcp-
server/skills/dsql-skill --scope $SCOPE
```

$SCOPE を workspace または user に置き換えます。

その後、Gemini で /dsql skill コマンドを使用できるようになります。Gemini は skill を使用するタイミングを自動的に検出します。

Codex Skill

$skill-installer skill を使用して、Codex CLI または TUI から skill インストーラを使用します。

セットアップ

```
$skill-installer install dsql skill: https://github.com/awslabs/mcp/tree/main/src/
aurora-dsql-mcp-server/skills/dsql-skill
```

Codex を再起動して skill を取得します。その後、`$dsql` を使用して skill をアクティブ化できます。

Aurora DSQL クエリエディタの使用を開始する

Aurora DSQL クエリエディタを使用すると、外部クライアントをインストールまたは設定することなく、Aurora DSQL クラスターに安全に接続し、AWS マネジメントコンソールから直接 SQL クエリを実行できます。組み込みの構文強調表示、自動補完、インテリジェントなコード支援を備えた直感的なワークスペースを提供します。スキーマオブジェクトの探索、SQL クエリの開発と実行、結果の表示をすべて 1 つのインターフェイスですばやく行うことができます。

このトピックでは、クラスターへの接続、クエリの実行、結果の表示、実行プランなどの高度な機能の詳細を確認する手順について説明します。

> **Note**
>
> クエリエディタは、Aurora DSQL がサポートされているすべてのリージョンで利用できます。各リージョンでご利用いただけるサービスの詳細については、「AWS リージョン別のサービス」を参照してください。

前提条件

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。

- 使用可能な Aurora DSQL クラスターが少なくとも 1 つあります。クラスターの作成の詳細については、「ステップ 1: Aurora DSQL 単一リージョンクラスターを作成する」を参照してください。
- クラスターエンドポイントはパブリックアクセス可能です。クエリエディタは、リソースベースのポリシーによってパブリックアクセスがブロックされているクラスター、または VPC エンドポイントを介して管理されているクラスターをサポートしていません。アクセス制限の詳細については、「Aurora DSQL のリソースベースのポリシーでパブリックアクセスをブロックする」および「AWS PrivateLink を使用した Amazon Aurora DSQL クラスターの管理と接続」を参照してください。
- IAM ユーザーまたはロールには、クラスターにアクセスして接続するために必要なアクセス許可があります。アクセス許可の詳細については、「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。

クエリエディタの操作

クエリエディタを開く

クエリエディタを開くには

1. 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エリエディタを選択します。

または、[クラスター] ページから、クエリするクラスターを選択し、[クエリエディタと接続] を選択してエディタを直接起動します。

> **Note**
>
> 作業状態と接続状態は保存されません。Aurora DSQL コンソールから移動するか、ブラウザタブを閉じるか、サインアウトすると、接続、クエリテキスト、結果が失われます。

クラスターに接続する

クラスターに接続するには

1. クラスター接続が存在しない場合、エディタには [クラスターが接続されていません] と表示されます。既存のクラスターに接続するには、[クラスターエクスプローラー] ペインで [接続] または [+] (追加) を選択します。
2. (オプション) 異なるロールを使用して、複数のクラスターまたは同じクラスターに接続します。

クラスターオブジェクトを調べる

クラスターエクスプローラーには使用可能なすべてのクラスター接続が表示され、データベース、スキーマ、テーブル、ビューなどのオブジェクトを参照できます。また、[更新]、[テーブルの作成]、その他のコンテキスト固有のオプションなどの一般的なアクションも提供します。

クエリを実行する

クエリを実行する

1. クエリエディタのタブペインで、SQL ステートメントを入力します。例えば、次のようになります。

```
SELECT * FROM public.orders LIMIT 10;
```

2. クエリタブの右上に表示される [アクティブクラスターコンテキスト] を確認します。これは、現在のクエリタブに関連付けられているクラスター接続を示します。
3. (オプション) [接続] ドロップダウンを使用して、使用可能なすべての接続を確認するか、別のクラスターに切り替えます。接続を変更すると、そのタブのクエリが実行される場所が更新されます。
4. クエリを実行するには、[実行] を選択します。

> **Note**
>
> 各クエリは、結果ペインで最大 10,000 行を返すことができます。データセットが大きい場合は、フィルターまたは制限を使用してクエリを絞り込みます。

結果と実行計画を確認する

クエリが実行されたら、エディタの下部にある [結果パネル] で出力を確認します。デフォルトでは、各クエリ実行には [結果 (テーブル)] タブが表示され、表形式のクエリ出力が表示されます。

クエリ実行プランを取得するには、`EXPLAIN ANALYZE` または `EXPLAIN ANALYZE VERBOSE` を実行して、クエリのパフォーマンスに関する追加のインサイトを取得します。実行計画の詳細については、「Aurora DSQL EXPLAIN プランの読み取り」を参照してください。

> **Tip**
>
> `EXPLAIN ANALYZE VERBOSE` コマンドは、コンピューティング、読み取り、書き込み、合計 DPU 値を含む DPU 使用量の見積もりを表示し、個々の SQL ステートメントによって消費されるリソースを即座に可視化します。

クエリエディタ: Aurora DSQL での JupyterLab の使用

このガイドでは、JupyterLab と Python を使用し、Amazon Aurora DSQL に接続してクエリを実行するステップバイステップの手順について説明します。JupyterLab は、コード、テキスト、視覚化を 1 つのドキュメントにまとめた一般的なインタラクティブコンピューティング環境です。データサイエンスや研究のアプリケーションに広く使用されています。

以下の手順では、JupyterLab のローカルインストールと、データワークフロー用の UI を備えたホスト環境を提供するフルマネージド機械学習サービスである Amazon SageMaker AI の両方における Aurora DSQL の使用の基本について説明します。

開始方法

要件

- Aurora DSQL クラスター
- AWS 認証情報の設定 (ローカルインストールのみ)
- Python バージョン 3.9 以降 (ローカルインストールのみ)

ローカル JupyterLab の使用

JupyterLab の使用を開始するには、ユーザーはまず Python の pip を使用してアプリケーションをインストールする必要があります。

```
pip install jupyterlab
```

JupyterLab は、**`jupyter lab`** を実行して開くことができます。これにより、ブラウザからアクセスできる localhost:8888 で JupyterLab アプリケーションが開きます。続行する前に、ローカル環境で AWS 認証情報が設定されていることを確認してください。

Amazon SageMaker AI の使用

AWS コンソールで、Amazon SageMaker AI コンソールページに進み、[アプリケーションと IDE] の [ノートブック] セクションに進みます。そこから [ノートブックインスタンスの作成] を選択して、SageMaker 環境の作成を開始できます。[ノートブックインスタンスの作成] をクリックする前に、インスタンスタイプとプラットフォームを選択します。

セットアップとインスタンスオプションの詳細については、「Amazon SageMaker AI セットアップドキュメント」を参照してください。

> **Note**
>
> 警告: Amazon SageMaker AI を使用すると、AWS アカウントに料金が発生する可能性があります。

SageMaker インスタンスがアクティブになったら、[JupyterLab を開く] の [ノートブックインスタンス] セクションから開くことができます。ノートブックで Aurora DSQL の使用を開始する前に、SageMaker インスタンスの IAM ロールで DSQL クラスターへのアクセスを提供する必要があります。これを行う最も簡単な方法は、ノートブックインスタンスページの IAM ロールへのリンクに従うことです。そこから、SageMaker IAM ロールにアタッチされたポリシーを編集できます。Aurora DSQL へのアクセスを許可する IAM ポリシーの設定の詳細については、「認証と認可」を参照してください。

JupyterLab を使用した Aurora DSQL への接続

JupyterLab インスタンスを設定すると、Aurora DSQL に接続する手順はローカルと SageMaker AI で同じになります。空の Python 3 ノートブックを作成し、Python コードでセルを追加できます。

Python セルで、公式トラストストアから Amazon ルート証明書をダウンロードします。

```
import urllib.request
urllib.request.urlretrieve('https://www.amazontrust.com/repository/AmazonRootCA1.pem',
 'root.pem')
```

Aurora DSQL に接続するには、まず Python 用 Aurora DSQL コネクタ と Psycopg ドライバーを Python セルにインストールしてからインポートします。

```
pip install aurora_dsql_python_connector psycopg
```

```
import aurora_dsql_psycopg as dsql
```

コネクタをインポートすると、DSQL 設定を作成して接続できます。Aurora DSQL Python コネクタは、各接続での認証トークンの作成を自動的に処理します。

```
config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
```

```
    'region': "us-east-1",
    'user': "admin"
    }

conn = dsql.connect(**config)
```

コードを実行すると、Aurora DSQL への Psycopg 接続が可能になります。その後、Psycopg カーソルを使用して SQL クエリを指定してクエリを実行できます。Postgres 互換データベースで Psycopg を使用する方法の詳細については、「Psycopg ドキュメント」を参照してください。このクエリは、`results_list` のタプルのリストになります。

```
with conn:
    with conn.cursor() as cur:
        cur.execute("SELECT * FROM table")
        results_list = cur.fetchall()
```

その後、Pandas などの Python フレームワークを使用して、クエリ結果を分析または視覚化できます。次に例を示します。

```
pip install pandas

import pandas as pd

df = pd.DataFrame(tuples_list)
print(df)
print(f"Total records: {len(df)}")
```

サンプルノートブック

Aurora DSQL を使用するサンプルノートブックは、Aurora DSQL サンプルリポジトリで入手できます。

詳細情報

Amazon SageMaker AI セットアップドキュメント

Python 用 Aurora DSQL コネクタ

Pandas ドキュメント

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 マネジメントコンソール、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。

の開始方法AWS Backup

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。

シングルリージョンクラスターの復元

Aurora DSQL シングルリージョンクラスターを復元するには、コンソール (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。詳細な手順については、「Restore a single-Region Aurora DSQL cluster」を参照してください。

マルチリージョンクラスターの復元

Aurora DSQL マルチリージョンクラスターの復元は、コンソール (https://console.aws.amazon.com/backup) と AWS CLI の両方でサポートされています。詳細な手順については、「Restore a multi-Region Aurora DSQL cluster」を参照してください。

Aurora DSQL マルチリージョン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、マルチリージョンクラスターのすべての AWS リージョンに同一のバックアップコピーがあることを確認する必要があります。これらのコピーがまだない場合は、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。

バックアップコピーは、主要な AWS リージョンに作成し、堅牢なディザスタリカバリオプションを有効にして、コンプライアンス要件を満たすことをお勧めします。Aurora DSQL の利用可能な AWS リージョンを確認するには、「the section called “AWS リージョン の可用性”」を参照してください。

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。

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。

その他のリソース

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。

変更データキャプチャストリーム (プレビュー)

> **Important**
>
> この機能は AWS プレビューとして提供されており、変更される可能性があります。詳細については、「AWS のサービス条件」のセクション 2、「ベータ版とプレビュー」を参照してください。CDC ストリームの料金の詳細については、「Aurora DSQL の料金ページ」を参照してください。
> 一般提供する前に、ストリームペイロードに新しいオペレーションタイプ ("op": "u" 更新用) を追加します。アプリケーションがこれらの変更を修正せずに処理できるようにするには、`after` ペイロードを適用して、認識されない op 値をすべてアップサートとして扱います。詳細については、「CDC レコードについて」を参照してください。

Amazon Aurora DSQL 変更データキャプチャ (CDC) は、コミットされたデータベースの変更をほぼリアルタイムで Amazon Kinesis Data Streams に直接ストリーミングします。Aurora DSQL は、コミットされた各行レベルの変更を構造化された JSON レコードとして、設定した Kinesis データストリームに送信します。

CDC は、次のような場合に役立ちます。

- ダウンストリームシステムを同期させる – バッチジョブを使用せずに検索インデックス、キャッシュ、データウェアハウス、または分析システムに変更をレプリケートします。
- イベント駆動型アーキテクチャを構築する – データベースの変更に応じてワークフロー、通知、またはマイクロサービスのアクションをトリガーします。
- 監査証跡を維持する – コンプライアンス、デバッグ、または履歴分析のためにコミットされたすべての変更をキャプチャします。
- プロデューサーをコンシューマーから切り離す – データベースはトランザクションに専念し、ダウンストリームシステムはそれぞれのペースで変更を処理します。

仕組み

Aurora DSQL はコミットされたトランザクションを読み取り、各行の変更を構造化された JSON レコードとしてフォーマットし、設定した Kinesis データストリームに配信します。CDC は、クラスター内のすべてのユーザーテーブルにわたって、すべての `INSERT`、`UPDATE`、`DELETE` を自動的に

キャプチャします。各 CDC レコードの `source.schema` フィールドと `source.table` フィールドを使用して、ダウンストリームのアプリケーションにフィルタリングロジックを適用し、アプリケーションに必要なテーブルや変更に焦点を当てます。

CDC ストリームはフルマネージド型です。Aurora DSQL は、変更イベントをキャプチャするために必要なすべてのインフラストラクチャを管理し、ストリームの状態をモニタリングし、`GetStream` API オペレーションと CloudWatch メトリクスを通じてステータスを報告します。

CDC ストリームは、独自のターゲットを持ち込むモデルを使用します。お客様はご自身のアカウントで Kinesis データストリームを作成および管理し、Aurora DSQL はお客様が設定した IAM ロールを引き受けて、ユーザーに代わって CDC レコードを書き込みます。ターゲットの容量、暗号化、保持設定はお客様の責任となります。サポートされている最新のターゲットについては、Amazon Aurora DSQL API リファレンスの「CreateStream」の `TargetDefinition` パラメータを参照してください。CDC ストリーム API オペレーションの完全なリストについては、「Amazon Aurora DSQL API リファレンス」を参照してください。

このページのトピック

関連トピック

- CDC ストリームの使用開始
- IAM の設定
- CDC レコードについて
- ストリームのモニタリング

順序付けと配信のセマンティクス

配信の保証

Aurora DSQL CDC は、コミットされたすべての変更が少なくとも 1 回ターゲットに到達することを保証します。Aurora DSQL はレコードを複数回配信できます。重複を処理できるようにアプリケー

ションを設計してください。source.ts_ns とプライマリキーの値を比較することで、重複を特定できます。重複は、元の配信と同じ値を持っています。

順序付け

CDC ストリームは UNORDERED モードを使用します。実際には、Aurora DSQL は変更を順番に読み取り、発行するため、レコードはおおよそコミット順序で到着します。ただし、Aurora DSQL は厳密な順序を保証するものではありません。具体的には次のとおりです。

- Aurora DSQL は、さまざまなトランザクションからのレコードを任意の順序で配信できます。
- 異なるトランザクションからの同じプライマリキーのレコードは、コミット順とは異なる順で到着する可能性があります。
- 単一のトランザクションからのレコードは、他のトランザクションからのレコードとインターリーブする可能性があります。ワークフローで必要な場合は、source.txId フィールドを使用して、トランザクションごとにレコードをグループ化します。

各 CDC レコードには、トランザクションのコミットタイムスタンプ (ナノ秒単位) を含む source.ts_ns フィールドが含まれます。このフィールドを使用して、受信側でコミット順序を確立します。

コンシューマー戦略

レコードはコミット順とは異なる順序で到着したり、複数回出現したりする可能性があるため、アプリケーションは両方の状況に対応する必要があります。

> **Important**
>
> CDC に参加するすべてのテーブルでプライマリキーを定義します。プライマリキーがない場合、アプリケーションはレコードの重複排除や、削除と影響を受ける行の関連付けを行うことができません。

最後の書き込みを優先 (マテリアライズドビュー、キャッシュ)

プライマリキーごとに、最も大きい source.ts_ns 値を追跡します。source.ts_ns が追跡された値以下のレコードはすべて破棄します。これにより、重複レコードと順序通りでないレコードの両方がフィルタリングされ、各キーの最新の状態が維持されます。削除 (op: "d") を処理するとき

は、エントリを削除するのではなく、source.ts_ns 値を保持するプライマリキーのトゥームストーンを保存します。トゥームストーンは、削除後に到着した以前の source.ts_ns を使用した挿入または更新によって、行が誤って復元されないようにします。

すべての変更処理 (監査ログ記録、イベントソーシング)

source.ts_ns とプライマリキー値を組み合わせて比較して、重複を削除します。受信レコードをバッファし、処理前に source.ts_ns でソートしてコミット順序を再構築します。

マルチリージョン CDC ストリーム設定

CDC ストリームはリージョンのリソースです。各ストリームは単一の AWS リージョンに属し、同じリージョン内の Kinesis データストリームに変更を配信します。マルチリージョンクラスターでは、任意の 1 つのリージョンの CDC ストリームが、クラスター内のすべてのリージョンからのコミットされた書き込みをキャプチャします。つまり、書き込みの発生場所に関係なく、すべての変更をキャプチャするのに必要なストリームは 1 つだけです。CDC レコードを複数のリージョンに配信するには、リージョンごとに個別のストリームを作成します。各ストリームは、クラスター全体でコミットされたすべての変更を個別にキャプチャします。

Aurora DSQL クラスター、Kinesis データストリーム、IAM サービスロール、呼び出し元プリンシパルなどのすべてのリソースは、同じ AWS アカウントとリージョンに存在する必要があります。

CDC レコードのダウンストリーム処理

CDC レコードが Kinesis データストリームに到着したら、そのレコードを直接処理したり、AWS 統合サービスを使用して他の送信先にルーティングしたりできます。次の表は、一般的な処理パターンをまとめたものです。

CDC レコードの一般的な処理パターン

パターン	仕組み
直接消費	Read records from Kinesis by using the Amazon Kinesis Client Library (KCL), the AWS SDK, or a Kinesis Data Streams consumer. See KCL コンシューマーの開発 in the Amazon Kinesis Data Streams デベロッパーガイド.

パターン	仕組み
AWS Lambda	Configure a Lambda function as an event source for your Kinesis data stream to process each batch of CDC records as they arrive. See Amazon Kinesis での AWS Lambda の使用 in the AWS Lambda デベロッパーガイド.
Amazon Data Firehose	Deliver CDC records from Kinesis to Amazon S3, Amazon Redshift, Amazon OpenSearch Service, or other destinations for analytics and archival. See 配信ストリームへのデータの送信 in the Amazon Data Firehose デベロッパーガイド.
セルフマネージドコンシューマー	Run Apache Kafka Connect with the Kinesis source connector, Apache Flink, or other stream processing frameworks to transform and route records. For Apache Flink on AWS, see アプリケーション入力の設定 in the Amazon Managed Service for Apache Flink デベロッパーガイド.

各 CDC レコードには、処理ロジックでレコードをルーティングおよびフィルタリングするために使用できる `source.schema`、`source.table`、`op` などのフィールドが含まれています。レコードスキーマ全体については、「CDC レコードについて」を参照してください。

CDC ストリームの使用開始

> **Important**
>
> この機能は AWS プレビューとして提供されており、変更される可能性があります。詳細については、「AWS のサービス条件」のセクション 2、「ベータ版とプレビュー」を参照してください。CDC ストリームの料金の詳細については、「Aurora DSQL の料金ページ」を参照してください。
> 一般提供する前に、ストリームペイロードに新しいオペレーションタイプ ("op": "u" 更新用) を追加します。アプリケーションがこれらの変更を修正せずに処理できるようにするに

は、after ペイロードを適用して、認識されない op 値をすべてアップサートとして扱います。詳細については、「CDC レコードについて」を参照してください。

このガイドでは、Aurora DSQL クラスターから Amazon Kinesis データストリームにコミットされた行レベルの変更をストリーミングするために必要なすべてのステップについて説明します。このガイドの最後には、動作する CDC パイプラインと、変更レコードの読み取りと出力を行う Python スクリプトを作成しているはずです。

前提条件

開始する前に、以下の点を確認してください。

- ACTIVE ステータスの Aurora DSQL クラスターを作成しました。クラスターがアイドル状態の場合は、CDC ストリームを作成する前に、PostgreSQL 互換クライアントを使用してクラスターに接続し、起動します。CreateStream は、クラスターのステータスが ACTIVE でない場合は検証エラーを返します。
- Aurora DSQL では、クラスター、Amazon Kinesis データストリーム、IAM サービスロール、呼び出し元プリンシパルなど、すべての CDC リソースが同じ AWS アカウントに存在する必要があります。
- Amazon Kinesis データストリームは、Aurora DSQL クラスターと同じ AWS リージョンにあります。
- IAM ロールと Amazon Kinesis Data Streams を作成するアクセス許可を持つ認証情報を使用して、AWS CLI をインストールして設定しました。

ステップ 1: Amazon Kinesis データストリームを作成する

Aurora DSQL クラスターと同じ AWS アカウントおよびリージョンに Kinesis データストリームを作成します。CDC レコードは、JSON 形式に列名、メタデータ、エンコーディングのオーバーヘッドが含まれるため、対応する Aurora DSQL 行よりもサイズが大きくなります。

Kinesis データストリームのサイズ設定

Aurora DSQL CDC は、すべての変更に対して完全な行を配信します。単一の列のみを変更する更新を行うと、その行のすべての列を含むレコードが生成されます。削除レコードは例外であり、プライマリキー列のみが含まれます。

平均レコードサイズの見積もり

CDC が生成するボリュームを把握し、サイズの大きいレコードを予測するために、ディスク上の平均行サイズを測定します。次のクエリは、テーブルのタプルの平均サイズ (バイト単位) を返します。

```
SELECT avg(pg_column_size(t.*)) FROM your_table t;
```

CDC レコードエンベロープは、行サイズに加えて、列名、メタデータ、エンコーディングのオーバーヘッドを追加します。正確なレコード形式については、「レコードのペイロード」を参照してください。Aurora DSQL が Kinesis のレコードサイズ制限を超えるレコードを処理する方法については、「サイズの大きいレコードの処理」を参照してください。Kinesis サービスの制限の詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の「Amazon Kinesis Data Streams のクォータと制限」を参照してください。

> **Important**
>
> Kinesis データストリームを作成するときは、以下を設定します。
>
> - `MaxRecordSizeInKiB` を `10240` (10 MiB) に変更します。Kinesis のデフォルトの最大 1 MiB は、Aurora DSQL CDC レコードに十分な大きさとは限りません。設定された Kinesis レコードサイズを超えるレコードがあると、CDC ストリームが `KINESIS_OVERSIZE_RECORD` で損なわれます。Aurora DSQL は、サイズの大きいレコードをそれぞれ 10 MiB 近くになるフラグメントに分割するため、Kinesis データストリームはそのサイズのレコードを受け入れる必要があります。詳細については、「サイズの大きいレコードの処理」を参照してください。
> - `StreamMode` を `ON_DEMAND` へオンデマンドモードでは、シャード容量が自動的にスケーリングされ、予期しない急増時のプロビジョニング不足を防ぎます。容量がスケールアップしても、秒単位の急激なバーストが発生した場合、Kinesis が `WriteProvisionedThroughputExceeded` を返す可能性があります。短いスロットリングイベントを想定して計画します。

`AWS/Kinesis` 名前空間の `IncomingBytes` と `WriteProvisionedThroughputExceeded` に対して CloudWatch アラームを作成します。Kinesis のスロットリングは CDC の配信を遅らせ、レプリケーションの遅延を増加させます。Aurora DSQL 側のメトリクスとアラームガイダンスについては、「モニタリングのベストプラクティス」を参照してください。

次の例では AWS CLI を使用しています。お使いの AWS CLI のバージョンで `--max-record-size-in-ki-b` パラメータがサポートされていない場合は、AWS SDK を使用して Kinesis CreateStream オペレーションを呼び出します。

```
aws kinesis create-stream \
  --stream-name my-cdc-stream \
  --stream-mode-details StreamMode=ON_DEMAND \
  --max-record-size-in-ki-b 10240 \
  --region region
```

ストリームがアクティブになるまで待機します。

```
aws kinesis describe-stream-summary \
  --stream-name my-cdc-stream \
  --region region \
  --query 'StreamDescriptionSummary.StreamStatus'
```

ストリームの準備ができると、コマンドは `"ACTIVE"` を返します。

出力からストリーム ARN を記録します。以降のステップで必要になります。ARN の形式は `arn:aws:kinesis:region:account-id:stream/my-cdc-stream` です。

ステップ 2: Aurora DSQL 用の IAM ロールを作成する

Aurora DSQL は、Kinesis データストリームに CDC レコードを書き込む IAM ロールを引き受けます。このステップでは、信頼ポリシーを持つロールを作成し、アクセス許可ポリシーをアタッチします。各ポリシー要素の詳細については、「IAM の設定」を参照してください。

信頼ポリシーファイルを作成する

次の JSON を `trust-policy.json` として保存します。*your-account-id*、*region*、*cluster-id* をお客様の値に置き換えます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAccess",
            "Effect": "Allow",
            "Principal": {
```

```
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
                }
            }
        }
    ]
}
```

ロールを作成する

次の コマンドを実行して、IAM ロールを作成します。

```
aws iam create-role \
  --role-name dsql-cdc-role \
  --assume-role-policy-document file://trust-policy.json
```

アクセス許可ポリシーファイルを作成する

次の JSON を permissions-policy.json として保存します。プレースホルダー値を Kinesis データストリームの ARN に置き換えます。KMSAccess ステートメントは、Kinesis データストリームが AWS KMS カスタマーマネージドキーを使用する場合にのみ必要ですが、カスタマーマネージドキーを後で追加しても CDC ストリームが破損しないように、事前に含めることができます。各条件の詳細については、「サービスロールのアクセス許可ポリシー」を参照してください。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord",
                "kinesis:PutRecords",
                "kinesis:DescribeStreamSummary",
```

```
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws:kinesis:region:your-account-id:stream/my-cdc-stream"
        },
        {
            "Sid": "KMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:GenerateDataKey"
            ],
            "Resource": "arn:aws:kms:*:*:key/*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "kinesis.region.amazonaws.com",
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:region:your-account-id:stream/my-cdc-stream",
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        }
    ]
}
```

アクセス許可ポリシーをアタッチする

次のコマンドを実行します。

```
aws iam put-role-policy \
  --role-name dsql-cdc-role \
  --policy-name dsql-cdc-kinesis-access \
  --policy-document file://permissions-policy.json
```

create-role 出力からロール ARN を記録します。ARN の形式は arn:aws:iam::*your-account-id*:role/*dsql-cdc-role* です。

ステップ 3: CDC ストリームを作成する

AWS CLI を使用して、Aurora DSQL クラスターを Kinesis データストリームに接続する CDC ストリームを作成します。プレースホルダー値を、ステップ 1 の Kinesis ストリーム ARN、ステップ 2 の IAM ロール ARN、およびクラスター識別子に置き換えます。

```
aws dsql create-stream \
```

```
  --cluster-identifier cluster-id \
  --target-definition '{"kinesis":{"streamArn":"kinesis-stream-arn","roleArn":"role-
arn"}}' \
  --ordering UNORDERED \
  --format JSON \
  --tags '{"Name":"my-cdc-stream"}' \
  --region region
```

レスポンスには、ストリーム識別子とステータス CREATING が含まれます。ストリームの作成には通常 1～3 分かかります。

ストリームがアクティブになるまで待機する

ACTIVE になるまでストリームステータスをポーリングします。

```
aws dsql get-stream \
  --cluster-identifier cluster-id \
  --stream-identifier stream-id \
  --region region \
  --query 'status'
```

AWS SDK で StreamActive ウェーターを使用して自動的にポーリングすることもできます。

ストリームが ACTIVE になると、Aurora DSQL はコミットされた行レベルの変更を Kinesis データストリームに配信し始めます。

> **Note**
>
> 各 Aurora DSQL クラスターには、CDC ストリームの最大数があります。この制限に達すると、CreateStream は ServiceQuotaExceededException を返します。デフォルトの制限については、「クォータと制限」を参照してください。

ステップ 4: レコードが流れていることを確認する

Aurora DSQL クラスターのテーブルに行を挿入します。例えば、次のようになります。

```
CREATE TABLE IF NOT EXISTS test_cdc (
    id INT PRIMARY KEY,
    message TEXT
```

```
);

INSERT INTO test_cdc VALUES (1, 'hello cdc');
```

Kinesis データストリームから読み取り、CDC レコードが到着したことを確認します。

```
SHARD_ITERATOR=$(aws kinesis get-shard-iterator \
  --stream-name my-cdc-stream \
  --shard-id shardId-000000000000 \
  --shard-iterator-type TRIM_HORIZON \
  --region region \
  --query 'ShardIterator' --output text)

aws kinesis get-records \
  --shard-iterator "$SHARD_ITERATOR" \
  --region region
```

各レコードの Data フィールドには、JSON ペイロードが含まれています。AWS CLI を使用する場合、ペイロードはレスポンスで Base64 エンコードされます。boto3 SDK を使用すると、SDK が自動的にデコードします。デコードされた JSON は以下のようになります。

```
{
    "type": "full",
    "op": "c",
    "before": null,
    "after": {"id": 1, "message": "hello cdc"},
    "source": {
        "version": "1.0",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
        "table": "test_cdc",
        "db": "postgres",
        "cluster": "cluster-id"
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291
}
```

各フィールドの詳細な説明については、「CDC レコードについて」を参照してください。

ステップ 5: Python スクリプトを使用してレコードを消費する

次の Python スクリプトは、Kinesis データストリームから CDC レコードを読み取り、各変更イベントを出力します。このスクリプトは、boto3 Amazon Kinesis クライアントを使用してシャードを反復処理し、各レコードをデコードします。Aurora DSQL CDC は少なくとも 1 回の配信を使用するため、スクリプトは同じレコードを複数回出力することがあります。

```
"""
Read CDC records from an Amazon Kinesis data stream.

Usage:
    pip install boto3
    python consume_cdc.py --stream-name my-cdc-stream --region us-east-1
"""
from __future__ import annotations

import argparse
import json

import boto3


def consume_cdc(stream_name: str, region: str) -> None:
    kinesis = boto3.client("kinesis", region_name=region)

    # List all shards (paginate if the stream has many shards)
    shard_ids: list[str] = []
    paginator = kinesis.get_paginator("list_shards")
    for page in paginator.paginate(StreamName=stream_name):
        shard_ids.extend(s["ShardId"] for s in page["Shards"])
    print(f"Reading from {stream_name} ({len(shard_ids)} shard(s))")

    for shard_id in shard_ids:
        iterator_response = kinesis.get_shard_iterator(
            StreamName=stream_name,
            ShardId=shard_id,
            ShardIteratorType="TRIM_HORIZON",
        )
        shard_iterator = iterator_response["ShardIterator"]

        while shard_iterator:
            records_response = kinesis.get_records(
                ShardIterator=shard_iterator, Limit=100
```

```
            )
            shard_iterator = records_response.get("NextShardIterator")

            for record in records_response["Records"]:
                # boto3 decodes Base64 automatically; record["Data"] is bytes.
                payload = json.loads(record["Data"])

                # A record's "type" field identifies its structure.
                # "full": inlined record with before/after values.
                # "chunked": main record that references fragments for a split image.
                # "fragment": one piece of a chunked image; reassemble in production
code.
                # For details, see cdc-record-format.html#cdc-oversized-records.
                record_type = payload.get("type", "full")
                if record_type == "fragment":
                    print(f"[FRAGMENT] chunk_id={payload['chunk_id']}
index={payload['index']}")
                    continue

                source = payload["source"]
                op = payload["op"]
                ts_ns = source["ts_ns"]
                tx_id = source["txId"]
                table = f"{source['schema']}.{source['table']}"

                # Aurora DSQL currently emits "c" for both inserts and updates. A
subsequent
                # release will emit "u" for updates, and "c" for inserts. Design your
                # consumer to handle all three values; this map stays correct across
the
                # transition.
                op_labels = {"c": "INSERT/UPDATE", "u": "UPDATE", "d": "DELETE"}
                print(
                    f"[{op_labels.get(op, op)}] {table} "
                    f"txId={tx_id} ts_ns={ts_ns} type={record_type}"
                )
                if payload.get("after"):
                    print(f"  after:  {json.dumps(payload['after'])}")
                if payload.get("before"):
                    print(f"  before: {json.dumps(payload['before'])}")
                if record_type == "chunked":
                    print(f"  chunked: {json.dumps(payload['chunked'])}")

            if not records_response["Records"]:
```

```
                break  # No more records in this shard


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Consume DSQL CDC records from Kinesis"
    )
    parser.add_argument("--stream-name", required=True, help="Kinesis stream name")
    parser.add_argument("--region", required=True, help="AWS Region")
    args = parser.parse_args()
    consume_cdc(args.stream_name, args.region)
```

スクリプトを実行します。

```
pip install boto3
python consume_cdc.py \
  --stream-name my-cdc-stream \
  --region region
```

スクリプトは、各変更イベントが到着するたびにそれを出力します。以下のような出力結果が表示されます。

```
Reading from my-cdc-stream (4 shard(s))
[INSERT/UPDATE] public.test_cdc txId=ffthunp5stx6ffs2vyfqoatmfu
 ts_ns=1705318200000000000 type=full
  after:  {"id": 1, "message": "hello cdc"}
```

最後の書き込みを優先重複排除の追加

Aurora DSQL CDC は少なくとも 1 回の配信を使用するため、本番アプリケーションではレコードの重複排除と順序付けを行う必要があります。次のコード例はハイウォーターマークアプローチを示しています。プライマリキーごとに、これまでに検出された最大の source.ts_ns を追跡し、同じまたはそれ以前のタイムスタンプを持つレコードを破棄します。PK_COLUMNS には、処理するテーブルのプライマリキー列名を設定します。複数のテーブルや削除を処理する戦略については、「コンシューマー戦略」を参照してください。

```
# Set PK_COLUMNS to the primary key column(s) of your table.
PK_COLUMNS = ["id"]

# Maps each primary key value to the highest ts_ns seen for that key.
high_water: dict[tuple, int] = {}
```

```
def process_record(payload: dict) -> bool:
    """Return True if the record is new, False if it's a duplicate or stale.

    Skip fragment records; reassemble them into a full image before calling this.
    """
    if payload.get("type") == "fragment":
        return False  # Fragments are reassembled upstream, not deduplicated here.

    source = payload["source"]
    ts_ns = source["ts_ns"]
    op = payload["op"]

    # For inserts/updates the row is in "after"; for deletes it's in "before".
    row = payload.get("after") or payload.get("before") or {}
    pk = tuple(row.get(col) for col in PK_COLUMNS)

    prev_ts = high_water.get(pk, -1)
    if ts_ns <= prev_ts:
        return False  # Duplicate or out-of-order record

    high_water[pk] = ts_ns
    return True
```

CDC ストリームの管理

ストリームを一覧表示する

クラスターのすべての CDC ストリームを一覧表示するには、ListStreams オペレーションを使用します。

```
aws dsql list-streams \
  --cluster-identifier cluster-id \
  --region region
```

ストリームを削除する

CDC ストリームを削除するには、次のコマンドを実行します。

```
aws dsql delete-stream \
  --cluster-identifier cluster-id \
  --stream-identifier stream-id \
```

```
  --region region
```

StreamNotExists ウェーターを使用して GetStream をポーリングし、Aurora DSQL がストリームを完全に削除したことを示す ResourceNotFoundException が返されるまで続けることができます。

IAM の設定

> **Important**
>
> この機能は AWS プレビューとして提供されており、変更される可能性があります。詳細については、「AWS のサービス条件」のセクション 2、「ベータ版とプレビュー」を参照してください。CDC ストリームの料金の詳細については、「Aurora DSQL の料金ページ」を参照してください。
> 一般提供する前に、ストリームペイロードに新しいオペレーションタイプ ("op": "u" 更新用) を追加します。アプリケーションがこれらの変更を修正せずに処理できるようにするには、after ペイロードを適用して、認識されない op 値をすべてアップサートとして扱います。詳細については、「CDC レコードについて」を参照してください。

CDC ストリームには、2 つの異なる IAM アクセス許可のセットが必要です。

- 発信者のアクセス許可 – CDC ストリーム API オペレーション (CreateStream、GetStream、DeleteStream、ListStreams) を呼び出す IAM プリンシパルには、これらのアクションと iam:PassRole のアクセス許可が必要です。
- サービスロール – CDC レコードをターゲットに書き込むために Aurora DSQL が実行時に引き受ける IAM ロール。このロールを作成し、Aurora DSQL サービスプリンシパルがロールを引き受けることを許可する信頼ポリシーをアタッチし、ターゲットへの書き込みアクセスを許可するアクセス許可ポリシーをアタッチします。

> **Note**
>
> CDC サービスロールは、Aurora DSQL クラスターのリソースベースのポリシーとは別のものです。クラスターリソースベースのポリシーは、クラスターに接続してクエリできるプリンシパルを制御します。CDC サービスロールは、Aurora DSQL が CDC レコードを書き込むことができるターゲットを制御します。

発信者のアクセス許可

CDC ストリーム API オペレーションを呼び出す IAM プリンシパルには、関連する dsql アクションおよび iam:PassRole に対するアクセス許可が必要です。CreateStream オペレーションでは、サービスロール ARN を Aurora DSQL に渡すため、iam:PassRole が必要です。次に、ポリシーの例を示します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLStreamActions",
            "Effect": "Allow",
            "Action": [
                "dsql:CreateStream",
                "dsql:GetStream",
                "dsql:ListStreams",
                "dsql:DeleteStream"
            ],
            "Resource": [
                "arn:aws:dsql:region:your-account-id:cluster/cluster-id",
                "arn:aws:dsql:region:your-account-id:cluster/cluster-id/stream/*"
            ]
        },
        {
            "Sid": "PassServiceRole",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::your-account-id:role/dsql-cdc-role",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "dsql.amazonaws.com"
                }
            }
        }
    ]
}
```

Resource 要素には、クラスター ARN (CreateStream および ListStreams で必要) とストリーム ARN パターン (GetStream と DeleteStream で必要) の両方が含まれます。

各オペレーションに必要なアクセス許可の完全なリストについては、「Amazon Aurora DSQL API リファレンス」の「CreateStream」、「GetStream」、「DeleteStream」、および「ListStreams」を参照してください。

サービスロール

サービスロールとは、Aurora DSQL が CDC レコードをターゲットに書き込むために引き受ける IAM ロールです。このロールを作成し、`CreateStream` を呼び出すときに、その ARN を `targetDefinition.kinesis.roleArn` フィールドに渡します。ロールには、信頼ポリシーとアクセス許可ポリシーが必要です。

サービスロール信頼ポリシー

信頼ポリシーでは、Aurora DSQL サービスプリンシパルがロールを引き受けることを許可する必要があります。混乱した代理攻撃を防ぐには、`aws:SourceAccount` および `aws:SourceArn` 条件キーを使用します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
                }
            }
        }
    ]
}
```

aws:SourceArn 条件は、ロールを特定のクラスターのストリームに制限します。Aurora DSQL がまだストリーム識別子を割り当てていないため、ストリームを作成するときはワイルドカード (stream/*) を使用する必要があります。ストリームを作成したら、ロールが単一のストリームを提供する場合、条件を正確なストリーム ARN (arn:aws:dsql:*region*:*your-account-id*:cluster/*cluster-id*/stream/*stream-id*) に絞り込むことができます。

アカウントの任意のクラスターで、ストリームを使用してロールを使用するには、より広範なワイルドカード arn:aws:dsql:*region*:*your-account-id*:cluster/*/stream/* を使用します。

混乱した代理の防止の詳細については、このガイドの「サービス間の混乱した代理の防止」を参照してください。

サービスロールのアクセス許可ポリシー

アクセス許可ポリシーにより、サービスロールは Kinesis データストリームにレコードを書き込むためのアクセス許可を取得します。次のポリシーには、Kinesis の書き込みアクセス許可と AWS KMS アクセス許可の両方が含まれています。KMSAccess ステートメントは、Kinesis データストリームが AWS KMS カスタマーマネージドキーを使用する場合にのみ必要ですが、カスタマーマネージドキーを後で追加しても CDC ストリームが破損しないように、事前に含めることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord",
                "kinesis:PutRecords",
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws:kinesis:region:your-account-id:stream/kinesis-stream-name"
        },
        {
            "Sid": "KMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:GenerateDataKey"
            ],
```

```
            "Resource": "arn:aws:kms:*:*:key/*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "kinesis.region.amazonaws.com",
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:region:your-account-id:stream/kinesis-stream-name",
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        }
    ]
}
```

AWS KMS ステートメントの条件は、次の保護を提供します。

- `kms:ViaService` – キーの使用を、指定されたリージョンで Kinesis サービスを経由するリクエストに制限します。
- `kms:EncryptionContext:aws:kinesis:arn` – キーの使用を、指定された Kinesis データストリームの暗号化オペレーションに制限します。
- `aws:ResourceAccount` – キーは呼び出し元のプリンシパルと同じ AWS アカウントに属している必要があり、これによりクロスアカウントでのキーの使用が防止されます。

> **Note**
>
> ここで参照される AWS KMS キーは、クラスターの AWS KMS キーではなく、Kinesis データストリームの暗号化キーです。クラスターの暗号化キーは、Aurora DSQL 境界内の CDC データを保護します。Kinesis 暗号化キーは、Aurora DSQL が Kinesis データストリームに書き込んだ後の CDC データを保護します。

データ保護

Aurora DSQL は Transport Layer Security (TLS) を使用して、Aurora DSQL とターゲット間で転送中の CDC データを暗号化します。Aurora DSQL 境界内では、Aurora DSQL はクラスターの暗号化キーを使用して保管中の CDC データを暗号化します。

クラスターが AWS KMS カスタマーマネージドキーを使用していて、そのキーにアクセスできなくなった場合、`ACTIVE` または `IMPAIRED` ストリームはエラーコード `CLUSTER_CMK_INACCESSIBLE`

で IMPAIRED に移行します。ストリームの作成が完了する前にキーにアクセスできなくなった場合、ストリームは FAILED に直接移行します。

Aurora DSQL での暗号化の詳細については、このガイドの「Amazon Aurora DSQL のデータ暗号化」を参照してください。

CDC レコードについて

> **Important**
>
> この機能は AWS プレビューとして提供されており、変更される可能性があります。詳細については、「AWS のサービス条件」のセクション 2、「ベータ版とプレビュー」を参照してください。CDC ストリームの料金の詳細については、「Aurora DSQL の料金ページ」を参照してください。
> 一般提供する前に、ストリームペイロードに新しいオペレーションタイプ ("op": "u" 更新用) を追加します。アプリケーションがこれらの変更を修正せずに処理できるようにするには、after ペイロードを適用して、認識されない op 値をすべてアップサートとして扱います。詳細については、「CDC レコードについて」を参照してください。

Aurora DSQL CDC は、各変更を JSON レコードとして配信します。レコードは、オペレーションタイプ、前後の行イメージ、ソースメタデータを含むエンベロープ構造を使用します。

レコードが Amazon Kinesis にマッピングされる方法

Aurora DSQL は、各 CDC レコードを単一の Kinesis レコードとして書き込みます。Kinesis レコードの Data フィールドには、JSON ペイロードが含まれています。Aurora DSQL は、ランダム化された Kinesis パーティションキーを使用して CDC レコードをシャード間で均等に分散します。すべての変更を読み取るには、Kinesis データストリームのすべてのシャードを消費します。レコードが Kinesis レコードのサイズ制限を超えると、Aurora DSQL はそれを複数の Kinesis レコードに分割します。詳細については、「サイズの大きいレコードの処理」を参照してください。

> **Note**
>
> Kinesis レコードには 1 つの Data BLOB があります。プライマリキー値は、削除の場合は JSON ペイロードの before フィールドに、挿入および更新の場合は after フィールドに表示されます。ダウンストリーム処理用のプライマリキーを抽出するには、ペイロードの適切なフィールドから読み取ります。

ペイロードのプライマリキー

プライマリキーを持つテーブルの場合、プライマリキー列の値がペイロードに表示されます。

- 挿入と更新の場合、ペイロードにはプライマリキー列と after フィールド内の他のすべての列が含まれます。
- 削除の場合、プライマリキー列が before フィールドに表示されます。

例えば、複合プライマリキーを持つテーブルがあるとします。

```
CREATE TABLE order_items (
    order_id INT,
    item_id INT,
    quantity INT,
    price NUMERIC,
    PRIMARY KEY (order_id, item_id)
);
```

このテーブルを削除すると、"before": {"order_id": 1001, "item_id": 42} というペイロードが生成されます。

レコードのペイロード

ペイロードは次の JSON エンベロープ形式を使用します。

INSERT の例

次の例は、挿入オペレーションの CDC レコードを示しています。

```
{
    "type": "full",
    "op": "c",
    "before": null,
    "after": {"order_id": 1001, "item_id": 42, "quantity": 5, "price": "29.99"},
    "source": {
        "version": "1.0",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
```

```
        "table": "order_items",
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291
}
```

UPDATE の例

次の例は、Aurora DSQL が op: "u" の出力を開始した後、UPDATE ステートメントによって生成された CDC レコードがどのようになるかを示しています。

> **Important**
>
> 現在、Aurora DSQL は挿入と更新の両方に対して op: "c" を出力します。後続のリリースでは、更新の場合は op: "u"、挿入の場合は op: "c" が出力されます。c、u、および d を処理するようにアプリケーションを設計し、コンシューマーが移行後も動作し続けるようにします。

```
{
    "type": "full",
    "op": "u",
    "before": null,
    "after": {"order_id": 1001, "item_id": 42, "quantity": 10, "price": "29.99"},
    "source": {
        "version": "1.0",
        "ts_ms": 1705318300000,
        "ts_ns": 1705318300000000000,
        "txId": "qvtiesgmd55cvlfukm3dfuotji",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318300125,
    "ts_ns": 1705318300125483291
}
```

DELETE の例

プライマリキーを持つテーブルの削除の場合、before フィールドには削除された行のプライマリキー値が含まれます。

```
{
    "type": "full",
    "op": "d",
    "before": {"order_id": 1001, "item_id": 42},
    "after": null,
    "source": {
        "version": "1.0",
        "ts_ms": 1705318400000,
        "ts_ns": 1705318400000000000,
        "txId": "xyzabc123def456ghi789jklmno",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "kmabugltfmjdaj2siqr2qbxgju"
    },
    "ts_ms": 1705318400125,
    "ts_ns": 1705318400125483291
}
```

ペイロードフィールド

フィールド	説明
#	The record type. ## for a complete record that includes inline ## and ## values. `chunked` for a main record that references fragment records for one or both images. `fragment` for an individual piece of a chunked image. For details, see サイズの大きいレコードの処理.
op	Operation type. c = create (insert), u = update, d = delete. Currently Aurora DSQL emits c for both inserts and updates. A subsequent release will emit u for updates, and c for inserts. Design your app to handle all three values.

フィールド	説明
`##`	For deletes on tables with a primary key, contains the primary key values of the deleted row. Aurora DSQL sets this field to `null` for inserts, updates, and deletes on tables without a primary key.
`##`	The full row state after the change, including all columns. Aurora DSQL sets this field to `null` for deletes.
`chunked`	Present only when # is `chunked`. Contains reassembly metadata for the ## image, the ## image, or both. Aurora DSQL omits the chunked image from the top-level ## or ## field and places it under `chunked` instead. For details, see サイズの大きいレコードの処理.
`source.version`	The CDC source metadata format version. The current version is `1.0`.
`source.ts_ms`	The transaction commit timestamp in milliseconds since the Unix epoch, Coordinated Universal Time (UTC).
`source.ts_ns`	Transaction commit timestamp in nanoseconds, UTC. The highest precision timestamp available. Use this field to establish a total order of transactions.
`source.txId`	A unique transaction identifier, encoded as base32. All records from the same transaction share the same `txId` value. Use this field to group records that belong to the same transaction.

フィールド	説明
source.schema	The PostgreSQL schema name (for example, ####).
source.table	The table name.
source.db	The database name. Always postgres for Aurora DSQL.
source.cluster	The Aurora DSQL cluster identifier.
ts_ms	The time at which the CDC system processed the record, in milliseconds, UTC. The difference between ts_ms and source.ts_ms is a measure of replication lag.
ts_ns	The time at which the CDC system processed the record, in nanoseconds, UTC.

フォーマットの詳細

以下の詳細では、Aurora DSQL CDC がレコードをフォーマットする方法について説明します。これらの動作に対応できるようにアプリケーションを設計します。

- 挿入と更新の完全な後のイメージ。Aurora DSQL は、すべての書き込みの after フィールドに完全な行状態を含めます。挿入および更新の場合、before フィールドは null になります。現在、挿入と更新の両方で op: "c" が使用されますが、それ以降のリリースでは更新のために op: "u" が出力されます。挿入と更新を区別するために op フィールドに依存するのではなく、プライマリキーごとに source.ts_ns を使用して順序付けするようにアプリケーションを設計します。
- 変更後の行の状態のみ。CDC レコードには、変更後の各行の状態がすべて含まれます。更新前の行の状態は含まれません。プライマリキーを持つテーブルの削除の場合、before フィールドにはプライマリキー値が含まれます。
- 文字列としてシリアル化された数値型。Aurora DSQL は、正確な精度を維持するために numeric および decimal 値を JSON 文字列としてシリアル化します。
- Base64 でエンコードされたバイナリデータ。Aurora DSQL は、bytea 値を Base64 文字列としてエンコードします。

- 特殊な浮動小数点値と数値。Aurora DSQL は NaN と ±Infinity を文字列 "NaN"、"Infinity"、および "-Infinity" としてシリアル化します。これは、real、double precision、および numeric タイプに適用されます。
- JSON 文字列としてシリアル化された JSON 列。Aurora DSQL は、json 列に格納されている未加工の JSON テキストを含む JSON 文字列として列の値をシリアル化します。アプリケーションで文字列値を解析し (JavaScript の JSON.parse や Python の json.loads など)、基になる JSON 値にアクセスします。
- null として出力されるオーバーフロー値。シリアル化中にターゲット JSON タイプで値を表現できない場合、Aurora DSQL はその列に対して JSON の null を出力します。これは、合計マイクロ秒が 64 ビット符号付き整数範囲 (±9,223,372,036,854,775,807 マイクロ秒、約 ±292,271 年) を超える interval 値に適用されます。データベーススキーマで null を許容しない列における予期しない null 値を処理できるようにアプリケーションを設計します。
- サイズの大きいレコードはチャンクに分割されます。レコードが Amazon Kinesis レコードのサイズ制限を超えると、Aurora DSQL は影響を受ける before または after イメージをフラグメントに分割し、個別の Kinesis レコードとして配信するため、変更を受け取ることができます。イメージを再構成するようにアプリケーションを設計します。詳細については、「サイズの大きいレコードの処理」を参照してください。

サイズの大きいレコードの処理

CDC レコードのシリアル化された JSON が 9 MiB を超えると、Aurora DSQL は before および/または after イメージを分割し、複数の Kinesis レコードを配信します。各レコードには、その構造を示す最上位の type フィールドが含まれています。完全なレコードの場合は full、フラグメントを参照するメインレコードの場合は chunked、チャンクイメージの個々の部分の場合は fragment です。チャンク化されたメインレコードの op、source、ts_ms、および ts_ns フィールドは、完全なレコードの場合と同じように動作します。単一の Kinesis レコードに収まるレコードは type が full に設定されており、追加の処理は必要ありません。

chunk_id は再試行間で安定しています。Aurora DSQL がフラグメントを再配信する場合、元の配信と同じ chunk_id を保持するため、アプリケーションは以前の試行の部分的なセットを処理せずに、同じ識別子でバッファリングを続行できます。

メインレコード

チャンク化されたメインレコードは、分割されたイメージの最上位の before または after フィールドを、再構成する方法を説明する chunked オブジェクトに置き換えます。chunked の各エント

リには、chunk_id (フラグメントをこのレコードにリンクする識別子)、total_fragments (そのイメージを構成するフラグメントの数)、crc32c (再アセンブルされたイメージテキストに対する 10 進文字列の CRC32C チェックサム) があります。1 つのイメージがインラインで、もう 1 つのイメージがチャンク化されている場合、インラインイメージは依然として最上位レベルに値または null として表示されます。

```
{
    "type": "chunked",
    "op": "c",
    "before": null,
    "after": null,
    "source": {
        "version": "1.0",
        "ts_ms": 1705318200000,
        "ts_ns": 1705318200000000000,
        "txId": "ffthunp5stx6ffs2vyfqoatmfu",
        "schema": "public",
        "table": "order_items",
        "db": "postgres",
        "cluster": "cluster-id"
    },
    "chunked": {
        "after": {
            "chunk_id": "chunk-id",
            "total_fragments": 3,
            "crc32c": "2073618257"
        }
    },
    "ts_ms": 1705318200125,
    "ts_ns": 1705318200125483291
}
```

フラグメントレコード

各フラグメントは、type が fragment に設定され、3 つのフィールドを持つ独自の Kinesis レコードです。chunk_id は、メインレコードの対応する chunked.before.chunk_id または chunked.after.chunk_id の値と一致し、index はイメージ内のフラグメントのゼロベースの位置であり、data は UTF-8 文字境界で分割されたイメージの JSON テキストのセグメントです (各フラグメントの data 値は、それ自体が有効な UTF-8 文字列です)。Aurora DSQL CDC は UNORDERED モードとランダム化されたパーティションキーを使用するため、フラグメントとメインレコードは異なるシャードに任意の順序で到着する可能性があります。すべてのフラグメントを読み

取るには、Kinesis データストリーム上のすべてのシャードを消費します。配信の順序付けの詳細については、「順序付け」を参照してください。

```
{
    "type": "fragment",
    "chunk_id": "chunk-id",
    "index": 0,
    "data": "partial-JSON-text"
}
```

サイズの大きいイメージを再構成するには、type が fragment の各レコードを chunk_id でバッファします。type が chunked のメインレコードを受け取ったら、chunked.before または chunked.after で参照される各 chunk_id に対して、total_fragments のフラグメントが揃うまで待機し、フラグメントを index の昇順にソートして、data 文字列を連結します。連結結果は、元の before または after オブジェクトを JSON テキストとして表したものです。これを解析して列の値にアクセスします。配信の整合性を検証するには、連結された文字列に対して CRC32C を計算し、結果を chunked.before.crc32c または chunked.after.crc32c と比較します。

データ型のシリアル化

次の表は、Aurora DSQL が CDC レコードの各 PostgreSQL データ型をシリアル化する方法を示しています。

整数タイプ

PostgreSQL タイプ	JSON 表現	例
smallint (int2)	JSON number	42
integer (int4)	JSON number	1001
bigint (int8)	JSON number	9223372036854775807
oid	JSON number (unsigned)	16384

JavaScript 環境では、bigint の値が ±2^53 を超えると精度が低下する可能性があります。このような場合は、BigInt または任意精度のライブラリを使用します。

浮動小数点型

PostgreSQL タイプ	JSON 表現	例	注意事項
`real` (`float4`)	JSON number	`3.14159`	NaN and ±Infinity are serialized as the strings `"NaN"`, `"Infinity"` , `"-Infinity"` .
`double precision` (`float8`)	JSON number	`3.141592653589793`	Same special value handling as `real`.
`numeric` / `decimal`	JSON string	`"123.45"`	Always a string to preserve exact precision. NaN and ±Infinity are serialized as the strings `"NaN"`, `"Infinity"` , `"-Infinity"` .

ブール値

PostgreSQL タイプ	JSON 表現	例
`boolean`	JSON boolean	`true` or `false`

文字型

PostgreSQL タイプ	JSON 表現	例
`varchar` / `####`	JSON string	`"Hello, world!"`
`bpchar` (`char(n)`)	JSON string	`"ABC"` (trailing spaces stripped)

PostgreSQL タイプ	JSON 表現	例
##	JSON string	"pg_class"
"char" (single-byte)	JSON string	#A#

バイナリ

PostgreSQL タイプ	JSON 表現	例
bytea	JSON string (Base64)	"SGVsbG8gV29ybGQh"

日付と時刻の型

PostgreSQL タイプ	JSON 表現	例	注意事項
date	JSON number (days since Unix epoch)	19,797	+infinity and -infinity are represented as sentinel day counts derived from epoch-offset arithmeti c. These values don't correspond to meaningful calendar dates.
##	JSON number (microseconds since midnight)	52200123456	
timetz	JSON number (microseconds since midnight, UTC)	52200123456	The local time is adjusted to UTC by applying the stored timezone offset (seconds west of

PostgreSQL タイプ	JSON 表現	例	注意事項
			UTC). The result is wrapped to the range [0, 86400000000) microseconds.
`timestamp`	JSON number (microseconds since Unix epoch)	`1710510600123456`	±Infinity maps to sentinel values: `9223372036825200000` for `+infinity` and `-9223372036832400000` for `-infinity` .
`timestamptz`	JSON number (microseconds since Unix epoch)	`1710510600123456`	Stored and emitted in UTC. Same ±infinity sentinel values as `timestamp` .

PostgreSQL タイプ	JSON 表現	例	注意事項
`interval`	JSON number (approximate total microseconds)	`2802603000000`	Months are approximated as 30.4375 days (2,629,800 seconds). The total is computed as `(# × 2,629,800 + # × 86,400) × 1,000,000 + ####`. If the result exceeds the 64-bit signed integer range (±9,223,372,036,854,775,807 microseconds, approximately ±292,271 years), Aurora DSQL emits JSON `null` for the column.

その他の型

PostgreSQL タイプ	JSON 表現	例
`uuid`	JSON string (standard 8-4-4-4-12 hex format)	`"550e8400-e29b-41d4-a716-446655440000"`
`oidvector`	JSON empty array	`[]`
`json`	JSON string containing the raw JSON text	`"{\"key\": \"value\"}"`

NULL 値

どのデータ型でも、NULL 列の値は JSON の `null` として表されます。

CDC レコードにおけるスキーマの進化

列の追加、削除、名前の変更などによってテーブルのスキーマを変更すると、CDC レコードには DDL の変更をコミットしたトランザクション以降の変更が反映されます。DDL の変更前にコミットされたトランザクションのレコードは、以前のスキーマを使用します。例えば、次のようになります。

- 列を追加すると、それ以前のトランザクションのレコードに新しい列が含まれなくなります。追加トランザクション以降のレコードには、新しい列が含まれます。
- 列を削除すると、削除されたトランザクション以降のレコードにその列が含まれなくなります。
- 列の名前を変更すると、名前変更トランザクション以降のレコードでは新しい列名が使用されます。

各レコードの `after` および `before` フィールドに存在する列名を調べて、ダウンストリームのコンシューマーのスキーマの変更を追跡します。各レコードの `source.version` フィールドは、CDC エンベロープ形式を識別します。

ストリームのモニタリング

> **Important**
>
> この機能は AWS プレビューとして提供されており、変更される可能性があります。詳細については、「AWS のサービス条件」のセクション 2、「ベータ版とプレビュー」を参照してください。CDC ストリームの料金の詳細については、「Aurora DSQL の料金ページ」を参照してください。
> 一般提供する前に、ストリームペイロードに新しいオペレーションタイプ ("op": "u" 更新用) を追加します。アプリケーションがこれらの変更を修正せずに処理できるようにするには、`after` ペイロードを適用して、認識されない op 値をすべてアップサートとして扱います。詳細については、「CDC レコードについて」を参照してください。

Aurora DSQL で CDC レコードの配信エラーが発生すると、ストリームは `IMPAIRED` ステータスに移行します。障害のあるストリームは引き続き他のレコードを処理して配信します。Aurora DSQL は失敗したレコードのみを再試行します。Aurora DSQL は、最も古い未配信レコードからのレプリケーション遅延を測定し、問題を解決するまで遅延は増加し続けます。Aurora DSQL は、未配信の変更を内部で 1 週間保持します。

このウィンドウ内に根本的な問題が解決されれば、次の再試行は成功し、エラー状態はクリアされ、ストリームは ACTIVE に戻ります。外部の問題 (IAM ポリシー、AWS KMS キー、Amazon Kinesis の容量など) を修正すると Aurora DSQL は自動的に再試行します。

レプリケーション遅延が失敗しきい値を超えると、ストリームは FAILED に移行します。

> **Important**
>
> 失敗したストリームは復旧できません。失敗したストリームを削除して、新しいストリームを作成する必要があります。

ストリームライフサイクル

ストリームは、ライフサイクル中に次のステータスに移行します。

- **CREATING** – Aurora DSQL はストリームをセットアップしています。Aurora DSQL はまだ CDC レコードを配信していません。
- **ACTIVE** – ストリームは動作し、CDC レコードをターゲットに配信します。
- **IMPAIRED** – ストリームで、アクションを必要とする問題が発生しました。Aurora DSQL は、失敗したレコードをエクスポネンシャルバックオフを使用して再試行しますが、他のレコードは引き続き配信できます。Aurora DSQL は、最も古い未配信レコードからのレプリケーション遅延を測定し、問題を解決するまで遅延は増加し続けます。Aurora DSQL は、未配信の変更を内部で 1 週間バッファします。「エラーコードのリファレンス」を参照してください。
- **FAILED** – ストリームで永続的なエラーが発生し、CDC レコードが配信されなくなりました。失敗したストリームは復旧できないため、削除する必要があります。ストリームがこの状態になる条件については、「エラーコードのリファレンス」を参照してください。
- **DELETING** – Aurora DSQL はストリームリソースを削除しています。
- **DELETED** – Aurora DSQL がストリームを削除しました。削除が完了すると、GetStream は ResourceNotFoundException を返します。

GetStream を呼び出して、いつでもストリームのステータスを表示できます。ストリームが IMPAIRED または FAILED の場合、レスポンスにはエラーコードとタイムスタンプを含む statusReason オブジェクトが含まれます。GetStream レスポンスフィールドの詳細については、「Amazon Aurora DSQL API リファレンス」の「GetStream」を参照してください。

ストリームの障害または失敗のトラブルシューティング

CDC ストリームに障害が発生したり、失敗した場合は、以下の手順に従います。ストリームが FAILED の場合、復元できません。ストリームを削除し、根本的な問題を解決して、新しいストリームを作成します。

1. ストリームのステータスを取得します。GetStream を呼び出して status フィールドを確認します。ステータスが ACTIVE の場合、ストリームは正常です。

```
aws dsql get-stream \
  --cluster-identifier cluster-id \
  --stream-identifier stream-id \
  --region region
```

2. エラーコードを確認します。ステータスが IMPAIRED または FAILED の場合、レスポンスには statusReason オブジェクトが含まれます。error フィールドにはエラーコードが含まれています。

```
{
    "status": "IMPAIRED",
    "statusReason": {
        "error": "KINESIS_THROUGHPUT_EXCEEDED",
        "updatedAt": "2025-01-15T14:30:00Z"
    }
}
```

3. 修復手順に従います。ストリームが IMPAIRED の場合、次の表のエラーコードを確認し、推奨される修正を適用します。根本的な問題が解決されると、Aurora DSQL は自動的に再試行します。ストリームが FAILED の場合は、ストリームを削除して、問題を解決してから、新しいストリームを作成します。

エラーコードのリファレンス

次の表は、各エラーコード、その原因、ストリームが復旧できるかどうか、および解決する手順を示しています。

エラーコード	原因	回復可能か?	解決方法
KINESIS_THROUGHPUT_EXCEEDED	Your Kinesis data stream exceeded its throughput limit, or AWS KMS throttled encryption operations on the Kinesis data stream, and the replication lag has grown.	Yes	Increase the number of shards on your Kinesis data stream, or switch to on-demand capacity mode. If the Kinesis data stream uses an AWS KMS customer managed key, verify that the key's request quota is large enough. After you increase capacity, Aurora DSQL retries automatically.
KINESIS_STREAM_NOT_FOUND	The target Kinesis data stream no longer exists.	No	The stream transitions directly to FAILED. Delete the CDC stream and create a new one pointing to a valid Kinesis data stream.
ROLE_ACCESS_DENIED	Aurora DSQL can't assume the IAM role specified in the target definition. The AWS STS AssumeRole call returned ######.	Yes	Verify the role's trust policy allows the Aurora DSQL service principal (dsql.amazonaws.com) to assume it. Verify the aws:SourceAccount and aws:SourceArn conditions match your cluster. For details,

エラーコード	原因	回復可能か?	解決方法
			see サービスロール信頼ポリシー. After you fix the trust policy, Aurora DSQL retries automatically.
KINESIS_ACCESS_DENIED	The assumed role doesn't have permission to write to the Kinesis data stream. Kinesis returned AccessDeniedException .	Yes	Add kinesis:PutRecord and kinesis:PutRecords permissions to the role's policy for the target Kinesis data stream Amazon Resource Name (ARN). After you fix the policy, Aurora DSQL retries automatically.

エラーコード	原因	回復可能か?	解決方法
`KINESIS_KMS_ACCESS_DENIED`	The assumed role doesn't have permission to use the AWS KMS key that encrypts the Kinesis data stream. This error covers AWS KMS access denial and invalid key states.	Yes	Verify the role has `kms:GenerateDataKey` permission on the AWS KMS key that the Kinesis data stream uses. Also verify that the AWS KMS key is in an enabled and valid state. This key is the encryption key on the Kinesis data stream, not the cluster's AWS KMS key. For details, see サービスロールのアクセス許可ポリシー. After you fix the permissions or key state, Aurora DSQL retries automatically.

エラーコード	原因	回復可能か?	解決方法
KINESIS_OVERSIZE_RECORD	A CDC record exceeded the maximum record size configured on the Kinesis data stream.	Yes	Increase MaxRecordSizeInKiB on the Kinesis data stream to 10240 (10 MiB). You can update this setting on an existing Kinesis data stream without deleting it. After you increase the limit, Aurora DSQL retries the oversized record automatically and the stream transitions back to ACTIVE.
CLUSTER_CMK_INACCESSIBLE	The AWS KMS customer managed key that encrypts the Aurora DSQL cluster is inaccessible.	Yes	Verify the AWS KMS key policy and key state. Re-enable or restore access to the key. After the key becomes accessible again, the stream transitions back to ACTIVE.

上記の表には、すべての StreamFailureErrorCode の値が一覧表示されています。statusReason レスポンスフィールドの詳細については、「Amazon Aurora DSQL API リファレンス」の「GetStream」を参照してください。

障害のあるストリームの復旧

ほとんどのエラーは、まずストリームを IMPAIRED に移行させます。障害のあるストリームは、引き続き他のレコードを処理し、失敗したレコードを自動的に再試行します。FAILED ストリームは復元できません。ストリームを削除して新しいストリームを作成する必要があります。

- 回復可能なエラーの場合: 外部の問題 (IAM ポリシー、AWS KMS キー、Kinesis の容量、または Kinesis レコードサイズ制限) を修正します。次の再試行が成功すると、エラー状態が解消され、ストリームは ACTIVE に戻ります。
- **KINESIS_STREAM_NOT_FOUND** の場合: ストリームは FAILED に直接移行します。失敗したストリームを削除し、有効な Kinesis データストリームを指す新しいストリームを作成します。

その他のすべてのエラーコードでは、問題を解決する前にレプリケーション遅延が失敗しきい値を超えると、ストリームは IMPAIRED から FAILED に移行します。失敗したストリームを ACTIVE に戻すことはできません。失敗したストリームを削除し、根本的な問題を解決してから、新しいストリームを作成します。

ストリームの状態のモニタリング

CloudWatch メトリクスと GetStream API を使用して、ストリームの状態をモニタリングします。CloudWatch メトリクスは、CDC パイプラインのパフォーマンスを継続的に可視化し、GetStream はストリームに障害が発生したり、失敗した場合に特定のエラーコードを提供します。

IsImpaired、BehindSourceLag、PublishedBytes、PublishedRecords を含む CDC メトリクスの完全なリストについては、「CDC ストリームの CloudWatch メトリクス」を参照してください。GetStream レスポンスフィールドの詳細については、「Amazon Aurora DSQL API リファレンス」の「GetStream」を参照してください。

CDC ストリームの CloudWatch メトリクス

次の CloudWatch メトリクスを使用して、各 CDC ストリームの状態とスループットをモニタリングします。Aurora DSQL は、これらのメトリクスを AWS/AuroraDSQL 名前空間に ClusterId と StreamId というディメンションとともに発行します。最後のメトリクスは、ダウンストリームの読み取り遅延を測定する AWS/Kinesis 名前空間内の標準の Amazon Kinesis メトリクスです。

> **Note**
>
> Aurora DSQL は、使用状況と請求の追跡のために、AWS/AuroraDSQL 名前空間に BytesStreamed および StreamDPU メトリクスも発行しています。説明については、「CDC ストリームメトリクス」を参照してください。

メトリクス名	有用な統計	説明
IsImpaired	Maximum	Indicates whether the stream is impaired. The value is 1 when the stream is in the IMPAIRED state, and 0 when the stream is healthy. Aurora DSQL emits this metric continuously for each active or impaired stream. Use this metric to create a CloudWatch alarm that notifies you when a stream becomes impaired.
BehindSourceLag	Average	The delay, in milliseconds, between when a transacti on commits in Aurora DSQL and when the CDC system processes the resulting record. A rising value indicates that the CDC pipeline is falling behind the write workload.
PublishedBytes	Sum	The total bytes of CDC records that Aurora DSQL wrote to the target during the period. Use this metric together with your Kinesis shard count to determine whether you've provisioned enough write capacity.
PublishedRecords	Sum	The total number of CDC records that Aurora DSQL wrote to the target during the period. Each committed row change produces one record.

メトリクス名	有用な統計	説明
GetRecords.IteratorAgeMilliseconds (AWS/Kinesis)	Average	A standard Kinesis metric that reports the age of the last record read from the Kinesis data stream by your downstream app, in milliseconds. Use the StreamName dimension. A rising value indicates that your downstream app can't keep up with the rate at which Aurora DSQL writes CDC records to Kinesis.

Aurora DSQL コンソールの [モニタリング] タブには、BehindSourceLag (CDC ソースのレイテンシー) と GetRecords.IteratorAgeMilliseconds (Kinesis リーダーのラグ) を組み合わせた [エンドツーエンドの平均レイテンシー] 値が表示されます。この結合値は、データベースへのコミットからダウンストリーム読み取りまでの遅延の合計を表します。

モニタリングのベストプラクティス

次のプラクティスを使用して、ダウンストリームシステムに影響を与える前に CDC パイプラインの問題を検出して解決します。

BehindSourceLag にアラームを設定する

BehindSourceLag がワークロードにとって重要なしきい値を超えたときに発生する CloudWatch アラームを作成します。例えば、1 分間のレイテンシー目標に対して 60 秒を設定します。このメトリクスの継続的な増加は、CDC パイプラインが遅れていることを意味します。遅延が失敗しきい値に達すると、ストリームは FAILED に移行します。トレンドをキャッチすることで、ストリームが劣化する前に Kinesis 容量を増やしたり、スループットのボトルネックを調査したりする時間を確保できます。

Kinesis 側で **GetRecords.IteratorAgeMilliseconds** をモニタリングする

Aurora DSQL がレコードを時間どおりに配信しても、ダウンストリームアプリケーションが遅延する可能性があります。GetRecords.IteratorAgeMilliseconds (AWS/Kinesis 名前空間のディメンション StreamName) で CloudWatch アラームを作成して、ダウンストリームの遅延を個別に検

出します。このメトリクスが増加して、BehindSourceLag が横ばいに保たれる場合、ボトルネックは Aurora DSQL ではなくダウンストリームアプリケーションにあります。

PublishedBytes を Kinesis シャード容量と比較して追跡する

各 Kinesis シャードは、書き込みに対して 1 秒あたり最大 1 MiB をサポートします。1 分あたりの PublishedBytes の合計を、シャード書き込み容量 (シャード数 × 60 MiB/分) の合計と比較します。使用量が 80% に近づいた場合は、スロットリングが KINESIS_THROUGHPUT_EXCEEDED をトリガーする前に、シャードを追加するか、オンデマンドキャパシティモードに切り替えます。

IsImpaired アラームで即時に障害を検出する

1 つの評価期間で IsImpaired の Maximum が 1 以上になったときに発生する CloudWatch アラームを作成します。これにより、API をポーリングしなくても、ストリームが IMPAIRED 状態になったときに、直接シグナルを受け取ることができます。アラームが発生したら、GetStream を呼び出して statusReason.error フィールドを読み取り、「ストリームの障害または失敗のトラブルシューティング」の修復ステップに従います。

GetStream をポーリングして詳細ステータスを確認する

IsImpaired メトリクスはストリームに障害があることを知らせますが、GetStream API は特定のエラーコードとタイムスタンプを提供します。GetStream をスケジュールに基づいて (5 分ごとなど) または IsImpaired アラームに応じてポーリングします。statusReason.error フィールドには、何が問題だったかが示されます。これを「ストリームの障害または失敗のトラブルシューティング」のトラブルシューティングステップと組み合わせて、迅速に解決します。

ダッシュボードを使用してメトリクスを関連付ける

IsImpaired、BehindSourceLag、PublishedRecords、PublishedBytes、および GetRecords.IteratorAgeMilliseconds を並べて表示する CloudWatch ダッシュボードを作成します。これらのメトリクスを相関させることで、CDC パイプラインの問題 (BehindSourceLag の上昇) とダウンストリームの読み取りの問題 (BehindSourceLag が安定していて IteratorAge が上昇) を区別できます。

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。

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
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	Unit	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.
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

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

CDC ストリームメトリクス

Aurora DSQL は、変更データキャプチャ (CDC) ストリームについて次のメトリクスを発行します。これらのメトリクスは `ClusterId` および `StreamId` ディメンションを使用するため、各 CDC ストリームを個別にモニタリングできます。CDC ストリームの詳細については、「変更データキャプチャ (CDC) ストリーム」を参照してください。

CloudWatch メトリクス名	メトリクス	単位	Description
IsImpaired	Is impaired	none	Indicates whether the stream is impaired. The value is 1 when the stream is in the `IMPAIRED` state, and `0` when the stream is healthy. Use this metric to create a CloudWatch alarm that notifies you when a stream becomes impaired.

CloudWatch メトリクス名	メトリクス	単位	Description
PublishedBytes	Published bytes	bytes	The total number of bytes that Aurora DSQL wrote to the target Kinesis data stream.
PublishedRecords	Published records	none	The number of CDC records that Aurora DSQL wrote to the target Kinesis data stream.
BehindSourceLag	Behind source lag	milliseconds	The delay, in milliseconds, between when a transaction commits in Aurora DSQL and when the CDC system processes the resulting record. A rising value indicates that the CDC pipeline is falling behind the write workload. If lag grows beyond the failure threshold, the stream transitions to `FAILED`.

CloudWatch メトリクス名	メトリクス	単位	Description
BytesStreamed	Bytes streamed	bytes	The total bytes streamed through the CDC pipeline for billing purposes. This metric reflects the data volume used to calculate streaming charges.
StreamDPU	Stream DPU	DPU	The Distributed Processing Units (DPU) consumed by the CDC stream. This metric reflects the processing cost of streaming change data.

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

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。

- CreateCluste:
- DeleteCluster
- GetCluster
- GetVpcEndpointServiceName
- ListClusters
- ListTagsForResource
- TagResource
- UntagResource

- UpdateCluster

CDC ストリームログ

Amazon Aurora DSQL は、次の CDC ストリームオペレーションを管理イベントとして CloudTrail に記録します。CDC ストリームの詳細については、「変更データキャプチャ (CDC) ストリーム」を参照してください。

- CreateStream
- DeleteStream
- GetStream
- ListStreams

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 がどのようにロ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 マネジメントコンソール 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 とお客様の間の共有責任です。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 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 のフォールトトレランス
- 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許可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アクセス許可を更新すると、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、次のことができます。

- マルチリージョンクラスターを含む Aurora DSQL クラスターの作成、削除、更新

- クラスターのインラインポリシーの管理 (ポリシーの作成、表示、更新、削除)
- クラスターの CDC ストリームを作成、更新、削除、管理する
- クラスターおよび CDC ストリームのタグの追加または削除
- クラスターの一覧表示と個別のクラスターに関する情報の表示
- Aurora DSQL クラスターにアタッチされたタグの表示
- 管理者を含む任意のユーザーでのデータベースへの接続
- ターゲット送信先へのデータ配信のために IAM ロールを CDC ストリームに渡す
- 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の実行
- クラスター暗号化へのカスタマーマネージド AWS KMS キーの使用
- アカウントでの CloudWatch メトリクス (任意) の表示
- AWS Fault Injection Service (AWS FIS) を使用した、フォールトトレランステストのための Aurora DSQL クラスターへの障害の挿入
- 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の作成

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` – サービスにリンクされたロールを作成し、データ配信のために CDC ストリームにロールを渡す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検証に必要なアクセス許可を付与します。
- `fis` - AWS Fault Injection Service (AWS FIS) を使用し、フォールトトレランステストのために Aurora DSQL クラスターに障害を挿入するアクセス許可を付与する。

`AmazonAuroraDSQLFullAccess` ポリシーは IAM コンソール、および「AWS Managed Policy Reference Guide」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 クラスターおよび CDC ストリームにアタッチされたタグを確認したり、個々の CDC ストリームに関する情報を表示したり、クラスターのインラインポリシーを表示したりできます。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。
- iam – データ配信設定のために CDC ストリームにロールを渡すアクセス許可を付与します。

AmazonAuroraDSQLReadOnlyAccess ポリシーは IAM コンソール、および「AWS Managed Policy Reference Guide」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 マネジメントコンソール経由で 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、次のことができます。

- マルチリージョンクラスターを含む Aurora DSQL クラスターのコンソールでの作成、削除、更新
- コンソールからのクラスターインラインポリシーの管理 (ポリシーの作成、表示、更新、削除)
- クラスターの一覧表示と個別のクラスターに関する情報の表示
- アカウントの任意のリソースのタグの表示
- 管理者を含む任意のユーザーでのデータベースへの接続

- 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の実行
- クラスター暗号化へのカスタマーマネージド AWS KMS キーの使用
- AWS マネジメントコンソール からの AWS CloudShell の起動
- アカウントの CloudWatch の任意のメトリクスの表示
- AWS Fault Injection Service (AWS FIS) を使用した、フォールトトレランステストのための Aurora DSQL クラスターへの障害の挿入
- 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の作成

`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 マネジメントコンソール 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検証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。
- `fis` - AWS FIS を使用し、フォールトトレランステストのために Aurora DSQL クラスターに障害を挿入するアクセス許可を付与する。

- `access-analyzer:ValidatePolicy` は、ポリシーエディタのリンターのアクセス許可を付与し、これにより、現在のポリシーのエラー、警告、セキュリティの問題に関するリアルタイムのフィードバックが提供されます。
- `fis` - AWS Fault Injection Service (AWS FIS) を使用し、フォールトトレランステストのために Aurora DSQL クラスターに障害を挿入するアクセス許可を付与する。

`AmazonAuroraDSQLConsoleFullAccess` ポリシーは IAM コンソール、および「AWS Managed Policy Reference Guide」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 および AmazonAuroraDSQLConsoleFullAccess の更新	Aurora DSQL との AWS Fault Injection Service (AWS FIS) 統合のサポートを追加しました。これにより、単一リージョンおよびマルチリージョンの Aurora DSQL クラスターに障害を挿入して、アプリケーションのフォールトトレラ	2025 年 8 月 19 日

変更	説明	日付
	ンスをテストできます。AWS FIS コンソールで実験テンプレートを作成して障害シナリオを定義し、テスト用に特定の Aurora DSQL クラスターをターゲットにできます。 これらのポリシーの詳細については、「AmazonAuroraDSQLFullAccess」および「AmazonAuroraDSQLConsoleFullAccess」を参照してください。	
AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess、AmazonAuroraDSQLConsoleFullAccess update	`PutClusterPolicy` 、`GetClusterPolicy` 、`DeleteClusterPolicy` の新しいアクセス許可を持つリソースベースのポリシー (RBP) サポートを追加しました。これらのアクセス許可により、Aurora DSQL クラスターにアタッチされたインラインポリシーを管理し、きめ細かなアクセスコントロールを行うことができます。 詳細については、「AmazonAuroraDSQLFullAccess」、「AmazonAuroraDSQLReadOnlyAccess」、「AmazonAuroraDSQLConsoleFullAccess」を参照してください。	2025 年 10 月 15 日

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

変更	説明	日付
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「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

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 ネームスペース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」ブログ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 アイデンティティセンター 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 は必須ですが、TLS 1.3 を推奨します。
- 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。

FIPS 準拠

Aurora DSQL データプレーンエンドポイント (データベース接続に使用されるクラスターエンドポイント) は、デフォルトで FIPS 140-2 検証済み暗号化モジュールを使用します。クラスター接続に個別の FIPS エンドポイントは必要ありません。

コントロールプレーンオペレーションの場合、Aurora DSQL はサポートされているリージョン専用の FIPS エンドポイントを提供します。コントロールプレーン FIPS エンドポイントの詳細については、「AWS 全般のリファレンス」の「Aurora DSQL endpoints and quotas」を参照してください。

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。
- Direct Connect 接続。詳細については、「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。クライアントは次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。

監視リージョンでのデータ保護

マルチリージョンクラスターを作成すると、監視リージョンは、暗号化されたトランザクションの同期レプリケーションに参加することで、自動障害復旧を有効にするのに役立ちます。ピア接続されたクラスターが使用できなくなっても、監視リージョンは引き続きデータベースの書き込みを検証して処理できるため、可用性が失われることはありません。

監視リージョンは、次の設計機能を通じてデータを保護します。

- 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受信して保存します。暗号化キーをホスト、保存、または送信することはありません。
- 監視リージョンは、書き込みトランザクションのログ記録とクォーラム機能のみを行います。設計上、データを読み取ることはできません。
- 監視リージョンは、クラスター接続エンドポイントまたはクエリプロセッサなしで動作します。これにより、ユーザーデータベースへのアクセスが防止されます。

監視リージョンの詳細については、「マルチリージョンクラスターの設定」を参照してください。

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 ルート認証局 (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局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、次の動作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
```

その他のリソース

- 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、すべての静止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厳格な規制要件を満たす
- 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厳格な規制要件. 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。

カスタマー管理のキー

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) を使用して静止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。キーアクセスをリストアすると、サービスは 15 分以内にリストア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、クラスターのリストア時間は、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、次のメリットがあります。

- 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。

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 マネジメントコンソールまたは 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 マネジメントコンソール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 マネジメントコンソールまたは 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 マネジメントコンソール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。
  - 静止データ - Aurora DSQL はパーシステント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 マネジメントコンソールと 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 マネジメントコンソール、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

AWS Identity and Access Management (IAM) の使用方法は、ロールによって異なります。

- サービスユーザー - 機能にアクセスできない場合は、管理者にアクセス許可をリクエストします (「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)。
- サービス管理者 - ユーザーアクセスを決定し、アクセス許可リクエストを送信します (「Amazon Aurora DSQL と IAM の連携」を参照)
- IAM 管理者 - アクセスを管理するためのポリシーを作成します (「Amazon Aurora DSQL のアイデンティティベースのポリシー例」を参照)

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、IAM ユーザー の AWS アカウントのルートユーザー 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 必要があります。

AWS IAM アイデンティティセンター (IAM アイデンティティセンター)、シングルサインオン認証、Google/Facebook 認証情報などの ID ソースからの認証情報を使用して、フェデレーテッドアイ

デンティティとしてサインインできます。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。

プログラムによるアクセスの場合、AWS はリクエストに暗号で署名するための SDK と CLI を提供します。詳細については、「IAM ユーザーガイド」の「API リクエストに対する AWS 署名バージョン 4」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と、すべての AWS のサービスとリソースに対する完全なアクセス権を持つ AWS アカウント ルートユーザーと呼ばれる 1 つのサインイン ID を使用して開始します。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。ルートユーザー認証情報を必要とするタスクについては、「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。

フェデレーテッドアイデンティティ

ベストプラクティスでは、人間のユーザーが一時的な認証情報を使用して AWS のサービス にアクセスする際、アイデンティティプロバイダーとのフェデレーションを使用することが求められます。

フェデレーテッドアイデンティティは、エンタープライズディレクトリ、ウェブ ID プロバイダー、Directory Service のユーザーであり、ID ソースからの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。フェデレーテッドアイデンティティは、一時的な認証情報を提供するロールを引き受けます。

アクセスを一元管理する場合は、AWS IAM アイデンティティセンター をお勧めします。詳細については、「AWS IAM アイデンティティセンター ユーザーガイド」の「IAM アイデンティティセンターとは」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、特定の個人やアプリケーションに対する特定のアクセス許可を持つアイデンティティです。長期認証情報を持つ IAM ユーザーの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。詳細は「IAM ユーザーガイド」の「人間のユーザーが一時的な認証情報を使用して AWS にアクセスするには ID プロバイダーとのフェデレーションの使用が必要です」を参照してください。

IAM グループは、IAM ユーザーの集合を指定し、大量のユーザーに対するアクセス許可の管理を容易にします。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、特定のアクセス許可を持つアイデンティであり、一時的な認証情報を提供します。ユーザーから IAM ロール (コンソール) に切り替える、または AWS CLI や AWS API オペレーションを呼び出すことで、ロールを引き受けることができます。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 ロールは、フェデレーションユーザーアクセス、一時的な IAM ユーザーのアクセス許可、クロスアカウントアクセス、クロスサービスアクセス、および Amazon EC2 で実行するアプリケーションに役立ちます。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。

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、アイデンティティやリソースとの関連付けに伴うアクセス許可を定義します。AWS は、プリンシパルがリクエストを行うときに、これらのポリシーを評価します。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、ポリシーを使用して、どのプリンシパルがどのリソースに対して、どのような条件でアクションを実行できるかを定義することで、誰が何にアクセスできるかを指定します。

デフォルトでは、ユーザーやロールにアクセス許可はありません。IAM 管理者は IAM ポリシーを作成してロールに追加し、このロールをユーザーが引き受けられるようにします。IAM ポリシーは、オペレーションの実行方法を問わず、アクセス許可を定義します。

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のポリシーは、アイデンティティ (ユーザー、グループ、またはロール) にアタッチできる JSON アクセス許可ポリシードキュメントです。これらのポリシーは、アイデンティティがどのリソースに対してどのような条件下でどのようなアクションを実行できるかを制御します。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、インラインポリシー (単一の ID に直接埋め込む) または管理ポリシー (複数の ID にアタッチされたスタンドアロンポリシー) にすることができます。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。例としては、IAM ロール信頼ポリシーや Amazon S3 バケットポリシーなどがあります。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。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。

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 IAM の AWS マネージドポリシーを使用することはできません。

その他のポリシータイプ

AWS は、より一般的なポリシータイプで付与された最大数のアクセス許可を設定できる、追加のポリシータイプをサポートしています。

- アクセス許可の境界 – アイデンティティベースのポリシーで IAM エンティティに付与することのできるアクセス許可の数の上限を設定します。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) - AWS Organizations 内の組織または組織単位の最大のアクセス許可を指定します。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」を参照してください。
- リソースコントロールポリシー (RCP) – は、アカウント内のリソースで利用できる最大数のアクセス許可を定義します。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 – ロールまたはフェデレーションユーザーの一時セッションを作成する際にパラメータとして渡される高度なポリシーです。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アクセス許可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 機能	Aurora DSQL のサポート
アイデンティティベースのポリシー	あり
リソースベースのポリシー	はい
ポリシーアクション	あり
ポリシーリソース	あり
ポリシー条件キー	あり
ACL	なし
ABAC (ポリシー内のタグ)	あり
一時的な認証情報	あり
プリンシパルアクセス権限	あり
サービスロール	あり
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、ユーザーグループ、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。

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。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: あり

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コントロールできます。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、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、フェデレーティッドユーザー、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 IAM の AWS マネージドポリシーを使用することはできません。

Aurora DSQL クラスターのリソースベースのポリシーを作成および管理する方法については、「Resource-based policies for Aurora DSQL」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ようなリソースにどのような条件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。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。

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。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。リソースレベルのアクセス許可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` 要素は、定義された基準に基づいてステートメントが実行される時期を指定します。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。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アクセス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、タグと呼ばれる属性に基づいてアクセス許可を定義する認可戦略です。IAM エンティティと AWS リソースにタグを付けることで、プリンシパルのタグがリソースタグと一致するときに操作を許可する ABAC ポリシーを設計できます。

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

一時的な認証情報は、AWSリソースへの短期的なアクセスを提供し、フェデレーションの使用時またはロールの切り替え時に自動的に作成されます。AWS では、長期的なアクセスキーを使用する代わりに、一時的な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 ユーザーガイド」の「IAM の一時的な認証情報」および「AWS のサービス と IAM との連携」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

転送アクセスセッション (FAS) は、AWS のサービス を呼び出すプリンシパルのアクセス許可を 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。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。

これらのサンプルの 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
- クラスター管理とデータベース接続を許可する

- タグに基づく Aurora DSQL リソースアクセス

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では、AWS アカウント に費用が発生する場合があります。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の使用を開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードへのアクセス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アクセス許可を付与する 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。ユースケースに固有の AWS 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。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 ポリシーでアクセス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。たとえば、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、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 アクセスアナライザー 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 アクセスアナライザー 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アクセス許可が含まれています。

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

クラスター管理とデータベース接続を許可する

次のポリシーは、特定の Aurora DSQL クラスターを管理および接続するアクセス許可を IAM ユーザーに付与します。このポリシーは、クラスター管理アクションと接続アクションを単一のクラスター Amazon リソースネーム (ARN) にスコープする一方で、`dsql:ListClusters` アクションについては、リソースレベルのアクセス許可をサポートしていないため、すべてのリソースに対してこのアクションを許可します。

この例では、`dsql:DbConnectAdmin` を使用して `admin` ロールに接続します。代わりにカスタムデータベースロールに接続するには、`dsql:DbConnectAdmin` を `dsql:DbConnect` に置き換えます。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。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowClusterManagement",
            "Effect": "Allow",
            "Action": [
                "dsql:GetCluster",
```

```
                "dsql:UpdateCluster",
                "dsql:DeleteCluster",
                "dsql:DbConnectAdmin",
                "dsql:TagResource",
                "dsql:ListTagsForResource",
                "dsql:UntagResource"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/my-cluster-id"
        },
        {
            "Sid": "AllowListClusters",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        }
    ]
}
```

タグに基づく Aurora DSQL リソースアクセス

アイデンティティベースのポリシーの条件を使用することで、タグに基づいて Aurora DSQL リソースへのアクセスを制御できます。次の例では、クラスターの表示を許可するポリシーの作成方法を示します。ただし、このポリシーがアクセス許可を付与するのは、クラスターのタグ `Owner` にそのユーザーのユーザー名の値がある場合に限ります。このポリシーでは、このアクションをコンソールで実行するために必要なアクセス許可も付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListClustersInConsole",
            "Effect": "Allow",
            "Action": "dsql:ListClusters",
            "Resource": "*"
        },
        {
            "Sid": "ViewClusterIfOwner",
            "Effect": "Allow",
            "Action": "dsql:GetCluster",
```

```
            "Resource": "arn:aws:dsql:*:*:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

このポリシーをアカウントの IAM ユーザーにアタッチできます。richard-roe という名前のユーザーが Aurora DSQL クラスターを表示する場合、クラスターには Owner=richard-roe または owner=richard-roe のタグを付ける必要があります。それ以外の場合、IAM はアクセスを拒否します。条件キー名では大文字と小文字が区別されないため、条件タグキー Owner は Owner と owner の両方に一致します。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。

次のポリシーでは、クラスターに独自のユーザー名を Owner としてタグ付けする場合にのみ、クラスターの作成をユーザーに許可します。また、ユーザーが既に所有しているクラスターに対してのみタグ付けを許可します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateTaggedCluster",
            "Effect": "Allow",
            "Action": "dsql:CreateCluster",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "${aws:username}"
                }
            }
        },
        {
            "Sid": "AllowTagOwnedClusters",
            "Effect": "Allow",
```

```
            "Action": "dsql:TagResource",
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "${aws:username}"
                }
            }
        }
    ]
}
```

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や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できます。そのためには、サービスにロールを渡すアクセス許可が必要です。

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アクセス許可が必要です。Mary には、ロールをサービスに渡すアクセス許可がありません。

```
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 のリソースベースのポリシー

Aurora DSQL のリソースベースのポリシーを使用し、クラスターリソースに直接アタッチする JSON ポリシードキュメントを通じてクラスターへのアクセスを制限または許可します。これらのポリシーは、クラスターにアクセスできるユーザーとその条件をきめ細かく制御します。

Aurora DSQL クラスターは、デフォルトではパブリックインターネットからアクセスでき、IAM 認証をプライマリセキュリティコントロールとして使用します。リソースベースのポリシーを使用すると、特にパブリックインターネットからのアクセスをブロックするためのアクセス制限を追加できます。

リソースベースのポリシーは、IAM アイデンティティベースのポリシーと連携して機能します。AWS は、両方のタイプのポリシーを評価して、クラスターへのアクセスリクエストの最終アクセス許可を決定します。デフォルトでは、Aurora DSQL クラスターはアカウント内でアクセス可能です。IAM ユーザーまたはロールに Aurora DSQL のアクセス許可がある場合、リソースベースのポリシーがアタッチされていないクラスターにアクセスできます。

> **Note**
>
> リソースベースのポリシーへの変更は結果的に整合性があり、通常は 1 分以内に有効になります。

アイデンティティベース (IAM) のポリシーおよびリソースポリシーの間の詳細な相違点については、「IAM ユーザーガイド」の「アイデンティティベースおよびリソースベースのポリシー」を参照してください。

> **Warning**
>
> クラスターを所有するアカウントの AWS アカウントルートユーザーは、VPC ソースの制限を含むリソースベースのポリシー条件に関係なく、常に自身のリソースへのフルアクセスを保持します。リソースベースのポリシーは、ルートユーザーのアクセスを制限しません。クラスターへのルートユーザーのアクセスを制限するには、IAM アイデンティティベースのポリシーを使用するか、データベース接続にルート認証情報を使用しないようにします。詳細については、「AWS アカウントのルートユーザーのベストプラクティス」を参照してください。

どのようなときにリソースベースのポリシーを使うか

リソースベースのポリシーは、以下のシナリオで特に役立ちます。

- ネットワークベースのアクセスコントロール - リクエストの送信元の VPC または IP アドレスに基づいてアクセスを制限するか、パブリックインターネットのアクセスをすべてブロックします。`aws:SourceVpc` や `aws:SourceIp` などの条件キーを使用して、ネットワークアクセスを制御します。
- 複数のチームまたはアプリケーション - 複数のチームまたはアプリケーションに同じクラスターへのアクセスを許可します。各プリンシパルの個々の IAM ポリシーを管理するのではなく、1 回でクラスターのアクセスルールを定義します。
- 複雑な条件付きアクセス - ネットワーク属性、リクエストコンテキスト、ユーザー属性などの複数の要因に基づいてアクセスを制御します。複数の条件を 1 つのポリシーに組み合わせることができます。
- 集中型セキュリティガバナンス - クラスター所有者は、既存のセキュリティプラクティスと統合された使い慣れた AWS ポリシー構文を使用してアクセスを制御できます。

> **Note**
>
> クロスアカウントアクセスは、Aurora DSQL のリソースベースのポリシーではまだサポートされていませんが、今後のリリースで追加されます。

誰かが Aurora DSQL クラスターに接続しようとすると、AWS は認可コンテキストの一部としてリソースベースのポリシーと、関連する IAM ポリシーを評価して、リクエストを許可または拒否するかどうかを判断します。

リソースベースのポリシーは、クラスターと同じ AWS アカウント内のプリンシパルにアクセスを許可できます。マルチリージョンクラスターの場合、各リージョンクラスターに独自のリソースベースのポリシーがあり、必要に応じてリージョン固有のアクセスコントロールが可能になります。

> **Note**
>
> 条件コンテキストキーは、リージョン (VPC ID など) によって異なる場合があります。

トピック

リソースベースのポリシーを使用したクラスターの作成

新しいクラスターを作成するときにリソースベースのポリシーをアタッチして、最初から確実にアクセス制御を実行できます。各クラスターには、クラスターに直接アタッチされた 1 つのインラインポリシーを設定できます。

AWS マネジメントコンソール

クラスターの作成中にリソースベースのポリシーを追加するには

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. [クラスターを作成] を選択します。

3. 必要に応じて、クラスター名、タグ、マルチリージョン設定を設定します。
4. [クラスター設定] セクションで、[リソースベースのポリシー] オプションを見つけます。
5. [リソースベースのポリシーを追加] を有効にします。
6. JSON エディタで、ポリシードキュメントを入力します。例えば、パブリックインターネットアクセスをブロックするには、次のとおりにします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

7. [ステートメントを編集] または [新しいステートメントを追加] を使用してポリシーを作成できます。
8. 残りのクラスター設定を完了し、[クラスターを作成] を選択します。

AWS CLI

クラスターを作成するときに `--policy` パラメータを使用して、インラインポリシーをアタッチします。

```
aws dsql create-cluster --policy '{
    "Version": "2012-10-17",
    "Statement": [{
```

```
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
        }
    }]
}'
```

AWS SDK

Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
        }
    }]
}

response = client.create_cluster(
    policy=json.dumps(policy)
)

print(f"Cluster created: {response['identifier']}")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;
```

```
DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
    }
  }]
}
""";

CreateClusterRequest request = CreateClusterRequest.builder()
    .policy(policy)
    .build();

CreateClusterResponse response = client.createCluster(request);
System.out.println("Cluster created: " + response.identifier());
```

クラスターのリソースベースのポリシーの追加と編集

AWS マネジメントコンソール

リソースベースのポリシーを既存のクラスターに追加するには

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. [クラスターリスト] からクラスターを選択して、クラスターの詳細ページを開きます。
3. [アクセス許可] タブを選択します。
4. [リソースベースのポリシー] セクションで、[ポリシーを追加] を選択します。
5. JSON エディタで、ポリシードキュメントを入力します。[ステートメントを編集] または [新しいステートメントを追加] を使用してポリシーを作成できます。
6. [Add policy] (ポリシーの追加) を選択します。

既存のリソースベースのポリシーを編集するには

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. [クラスターリスト] からクラスターを選択して、クラスターの詳細ページを開きます。
3. [アクセス許可] タブを選択します。
4. [リソースベースのポリシー] セクションで、[編集]を選択します。
5. JSON エディタでポリシードキュメントを変更します。[ステートメントを編集] または [新しいステートメントを追加] を使用して、ポリシーを更新できます。
6. [Save changes] (変更の保存) をクリックします。

AWS CLI

`put-cluster-policy` コマンドを使用して、新しいポリシーをアタッチするか、クラスターの既存のポリシーを更新します。

```
aws dsql put-cluster-policy --identifier your_cluster_id --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": { "aws:SourceVpc": "true" }
        }
    }]
}'
```

AWS SDK

Python

```
import boto3
import json

client = boto3.client('dsql')
```

```
policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
    }]
}

response = client.put_cluster_policy(
    identifier='your_cluster_id',
    policy=json.dumps(policy)
)
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.PutClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "Null": {"aws:SourceVpc": "true"}
    }
  }]
}
""";

PutClusterPolicyRequest request = PutClusterPolicyRequest.builder()
```

```
    .identifier("your_cluster_id")
    .policy(policy)
    .build();

client.putClusterPolicy(request);
```

リソースベースのポリシーの表示

クラスターにアタッチされたリソースベースのポリシーを表示して、現在のアクセスコントロールを理解できます。

AWS マネジメントコンソール

リソースベースのポリシーを表示するには

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. [クラスターリスト] からクラスターを選択して、クラスターの詳細ページを開きます。
3. [アクセス許可] タブを選択します。
4. [リソースベースのポリシー] セクションでアタッチされたポリシーを表示します。

AWS CLI

`get-cluster-policy` コマンドを使用して、クラスターのリソースベースのポリシーを表示します。

```
aws dsql get-cluster-policy --identifier your_cluster_id
```

AWS SDK

Python

```
import boto3
import json

client = boto3.client('dsql')
```

```
response = client.get_cluster_policy(
    identifier='your_cluster_id'
)

# Parse and pretty-print the policy
policy = json.loads(response['policy'])
print(json.dumps(policy, indent=2))
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyResponse;

DsqlClient client = DsqlClient.create();

GetClusterPolicyRequest request = GetClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

GetClusterPolicyResponse response = client.getClusterPolicy(request);
System.out.println("Policy: " + response.policy());
```

リソースベースのポリシーの削除

クラスターからリソースベースのポリシーを削除して、アクセスコントロールを変更できます。

> **Important**
>
> クラスターからすべてのリソースベースのポリシーを削除すると、アクセスは IAM アイデンティティベースのポリシーによって完全に制御されます。

AWS マネジメントコンソール

リソースベースのポリシーを削除するには

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. [クラスターリスト] からクラスターを選択して、クラスターの詳細ページを開きます。
3. [アクセス許可] タブを選択します。
4. [リソースベースのポリシー] セクションで、[削除] を選択します。
5. 確認ダイアログで、**confirm** と入力して削除を確認します。
6. [削除] を選択します。

AWS CLI

`delete-cluster-policy` コマンドを使用して、クラスターからポリシーを削除します。

```
aws dsql delete-cluster-policy --identifier your_cluster_id
```

AWS SDK

Python

```
import boto3

client = boto3.client('dsql')

response = client.delete_cluster_policy(
    identifier='your_cluster_id'
)

print("Policy deleted successfully")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterPolicyRequest;

DsqlClient client = DsqlClient.create();
```

```
DeleteClusterPolicyRequest request = DeleteClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

client.deleteClusterPolicy(request);
System.out.println("Policy deleted successfully");
```

一般的なリソースベースのポリシーの例

これらの例は、Aurora DSQL クラスターへのアクセスを制御する一般的なパターンを示しています。これらのパターンを組み合わせて変更し、特定のアクセス要件を満たすことができます。

パブリックインターネットアクセスをブロックする

このポリシーは、パブリックインターネット (非 VPC) から Aurora DSQL クラスターへの接続をブロックします。このポリシーでは、ユーザーが接続できる VPC は指定されず、VPC から接続する必要があるだけです。特定の VPC へのアクセスを制限するには、`aws:SourceVpc` と `StringEquals` 条件演算子を使用します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

> **Note**
>
> この例では aws:SourceVpc のみを使用して VPC 接続をチェックします。aws:VpcSourceIp および aws:SourceVpce 条件キーによりさらにきめ細かくできますが、基本的な VPC のみのアクセスコントロールには必要ありません。

特定のロールに対する例外を指定するには、代わりにこのポリシーを使用します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyAccessFromOutsideVPC",
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        },
        "StringNotEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:role/ExceptionRole",
            "arn:aws:iam::123456789012:role/AnotherExceptionRole"
          ]
        }
      }
    }
  ]
}
```

AWS 組織へのアクセスを制限する

このポリシーは、AWS 組織内のプリンシパルへのアクセスを制限します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalOrgID": "o-exampleorgid"
        }
      }
    }
  ]
}
```

特定の組織単位へのアクセスを制限する

このポリシーは、AWS 組織の特定の組織単位 (OU) 内のプリンシパルへのアクセスを制限し、組織全体のアクセスよりもきめ細かい制御を提供します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotLike": {
```

```
          "aws:PrincipalOrgPaths": "o-exampleorgid/r-examplerootid/ou-exampleouid/*"
        }
      }
    }
  ]
}
```

マルチリージョンクラスターポリシー

マルチリージョンクラスターの場合、各リージョンクラスターは独自のリソースポリシーを維持し、リージョン固有の制御が可能です。リージョンごとにポリシーが異なる例を示します。

us-east-1 ポリシー:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-east1-id"
        },
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

us-east-2 ポリシー:

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": "vpc-east2-id"
        }
      }
    }
  ]
}
```

> **Note**
>
> 条件コンテキストキーは AWS リージョン (VPC ID など) によって異なる場合があります。

Aurora DSQL のリソースベースのポリシーでパブリックアクセスをブロックする

ブロックパブリックアクセス (BPA) は、AWS アカウント全体で Aurora DSQL クラスターへのパブリックアクセスを許可するリソースベースのポリシーを特定し、アタッチされないように防ぐ機能です。BPA を使用すると、Aurora DSQL リソースへのパブリックアクセスを阻止できます。BPA はリソースベースのポリシーの作成時や変更時にチェックを行い、Aurora DSQL のセキュリティ体制の改善に貢献します。

BPA は自動推論を活用して、リソースベースのポリシーで許可されるアクセスを分析し、該当するアクセス許可がリソースベースのポリシーの管理時に見つかった場合は警告します。分析では、リソースベースのポリシーのステートメント、アクション、ポリシーで使用されている条件キーセットをすべてまたいでアクセスを検証します。

> **Important**
>
> BPA は、Aurora DSQL リソース (クラスターなど) に直接アタッチされたリソースベースのポリシーによってパブリックアクセスが許可されることがないように防いで、リソースを保護します。BPA を利用したうえでさらに、次のポリシーを注意深く調べて、パブリックアクセスが許可されていないことを確かめてください。
>
> - 関連する AWS プリンシパル (IAM ロールなど) にアタッチされているアイデンティティベースのポリシー
> - 関連する AWS リソース (AWS Key Management Service (KMS) キーなど) にアタッチされているリソースベースのポリシー

プリンシパルに * エントリを含めないでください。または、指定された条件キーのいずれかでプリンシパルからリソースへのアクセスが制限されていることを確認する必要があります。リソースベースのポリシーが AWS アカウント間のクラスターへのパブリックアクセスを許可している場合、ポリシー内の指定内容が修正され、非パブリックと判断されない限り、Aurora DSQL はそのポリシーの作成または変更を阻止します。

Principal ブロック内に 1 つ以上のプリンシパルを指定することで、ポリシーを非パブリックにすることができます。次のリソースベースのポリシーの例では、2 つのプリンシパルを指定することで、パブリックアクセスをブロックしています。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id"
}
```

特定の条件キーを指定してアクセスを制限するポリシーも、パブリックとは見なされません。リソースベースのポリシーで指定されているプリンシパルを評価するほかに、以下の信頼できる条件キーも、リソースベースのポリシーが付与するアクセスが非パブリックであるという評価の補足に使用できます。

- aws:PrincipalAccount
- aws:PrincipalArn
- aws:PrincipalOrgID
- aws:PrincipalOrgPaths
- aws:SourceAccount
- aws:SourceArn
- aws:SourceVpc
- aws:SourceVpce
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- aws:Ec2InstanceSourceVpc
- aws:SourceOrgID
- aws:SourceOrgPaths

さらに、リソースベースのポリシーが非パブリックとなるには、Amazon リソースネーム (ARN) と文字列キーの値にワイルドカードや変数が含まれていてはいけません。リソースベースのポリシーで aws:PrincipalIsAWSService キーが使用されている場合は、キー値を true に設定する必要があります。

次のポリシーでは、指定されたアカウントのユーザー Ben にアクセスが限定されています。この条件により Principal が制限されるため、パブリックとは見なされません。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/Ben"
    }
  }
```

```
}
```

次の例は非パブリックのリソースベースのポリシーです。StringEquals 演算子を使用して sourceVPC を制限しています。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dsql:*",
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
      }
    }
  ]
}
```

Aurora DSQL API オペレーションとリソースベースのポリシー

Aurora DSQL のリソースベースのポリシーは、特定の API オペレーションへのアクセスを制御します。以下のセクションでは、すべての Aurora DSQL API オペレーションをカテゴリ別に分類し、どのオペレーションがリソースベースのポリシーをサポートしているかを示します。

RBP をサポート列は、ポリシーがクラスターにアタッチされたときに API オペレーションがリソースベースのポリシー評価の対象となるかどうかを示します。

タグ付け API

API 操作	説明	RBP をサポート
ListTagsForResource	Aurora DSQL リソースのタグを一覧表示します	はい

API 操作	説明	RBP をサポート
TagResource	Aurora DSQL リソースにタグを追加します	はい
UntagResource	Aurora DSQL リソースからタグを削除します	はい

クラスター管理 API

API 操作	説明	RBP をサポート
CreateCluste:	新しいクラスターを作成します。	いいえ
DeleteCluster	クラスターを削除する	はい
GetCluster	クラスターに関する情報を取得します	はい
GetVpcEndpointServiceName	クラスターの VPC エンドポイントサービス名を取得します	はい
ListClusters	アカウントのクラスターを一覧表示します	いいえ
UpdateCluster	クラスターの設定を更新します	はい

マルチリージョンプロパティ API

API 操作	説明	RBP をサポート
AddPeerCluster	マルチリージョン設定にピアクラスターを追加します	はい
PutMultiRegionProperties	クラスターのマルチリージョンプロパティを設定します	はい
PutWitnessRegion	マルチリージョンクラスターの監視リージョンを設定します	はい

リソースベースのポリシーの API

API 操作	説明	RBP をサポート
DeleteClusterPolicy	クラスターからリソースベースのポリシーを削除します	はい
GetClusterPolicy	クラスターのリソースベースのポリシーを取得します	はい
PutClusterPolicy	クラスターのリソースベースのポリシーを作成または更新します	はい

AWS Fault Injection Service API

API 操作	説明	RBP をサポート
InjectError	フォールトインジェクションテストのエラーを挿入します	いいえ

バックアップおよび復元 API

API 操作	説明	RBP をサポート
GetBackupJob	バックアップジョブに関する情報を取得します	いいえ
GetRestoreJob	復元ジョブに関する情報を取得します	いいえ
StartBackupJob	クラスターのバックアップジョブを開始します	はい
StartRestoreJob	バックアップから復元ジョブを開始します	いいえ
StopBackupJob	実行中のバックアップジョブを停止します	いいえ
StopRestoreJob	実行中の復元ジョブを停止します	いいえ

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」を参照し、[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**
>
> サービスリンクロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) 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。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "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。

JSON

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

```
            "Action": ["dsql:CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

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。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "dsql:CreateCluster",
          "dsql:PutMultiRegionProperties",
          "dsql:PutWitnessRegion",
          "dsql:AddPeerCluster"
        ],
        "Resource": [
           "arn:aws:dsql:us-east-1:123456789012:cluster/*",
           "arn:aws:dsql:us-east-2:123456789012:cluster/*"
        ]
      }
    ]
```

```
}
```

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。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dsql:CreateCluster",
                "dsql:PutMultiRegionProperties",
                "dsql:AddPeerCluster"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dsql:PutWitnessRegion"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": [
                        "us-west-2"
                    ]
                }
            }
        }
    ]
}
```

Amazon Aurora DSQL 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」を参照して、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お客様のコンプライアンス責任は、お客様のデータの機密性や貴社のコンプライアンス目的、適用可能な法律および規制によって決定されます。AWS のサービス を使

用する際のコンプライアンス責任の詳細については、「AWS セキュリティドキュメント」を参照してください。

Amazon Aurora DSQL のフォールトトレランス

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ー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、フォールトトレランス、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 のフォールトトレランスのすべてのメリットを提供しながら、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 (AZ) 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、データのフォールトトレランス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。ウィットネスリージョンに障害が発生した場合、クラスターの可用性には影響しません。書き込みトランザクションでは、ウィットネスリージョンが回復するまでレイテンシーがわずかに増加することがあります。

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/セカンダリフェー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ー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 のフォールトトレランスは、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。

フォールトインジェクションテスト

Amazon Aurora DSQL は、アプリケーションのフォールトトレランスを向上させるために実行中の制御されたフォールトインジェクション実験のフルマネージドサービスである AWS Fault Injection Service (AWS FIS) と統合します。AWS FIS を使用して、以下のことができます。

- 特定の障害シナリオを定義する実験テンプレートを作成する
- 障害 (クラスター接続エラー率が高い) を挿入し、アプリケーションのエラー処理と復旧メカニズムを検証する
- マルチリージョンアプリケーションの動作をテストし、一つの AWS リージョン の接続エラー率が高いときの AWS リージョン 間のアプリケーショントラフィックのシフトを検証する

例えば、米国東部 (バージニア北部) と米国東部 (オハイオ) にまたがるマルチリージョンクラスターでは、米国東部 (オハイオ) で実験を実行して障害をテストし、米国東部 (バージニア北部) で通常の運用を継続できます。この制御されたテストは、本番稼働用ワークロードに影響を与える前に潜在的な問題を特定して解決するのに役立ちます。

AWS FIS のサポートされているアクションの完全なリストについては、「AWS FIS ユーザーガイド」の「アクションターゲット」を参照してください。

AWS FIS で使用できる Amazon Aurora DSQL アクションの詳細については、「AWS FIS ユーザーガイド」の「Amazon Aurora DSQL アクションリファレンス」を参照してください。

フォールトインジェクション実験の実行を開始するには、「AWS FIS ユーザーガイド」の「AWS FIS 実験の計画」を参照してください。

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 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、「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「インターフェイス VPC エンドポイントを使用して AWS のサービスにアクセスする」と「AWS PrivateLink のクォータ」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.dsql.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。

JSON

```
{
  "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);
```

Direct Connect または Amazon VPC ピアリング経由で接続する際の追加設定

Amazon VPC ピアリングまたは Direct Connect を介してオンプレミスのデバイスから AWS PrivateLink 接続エンドポイントを使用して Aurora DSQL クラスターに接続するには、追加の設定が必要になる場合があります。アプリケーションが AWS PrivateLink エンドポイントと同じ Amazon VPC で実行されている場合、この設定は必要ありません。上記で作成したプライベート DNS エントリは、エンドポイントの Amazon VPC の外部で正しく解決されませんが、AWS PrivateLink 接続エンドポイントに解決される独自のプライベート DNS レコードを作成できます。

AWS PrivateLink エンドポイントの完全修飾ドメイン名を指すプライベート CNAME DNS レコードを作成します。作成された DNS レコードのドメイン名は、次のコンポーネントから構成する必要があります。

1. サービス名のサービス識別子。例: `dsql-fnh4`
2. AWS リージョン

次の形式のドメイン名を使用して CNAME DNS レコードを作成します。`*.service-identifier.region.on.aws`

ドメイン名の形式は、次の 2 つの理由で重要です。

1. `verify-full` SSL モードを使用する場合、Aurora DSQL への接続に使用するホスト名は、Aurora DSQL のサーバー証明書と一致する必要があります。これにより、最高レベルの接続セキュリティが保証されます。
2. Aurora DSQL は、Aurora DSQL への接続に使用されるホスト名のクラスター ID 部分を使用して、接続するクラスターを識別します。

プライベート DNS レコードを作成できない場合でも、引き続き Aurora DSQL に接続できます。「プライベート DNS のない AWS PrivateLink エンドポイントを使用して Aurora DSQL クラスターに接続する」を参照してください。

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。例: `us-east-1`

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
```

プライベート DNS のない AWS PrivateLink エンドポイントを使用して Aurora DSQL クラスターに接続する

上記の接続手順は、プライベート DNS レコードに依存しています。アプリケーションが AWS PrivateLink エンドポイントと同じ Amazon VPC で実行されている場合、DNS レコードが自動的に作成されます。または、オンプレミスのデバイスから Amazon VPC ピアリングまたは Direct Connect 経由で接続する場合は、独自のプライベート DNS レコードを作成できます。ただし、セキュリティチームが課すネットワーク制限により、DNS レコードの設定が常に可能になるとは限りません。アプリケーションが Direct Connect またはピアリングされた Amazon VPC を使用して接続する必要があり、DNS レコードを設定できない場合でも、Aurora DSQL に接続できます。

Aurora DSQL は、ホスト名のクラスター ID 部分を使用して接続クラスターを識別しますが、DNS レコードを設定できない場合は、Aurora DSQL は `amzn-cluster-id` 接続オプションを使用してターゲットクラスターを指定することをサポートしています。このオプションを使用すると、接続時に AWS PrivateLink エンドポイントの完全修飾ドメイン名をホスト名として使用できます。

> **Important**
>
> AWS PrivateLink エンドポイントの完全修飾ドメイン名または IP アドレスで接続する場合、`verify-full` SSL モードはサポートされません。そのため、プライベート DNS を設定することをお勧めします。

例: PostgreSQL を使用してクラスター ID 接続オプションを指定する

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export HOSTNAME=vpce-04037adb76c111221-d849uc2p.dsql-fnh4.us-east-1.vpce.amazonaws.com
 # This should match your endpoint's fully-qualified domain name

# Construct the hostname used to generate the authentication token
export AUTH_HOSTNAME="$CLUSTERID.dsql.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $AUTH_HOSTNAME)

# Specify the amzn-cluster-id connection option
export PGOPTIONS="-c amzn-cluster-id=$CLUSTERID"

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

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。

問題	考えられる原因	ソリューション
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、ディザスターリカバリー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
- Amazon Web Services: 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。これを防ぐために、AWS には、アカウント内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が用意されています。

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:*dsql*:*:*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、サービスロールが代理する Aurora DSQL リソースの ARN 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。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "backup.amazonaws.com"
    },
    "Action": "dsql:GetCluster",
    "Resource": [
```

```
      "arn:aws:dsql:*:123456789012:cluster/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:backup:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

CDC ストリームサービスロール

変更データキャプチャ (CDC) ストリームには、Aurora DSQL が CDC レコードをターゲットに書き込むために引き受ける IAM サービスロールが必要です。このロールを作成するときは、信頼ポリシーで aws:SourceAccount および aws:SourceArn 条件を使用して、アカウントの CDC ストリームのみがロールを引き受けられるようにします。

aws:SourceArn を、ロールを使用するクラスターのストリーム ARN パターンに設定します。Aurora DSQL はストリームの作成時にストリーム識別子を割り当てていないため、ARN のストリーム部分にはワイルドカードを使用します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DSQLAssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": "dsql.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "your-account-id"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:dsql:region:your-account-
id:cluster/cluster-id/stream/*"
```

```
                }
            }
        }
    ]
}
```

ストリームを作成したら、ロールが単一のストリームを提供する場合、aws:SourceArn を正確なストリーム ARN に絞り込むことができます。CDC サービスロールの信頼ポリシーとアクセス許可ポリシーの詳細については、「IAM の設定」を参照してください。

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 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。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。

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局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、クラスターや CDC ストリームなどの 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 マネジメントコンソール、AWS CLI、または AWS SDK を使用して、Aurora DSQL クラスターおよび CDC ストリームのタグを追加、一覧表示、削除できます。AWS コンソールを使用して、リソース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リソース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。

CDC ストリームのタグ付け

CDC ストリームは個別にタグ付け可能なリソースです。ストリームの作成時に、`CreateStream` に `--tags` パラメータを渡すことでタグを追加でき、既存のストリームに対して、ストリーム ARN を指定して `ListTagsForResource`、`TagResource`、`UntagResource` を使用することで、タグの読み取り、追加、削除を行うことができます。CDC ストリームのタグは、親クラスターのタグや、送信先の Amazon Kinesis データストリームのタグとは別のものです。

CDC ストリーム ARN の形式は `arn:aws:dsql:`*`region`*`:`*`account-id`*`:cluster/`*`cluster-id`*`/stream/`*`stream-id`* です。CDC ストリームの詳細については、「変更データキャプチャ (CDC) ストリーム」を参照してください。

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。

- DROP TABLE コマンドを実行した後、解放されたストレージを反映されるまで、ストレージ制限の計算に時間がかかる場合があります。追加のストレージ容量が必要な場合は、「クラスタークォータ」を参照してクォータの更新をリクエストしてください。
- Aurora DSQL の大きなテーブルの場合、COUNT(*) オペレーションの代わりにシステムカタログを使用してテーブル行数を取得します。詳細については、「Aurora DSQL でのシステムテーブルとコマンドの使用」を参照してください。
- Aurora DSQL は、スキーマレベルの許可を通じてアクセス許可を管理します。管理者ユーザーは、CREATE SCHEMA を使用してスキーマを作成し、GRANT USAGE ON SCHEMA を使用して他のロールへのアクセスを許可します。管理者ユーザーはパブリックスキーマのオブジェクトを管理し、管理者以外のユーザーはユーザーが作成したスキーマ内にオブジェクトを作成します。管理者ロールは、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取得するために、自身に他のロールを付与できます。詳細については、「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」を参照してください。
- ドライバーが PG_PREPARED_STATEMENTS を呼び出すと、Aurora DSQL はキャッシュされたプリペアドステートメントのクラスター全体のビューを提供します。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、準備中にステートメント名を動的に管理します。
- IPv4 専用インスタンスから接続する場合は、クライアントが IPv4 接続用に設定されていることを確認します。一部の PostgreSQL クライアントは、デュアルスタックモードで IPv4 と IPv6 の両方の接続を試行します。IPv4 接続でスロットリングが発生した場合、クライアントは IPv6 を試行し、IPv4 専用ホストで NetworkUnreachable エラーを返すことがあります。この動作を避けるために IPv4 を明示的に使用するようにクライアントを設定します。
- 管理者ユーザーが新しいスキーマを作成すると、GRANT および REVOKE の変更は接続有効期間内 (最大 1 時間) に既存の接続に伝播されます。すぐに有効にするには、アクセス許可の変更後に新しい接続を確立します。
- まれに発生するマルチリージョンリンククラスター復旧シナリオでは、自動クラスター復旧オペレーションによって高可用性が維持されますが、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、影響を受けるのはワークロードの一部のみです。このよう

な一時的な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、スキーマ情報を入力するために広範なシステムメタデータをリクエストします。Aurora DSQL は、SQL クエリ機能のコアメタデータを提供します。これらのクライアントのスキーマ表示では、完全な機能セットと比較して、限られた情報が表示される場合があります。
- クエリが新しく作成されたスキーマとテーブルを認識するようにするには、データベースオブジェクトを作成または削除した後に接続を更新します。これには、スキーマを削除した後、または別の接続で作成されたオブジェクトをクエリするときに `Schema Already Exists` エラーが表示されるシナリオが含まれます。切断して再接続するか、`SET search_path` を再度実行してカタログキャッシュを更新します。
- 複雑なクエリの場合は、`EXPLAIN ANALYZE VERBOSE` を使用して高レイテンシーオペレーションを特定し、クエリプランを最適化します。インデックスをカバーすると、完全なテーブルスキャンではなくインデックスのみのスキャンが可能になり、DPU コストを大幅に削減できます。詳細については、「Aurora DSQL EXPLAIN プランの操作」を参照してください。
- 接続制限はクラスターレベルで管理されます。クォータの更新をリクエストするには、「クラスタークォータ」を参照してください。

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 ServiceQuotaExceededExcepti
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 ServiceQuotaExceededExcepti
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 256 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
クラスターあたりの最大 CDC ストリーム数	5 ストリーム	いいえ	API エラーコード `ServiceQuotaExceededExcepti`

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	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
の最大合計サイズ				
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
シーケンスの最大数	5,000	いいえ	54000	ERROR: creating more than 500 not allowed

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 マネジメントコンソールおよび 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

error: unrecognized SSL error code: 6 または unable to accept connection, sni was not received

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。

ユーザーとアクションのマッピングが正しくない

このエラーは、認証トークンタイプがデータベースロールと一致しない場合に発生します。Aurora DSQL は、admin ロール用の DbConnectAdmin とカスタムデータベースロール用の DbConnect という 2 つのトークンタイプを使用します。

- `Wrong user to action mapping. user: admin, action: DbConnect` が表示された場合は、`generate-db-connect-admin-auth-token` を `generate-db-connect-auth-token` の代わりに使用します。
- `Wrong user to action mapping. user: myusername, action: DbConnectAdmin` が表示された場合は、`generate-db-connect-auth-token` を `generate-db-connect-admin-auth-token` の代わりに使用します。

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。「Aurora DSQL でサポートされている SQL コマンドのサブセット」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。「Aurora DSQL でサポートされている SQL コマンドのサブセット」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: 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。

同時実行制御レスポンスのトラブルシューティング

OC000 “ERROR: change conflicts with another transaction (OC000)”

このトランザクションは、別の同時実行トランザクションと同じタプルを変更しようとしました。これは、変更されたタプルの競合を示します。詳細については、「Aurora DSQL の同時実行制御」を参照してください。

OC001 “ERROR: schema has been updated by another transaction (OC001)”

セッションには、バージョン V1 のスキーマカタログのキャッシュされたコピーがあり、時間 T1 にロードされました。

別のトランザクションでは、時刻 T2 にカタログがバージョン V2 に更新されました。

時間 T3 で、セッションがクエリを実行すると、カタログが古いことが検出され、新しいカタログの変更へのリベースを試みます。状況によっては、リベースが成功せず、Aurora DSQL が `40001 OC001` レスポンスを返す場合があります。クエリプロセッサは、プロアクティブ更新を受信するのではなく、カタログの変更を事後的に検出するため、T2 と T3 の間の時間は数ミリ秒から数分の範囲になります。

同じセッションから再試行すると、Aurora DSQL はカタログキャッシュを更新します。再試行されたトランザクションはカタログ V2 を使用し、T2 以降にカタログの変更が発生していない限り、成功します。

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 に関するフィードバックの提供

移行に不可欠であるにもかかわらず、Aurora DSQL で現在サポートされていない機能に遭遇した場合、AWS にはフィードバック用のチャネルがいくつか用意されています。

フィードバックチャネル

Aurora DSQL Discord サーバー

Aurora DSQL Discord サーバーに参加して、AWS チームおよびコミュニティとつながります。機能リクエストを共有し、移行の課題について話し合い、リアルタイムのフィードバックを得ることができます。

AWS Support

AWS サポートプランをお持ちの場合は、サポートケースを作成して、特定の要件とタイムラインのニーズについて話し合います。

AWS re:Post

AWS re:Post を使用して質問をし、コミュニティや AWS の専門家とフィードバックを共有します。

有効な機能リクエスト

機能をリクエストするときは、以下を提供します。

- ユースケースの説明: 何を達成しようとしているのか、その理由を説明します
- 現在の回避策: 試した代替案があれば説明します
- ビジネスへの影響: 不足している機能が移行タイムラインまたはアプリケーション機能にどのように影響するかを説明します
- 優先度レベル: これが移行をブロックしているか、あれば望ましい改善点なのかを示します

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
新しいコンテンツ: 変更データキャプチャ (CDC) ストリーム	コミットされた行レベルの変更をキャプチャして Amazon Kinesis Data Streams に配信する Aurora DSQL 変更データキャプチャ (CDC) ストリームに関するドキュメントを追加しました。詳細については、「変更データキャプチャ (CDC) ストリーム」を参照してください。	2026 年 5 月 13 日
新しいコンテンツ: CREATE TABLE と ALTER TABLE における STORAGE キーワードのサポート	列のストレージモード (PLAIN、EXTERNAL、EXTENDED、MAIN、DEFAULT) を設定する、CREATE TABLE および ALTER TABLE の STORAGE 句に関するドキュメントを追加しました。これは、json などの可変長データ型の圧縮動作を制御するために使用できます。詳細については、「CREATE TABLE」および「ALTER TABLE」を参照してください。	2026 年 5 月 1 日
新しいコンテンツ: JSON データ型	自動圧縮動作、1 MiB の圧縮サイズ制限、サポートされている JSON 関数と演算子など、Aurora DSQL の json データ型に関するドキュメントを追加しました。ネイティブ json ストレージの可用性	2026 年 5 月 1 日

	を反映するために、JSONB ランタイムタイプのガイダンスを更新しました。詳しくは「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。	
新しいコンテンツ: ALTER TABLE ADD CONSTRAINT USING INDEX	`ALTER TABLE ... ADD` *`table_constraint_using_index`* のドキュメントを追加しました。これにより、既存の一意のインデックスに基づいてテーブルに UNIQUE 制約が追加されます。詳細については、「ALTER TABLE」を参照してください。	2026 年 4 月 20 日
新しいコンテンツ: PHP PDO_PGSQL 用 Aurora DSQL コネクタ	自動 IAM 認証で PDO_PGSQL をラップする PHP PDO_PGSQL 用 Aurora DSQL コネクタのドキュメントを追加しました。コネクタは、PHP アプリケーションのトークン生成、SSL 設定、接続管理を処理します。詳細については、「PHP コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2026 年 4 月 10 日

新しいコンテンツ: Rust SQLx 用 Aurora DSQL コネクタ	自動 IAM 認証で SQLx をラップする Rust SQLx 用 Aurora DSQL コネクタのドキュメントを追加しました。コネクタは、Rust アプリケーションのトークン生成、SSL 設定、接続管理を処理します。詳細については、「Rust コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2026 年 3 月 27 日
新しいコンテンツ: .NET Npgsql 用 Aurora DSQL コネクタ	自動 IAM 認証で Npgsql をラップする、.NET Npgsql 用 Aurora DSQL コネクタのドキュメントを追加しました。コネクタは、.NET アプリケーションのトークン生成、SSL 設定、および接続管理を処理します。詳細については、「.NET Npgsql コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2026 年 3 月 20 日

更新された内容: SQL コマンドリファレンスとシステムクエリ	END と ABORT をエイリアスとする、START TRANSACTION と ROLLBACK をトランザクションコントロールコマンドリファレンスに追加しました。Aurora DSQL および PostgreSQL のバージョン情報を取得するために役立つシステムクエリを追加しました。詳細については、「PostgreSQL の互換性リファレンス」を参照してください。	2026 年 3 月 13 日
更新された内容: Aurora DSQL へのデータのロード	データロードガイドを更新し、クライアント側の \copy の使用状況、INSERT のベストプラクティス、ロード前にテーブルを事前に作成するためのガイダンスを追加しました。詳細については、「Aurora DSQL へのデータのロード」を参照してください。	2026 年 3 月 13 日

新しいコンテンツ: Ruby pg 用 Aurora DSQL コネクタ	自動 IAM 認証で pg gem をラップする、Ruby pg 用 Aurora DSQL コネクタのドキュメントを追加しました。コネクタは、Ruby アプリケーションのトークン生成、SSL 設定、および接続管理を処理します。詳細については、「Ruby pg コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2026 年 3 月 12 日
更新された内容: 非同期 DDL ジョブ	sys.jobs ドキュメントを更新し、非同期 DDL オペレーションのモニタリングと管理の詳細を拡張しました。詳細については、「PostgreSQL の互換性リファレンス」を参照してください。	2026 年 3 月 6 日
更新された内容: PHP 認証トークンの生成	認証トークンの生成ページに PHP SDK タブを追加しました。詳細については、「認証トークンの生成」を参照してください。	2026 年 3 月 5 日
新しいコンテンツ: Aurora DSQL へのデータのロード	Aurora DSQL ローダーユーティリティの使用など、Aurora DSQL クラスターにデータをロードするためのガイドを追加しました。詳細については、「Aurora DSQL へのデータのロード」を参照してください。	2026 年 3 月 5 日

新しいコンテンツ: シーケンスと ID 列	シーケンスと ID 列のサポートが追加されました。CREATE SEQUENCE、ALTER SEQUENCE、DROP SEQUENCE、およびシーケンス操作関数の新しい SQL コマンドリファレンスページ。ID 列構文を含めるように CREATE TABLE と ALTER TABLE を更新しました。識別子タイプとキャッシュサイズを選択するための新しいガイドを追加しました。詳細については、「シーケンスと ID 列」を参照してください。	2026 年 2 月 11 日
新しいコンテンツ: Go 用 Aurora DSQL コネクタ	自動 IAM 認証で pgx をラップする Go 用 Aurora DSQL コネクタのドキュメントを追加しました。コネクタは、Go アプリケーションのトークン生成、SSL 設定、および接続管理を処理します。詳細については、「Go コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2026 年 2 月 5 日

更新されたコンテンツ: Amazon Aurora DSQL クラスター接続ツール	クラスター接続ツールのドキュメントを再編成し、AWS が提供するコネクタ、アダプター、サードパーティー製ツールの区別を明確にしました。コードサンプルへの欠落していたリンクを追加しました。詳細については、「Amazon Aurora DSQL クラスター接続ツール」を参照してください。	2026 年 1 月 26 日
新しいコンテンツ: DBeaver Community Edition 用 Aurora DSQL プラグイン	Aurora DSQL クラスターの IAM 認証と簡素化された接続設定を可能にする DBeaver Community Edition 用 Aurora DSQL プラグインのドキュメントを追加しました。インストール手順、接続設定、トラブルシューティングガイダンスが含まれています。詳細については、「DBeaver を使用して Aurora DSQL にアクセスする」を参照してください。	2026 年 1 月 26 日
新しいコンテンツ: Aurora DSQL Driver for SQLTools	開発者が自動 IAM 認証を使用して VS Code から直接 Aurora DSQL データベースに接続してクエリできるようにする Visual Studio Code 拡張機能である Aurora DSQL Driver for SQLTools のドキュメントを追加しました。詳細については、「Aurora DSQL Driver for SQLTools を使用する」を参照してください。	2026 年 1 月 26 日

更新されたコンテンツ: Aurora DSQL ステアリング: Skills と Powers	エージェントに依存しないサポートのために、Skills CLI を使用したインストールをサポートするドキュメントを追加しました。Skills CLI は、Claude Code、Cursor、Copilot、Gemini など、複数の AI コーディングアシスタントで動作するシンプルなセットアップ方法を提供します。詳細については、「Aurora DSQL ステアリング: Skills と Powers」を参照してください。	2026 年 1 月 23 日
新しいコンテンツ: Tortoise ORM 用の Aurora DSQL アダプター	Python 非同期 ORM フレームワークである Tortoise ORM のサポートが追加されました。Tortoise ORM 用の Aurora DSQL アダプターを使用すると、デベロッパーは Tortoise ORM を Aurora DSQL クラスターで使用できるようになります。詳細については、「Aurora DSQL アダプターとダイアレクト」を参照してください。	2026 年 1 月 23 日

新しいコンテンツ: Aurora DSQL ステアリング: Skills と Powers	Skills と Powers を使用して Aurora DSQL で AI ステアリングを設定するための新しいドキュメントを追加しました。Kiro Powers、Claude Skills、Gemini Skills、および Codex Skills のセットアップ手順が含まれています。詳細については、「Aurora DSQL ステアリング: Skills と Powers」を参照してください。	2026 年 1 月 16 日
数値データ型インデックスのサポート	Aurora DSQL の `numeric` データ型のインデックスサポートを追加しました。`numeric` 列をプライマリキーおよびセカンダリインデックスとして使用できるようになりました。詳しくは「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。	2026 年 1 月 13 日

更新されたコンテンツ: サポートされている SQL コマンドのサブセット	ナビゲーションとわかりやすさを改善するために、SQL コマンドのドキュメントを個別のページに再編成しました。各コマンド (CREATE TABLE、ALTER TABLE、CREATE VIEW、ALTER VIEW、DROP VIEW) に専用ページが用意されました。詳細については、「サポートされている SQL コマンドのサブセット」を参照してください。	2026 年 1 月 6 日
更新されたコンテンツ: AWS ラボ Aurora DSQL MCP サーバー	CLI ベースのセットアップと設定ファイルの例を含む、Claude Code および Codex の詳細なインストール方法を記載した MCP サーバードキュメントが更新されました。さまざまな開発ツール間で MCP クライアント設定ファイルを検索するための包括的なガイダンスを追加しました。詳細については、「AWS ラボ Aurora DSQL MCP サーバー」を参照してください。	2025 年 12 月 19 日

更新されたコンテンツ: PostgreSQL から Aurora DSQL への移行	PostgreSQL の互換性セクションを包括的な移行ガイドとして改訂しました。フレームワークの互換性情報、一般的な移行パターン、アーキテクチャの違い、AI 支援による移行ガイダンスが含まれます。Aurora DSQL に関するフィードバックを提供するための新しい章を追加しました。詳細については、「PostgreSQL から Aurora DSQL への移行」を参照してください。	2025 年 12 月 16 日
更新されたコンテンツ: AWS PrivateLink を使用した Aurora DSQL への接続	AWS PrivateLink を Direct Connect または Amazon VPC ピアリングで使用しているお客様をサポートするために、プライベート DNS セットアップとクラスター ID 接続オプションに関するドキュメントを追加しました。`amzn-cluster-id` 接続オプションを使用してプライベート DNS なしで接続するためのガイダンスが含まれています。詳細については、「AWS PrivateLink を使用した Aurora DSQL クラスターの管理と接続」を参照してください。	2025 年 12 月 11 日

更新されたコンテンツ: Aurora DSQL クラスターのライフサイクル	Aurora DSQL クラスターのライフサイクル管理に関するドキュメントが更新されました。クラスターステータス定義、状態遷移、およびアイドル状態と非アクティブ状態で使用可能なオペレーションについて説明します。詳細については、「Aurora DSQL クラスターのライフサイクル」を参照してください。	2025 年 12 月 4 日
新しいコンテンツ: Python および Node.js 用の Aurora DSQL コネクタ	Python (psycopg、psycopg2、asyncpg) および Node.js (node-postgres、Postgres.js) 用の Aurora DSQL コネクタに関するドキュメントを追加しました。これらのコネクタは、アプリケーションを Aurora DSQL クラスターに接続するための IAM 認証を統合します。詳細については、「Aurora DSQL コネクタ」を参照してください。	2025 年 11 月 21 日

新しいコンテンツ: Aurora DSQL での JupyterLab の使用	Python で JupyterLab を使用して Aurora DSQL に接続し、クエリを実行するためのステップバイステップガイドを追加しました。ローカル JupyterLab インストールと Amazon SageMaker AI 環境の両方の手順が含まれています。詳細については、「Aurora DSQL での JupyterLab の使用」を参照してください。	2025 年 11 月 20 日
更新されたコンテンツ: Aurora DSQL のクォータ	クラスターの最大ストレージクォータを 128 TiB から 256 TiB に更新しました。詳細については、「Aurora DSQL のクォータ」を参照してください。	2025 年 11 月 19 日
新しいコンテンツ: Aurora DSQL クエリエディタの使用を開始する	AWS マネジメントコンソールで Aurora DSQL クエリエディタを使用するためのドキュメントを追加しました。前提条件、接続設定、クエリ実行手順が含まれています。詳細については、「Aurora DSQL クエリエディタの使用を開始する」を参照してください。	2025 年 11 月 18 日

Amazon Aurora DSQL のリソースベースのポリシーのサポート	`PutClusterPolicy` 、`GetClusterPolicy` 、`DeleteClusterPolicy` の新しいアクセス許可を持つリソースベースのポリシー (RBP) サポートを追加しました。これらのアクセス許可により、Aurora DSQL クラスターにアタッチされたインラインポリシーを管理し、きめ細かなアクセスコントロールを行うことができます。マネージドポリシー AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess、AmazonAuroraDSQLConsoleFullAccess を更新し、RBP 機能を含めました。詳細については、「Amazon Aurora DSQL の AWS マネージドポリシー」を参照してください。	2025 年 10 月 15 日
Aurora DSQL JDBC コネクタ	Java アプリケーションを Amazon Aurora DSQL クラスターに接続するための IAM 認証を統合した PgJDBC ラッパーである Aurora DSQL JDBC コネクタに関するドキュメントを追加しました。詳細については、「JDBC コネクタを使用した Aurora DSQL クラスターへの接続」を参照してください。	2025 年 9 月 2 日

AWS FIS 統合のための AWS マネージドポリシーの更新	AWS Fault Injection Service と Aurora DSQL との統合をサポートするように AmazonAuroraDSQLFullAccess および AmazonAuroraDSQLConsoleFullAccess ポリシーを更新しました。これにより、単一リージョンおよびマルチリージョンの Aurora DSQL クラスターに障害を挿入して、アプリケーションの耐障害性をテストできます。これらのポリシーの詳細については、「AWS managed policy updates」を参照してください。	2025 年 8 月 19 日
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「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

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「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

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` ネームスペース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

AWS PrivateLinkAmazon Aurora DSQL の	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