aws

사용자 가이드

Amazon Aurora DSQL

Amazon Aurora DSQL: 사용자 가이드

Table of Contents

Amazon Aurora DSQL이란 무엇인가요?

Amazon Aurora DSQL은 트랜잭션 워크로드에 최적화된 서버리스 분산 관계형 데이터베이스 서비스입니다. Aurora DSQL은 사실상 무제한의 규모를 제공하며 사용자가 인프라를 관리할 필요가 없습니다. 가용성이 매우 높은 액티브-액티브 아키텍처는 단일 리전의 경우 99.99%, 다중 리전의 경우 99.999%의 가용성을 제공합니다.

Aurora DSQL을 사용해야 하는 경우

Aurora DSQL은 ACID 트랜잭션 및 관계형 데이터 모델의 이점을 활용하는 트랜잭션 워크로드에 최적화되어 있습니다. 서버리스이기 때문에 Aurora DSQL은 마이크로서비스, 서버리스 및 이벤트 기반 아키텍처의 애플리케이션 패턴에 적합합니다. Aurora DSQL은 PostgreSQL과 호환되므로 익숙한 드라이버, 객체 관계 매핑(ORM), 프레임워크 및 SQL 기능을 사용할 수 있습니다.

Aurora DSQL은 시스템 인프라를 자동으로 관리하고 워크로드에 따라 컴퓨팅, I/O 및 스토리지 규모를 조정합니다. 프로비저닝하거나 관리할 서버가 없으므로 프로비저닝, 패치 적용 또는 인프라 업그레이드와 관련된 유지 관리 가동 중지 시간에 대해 걱정할 필요가 없습니다.

Aurora DSQL은 어느 규모에서나 항상 사용할 수 있는 엔터프라이즈 애플리케이션을 구축하고 유지 관리하는 데 도움이 됩니다. 액티브-액티브 서버리스 설계는 장애 복구를 자동화하므로 기존 데이터베이스 장애 조치에 대해 걱정할 필요가 없습니다. 애플리케이션은 다중 AZ 및 다중 리전 가용성의 이점을 누릴 수 있으며 장애 조치와 관련된 최종 일관성이나 누락된 데이터에 대해 걱정할 필요가 없습니다.

Aurora DSQL의 주요 기능

다음 주요 기능은 고가용성 애플리케이션을 지원하는 서버리스 분산 데이터베이스를 생성하는 데 도움이 됩니다.

분산 아키텍처

Aurora DSQL은 다음과 같은 다중 테넌트 구성 요소로 구성됩니다.

- 릴레이 및 연결
- 컴퓨팅 및 데이터베이스
- 트랜잭션 로그, 동시성 제어 및 격리
- 스토리지

컨트롤 플레인은 이전 구성 요소를 조정합니다. 각 구성 요소는 구성 요소 장애 시 자동 클러스터 규모 조정 및 자가 복구를 통해 세 개의 가용 영역(AZ)에 걸쳐 중복성을 제공합니다. 이 아키텍처가 고가용성을 지원하는 방법에 대한 자세한 내용은 Amazon Aurora DSQL의 복원력 섹션을 참조하세요.

다중 리전 및 단일 리전 클러스터

Aurora DSQL 클러스터는 다음과 같은 이점을 제공합니다.

- 동기식 데이터 복제
- 일관된 읽기 작업
- 자동 장애 복구
- 여러 AZ 또는 리전에서의 데이터 일관성

인프라 구성 요소가 실패하면 Aurora DSQL은 수동 개입 없이 요청을 정상 인프라로 자동으로 라우팅합니다. Aurora DSQL은 강력한 일관성, 스냅샷 격리, 원자성, 교차 AZ 및 교차 리전 내구성을 갖춘 원자성, 일관성, 격리 및 내구성(ACID) 트랜잭션을 제공합니다.

다중 리전 피어링된 클러스터는 단일 리전 클러스터와 동일한 복원력과 연결을 제공합니다. 그러나 피어링된 클러스터 리전마다 하나씩, 총 두 개의 리전 엔드포인트를 제공하여 가용성을 개선합니다. 피어링된 클러스터의 두 엔드포인트 모두 단일 논리적 데이터베이스를 제공합니다. 동시 읽기 및 쓰기 작업에 사용할 수 있으며 강력한 데이터 일관성을 제공합니다. 성능과 복원력을 위해 여러 리전에서 동시에 실행되는 애플리케이션을 빌드할 수 있으며 독자는 항상 동일한 데이터를 볼 수 있습니다.

PostgreSQL 데이터베이스와의 호환성

Aurora DSQL의 분산 데이터베이스 계층(컴퓨팅)은 PostgreSQL의 현재 메이저 버전을 기반으로 합니다. `psql`과 같은 친숙한 PostgreSQL 드라이버 및 도구를 사용하여 Aurora DSQL에 연결할 수 있습니다. Aurora DSQL은 현재 PostgreSQL 버전 16과 호환되며 PostgreSQL 기능, 표현식 및 데이터 유형의 하위 집합을 지원합니다. 지원되는 SQL 기능에 대한 자세한 내용은 Aurora DSQL의 SQL 기능 호환성 섹션을 참조하세요.

Aurora DSQL의 리전 가용성

Amazon Aurora DSQL을 사용하면 여러 AWS 리전에 데이터베이스 인스턴스를 배포하여 글로벌 애플리케이션을 지원하고 데이터 레지던시 요구 사항을 충족할 수 있습니다. 리전 가용성에 따라 Aurora DSQL 데이터베이스 클러스터를 생성하고 관리할 수 있는 위치가 결정됩니다. 가용성이 높고 전역에 분산된 데이터베이스 시스템을 설계해야 하는 데이터베이스 관리자와 애플리케이션 아키텍트는 워크로드에 대한 리전 지원을 이해해야 하는 경우가 많습니다. 일반적인 사용 사례에는 교차 리전 재해 복

구를 설정하고, 지리적으로 가까운 데이터베이스 인스턴스에서 사용자에게 서비스하여 지연 시간을 줄이고, 규정 준수를 위해 특정 위치에 데이터 복사본을 유지하는 것이 포함됩니다.

다음 표에는 현재 Aurora DSQL을 사용할 수 있는 AWS 리전 및 각 AWS 리전의 엔드포인트가 나와 있습니다.

지원되는 AWS 리전 및 엔드포인트

지역명	지역	엔드포인트	프로토콜
미국 동부(버지니아 북부)	us-east-1	dsql.us-east-1.api.aws	HTTPS
미국 동부(오하이오)	us-east-2	dsql.us-east-2.api.aws	HTTPS
미국 서부(오레곤)	us-west-2	dsql.us-west-2.api.aws	HTTPS
유럽(런던)	eu-west-2	dsql.eu-west-2.api.aws	HTTPS
유럽(아일랜드)	eu-west-1	dsql.eu-west-1.api.aws	HTTPS
유럽(파리)	eu-west-3	dsql.eu-west-3.api.aws	HTTPS
아시아 태평양(오사카)	ap-northeast-3	dsql.ap-northeast-3.api.aws	HTTPS
아시아 태평양(도쿄)	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS

> **Note**
>
> Aurora DSQL 다중 리전 클러스터는 현재 다음 세 개의 AWS 리전에서 지원됩니다.
>
> - 미국 동부(버지니아 북부)
> - 미국 동부(오하이오)
> - 미국 서부(오리건)

Aurora DSQL 요금

비용 정보는 Aurora DSQL pricing을 참조하세요.

다음 단계

Aurora DSQL의 핵심 구성 요소에 대한 자세한 내용을 알아보고 서비스를 시작하려면 다음을 참조하세요.

- Aurora DSQL 시작하기
- Aurora DSQL의 SQL 기능 호환성
- Aurora DSQL 액세스
- Aurora DSQL 및 PostgreSQL

Aurora DSQL 시작하기

Amazon Aurora DSQL은 트랜잭션 워크로드에 최적화된 서버리스 분산 관계형 데이터베이스입니다. 다음 섹션에서는 단일 리전 및 다중 리전 Aurora DSQL 클러스터를 생성하고, 클러스터에 연결하고, 샘플 스키마를 생성하고 로드하는 방법을 알아봅니다. AWS Management Console을 사용하여 클러스터에 액세스하고 `psql` 유틸리티를 사용하여 데이터베이스와 상호 작용합니다. 결국 작동하는 Aurora DSQL 클러스터가 설정되고 테스트 또는 프로덕션 워크로드에 사용할 준비가 됩니다.

주제

사전 조건

Aurora DSQL 사용을 시작하기 전에 다음 사전 조건을 충족하는지 확인하세요.

- IAM ID에 AWS Management Console에 로그인할 수 있는 권한이 있어야 합니다.
- IAM ID는 다음 기준 중 하나를 충족해야 합니다.
  - AWS 계정의 어떤 리소스에서나 작업을 수행할 수 있는 액세스 권한
  - IAM 권한 `iam:CreateServiceLinkedRole` 및 IAM 정책 작업 `dsql:*`에 대한 액세스 권한
- Unix와 유사한 환경에서 AWS CLI를 사용하는 경우 Python 버전 3.8+ 및 `psql` 버전 14+가 설치되어 있는지 확인합니다. 애플리케이션 버전을 확인하려면 다음 명령을 실행합니다.

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

  다른 환경에서 AWS CLI를 사용하는 경우 Python 버전 3.8+ 및 `psql` 버전 14+를 수동으로 설정해야 합니다.

- AWS CloudShell을 사용하여 Aurora DSQL에 액세스하려는 경우 Python 버전 3.8+ 및 `psql` 버전 14+가 추가 설정 없이 제공됩니다. AWS CloudShell에 대한 자세한 내용은 AWS CloudShell란 무엇인가요?를 참조하세요.
- GUI를 사용하여 Aurora DSQL에 액세스하려면 DBeaver 또는 JetBrains DataGrip을 사용합니다. 자세한 내용은 DBeaver를 사용하여 Aurora DSQL에 액세스 및 JetBrains DataGrip을 사용하여 Aurora DSQL에 액세스(을)를 참조하세요.

Aurora DSQL 액세스

다음 방법을 통해 Aurora DSQL에 액세스할 수 있습니다. CLI, API 및 SDK를 사용하는 방법을 알아보려면 Aurora DSQL 액세스 섹션을 참조하세요.

주제

- AWS Management Console을 통해 Aurora DSQL에 액세스
- SQL 클라이언트를 사용하여 Aurora DSQL에 액세스
- Aurora DSQL에서 PostgreSQL 프로토콜 사용

AWS Management Console을 통해 Aurora DSQL에 액세스

https://console.aws.amazon.com/dsql에서 Aurora DSQL용 AWS Management Console에 액세스할 수 있습니다.

SQL 클라이언트를 사용하여 Aurora DSQL에 액세스

Aurora DSQL은 PostgreSQL 프로토콜을 사용합니다. 클러스터에 연결할 때 서명된 IAM 인증 토큰을 암호로 제공하여 선호하는 대화형 클라이언트를 사용합니다. 인증 토큰은 Aurora DSQL이 AWS Signature Version 4를 사용하여 동적으로 생성하는 고유한 문자열입니다.

Aurora DSQL은 인증에만 토큰을 사용합니다. 토큰은 연결 설정 후 연결에 영향을 주지 않습니다. 만료된 토큰을 사용하여 다시 연결하려고 하면 연결 요청이 거부됩니다. 자세한 내용은 Amazon Aurora DSQL에서 인증 토큰 생성 섹션을 참조하세요.

주제

- psql을 사용하여 Aurora DSQL에 액세스(PostgreSQL 대화형 터미널)
- DBeaver를 사용하여 Aurora DSQL에 액세스
- JetBrains DataGrip을 사용하여 Aurora DSQL에 액세스

psql을 사용하여 Aurora DSQL에 액세스(PostgreSQL 대화형 터미널)

psql 유틸리티는 PostgreSQL에 대한 터미널 기반 프런트엔드입니다. 이를 통해 대화형으로 쿼리를 입력하고, PostgreSQL에 발급하고, 쿼리 결과를 볼 수 있습니다. 쿼리 응답 시간을 개선하려면 PostgreSQL 버전 17 클라이언트를 사용합니다.

PostgreSQL Downloads 페이지에서 운영 체제의 설치 프로그램을 다운로드합니다. psql에 대한 자세한 내용은 https://www.postgresql.org/docs/current/app-psql.htm을 참조하세요.

AWS CLI가 이미 설치되어 있는 경우 다음 예시를 사용하여 클러스터에 연결합니다. psql이 사전 설치되어 있는 AWS CloudShell을 사용하거나 psql을 직접 설치할 수 있습니다.

```
# 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
```

DBeaver를 사용하여 Aurora DSQL에 액세스

DBeaver는 오픈 소스 GUI 기반 데이터베이스 도구입니다. DBeaver를 사용하여 데이터베이스에 연결하고 데이터베이스를 관리할 수 있습니다. DBeaver를 다운로드하려면 DBeaver 커뮤니티 웹사이트의 다운로드 페이지를 참조하세요. 이어지는 단계에서는 DBeaver를 사용하여 클러스터에 연결하는 방법을 설명합니다.

DBeaver에서 새 Aurora DSQL 연결을 설정하는 방법

1. 새 데이터 연결을 선택합니다.

2. 새 데이터베이스 연결 창에서 PostgreSQL을 선택합니다.
3. 연결 설정/메인 탭에서 연결 기준: 호스트를 선택하고 다음 정보를 입력합니다.

   - 호스트 - 클러스터 엔드포인트를 사용합니다.

     데이터베이스 - `postgres`를 입력합니다.

     인증 - `Database Native`를 선택합니다.

     사용자 이름 - `admin`을 입력합니다.

     암호 - 인증 토큰을 생성합니다. 생성된 토큰을 복사하여 암호로 사용합니다.
4. 경고가 나오면 무시하고 DBeaver 암호 필드에 인증 토큰을 붙여 넣습니다.

> **Note**
>
> 클라이언트 연결에서 SSL 모드를 설정해야 합니다. Aurora DSQL은 `SSLMODE=require`를 지원합니다. Aurora DSQL은 서버 측에서 SSL 통신을 적용하고 SSL이 아닌 연결을 거부합니다.

5. 클러스터에 연결되어 있어야 하며 SQL 문 실행을 시작할 수 있습니다.

> **Important**
>
> PostgreSQL 데이터베이스에 대해 DBeaver에서 제공하는 관리 기능(예: 세션 관리자 및 잠금 관리자)은 고유한 아키텍처로 인해 데이터베이스에 적용되지 않습니다. 이러한 화면은 액세스 가능하지만 데이터베이스 상태에 대한 신뢰할 수 있는 정보를 제공하지 않습니다.

DBeaver용 인증 자격 증명 만료

설정된 세션은 최대 1시간 동안 또는 DBeaver 연결이 해제되거나 제한 시간이 초과될 대까지 인증된 상태로 유지됩니다. 새 연결을 설정하려면 연결 설정의 암호 필드에 유효한 인증 토큰을 제공하세요. 새 세션을 열려고 하면(예: 새 테이블 또는 새 SQL 콘솔 나열) 새 인증 시도가 강제로 수행됩니다. 연결 설정에 구성된 인증 토큰이 더 이상 유효하지 않으면 새 세션이 실패하고 DBeaver가 이전에 연 모든 세션을 무효화합니다. `expires-in` 옵션을 사용하여 IAM 인증 토큰의 기간을 선택할 때는 이 점을 염두에 두세요.

JetBrains DataGrip을 사용하여 Aurora DSQL에 액세스

JetBrains DataGrip은 PostgreSQL을 포함한 데이터베이스와 SQL 작업을 위한 교차 플랫폼 IDE입니다. DataGrip에는 지능형 SQL 편집기가 있는 강력한 GUI가 포함되어 있습니다. DataGrip을 다운로드하려면 JetBrains 웹사이트의 다운로드 페이지로 이동합니다.

JetBrains DataGrip에서 새 Aurora DSQL 연결을 설정하는 방법

1. 새 데이터 소스를 선택하고 PostgreSQL을 선택합니다.
2. 데이터 소스/일반 탭에서 다음 정보를 입력합니다.
   - 호스트 - 클러스터 엔드포인트를 사용합니다.

     포트 - Aurora DSQL은 PostgreSQL 기본값인 `5432`를 사용합니다.

     데이터베이스 - Aurora DSQL은 PostgreSQL 기본값인 `postgres`를 사용합니다.

     인증 - `User & Password` 를 선택합니다.

     사용자 이름 - `admin`을 입력합니다.

     암호 - 토큰을 생성하여 이 필드에 붙여 넣습니다.

     URL - 이 필드는 수정하지 마세요. 다른 필드에 따라 자동으로 채워집니다.
3. 암호 - 인증 토큰을 생성하여 제공합니다. 토큰 생성기의 결과 출력을 복사하여 암호 필드에 붙여 넣습니다.

   > **Note**
   >
   > 클라이언트 연결에서 SSL 모드를 설정해야 합니다. Aurora DSQL은 `PGSSLMODE=require`를 지원합니다. Aurora DSQL은 서버 측에서 SSL 통신을 적용하고 SSL이 아닌 연결을 거부합니다.
4. 클러스터에 연결되어 있어야 하며 SQL 문 실행을 시작할 수 있습니다.

> **Important**
>
> PostgreSQL 데이터베이스에 대해 DataGrip에서 제공하는 일부 뷰(예: 세션)는 고유한 아키텍처로 인해 데이터베이스에 적용되지 않습니다. 이러한 화면은 액세스 가능하지만 데이터베이스에 연결된 실세 세션에 대한 신뢰할 수 있는 정보를 제공하지 않습니다.

인증 자격 증명 만료

설정된 세션은 최대 1시간 동안 또는 명시적 연결 해제나 클라이언트 측 제한 시간 초과가 발생할 때까지 인증된 상태로 유지됩니다. 새 연결을 설정해야 하는 경우 새 인증 토큰을 생성하여 데이터 소스 속성의 암호 필드에 제공해야 합니다. 새 세션을 열려고 하면(예: 새 테이블 또는 새 SQL 콘솔 나열) 새 인증 시도가 강제로 수행됩니다. 연결 설정에 구성된 인증 토큰이 더 이상 유효하지 않으면 새 세션이 실패하고 이전에 연 모든 세션이 무효화됩니다.

Aurora DSQL에서 PostgreSQL 프로토콜 사용

PostgreSQL은 클라이언트와 서버 간의 통신을 위해 메시지 기반 프로토콜을 사용합니다. 프로토콜은 TCP/IP 및 Unix 도메인 소켓을 통해 지원됩니다. 다음 표는 Aurora DSQL이 PostgreSQL 프로토콜을 지원하는 방법을 보여줍니다.

PostgreSQL	Aurora DSQL	Notes
역할(사용자 또는 그룹이라고도 함)	데이터베이스 역할	Aurora DSQL은 `admin`이라는 역할을 생성합니다. 사용자 지정 데이터베이스 역할을 생성할 경우 클러스터에 연결할 때 인증을 위해 `admin` 역할을 사용하여 IAM 역할과 연결해야 합니다. 자세한 내용은 Configure custom database roles을 참조하세요.
호스트(호스트 이름 또는 호스트 사양이라고도 함)	클러스터 엔드포인트	Aurora DSQL 단일 리전 클러스터는 단일 관리형 엔드포인트를 제공하고 리전 내에서 사용할 수 없는 경우 트래픽을 자동으로 리디렉션합니다.
Port	해당 사항 없음 - 기본값인 5432 사용	PostgreSQL 기본값입니다.

PostgreSQL	Aurora DSQL	Notes
데이터베이스(dbname)	`postgres` 사용	사용자가 클러스터를 생성할 때 Aurora DSQL이 이 데이터베이스를 생성합니다.
SSL Mode	SSL은 항상 서버 측에서 활성화됨	Aurora DSQL에서 Aurora DSQL은 `require` SSL Mode를 지원합니다. SSL이 없는 연결은 Aurora DSQL에서 거부됩니다.
암호	인증 토큰	Aurora DSQL에는 수명이 긴 암호 대신 임시 인증 토큰이 필요합니다. 자세한 내용은 Amazon Aurora DSQL에서 인증 토큰 생성를 참조하세요.

1단계: Aurora DSQL 단일 리전 클러스터 생성

Aurora DSQL의 기본 단위는 데이터를 저장하는 클러스터입니다. 이 작업에서는 단일 AWS 리전에 클러스터를 생성합니다.

Aurora DSQL에 단일 리전 클러스터를 생성하는 방법

1. AWS Management Console에 로그인한 다음, https://console.aws.amazon.com/dsql에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 생성을 선택한 다음 단일 리전을 선택합니다.
3. (선택 사항) 클러스터 설정에서 다음 옵션 중 하나를 선택합니다.
   - 암호화 설정 사용자 지정(고급)을 선택하여 AWS KMS key를 선택하거나 생성합니다.
   - 삭제 작업으로 인해 클러스터가 삭제되지 않도록 방지하려면, 삭제 방지 활성화를 선택합니다. 삭제 방지 기능은 기본적으로 선택됩니다.
4. (선택 사항) 태그에서 이 클러스터의 태그를 선택하거나 입력합니다.
5. 클러스터 생성을 선택합니다.

2단계: Aurora DSQL 클러스터에 연결

클러스터 엔드포인트는 클러스터 ID 및 리전을 기반으로 Aurora DSQL 클러스터를 생성할 때 자동으로 생성됩니다. 이름 형식은 *clusterid*.dsql.*region*.on.aws입니다. 클라이언트는 엔드포인트를 사용하여 클러스터에 대한 네트워크 연결을 생성합니다.

인증을 외부에서 IAM을 사용해 관리하기 때문에 자격 증명을 데이터베이스에 저장할 필요가 없습니다. 인증 토큰이란 동적으로 생성되는 고유 문자열입니다. 토큰은 인증에만 사용되며 연결이 설정된 후에는 연결에 영향을 주지 않습니다. 연결을 시도하기 전에 사전 조건에 설명된 대로 IAM ID에 `dsql:DbConnectAdmin` 권한이 있는지 확인합니다.

> **Note**
>
> 데이터베이스 연결 속도를 최적화하려면 PostgreSQL 버전 17 클라이언트를 사용하고 `PGSSLNEGOTIATION`을 `PGSSLNEGOTIATION=direct`로 설정합니다.

인증 토큰을 사용하여 클러스터에 연결하는 방법

1. Aurora DSQL 콘솔에서 연결할 클러스터를 선택합니다.
2. 연결을 선택합니다.
3. 엔드포인트(호스트)에서 엔드포인트를 복사합니다.
4. 인증 토큰(암호) 섹션에서 관리자로 연결을 선택했는지 확인합니다.
5. 생성된 인증 토큰을 복사합니다. 이 토큰은 15분 동안 유효합니다.
6. 운영 체제 명령줄에서 다음 명령을 사용하여 `psql`을 시작하고 클러스터에 연결합니다. *your_cluster_endpoint*를 앞서 복사한 클러스터 엔드포인트로 바꿉니다.

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

   암호를 입력하라는 메시지가 표시되면 앞서 복사한 인증 토큰을 입력합니다. 만료된 토큰을 사용하여 다시 연결하려고 하면 연결 요청이 거부됩니다. 자세한 내용은 Amazon Aurora DSQL에서 인증 토큰 생성 섹션을 참조하세요.
7. Enter를 누릅니다. PostgreSQL 프롬프트가 표시됩니다.

```
postgres=>
```

액세스 거부 오류가 발생하면 IAM ID에 `dsql:DbConnectAdmin` 권한이 있는지 확인합니다. 권한이 있고 액세스 거부 오류가 계속 발생하는 경우 IAM 문제 해결 및 IAM 정책을 적용할 때 액세스 거부 또는 승인되지 않은 작업 오류를 해결하려면 어떻게 해야 합니까?를 참조하세요.

3단계: Aurora DSQL에서 샘플 SQL 명령 실행

SQL 문을 실행하여 Aurora DSQL 클러스터를 테스트합니다. 다음 샘플 문에는 `department-insert-multirow.sql`과 `invoice.csv`라는 데이터 파일이 필요하며, GitHub의 aws-samples/aurora-dsql-samples 리포지토리에서 다운로드할 수 있습니다.

Aurora DSQL에서 샘플 SQL 명령을 실행하는 방법

1. `example`이라는 스키마를 생성합니다.

```
CREATE SCHEMA example;
```

2. 자동으로 생성된 UUID를 프라이머리 키로 사용하는 인보이스 테이블을 생성합니다.

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

3. 빈 테이블을 사용하는 보조 인덱스를 생성합니다.

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

4. 부서 테이블을 생성합니다.

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

5. `psql \include` 명령을 사용하여 GitHub의 aws-samples/aurora-dsql-samples 리포지토리에서 다운로드한 `department-insert-multirow.sql` 파일을 로드합니다. *`my-path`*를 로컬 복사본의 경로로 바꿉니다.

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

6. psql \copy 명령을 사용하여 GitHub의 aws-samples/aurora-dsql-samples 리포지토리에서 다운로드한 invoice.csv 파일을 로드합니다. *my-path*를 로컬 복사본의 경로로 바꿉니다.

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

7. 부서를 쿼리하고 총 매출을 기준으로 정렬합니다.

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

다음 샘플 출력은 세 번째 부서가 매출이 가장 많음을 보여줍니다.

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

4단계: 다중 리전 클러스터 생성

다중 리전 클러스터를 생성할 때 다음 리전을 지정합니다.

원격 리전

두 번째 클러스터를 생성하는 리전입니다. 이 리전에서 두 번째 클러스터를 생성하고 첫 번째 클러스터에 피어링합니다. Aurora DSQL은 첫 번째 클러스터의 모든 쓰기를 피어링된 클러스터에 복제합니다. 어느 클러스터에서나 읽고 쓸 수 있습니다.

감시 리전

이 리전은 다중 리전 클러스터에 기록된 모든 데이터를 수신합니다. 하지만 감시 리전은 클라이언트 엔드포인트를 호스팅하지 않으며 사용자 데이터 액세스를 제공하지 않습니다. 암호화된 트랜잭션 로그의 제한된 기간은 감시 리전에 유지됩니다. 이 로그를 통해 복구를 용이하게 하고 리전을 사용할 수 없는 경우 트랜잭션 쿼럼을 지원합니다.

다음 예시에서는 첫 번째 클러스터를 생성하고 다른 리전에 두 번째 클러스터를 생성한 다음 두 클러스터를 피어링하여 다중 리전 클러스터를 생성하는 방법을 보여줍니다. 또한 리전 간 쓰기 복제와 두 리전 엔드포인트의 일관된 읽기를 보여줍니다.

다중 리전 클러스터를 생성하는 방법

1. AWS Management Console에 로그인한 다음, https://console.aws.amazon.com/dsql에서 Aurora DSQL 콘솔을 엽니다.
2. 탐색 창에서 클러스터를 선택합니다.
3. 클러스터 생성을 선택한 다음 다중 리전을 선택합니다.
4. (선택 사항) 클러스터 설정에서 첫 번째 클러스터에 대해 다음 옵션 중 하나를 선택합니다.
   - 암호화 설정 사용자 지정(고급)을 선택하여 AWS KMS key를 선택하거나 생성합니다.
   - 삭제 작업으로 인해 클러스터가 삭제되지 않도록 방지하려면, 삭제 방지 활성화를 선택합니다. 삭제 방지 기능은 기본적으로 선택됩니다.
5. 다중 리전 설정에서 첫 번째 클러스터에 대해 다음 옵션을 선택합니다.
   - 감시 리전에서 리전을 선택합니다. 현재 미국 기반 리전에서만 다중 리전 클러스터의 감시 리전이 지원됩니다.
   - (선택 사항) 원격 리전 클러스터 ARN에 다른 리전의 기존 클러스터에 대한 ARN을 입력합니다. 다중 리전 클러스터에서 두 번째 클러스터 역할을 할 클러스터가 없는 경우 첫 번째 클러스터를 생성한 후 설정을 완료합니다.
6. (선택 사항) 첫 번째 클러스터의 태그를 선택합니다.
7. 클러스터 생성을 선택하여 첫 번째 클러스터를 생성합니다. 이전 단계에서 ARN을 입력하지 않은 경우 콘솔에 클러스터 설정 보류 중 알림이 표시됩니다.
8. 클러스터 설정 보류 중 알림에서 다중 리전 클러스터 설정 완료를 선택합니다. 이 작업은 다른 리전에서 두 번째 클러스터 생성을 시작합니다.
9. 두 번째 클러스터에 대해 다음 옵션 중 하나를 선택합니다.

- 원격 리전 클러스터 ARN 추가 - 클러스터가 존재하고 다중 리전 클러스터의 두 번째 클러스터가 되도록 하려면 이 옵션을 선택합니다.
- 다른 리전에서 클러스터 생성 - 두 번째 클러스터를 생성하려면 이 옵션을 선택합니다. 원격 리전에서 이 두 번째 클러스터의 리전을 선택합니다.

10. ***your-second-region***에서 클러스터 생성을 선택합니다. 여기서 *your-second-region*은 두 번째 클러스터의 위치입니다. 콘솔이 두 번째 리전에서 열립니다.
11. (선택 사항) 두 번째 클러스터의 클러스터 설정을 선택합니다. 예를 들어 AWS KMS key를 선택할 수 있습니다.
12. 클러스터 생성을 선택하여 두 번째 클러스터를 생성합니다.
13. ***initial-cluster-region***에서 피어링를 선택합니다. 여기서 *initial-cluster-region*은 생성한 첫 번째 클러스터를 호스팅하는 리전입니다.
14. 메시지가 나타나면 확인을 선택합니다. 이 단계에서는 다중 리전 클러스터 생성을 완료합니다.

두 번째 클러스터에 연결하는 방법

1. Aurora DSQL 콘솔을 열고 두 번째 클러스터의 리전을 선택합니다.
2. 클러스터를 선택하세요.
3. 다중 리전 클러스터에서 두 번째 클러스터의 행을 선택합니다.
4. 작업에서 CloudShell에서 열기를 선택합니다.
5. 관리자로 연결을 선택합니다.
6. CloudShell 시작을 선택합니다.
7. Run(실행)을 선택합니다.
8. 3단계: Aurora DSQL에서 샘플 SQL 명령 실행의 단계에 따라 샘플 스키마를 생성합니다.

   트랜잭션 예시

   Example

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

9. `psql` `copy` 및 `include` 명령을 사용하여 샘플 데이터를 로드합니다. 자세한 내용은 3단계: Aurora DSQL에서 샘플 SQL 명령 실행 섹션을 참조하세요.

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

첫 번째 클러스터를 호스팅하는 리전에서 두 번째 클러스터의 데이터를 쿼리하는 방법

1. Aurora DSQL 콘솔에서 첫 번째 클러스터의 리전을 선택합니다.
2. 클러스터를 선택하세요.
3. 다중 리전 클러스터에서 두 번째 클러스터의 행을 선택합니다.
4. 작업에서 CloudShell에서 열기를 선택합니다.
5. 관리자로 연결을 선택합니다.
6. CloudShell 시작을 선택합니다.
7. Run(실행)을 선택합니다.
8. 두 번째 클러스터에 삽입한 데이터를 쿼리합니다.

   Example

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

Aurora DSQL에 대한 인증 및 권한 부여

Aurora DSQL은 클러스터 권한 부여에 IAM 역할 및 정책을 사용합니다. 데이터베이스 권한 부여를 위해 IAM 역할을 PostgreSQL 데이터베이스 역할과 연결합니다. 이 접근 방식은 IAM의 이점과 PostgreSQL 권한을 결합합니다. Aurora DSQL은 이러한 기능을 사용하여 클러스터, 데이터베이스 및 데이터에 대한 포괄적인 권한 부여 및 액세스 정책을 제공합니다.

IAM을 사용하여 클러스터 관리

클러스터를 관리하려면 인증 및 권한 부여에 IAM을 사용합니다.

IAM 인증.

Aurora DSQL 클러스터를 관리할 때 IAM ID를 인증하려면 IAM을 사용해야 합니다. AWS Management Console, AWS CLI 또는 AWS SDK를 사용하여 인증을 제공할 수 있습니다.

IAM 권한 부여

Aurora DSQL 클러스터를 관리하려면 Aurora DSQL에 대한 IAM 작업을 사용하여 권한을 부여합니다. 예를 들어 클러스터를 생성하려면 다음 샘플 정책 작업과 같이 IAM ID에 IAM 작업 `dsql:CreateCluster`에 대한 권한이 있는지 확인합니다.

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

자세한 내용은 IAM 정책 작업을 사용하여 클러스터 관리 섹션을 참조하세요.

IAM을 사용하여 클러스터에 연결

클러스터에 연결하려면 인증 및 권한 부여에 IAM을 사용합니다.

IAM 인증.

클러스터에 연결할 권한이 있는 IAM ID를 사용하여 임시 인증 토큰을 생성합니다. 자세한 내용은 Amazon Aurora DSQL에서 인증 토큰 생성를 참조하세요.

IAM 권한 부여

클러스터의 엔드포인트에 대한 연결을 설정하는 데 사용하는 IAM ID에 다음 IAM 정책 작업을 부여합니다.

- admin 역할을 사용하는 경우 dsql:DbConnectAdmin을 사용합니다. Aurora DSQL은 이 역할을 자동으로 생성하고 관리합니다. 다음 샘플 IAM 정책 작업은 admin이 *my-cluster*에 연결하도록 허용합니다.

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

- 사용자 지정 데이터베이스 역할을 사용하는 경우 dsql:DbConnect를 사용합니다. 데이터베이스에서 SQL 명령을 사용하여 이 역할을 생성하고 관리합니다. 다음 샘플 IAM 정책 작업은 사용자 지정 데이터베이스 역할이 최대 1시간 동안 *my-cluster*에 연결하도록 허용합니다.

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

연결을 설정하면 역할에 최대 1시간 동안 연결 권한이 부여됩니다.

PostgreSQL 데이터베이스 역할 및 IAM 역할을 사용하여 데이터베이스와 상호 작용

PostgreSQL은 역할 개념을 사용하여 데이터베이스 액세스 권한을 관리합니다. 역할은 역할 설정 방식에 따라 데이터베이스 사용자 또는 데이터베이스 사용자 그룹으로 생각할 수 있습니다. SQL 명령을 사용하여 PostgreSQL 역할을 생성합니다. 데이터베이스 수준 권한 부여를 관리하려면 PostgreSQL 데이터베이스 역할에 PostgreSQL 권한을 부여합니다.

Aurora DSQL은 admin 역할 및 사용자 지정 역할이라는 두 가지 유형의 데이터베이스 역할을 지원합니다. Aurora DSQL은 Aurora DSQL 클러스터에 사전 정의된 admin 역할을 자동으로 생성합니다. admin 역할은 수정할 수 없습니다. admin 역할로 데이터베이스에 연결하면 SQL을 실행하여 IAM 역

할과 연결할 새 데이터베이스 수준 역할을 생성할 수 있습니다. IAM 역할이 데이터베이스에 연결되도록 하려면 사용자 지정 데이터베이스 역할을 IAM 역할과 연결합니다.

인증

admin 역할을 사용하여 클러스터에 연결합니다. 데이터베이스를 연결한 후 다음 예시와 같이 AWS IAM GRANT 명령을 사용하여 사용자 지정 데이터베이스 역할을 클러스터에 연결할 권한이 있는 IAM ID와 연결합니다.

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

자세한 내용은 클러스터에 연결할 데이터베이스 역할 권한 부여를 참조하세요.

권한 부여

admin 역할을 사용하여 클러스터에 연결합니다. SQL 명령을 실행하여 사용자 지정 데이터베이스 역할을 설정하고 권한을 부여합니다. 자세한 내용은 PostgreSQL 설명서의 PostgreSQL database roles 및 PostgreSQL privileges를 참조하세요.

Aurora DSQL에서 IAM 정책 작업 사용

사용하는 IAM 정책 작업은 클러스터에 연결하는 데 사용하는 역할이 admin인지 사용자 지정 데이터베이스 역할인지에 따라 달라집니다. 또한 정책은 이 역할에 필요한 IAM 작업에 따라 달라집니다.

IAM 정책 작업을 사용하여 클러스터에 연결

기본 데이터베이스 역할인 admin을 사용하여 클러스터에 연결할 때 다음 IAM 정책 작업을 수행할 권한이 있는 IAM ID를 사용합니다.

```
"dsql:DbConnectAdmin"
```

사용자 지정 데이터베이스 역할을 사용하여 클러스터에 연결할 때 먼저 IAM 역할을 데이터베이스 역할과 연결합니다. 클러스터에 연결하는 데 사용하는 IAM ID에는 다음 IAM 정책 작업을 수행할 수 있는 권한이 있어야 합니다.

```
"dsql:DbConnect"
```

사용자 지정 데이터베이스 역할에 대한 자세한 내용은 데이터베이스 역할 및 IAM 인증 사용 섹션을 참조하세요.

IAM 정책 작업을 사용하여 클러스터 관리

Aurora DSQL 클러스터를 관리할 때 역할이 수행해야 하는 작업에 대해서만 정책 작업을 지정합니다. 예를 들어, 역할이 클러스터 정보를 가져오기만 해야 하는 경우 다음 샘플 정책과 같이 역할 권한을 `GetCluster` 및 `ListClusters` 권한으로만 제한할 수 있습니다.

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

다음 예시 정책은 클러스터 관리에 사용할 수 있는 모든 IAM 정책 작업을 보여줍니다.

```
{
"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 및 PostgreSQL을 사용하여 권한 부여 취소

IAM 역할이 데이터베이스 수준 역할에 액세스할 수 있는 권한을 취소할 수 있습니다.

클러스터에 연결하는 관리자 권한 부여 취소

admin 역할을 클러스터에 연결할 수 있는 권한을 취소하려면 dsql:DbConnectAdmin에 대한 IAM ID의 액세스를 취소합니다. IAM 정책을 편집하거나 ID에서 정책을 분리합니다.

IAM ID에서 연결 권한 부여를 취소한 후 Aurora DSQL은 해당 IAM ID의 모든 새 연결 시도를 거부합니다. IAM ID를 사용하는 모든 활성 연결은 연결 기간 동안 권한을 유지할 수 있습니다. 한도와 할당량에 대한 자세한 내용은 Quotas and limits를 참조하세요.

클러스터에 연결하는 사용자 지정 권한 부여 취소

admin 외의 데이터베이스 역할에 대한 액세스를 취소하려면 dsql:DbConnect에 대한 IAM ID의 액세스를 취소합니다. IAM 정책을 편집하거나 ID에서 정책을 분리합니다.

데이터베이스에서 AWS IAM REVOKE 명령을 사용하여 데이터베이스 역할과 IAM 간의 연결을 제거할 수도 있습니다. 데이터베이스 역할에서 액세스를 취소하는 방법에 대한 자세한 내용은 IAM 역할에서 데이터베이스 권한 부여 취소 섹션을 참조하세요.

사전 정의된 admin 데이터베이스 역할의 권한은 관리할 수 없습니다. 사용자 지정 데이터베이스 역할에 대한 권한을 관리하는 방법을 알아보려면 PostgreSQL privileges를 참조하세요. 권한 수정은 Aurora DSQL이 수정 트랜잭션을 성공적으로 커밋한 후 다음 트랜잭션에 적용됩니다.

Amazon Aurora DSQL에서 인증 토큰 생성

SQL 클라이언트를 사용하여 Amazon Aurora DSQL에 연결하려면 암호로 사용할 인증 토큰을 생성합니다. 이 토큰은 연결을 인증하는 데만 사용됩니다. 연결이 설정되면 인증 토큰이 만료되더라도 연결이 유효합니다.

AWS Console을 사용하여 인증 토큰을 생성하면 토큰은 기본적으로 1시간 후에 자동으로 만료됩니다. AWS CLI 또는 SDK를 사용하여 토큰을 생성하는 경우 기본값은 15분입니다. 최대 기간은 1주일인 604,800초입니다. 클라이언트에서 Aurora DSQL에 다시 연결하려면 만료되지 않은 동일한 인증 토큰을 사용하거나 새 토큰을 생성할 수 있습니다.

토큰 생성을 시작하려면 Aurora DSQL에서 IAM 정책과 클러스터를 생성합니다. 그런 다음 AWS Console, AWS CLI 또는 AWS SDK를 사용하여 토큰을 생성합니다.

연결하는 데 사용하는 데이터베이스 역할에 따라 최소한 IAM을 사용하여 클러스터에 연결에 나열된 IAM 권한이 있어야 합니다.

주제

- AWS Console을 사용하여 Aurora DSQL에서 인증 토큰 생성
- AWS CloudShell을 사용하여 Aurora DSQL에서 인증 토큰 생성
- AWS CLI를 사용하여 Aurora DSQL에서 인증 토큰 생성
- SDK를 사용하여 Aurora DSQL에서 토큰 생성

AWS Console을 사용하여 Aurora DSQL에서 인증 토큰 생성

Aurora DSQL은 암호가 아닌 토큰으로 사용자를 인증합니다. 콘솔에서 토큰을 생성할 수 있습니다.

인증 토큰을 생성하는 방법

1. AWS Management Console에 로그인한 다음, https://console.aws.amazon.com/dsql에서 Aurora DSQL 콘솔을 엽니다.
2. 인증 토큰을 생성할 클러스터의 클러스터 ID를 선택합니다. 클러스터를 아직 생성하지 않은 경우 1단계: Aurora DSQL 단일 리전 클러스터 생성 또는 4단계: 다중 리전 클러스터 생성의 단계를 따릅니다.
3. 연결을 선택한 다음 토큰 가져오기를 선택합니다.
4. `admin`으로 연결할지, 아니면 사용자 지정 데이터베이스 역할로 연결할지를 선택합니다.
5. 생성된 인증 토큰을 복사하여 SQL 클라이언트를 사용하여 Aurora DSQL에 액세스에 사용합니다.

Aurora DSQL의 사용자 지정 데이터베이스 역할 및 IAM에 대한 자세한 내용은 인증 및 권한 부여 섹션을 참조하세요.

AWS CloudShell을 사용하여 Aurora DSQL에서 인증 토큰 생성

AWS CloudShell을 사용하여 인증 토큰을 생성하려면 먼저 다음 작업을 수행해야 합니다.

- Aurora DSQL 클러스터를 생성합니다.
- `get-object` Amazon S3 작업을 실행하여 조직 외부의 AWS 계정에서 객체를 검색할 수 있는 권한을 추가합니다. 자세한 내용은 Amazon S3 사용 설명서를 참조하세요.

AWS CloudShell을 사용하여 인증 토큰을 생성하는 방법

1. AWS Management Console에 로그인한 다음, https://console.aws.amazon.com/dsql에서 Aurora DSQL 콘솔을 엽니다.
2. AWS Console 왼쪽 하단에서 AWS CloudShell을 선택합니다.
3. AWS CLI를 설치하려면 최신 버전의 AWS CLI 설치 또는 업데이트에 나와 있는 단계를 따르세요.

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

4. 다음 명령을 실행하여 admin 역할에 대한 인증 토큰을 생성합니다. *us-east-1*을 실제 리전으로 바꾸고 *your_cluster_endpoint*를 실제 클러스터의 엔드포인트로 바꿉니다.

> **Note**
>
> admin으로 연결하지 않는 경우 generate-db-connect-auth-token을 대신 사용합니다.

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

문제가 발생하면 IAM 문제 해결 및 IAM 정책을 적용할 때 액세스 거부 또는 승인되지 않은 작업 오류를 해결하려면 어떻게 해야 합니까?를 참조하세요.

5. 다음 명령을 사용하여 psql을 사용해 클러스터에 대한 연결을 시작합니다.

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

6. 암호를 입력하라는 프롬프트가 표시됩니다. 생성한 토큰을 복사하고 추가 공백이나 문자를 포함하지 않도록 합니다. psql의 다음 프롬프트에 붙여 넣습니다.

```
Password for user admin:
```

7. Enter를 누릅니다. PostgreSQL 프롬프트가 표시됩니다.

```
postgres=>
```

액세스 거부 오류가 발생하면 IAM ID에 `dsql:DbConnectAdmin` 권한이 있는지 확인합니다. 권한이 있고 액세스 거부 오류가 계속 발생하는 경우 IAM 문제 해결 및 IAM 정책을 적용할 때 액세스 거부 또는 승인되지 않은 작업 오류를 해결하려면 어떻게 해야 합니까?를 참조하세요.

Aurora DSQL의 사용자 지정 데이터베이스 역할 및 IAM에 대한 자세한 내용은 인증 및 권한 부여 섹션을 참조하세요.

AWS CLI를 사용하여 Aurora DSQL에서 인증 토큰 생성

클러스터가 `ACTIVE`인 경우 `aws dsql` 명령을 사용하여 CLI에서 인증 토큰을 생성할 수 있습니다. 다음 중 한 가지 방법을 사용합니다.

- `admin` 역할로 연결하는 경우 `generate-db-connect-admin-auth-token` 옵션을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 `generate-db-connect-auth-token` 옵션을 사용합니다.

다음 예시에서는 다음 속성을 사용하여 `admin` 역할에 대한 인증 토큰을 생성합니다.

- *`your_cluster_endpoint`* - 클러스터의 엔드포인트입니다. 예시 `01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws`와 같이 *`your_cluster_identifier`*`.dsql.`*`region`*`.on.aws` 형식을 따릅니다.
- *`region`* - `us-east-2` 또는 `us-east-1`과 같은 AWS 리전입니다.

다음 예시에서는 토큰이 3,600초(1시간) 후에 만료되도록 만료 시간을 설정합니다.

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
```

SDK를 사용하여 Aurora DSQL에서 토큰 생성

클러스터가 ACTIVE 상태일 때 클러스터에 대한 인증 토큰을 생성할 수 있습니다. SDK 예시에서는 다음 속성을 사용하여 admin 역할에 대한 인증 토큰을 생성합니다.

- *your_cluster_endpoint*(또는 *yourClusterEndpoint*) - Aurora DSQL 클러스터의 엔드포인트입니다. 예시 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws와 같이 이름 지정 형식은 *your_cluster_identifier*.dsql.*region*.on.aws입니다.
- *region*(또는 *RegionEndpoint*) - us-east-2 또는 us-east-1과 같이 클러스터가 위치한 AWS 리전입니다.

Python SDK

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 generate_db_connect_admin_auth_token을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 generate_connect_auth_token을 사용합니다.

```
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

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 GenerateDBConnectAdminAuthToken을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 GenerateDBConnectAuthToken을 사용합니다.

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

using namespace Aws;
using namespace Aws::DSQL;

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

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

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

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

JavaScript SDK

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 getDbConnectAdminAuthToken을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 getDbConnectAuthToken을 사용합니다.

```
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

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 generateDbConnectAdminAuthToken을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 generateDbConnectAuthToken을 사용합니다.

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

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

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

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

Rust SDK

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 db_connect_admin_auth_token을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 db_connect_auth_token을 사용합니다.

```
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

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 generate_db_connect_admin_auth_token을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 generate_db_connect_auth_token을 사용합니다.

```
require 'aws-sdk-dsql'

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

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

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

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

.NET

> **Note**
>
> 공식 SDK for .NET에는 Aurora DSQL에 대한 인증 토큰을 생성하기 위한 기본 제공 API 직접 호출이 포함되어 있지 않습니다. 대신 유틸리티 클래스인 DSQLAuthTokenGenerator를 사용해야 합니다. 다음 코드 샘플은 .NET에 대한 인증 토큰을 생성하는 방법을 보여줍니다.

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 DbConnectAdmin을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 DbConnect을 사용합니다.

다음 예시에서는 DSQLAuthTokenGenerator 유틸리티 클래스를 사용하여 admin 역할이 있는 사용자의 인증 토큰을 생성합니다. *insert-dsql-cluster-endpoint*를 실제 클러스터 엔드포인트로 바꿉니다.

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

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

AWSCredentials credentials = FallbackCredentialsFactory.GetCredentials();

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

Console.WriteLine(token);
```

Golang

> **Note**
>
> Golang SDK는 미리 서명된 토큰을 생성하기 위한 기본 제공 메서드를 제공하지 않습니다. 다음 코드 예시와 같이 서명된 요청을 수동으로 구성해야 합니다.

다음 코드 예시에서는 PostgreSQL 사용자를 기반으로 `action`을 지정합니다.

- `admin` 역할로 연결하는 경우 `DbConnectAdmin` 작업을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 `DbConnect` 작업을 사용합니다.

다음 예시에서는 *`yourClusterEndpoint`* 및 *`region`* 외에도 *`action`*을 사용합니다. PostgreSQL 사용자를 기반으로 *`action`*을 지정합니다.

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

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

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

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

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

 return url, nil
}
```

데이터베이스 역할 및 IAM 인증 사용

Aurora DSQL은 IAM 역할과 IAM 사용자를 사용한 인증을 둘 다 지원합니다. 두 방법 중 하나를 사용하여 Aurora DSQL 데이터베이스를 인증하고 액세스할 수 있습니다.

IAM 역할

IAM 역할은 특정 권한이 있지만 특정 사용자와 연결되지 않은 AWS 계정 내의 ID입니다. IAM 역할을 사용하면 임시 보안 자격 증명이 제공됩니다. 다음과 같은 여러 가지 방법으로 IAM 역할을 일시적으로 수임할 수 있습니다.

- AWS Management Console에서 역할 전환
- AWS CLI 또는 AWS API 작업 직접 호출
- 사용자 지정 URL 사용

역할을 수임한 후 역할의 임시 자격 증명을 사용하여 Aurora DSQL에 액세스할 수 있습니다. 역할 사용 방법에 대한 자세한 정보는 IAM 사용 설명서의 IAM 자격 증명을 참조하세요.

IAM 사용자

IAM 사용자는 특정 권한을 가졌으며 한 명의 개인 또는 하나의 애플리케이션과 연결된 AWS 계정 내 ID입니다. IAM 사용자에게는 Aurora DSQL에 액세스하는 데 사용할 수 있는 암호 및 액세스 키와 같은 장기 자격 증명이 있습니다.

> **Note**
>
> IAM 인증으로 SQL 명령을 실행하려면 아래 예시에서 IAM 역할 ARN 또는 IAM 사용자 ARN을 사용할 수 있습니다.

클러스터에 연결할 데이터베이스 역할 권한 부여

IAM 역할을 생성하고 IAM 정책 작업 `dsql:DbConnect`를 사용하여 연결 권한을 부여합니다.

IAM 정책은 클러스터 리소스에 액세스할 수 있는 권한도 부여해야 합니다. 와일드카드(*)를 사용하거나 Amazon Aurora DSQL에서 IAM 조건 키 사용의 지침을 따릅니다.

데이터베이스 역할이 데이터베이스에서 SQL을 사용하도록 권한 부여

클러스터에 연결할 권한이 있는 IAM 역할을 사용해야 합니다.

1. SQL 유틸리티를 사용하여 Aurora DSQL 클러스터에 연결합니다.

   IAM 작업 `dsql:DbConnectAdmin`이 클러스터에 연결할 수 있도록 권한이 부여된 IAM ID와 함께 `admin` 데이터베이스 역할을 사용합니다.
2. `WITH LOGIN` 옵션을 지정하여 새 데이터베이스 역할을 생성합니다.

   ```
   CREATE ROLE example WITH LOGIN;
   ```
3. 데이터베이스 역할을 IAM 역할 ARN과 연결합니다.

   ```
   AWS IAM GRANT example TO 'arn:aws:iam::012345678912:role/example';
   ```
4. 데이터베이스 역할에 데이터베이스 수준 권한 부여

   다음 예시에서는 GRANT 명령을 사용하여 데이터베이스 내에서 권한 부여를 제공합니다.

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

자세한 내용은 PostgreSQL 설명서에서 PostgreSQL GRANT 및 PostgreSQL Privileges를 참조하세요.

IAM 역할에서 데이터베이스 권한 부여 취소

데이터베이스 권한 부여를 취소하려면 `AWS IAM REVOKE` 작업을 사용합니다.

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

권한 부여 취소에 대한 자세한 내용은 IAM 및 PostgreSQL을 사용하여 권한 부여 취소 섹션을 참조하세요.

Aurora DSQL 및 PostgreSQL

Aurora DSQL은 트랜잭션 워크로드용으로 설계된 PostgreSQL 호환 분산 관계형 데이터베이스입니다. Aurora DSQL은 구문 파서, 플래너, 옵티마이저 및 유형 시스템과 같은 핵심 PostgreSQL 구성 요소를 사용합니다.

Aurora DSQL 설계는 지원되는 모든 PostgreSQL 구문이 호환되는 동작을 제공하고 동일한 쿼리 결과를 산출하도록 보장합니다. 예를 들어 Aurora DSQL은 PostgreSQL과 동일한 유형 변환, 산술 연산, 수치 정밀도 및 규모를 제공합니다. 모든 편차가 문서화됩니다.

또한 Aurora DSQL에는 낙관적 동시성 제어 및 분산 스키마 관리와 같은 고급 기능이 도입되었습니다. 이러한 기능을 사용하면 현대적 클라우드 네이티브 분산 애플리케이션에 필요한 성능과 확장성을 활용하면서 친숙한 PostgreSQL 도구를 사용할 수 있습니다.

PostgreSQL 호환성 하이라이트

Aurora DSQL은 현재 PostgreSQL 버전 16을 기반으로 합니다. 주요 호환성은 다음과 같습니다.

와이어 프로토콜

Aurora DSQL은 표준 PostgreSQL v3 와이어 프로토콜을 사용합니다. 이를 통해 표준 PostgreSQL 클라이언트, 드라이버 및 도구와 통합할 수 있습니다. 예를 들어 Aurora DSQL은 `psql`, `pgjdbc` 및 `psycopg`와 호환됩니다.

SQL 호환성

Aurora DSQL은 트랜잭션 워크로드에 일반적으로 사용되는 다양한 표준 PostgreSQL 표현식 및 함수를 지원합니다. 지원되는 SQL 표현식은 다음을 포함하여 PostgreSQL과 동일한 결과를 산출합니다.

- Null 처리
- 정렬 순서 동작
- 숫자 연산에 대한 규모 및 정밀도
- 문자열 작업의 동등성

자세한 내용은 Aurora DSQL의 SQL 기능 호환성 섹션을 참조하세요.

트랜잭션 관리

Aurora DSQL은 ACID 트랜잭션 및 PostgreSQL Repeatable Read와 동등한 격리 수준과 같은 PostgreSQL의 기본 특성을 보존합니다. 자세한 내용은 Aurora DSQL의 동시성 제어 섹션을 참조하세요.

아키텍처의 주요 차이점

Aurora DSQL의 분산된 비공유 설계로 인해 기존 PostgreSQL과 몇 가지 근본적인 차이점이 있습니다. 이러한 차이점은 Aurora DSQL 아키텍처에 필수적이며 다양한 성능적 이점과 확장성 이점을 제공합니다. 주요 차이점은 다음과 같습니다.

낙관적 동시성 제어(OCC)

Aurora DSQL은 낙관적 동시성 제어 모델을 사용합니다. 이 비잠금 접근 방식은 트랜잭션이 서로 차단하는 것을 방지하고 교착 상태를 제거하며 처리량이 많은 병렬 실행을 활성화합니다. 이러한 기능 덕분에 Aurora DSQL은 대규모로 일관된 성능이 필요한 애플리케이션에 특히 유용합니다. 더 많은 예시는 Aurora DSQL의 동시성 제어 섹션을 참조하세요.

비동기식 DDL 작업

Aurora DSQL은 DDL 작업을 비동기적으로 실행하므로 스키마 변경 중에 중단 없는 읽기 및 쓰기가 가능합니다. 분산 아키텍처를 통해 Aurora DSQL은 다음 작업을 수행할 수 있습니다.

- DDL 작업을 백그라운드 작업으로 실행하여 중단을 최소화합니다.
- 카탈로그 변경 사항을 강력히 일관된 분산 트랜잭션으로 조정합니다. 이렇게 하면 장애 또는 동시 작업 중에도 모든 노드에서 원자성 가시성을 보장할 수 있습니다.
- 분리된 컴퓨팅 및 스토리지 계층을 사용하여 여러 가용 영역에서 완전 분산된 리더리스 방식으로 작업합니다.

자세한 내용은 Aurora DSQL의 DDL 및 분산 트랜잭션 섹션을 참조하세요.

Aurora DSQL의 SQL 기능 호환성

Aurora DSQL과 PostgreSQL은 모든 SQL 쿼리에 대해 동일한 결과를 반환합니다. Aurora DSQL은 `ORDER BY` 절이 없는 PostgreSQL과 다릅니다. 이어지는 섹션에서는 PostgreSQL 데이터 유형 및 SQL 명령에 대한 Aurora DSQL 지원에 대해 알아봅니다.

주제

- Aurora DSQL에서 지원되는 데이터 유형
- Aurora DSQL에 지원되는 SQL
- Aurora DSQL에서 지원되는 SQL 명령 하위 집합
- Aurora DSQL에서 지원되지 않는 PostgreSQL 기능

Aurora DSQL에서 지원되는 데이터 유형

Aurora DSQL은 일반적인 PostgreSQL 유형의 하위 집합을 지원합니다.

주제

- 숫자 데이터 유형
- 문자 데이터 유형
- 날짜 및 시간 데이터 유형
- 기타 데이터 유형
- 쿼리 런타임 데이터 유형

숫자 데이터 유형

Aurora DSQL은 다음 PostgreSQL 숫자 데이터 유형을 지원합니다.

명칭	에일리어스	범위 및 정밀도	스토리지 크기	인덱스 지원
`smallint`	`int2`	2 bytes	2 bytes	예
`integer`	`int`, `int4`	4 bytes	4비트	예
`bigint`	`int8`	-9,223,372,036,854,775,808~+9,223,372,036,854,775,807	8 bytes	예
`real`	`float4`	소수 자릿수 6까지의 정밀도	4비트	예

명칭	에일리어스	범위 및 정밀도	스토리지 크기	인덱스 지원
`double precision`	`float8`	소수 자릿수 15까지의 정밀도	8 bytes	예

`numeric [ (p, s) ]`	`decimal [ (p, s) ]`
	
	`dec[ (p,s)]`
		정밀도를 선택할 수 있는 정확한 숫자. 최대 정밀도는 38이며 최대 범위는 37이고[2] 기본값은 `numeric (18,6)`입니다.	정밀도 숫자당 8바이트 + 2바이트. 최대 크기는 27바이트입니다.	아니요

[2] - `CREATE TABLE` 또는 `ALTER TABLE ADD COLUMN`을 실행할 때 크기를 명시적으로 지정하지 않으면 Aurora DSQL이 기본값을 적용합니다. `INSERT` 또는 `UPDATE` 문을 실행할 때 Aurora DSQL은 제한을 적용합니다.

문자 데이터 유형

Aurora DSQL은 다음 PostgreSQL 문자 데이터 유형을 지원합니다.

명칭	에일리어스	설명	Aurora DSQL 제한	스토리지 크기	인덱스 지원
`character [ (n) ]`	`char [ (n) ]`	고정 길이 문자열	4,096B[1]	최대 4,100바이트까지 가변	예
`character varying [ (n) ]`	`varchar [ (n) ]`	가변 길이 문자열	65,535B[1]	최대 65,539바이트까지 가변	예
`bpchar [ (n) ]`		고정 길이인 경우 `char`의 별칭입니다. 가변 길이인 경우 이는 후행 공백이 의미상 유의하지	4,096B[1]	최대 4,100바이트까지 가변	예

명칭	에일리어스	설명	Aurora DSQL 제한	스토리지 크기	인덱스 지원
		않은 varchar의 별칭입니다.			
text		가변 길이 문자열	1MiB[1]	최대 1MiB까지 가변	예

1 - CREATE TABLE 또는 ALTER TABLE ADD COLUMN을 실행할 때 크기를 명시적으로 지정하지 않으면 Aurora DSQL이 기본값을 적용합니다. INSERT 또는 UPDATE 문을 실행할 때 Aurora DSQL은 제한을 적용합니다.

날짜 및 시간 데이터 유형

Aurora DSQL은 다음 PostgreSQL 날짜 및 시간 데이터 유형을 지원합니다.

명칭	에일리어스	설명	Range	해결 방법	스토리지 크기	인덱스 지원
date		날짜(년, 월, 일)	4713 BC~5874897 AD	1일	4비트	예
time [(*p*)] [without time zone]	time	시간대가 없는 시간	0~1	1마이크로초	8 bytes	예
time [(*p*)] with time zone	time	시간대를 포함한 시간	00:00:00+1,559~24:00:00 –1,559	1마이크로초	12바이트	아니요
timestamp [(*p*)] [without time zone]		시간대가 없는 날짜 및 시간	4713 BC~294276 AD	1마이크로초	8 bytes	예

명칭	에일리어스	설명	Range	해결 방법	스토리지 크기	인덱스 지원
`timestamp [(p)] with time zone`	`timestz`	시간대를 포함한 날짜 및 시간	4713 BC~294276 AD	1마이크로초	8 bytes	예
`interval [fields] [(p)]`		시간 범위	-178,000,000년~178,000,000년	1마이크로초	16바이트	아니요

기타 데이터 유형

Aurora DSQL은 다음과 같은 기타 PostgreSQL 데이터 유형을 지원합니다.

명칭	에일리어스	설명	Aurora DSQL 제한	스토리지 크기	인덱스 지원
`boolean`	`bool`	논리적 부울(true/false)		1바이트	예
`bytea`		바이너리 데이터('바이트 배열')	1MiB[1]	최대 1MiB까지 가변	아니요
`UUID`		범용 고유 식별자		16바이트	예

[1] - `CREATE TABLE` 또는 `ALTER TABLE ADD COLUMN`을 실행할 때 크기를 명시적으로 지정하지 않으면 Aurora DSQL이 기본값을 적용합니다. `INSERT` 또는 `UPDATE` 문을 실행할 때 Aurora DSQL은 제한을 적용합니다.

쿼리 런타임 데이터 유형

쿼리 런타임 데이터 유형은 쿼리 실행 시점에 사용되는 내부 데이터 유형입니다. 이러한 유형은 스키마에서 정의하는 `varchar` 및 `integer`와 같은 PostgreSQL 호환 유형과 다릅니다. 대신 이러한 유형은 Aurora DSQL이 쿼리를 처리할 때 사용하는 런타임 표현입니다.

다음 데이터 유형은 쿼리 런타임 중에만 지원됩니다.

배열 유형

Aurora DSQL은 지원되는 데이터 유형의 배열을 지원합니다. 예를 들어 정수 배열이 있을 수 있습니다. `string_to_array` 함수는 다음 예시처럼 쉼표 구분 기호(`,`)를 사용하여 문자열을 PostgreSQL 스타일 배열로 분할합니다. 쿼리 실행 중에 표현식, 함수 출력 또는 임시 계산에 배열을 사용할 수 있습니다.

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

이 함수는 다음과 유사한 응답을 반환합니다.

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

inet 유형

이 데이터 유형은 IPv4, IPv6 호스트 주소 및 해당 서브넷을 나타냅니다. 이 유형은 로그를 구문 분석하거나, IP 서브넷에서 필터링하거나, 쿼리 내에서 네트워크 계산을 수행할 때 유용합니다. 자세한 내용은 PostgreSQL 설명서의 inet을 참조하세요.

Aurora DSQL에 지원되는 SQL

Aurora DSQL은 다양한 핵심 PostgreSQL SQL 기능을 지원합니다. 이어지는 섹션에서는 일반적인 PostgreSQL 표현식 지원에 대해 알아볼 수 있습니다. 단, 이 목록이 전부는 아닙니다.

SELECT 명령

Aurora DSQL은 `SELECT` 명령의 다음 절을 지원합니다.

기본 절	지원 절
FROM	
GROUP BY	ALL, DISTINCT
ORDER BY	ASC, DESC, NULLS
LIMIT	
DISTINCT	
HAVING	
USING	
WITH(공통 테이블 표현식)	
INNER JOIN	ON
OUTER JOIN	LEFT, RIGHT, FULL, ON
CROSS JOIN	ON
UNION	ALL
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR UPDATE	

데이터 정의 언어(DDL)

Aurora DSQL은 다음과 같은 PostgreSQL DDL 명령을 지원합니다.

Command	기본 절	지원 절
CREATE	TABLE	CREATE TABLE 명령의 지원되는 구문에 대한 자세한 내용은 CREATE TABLE 섹션을 참조하세요.
ALTER	TABLE	ALTER TABLE 명령의 지원되는 구문에 대한 자세한 내용은 ALTER TABLE 섹션을 참조하세요.
DROP	TABLE	
CREATE	[UNIQUE] INDEX ASYNC	이 명령과 ON, NULLS FIRST, NULLS LAST 파라미터를 함께 사용할 수 있습니다. CREATE INDEX ASYNC 명령의 지원되는 구문에 대한 자세한 내용은 Aurora DSQL의 비동기 인덱스 섹션을 참조하세요.
DROP	INDEX	
CREATE	VIEW	CREATE VIEW 명령의 지원되는 구문에 대한 자세한 내용은 CREATE VIEW 섹션을 참조하세요.
ALTER	VIEW	ALTER VIEW 명령의 지원되는 구문에 대한 자세한 내용은 ALTER VIEW 섹션을 참조하세요.
DROP	VIEW	DROP VIEW 명령의 지원되는 구문에 대한 자세한 내용은 DROP VIEW 섹션을 참조하세요.
CREATE	ROLE, WITH	
CREATE	FUNCTION	LANGUAGE SQL
CREATE	DOMAIN	

데이터 조작 언어(DML)

Aurora DSQL은 다음과 같은 PostgreSQL DML 명령을 지원합니다.

Command	기본 절	지원 절
INSERT	INTO	VALUES SELECT
UPDATE	SET	WHERE (SELECT) FROM, WITH
DELETE	FROM	USING, WHERE

데이터 제어 언어(DCL)

Aurora DSQL은 다음과 같은 PostgreSQL DCL 명령을 지원합니다.

Command	지원 절
GRANT	ON, TO
REVOKE	ON, FROM, CASCADE, RESTRICT

트랜잭션 제어 언어(TCL)

Aurora DSQL은 다음과 같은 PostgreSQL TCL 명령을 지원합니다.

Command	지원 절
COMMIT	
BEGIN	[WORK | TRANSACTION] [READ ONLY | READ WRITE]

유틸리티 명령

Aurora DSQL은 다음과 같은 PostgreSQL 유틸리티 명령을 지원합니다.

- `EXPLAIN`
- `ANALYZE`(관계 이름만 해당)

Aurora DSQL에서 지원되는 SQL 명령 하위 집합

이 PostgreSQL 섹션에서는 광범위한 파라미터 세트 및 하위 명령이 있는 명령에 중점을 두고 지원되는 표현식에 대한 자세한 정보를 제공합니다. 예를 들어 PostgreSQL의 CREATE TABLE은 많은 절과 파라미터를 제공합니다. 이 섹션에서는 Aurora DSQL이 이러한 명령에 대해 지원하는 PostgreSQL 구문 요소를 모두 설명합니다.

주제

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

`CREATE TABLE`

`CREATE TABLE`은 새 테이블을 정의합니다.

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

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression )|
```

```
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] index_parameters |
  PRIMARY KEY index_parameters |

and table_constraint is:

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

and like_option is:

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

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

ALTER TABLE

ALTER TABLE은 테이블의 정의를 변경합니다.

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

where action is one of:

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

CREATE VIEW

CREATE VIEW는 새로운 영구 뷰를 정의합니다. Aurora DSQL은 임시 뷰를 지원하지 않으며 영구 뷰만 지원됩니다.

지원되는 구문

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

설명

CREATE VIEW는 쿼리 뷰를 정의합니다. 이 뷰는 물리적으로 구체화되지 않습니다. 대신 쿼리에서 뷰가 참조될 때마다 쿼리가 실행됩니다.

CREATE or REPLACE VIEW는 비슷하지만 동일한 이름의 뷰가 이미 있는 경우 대체됩니다. 새 쿼리는 기존 뷰 쿼리에서 생성된 것과 동일한 열(즉, 동일한 순서로 동일한 데이터 유형을 가진 동일한 열 이름)을 생성해야 하지만 목록 끝에 추가 열을 추가할 수 있습니다. 출력 열을 생성하는 계산은 다를 수 있습니다.

스키마 이름이 지정되어 있는 경우(예: CREATE VIEW myschema.myview ...), 뷰는 지정된 스키마에서 생성됩니다. 그렇지 않으면, 뷰가 현재 스키마에서 생성됩니다.

뷰의 이름은 동일한 스키마에 있는 다른 관계(테이블, 인덱스, 뷰)의 이름과 달라야 합니다.

파라미터

CREATE VIEW는 다양한 파라미터를 지원하여 자동으로 업데이트 가능한 뷰의 동작을 제어합니다.

RECURSIVE

재귀 뷰를 생성합니다. CREATE RECURSIVE VIEW [schema .] view_name (column_names) AS SELECT ...; 구문은 CREATE VIEW [schema .] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...) SELECT column_names FROM view_name;과 동일합니다.

재귀 뷰에 대해 뷰 열 이름 목록을 지정해야 합니다.

name

생성할 뷰의 이름으로, 선택적으로 스키마가 지정될 수 있습니다. 재귀 뷰에 열 이름 목록을 지정해야 합니다.

column_name

뷰에서 열에 사용할 이름의 선택적 목록입니다. 지정되지 않은 경우에는 열 이름이 쿼리에서 추론됩니다.

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

이 절은 뷰에 대한 선택적 파라미터를 지정합니다. 다음 파라미터가 지원됩니다.

- check_option (enum) -이 파라미터는 local 또는 cascaded일 수 있으며 WITH [CASCADED | LOCAL] CHECK OPTION 지정과 동일합니다.
- security_barrier (boolean) - 뷰가 행 수준 보안을 제공해야 하는 경우 이 옵션을 사용해야 합니다. Aurora DSQL은 현재 행 수준 보안을 지원하지 않지만 이 옵션은 뷰의 WHERE 조건(및 LEAKPROOF로 표시된 연산자를 사용하는 모든 조건)을 먼저 평가하도록 강제합니다.
- security_invoker (boolean) -이 옵션을 사용하면 뷰 소유자가 아닌 뷰 사용자의 권한을 기준으로 기본 관계를 확인할 수 있습니다. 자세한 내용은 아래 참고 사항을 참조하세요.

위의 모든 옵션은 기존 뷰에서 ALTER VIEW를 사용하여 변경할 수 있습니다.

query

뷰의 열과 행을 제공하는 SELECT 또는 VALUES 명령입니다.

- WITH [CASCADED | LOCAL] CHECK OPTION -이 옵션은 자동으로 업데이트 가능한 뷰의 동작을 제어합니다. 이 옵션을 지정하면 새 행이 뷰 정의 조건을 충족하는지 확인하기 위해 뷰의 INSERT 및 UPDATE 명령이 확인됩니다(즉, 새 행이 뷰를 통해 표시되는지 확인함). 충족하지 않으면 업데이트가 거부됩니다. CHECK OPTION이 지정되지 않은 경우 뷰의 INSERT 및 UPDATE 명령은 뷰를 통해 표시되지 않는 행을 생성할 수 있습니다. 다음 확인 옵션이 지원됩니다.
- LOCAL - 새 행은 뷰 자체에 직접 정의된 조건을 기준으로만 확인됩니다. 기본 뷰에 정의된 조건은 아무것도 확인되지 않습니다(CHECK OPTION을 지정하지 않는 한).
- CASCADED - 뷰 및 모든 기본 뷰의 조건을 기준으로 새 행이 확인됩니다. CHECK OPTION이 지정되고 LOCAL과 CASCADED는 지정되지 않으면 CASCADED가 수임됩니다.

> **Note**
>
> CHECK OPTION은 RECURSIVE 뷰와 함께 사용할 수 없습니다. CHECK OPTION은 자동으로 업데이트 가능한 뷰에서만 지원됩니다.

Notes

DROP VIEW 문을 사용하여 뷰를 삭제합니다.

뷰 열의 이름과 데이터 유형을 신중하게 고려해야 합니다. 예를 들어 CREATE VIEW vista AS SELECT 'Hello World';는 열 이름의 기본값이 ?column?;이므로 권장되지 않습니다. 또한 열 데이터 유형은 기본적으로 text로 설정되며, 이는 원하는 것과 다를 수 있습니다.

더 나은 방법은 CREATE VIEW vista AS SELECT text 'Hello World' AS hello;와 같이 열 이름과 데이터 유형을 명시적으로 지정하는 것입니다.

기본적으로 뷰에서 참조되는 기본 관계에 대한 액세스는 뷰 소유자의 권한에 따라 결정됩니다. 경우에 따라 기본 테이블에 대한 안전하고 제한된 액세스를 제공하는 데 사용할 수 있습니다. 그러나 모든 뷰가 변조로부터 안전한 것은 아닙니다.

- 뷰에 security_invoker 속성이 true로 설정된 경우 기본 관계에 대한 액세스는 뷰 소유자가 아닌 쿼리를 실행하는 사용자의 권한에 따라 결정됩니다. 따라서 보안 호출자 뷰의 사용자는 뷰 및 기본 관계에 대한 관련 권한이 있어야 합니다.
- 기본 관계가 보안 호출자 뷰인 경우 원래 쿼리에서 직접 액세스한 것처럼 처리됩니다. 따라서 보안 호출자 뷰는 security_invoker 속성이 없는 뷰에서 액세스하더라도 항상 현재 사용자의 권한을 사용하여 기본 관계를 확인합니다.
- 뷰에서 직접 호출된 함수는 뷰를 사용하여 쿼리에서 직접 호출된 것과 동일하게 처리됩니다. 따라서 뷰 사용자에게 뷰에서 사용하는 모든 함수를 직접 호출할 수 있는 권한이 있어야 합니다. 뷰의 함수는 함수가 SECURITY INVOKER와 SECURITY DEFINER 중 무엇으로 정의되었는지에 따라 쿼리를 실행하는 사용자 또는 함수 소유자의 권한으로 실행됩니다. 예를 들어 뷰에서 직접 CURRENT_USER를 호출하면 뷰 소유자가 아닌 호출하는 사용자가 항상 반환됩니다. 이는 뷰의 security_invoker 설정의 영향을 받지 않으므로 security_invoker가 false로 설정된 뷰는 SECURITY DEFINER 함수와 동일하지 않습니다.
- 뷰를 생성하거나 교체하는 사용자는 스키마에서 참조된 객체를 조회하기 위해 뷰 쿼리에서 참조된 스키마에 대한 USAGE 권한이 있어야 합니다. 그러나 이 조회는 뷰가 생성되거나 교체될 때만 발생합니다. 따라서 뷰 사용자에게는 뷰가 포함된 스키마에 대한 USAGE 권한만 필요하며, 보안 호출자 뷰의 경우에도 뷰 쿼리에서 참조되는 스키마에 대한 권한은 필요하지 않습니다.

- 기존 뷰에서 CREATE OR REPLACE VIEW를 사용하는 경우 뷰의 정의 SELECT 규칙과 WITH (...) 파라미터 및 해당 CHECK OPTION만 변경됩니다. 소유권, 권한 및 SELECT가 아닌 규칙을 포함한 다른 뷰 속성은 변경되지 않습니다. 뷰를 대체하려면 뷰를 소유해야 합니다(소유 역할의 멤버가 되는 것 포함).

업데이트 가능한 뷰

단순 뷰는 자동으로 업데이트할 수 있습니다. 시스템에서는 일반 테이블과 동일한 방식으로 뷰에서 INSERT, UPDATE 및 DELETE 문을 사용하도록 허용합니다. 뷰는 다음 조건을 모두 충족하는 경우 자동으로 업데이트됩니다.

- 뷰는 FROM 목록에 정확히 하나의 항목이 있어야 하며,이 항목은 테이블 또는 다른 업데이트 가능한 뷰여야 합니다.
- 뷰 정의에서 최상위 수준에 WITH, DISTINCT, GROUP BY, HAVING, LIMIT 또는 OFFSET 절이 포함되어서는 안 됩니다.
- 뷰 정의에서 최상위 수준에 세트 작업(UNION, INTERSECT 또는 EXCEPT)이 포함되어서는 안 됩니다.
- 뷰의 선택 목록에 집계, 창 함수 또는 집합 반환 함수가 포함되어서는 안 됩니다.

자동 업데이트 가능 뷰에 업데이트 가능 열과 업데이트 불가능 열이 혼합되어 있을 수 있습니다. 기본 관계의 업데이트 가능한 열에 대한 단순 참조인 경우 열을 업데이트할 수 있습니다. 그렇지 않으면 열은 읽기 전용이며 INSERT 또는 UPDATE 문이 값을 할당하려고 하면 오류가 발생합니다.

자동 업데이트 가능 뷰의 경우 시스템은 뷰의 모든 INSERT, UPDATE 또는 DELETE 문을 기본 관계의 상응하는 문으로 변환합니다. ON CONFLICT UPDATE 절이 있는 INSERT 문은 완전히 지원됩니다.

자동 업데이트 가능 뷰에 WHERE 조건이 포함된 경우 조건은 뷰의 UPDATE 및 DELETE 문이 수정할 수 있는 기본 관계의 행을 제한합니다. 그러나 UPDATE는 더 이상 WHERE 조건을 충족하지 않도록 행을 변경하여 뷰를 통해 표시되지 않도록 할 수 있습니다. 마찬가지로 INSERT 명령은 WHERE 조건을 충족하지 않는 기본 관계 행을 잠재적으로 삽입하여 뷰를 통해 표시되지 않게 만들 수 있습니다. ON CONFLICT UPDATE는 뷰를 통해 표시되지 않는 기존 행에 유사한 영향을 미칠 수 있습니다.

CHECK OPTION을 사용하여 INSERT 및 UPDATE 명령이 뷰를 통해 표시되지 않는 행을 생성하지 못하게 할 수 있습니다.

자동 업데이트 가능 뷰에 security_barrier 속성이 표시된 경우 뷰 사용자가 추가한 조건 전에 뷰의 모든 WHERE 조건(및 LEAKPROOF로 표시된 연산자를 사용하는 모든 조건)이 항상 먼저 평가됩니다. 이로 인해 최종적으로 반환되지 않는 행(사용자의 WHERE 조건을 통과하지 못하기 때문)이 여전히 잠길 수 있

습니다. EXPLAIN 을 사용하여 관계 수준에서 적용되는 조건(따라서 행을 잠그지 않음)과 그렇지 않은 조건을 확인할 수 있습니다.

이 모든 조건을 충족하지 않는 보다 복잡한 뷰는 기본적으로 읽기 전용입니다. 시스템은 해당 뷰에서 삽입, 업데이트 또는 삭제를 허용하지 않습니다.

> **Note**
>
> 해당 뷰에서 삽입, 업데이트 또는 삭제를 수행하는 사용자에게는 뷰에 대한 해당 삽입, 업데이트 또는 삭제 권한이 있어야 합니다. 기본적으로 뷰의 소유자는 기본 관계에 대한 관련 권한을 가지고 있어야 하지만 업데이트를 수행하는 사용자는 기본 관계에 대한 권한이 필요하지 않습니다. 그러나 뷰에 security_invoker가 true로 설정된 경우 뷰 소유자가 아닌 업데이트를 수행하는 사용자에게 기본 관계에 대한 관련 권한이 있어야 합니다.

예시

모든 코미디 영화로 구성된 뷰를 생성합니다.

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

이렇게 하면 뷰 생성 시 film 테이블에 있는 열이 포함된 뷰가 생성됩니다. *가 뷰를 생성하는 데 사용되었지만 나중에 테이블에 추가된 열은 뷰의 일부가 아닙니다.

LOCAL CHECK OPTION을 포함하여 뷰를 생성합니다.

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

이렇게 하면 새 행의 kind 및 classification을 모두 확인하는 뷰가 생성됩니다.

업데이트 가능한 열과 업데이트 불가능한 열이 혼합된 뷰를 생성합니다.

```
CREATE VIEW comedies AS
```

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

이 뷰는 INSERT, UPDATE, DELETE를 지원합니다. 영화 테이블의 모든 열은 업데이트 가능하지만 계산된 열 country와 avg_rating은 읽기 전용입니다.

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

> **Note**
>
> 재귀 뷰의 이름은 이 CREATE에서 스키마가 지정되었지만 내부 자체 참조는 스키마가 지정되지 않습니다. 이는 암시적으로 생성된 공통 테이블 표현식(CTE)의 이름에 스키마를 지정할 수 없기 때문입니다.

호환성

CREATE OR REPLACE VIEW는 PostgreSQL 언어 확장입니다. WITH (...) 절도 확장이며 보안 장벽 뷰 및 보안 호출자 뷰도 마찬가지입니다. Aurora DSQL은 이러한 언어 확장을 지원합니다.

ALTER VIEW

ALTER VIEW 문을 사용하면 기존 뷰의 다양한 속성을 변경할 수 있으며 Aurora DSQL은 이 명령에 대한 모든 PostgreSQL 구문을 지원합니다.

지원되는 구문

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

설명

ALTER VIEW는 뷰의 다양한 보조 속성을 변경합니다. (뷰의 정의 쿼리를 수정하려면 CREATE OR REPLACE VIEW를 사용하세요.) ALTER VIEW를 사용하려면 뷰를 소유해야 합니다. 뷰의 스키마를 변경하려면 새 스키마에 대한 CREATE 권한도 있어야 합니다. 소유자를 변경하려면 새 소유 역할에 SET ROLE을 수행할 수 있어야 하며 해당 역할에 뷰의 스키마에 대한 CREATE 권한이 있어야 합니다. 이러한 제한은 소유자를 변경해도 뷰를 삭제하고 다시 생성함으로써 수행할 수 없는 작업은 수행하지 않도록 강제합니다.

파라미터

ALTER VIEW 파라미터

name

기존 뷰의 이름(선택적으로 스키마 지정)입니다.

column_name

기존 열의 새 이름입니다.

IF EXISTS

뷰가 없는 경우 오류가 발생하지 않습니다. 이 경우 알림이 발행됩니다.

SET/DROP DEFAULT

이러한 양식은 열의 기본값을 설정하거나 제거합니다. 뷰 열의 기본값은 대상이 뷰인 INSERT 또는 UPDATE 명령으로 대체됩니다. 따라서 뷰의 기본값이 기본 관계의 기본값보다 우선합니다.

new_owner

뷰의 새 소유자의 사용자 이름입니다.

new_name

새 뷰의 이름입니다.

new_schema

뷰의 새 스키마입니다.

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

뷰 옵션을 설정하거나 재설정합니다. 지원되는 옵션은 다음과 같습니다.

- `check_option (enum)` - 뷰의 확인 옵션을 변경합니다. 값은 `local` 또는 `cascaded`여야 합니다.
- `security_barrier (boolean)` - 뷰의 보안 장벽 속성을 변경합니다. 값은 `true` 또는 `false`와 같은 부울 값이어야 합니다.
- `security_invoker (boolean)` - 뷰의 보안 장벽 속성을 변경합니다. 값은 `true` 또는 `false`와 같은 부울 값이어야 합니다.

Notes

과거 PostgreSQL의 사유로, 뷰에서 `ALTER TABLE`도 사용할 수 있지만 뷰에서 허용되는 `ALTER TABLE`의 유일한 변형은 이전에 표시된 변형과 동일합니다.

예시

`foo` 뷰의 이름을 `bar`로 변경합니다.

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

업데이트 가능한 뷰에 기본 열 값을 연결합니다.

```
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
```

호환성

`ALTER VIEW`는 Aurora DSQL이 지원하는 SQL 표준의 PostgreSQL 확장입니다.

DROP VIEW

`DROP VIEW` 문은 기존 뷰를 제거합니다. Aurora DSQL은 이 명령에 대한 전체 PostgreSQL 구문을 지원합니다.

지원되는 구문

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

설명

DROP VIEW는 기존 뷰를 삭제합니다. 이 명령을 실행하려면 뷰의 소유자여야 합니다.

파라미터

IF EXISTS

뷰가 없는 경우 오류가 발생하지 않습니다. 이 경우 알림이 발행됩니다.

name

제거할 뷰의 이름(선택적으로 스키마 지정)입니다.

CASCADE

뷰에 종속된 객체(예: 다른 뷰)와 해당 객체에 종속된 모든 객체가 자동으로 삭제됩니다.

RESTRICT

종속된 객체가 있는 경우 뷰를 삭제하는 것을 거부합니다. 이 값이 기본값입니다.

예시

```
DROP VIEW kinds;
```

호환성

이 명령은 SQL 표준을 준수합니다. 단, Aurora DSQL이 지원하는 PostgreSQL 확장인 IF EXISTS 옵션을 제외하고 표준에서 명령당 하나의 뷰만 삭제할 수 있습니다.

Aurora DSQL에서 지원되지 않는 PostgreSQL 기능

Aurora DSQL은 PostgreSQL과 호환됩니다. 즉, Aurora DSQL은 ACID 트랜잭션, 보조 인덱스, 조인, 삽입 및 업데이트와 같은 핵심 관계형 기능을 지원합니다. 지원되는 SQL 기능에 대한 개요는 Supported SQL expressions을 참조하세요.

다음 섹션에는 현재 Aurora DSQL에서 지원되지 않는 PostgreSQL 기능이 나와 있습니다.

지원되지 않는 객체

Aurora DSQL에서 지원하지 않는 객체에는 다음이 포함됩니다.

- 단일 Aurora DSQL 클러스터의 여러 데이터베이스
- 임시 테이블
- 트리거
- 유형(부분 지원)
- 테이블스페이스
- SQL 이외의 언어로 작성된 함수
- 시퀀스
- 파티션

지원되지 않는 제약 조건

- 외래 키
- 예외 제약 조건

지원되지 않는 명령

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

> **Note**
>
> Aurora DSQL에는 vacuum이 필요하지 않습니다. 시스템은 수동 vacuum 명령 없이 통계를 유지하고 스토리지 최적화를 자동으로 관리합니다.

지원되지 않는 확장

Aurora DSQL은 PostgreSQL 확장을 지원하지 않습니다. 다음 표에는 지원되지 않는 확장이 나와 있습니다.

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

지원되지 않는 SQL 표현식

다음 표에는 Aurora DSQL에서 지원되지 않는 절을 설명합니다.

범주	기본 절	지원되지 않는 절
CREATE	INDEX ASYNC	ASC | DESC
CREATE	INDEX [1]	
TRUNCATE		
ALTER	SYSTEM	모든 ALTER SYSTEM 명령이 차단됩니다.
CREATE	TABLE	COLLATE, AS SELECT, INHERITS, PARTITION
CREATE	FUNCTION	LANGUAGE *non-sql-lang* , 여기서 *non-sql-lang* 은 SQL 이외의 언어입니다.
CREATE	TEMPORARY	TABLES
CREATE	EXTENSION	
CREATE	SEQUENCE	
CREATE	MATERIALIZED	VIEW

범주	기본 절	지원되지 않는 절
CREATE	TABLESPACE	
CREATE	TRIGGER	
CREATE	TYPE	
CREATE	DATABASE	추가 데이터베이스를 생성할 수 없습니다.

[1] 지정된 테이블의 열에 인덱스를 생성하려면 Aurora DSQL의 비동기 인덱스 섹션을 참조하세요.

PostgreSQL 호환성을 위한 Aurora DSQL 고려 사항

Aurora DSQL을 사용할 때는 다음 호환성 제한 사항을 고려하세요. 일반적인 고려 사항은 Amazon Aurora DSQL 사용에 대한 고려 사항 섹션을 참조하세요. 할당량 및 한도는 Amazon Aurora DSQL의 클러스터 할당량 및 데이터베이스 한도 섹션을 참조하세요.

- Aurora DSQL은 `postgres`라는 단일 내장 데이터베이스를 사용합니다. 추가 데이터베이스를 생성하거나 `postgres` 데이터베이스를 삭제하거나 이름을 바꿀 수 없습니다.
- `postgres` 데이터베이스는 UTF-8 문자 인코딩을 사용합니다. 인코딩은 변경할 수 없습니다.
- 데이터베이스는 `C` 데이터 정렬만 사용합니다.
- Aurora DSQL은 `UTC`를 시스템 시간대로 사용합니다. 파라미터 또는 `SET TIMEZONE`과 같은 SQL 문을 사용하여 기본 시간대를 수정할 수 없습니다.
- 트랜잭션 격리 수준은 PostgreSQL `Repeatable Read`에서 고정됩니다.
- 트랜잭션의 제약 조건은 다음과 같습니다.
  - 하나의 트랜잭션에서 DDL 및 DML 작업을 혼합할 수 없습니다.
  - 하나의 트랜잭션에는 DDL 문이 하나만 포함될 수 있습니다.
  - 하나의 트랜잭션은 보조 인덱스 수와 관계없이 최대 3,000개의 행을 수정할 수 있습니다.
  - 행 3,000개 한도는 모든 DML 문(`INSERT`, `UPDATE`, `DELETE`)에 적용됩니다.
- 데이터베이스 연결은 1시간 후에 시간 초과됩니다.
- Aurora DSQL에서는 현재 `GRANT [permission] ON DATABASE`를 실행할 수 없습니다. 해당 문을 실행하려고 하면 Aurora DSQL이 `ERROR: unsupported object type in GRANT`라는 오류 메시지를 반환합니다.

- Aurora DSQL은 관리자가 아닌 사용자 역할이 `CREATE SCHEMA` 명령을 실행하도록 허용하지 않습니다. `GRANT [permission] on DATABASE` 명령을 실행하고 데이터베이스에 대한 `CREATE` 권한을 부여할 수 없습니다. 관리자가 아닌 사용자 역할이 스키마를 생성하려고 하면 Aurora DSQL은 `ERROR: permission denied for database postgres`라는 오류 메시지를 반환합니다.
- 관리자가 아닌 사용자는 퍼블릭 스키마에서 객체를 생성할 수 없습니다. 관리자 사용자만 퍼블릭 스키마에서 객체를 생성할 수 있습니다. 관리자 사용자 역할에는 관리자가 아닌 사용자에게 이러한 객체에 대한 읽기, 쓰기 및 수정 액세스 권한을 부여할 수 있는 권한이 있지만 퍼블릭 스키마 자체에는 `CREATE` 권한을 부여할 수 없습니다. 관리자가 아닌 사용자는 객체 생성에 다른 사용자 생성 스키마를 사용해야 합니다.
- Aurora DSQL은 `ALTER ROLE [] CONNECTION LIMIT` 명령을 지원하지 않습니다. 연결 한도를 늘려야 하는 경우 AWS Support에 문의하세요.
- Aurora DSQL은 Python용 비동기 PostgreSQL 데이터베이스 드라이버인 asyncpg를 지원하지 않습니다.

Aurora DSQL의 동시성 제어

동시성을 통해 여러 세션이 데이터 무결성과 일관성을 손상시키지 않고 동시에 데이터에 액세스하고 수정할 수 있습니다. Aurora DSQL은 현대적 비잠금 동시성 제어 메커니즘을 구현하면서 PostgreSQL 호환성을 제공합니다. 스냅샷 격리를 통해 완전한 ACID 규정 준수를 유지하여 데이터 일관성과 신뢰성을 보장합니다.

Aurora DSQL의 주요 이점은 일반적인 데이터베이스 성능 병목 현상을 제거하는 비잠금 아키텍처라는 점입니다. Aurora DSQL은 느린 트랜잭션이 다른 작업을 차단하는 것을 방지하고 교착 상태의 위험을 제거합니다. 이 접근 방식을 통해 Aurora DSQL은 성능과 확장성이 필수인 처리량이 많은 애플리케이션에 특히 유용합니다.

트랜잭션 충돌

Aurora DSQL은 기존 잠금 기반 시스템과 다르게 작동하는 낙관적 동시성 제어(OCC)를 사용합니다. OCC는 잠금을 사용하는 대신 커밋 시 충돌을 평가합니다. 동일한 행을 업데이트하는 동안 여러 트랜잭션이 충돌하는 경우 Aurora DSQL은 다음과 같이 트랜잭션을 관리합니다.

- 커밋 시간이 가장 빠른 트랜잭션은 Aurora DSQL에서 처리됩니다.
- 충돌하는 트랜잭션에 PostgreSQL 직렬화 오류가 수신되어 재시도해야 함을 나타냅니다.

충돌을 처리하기 위해 재시도 로직을 구현하도록 애플리케이션을 설계하세요. 이상적인 설계 패턴은 멱등적이므로 가능하면 트랜잭션 재시도를 첫 번째 수단으로 사용할 수 있습니다. 권장 로직은 표준 PostgreSQL 잠금 제한 시간 또는 교착 상태의 중단 및 재시도 로직과 유사합니다. 그러나 OCC에서는 애플리케이션이 이 로직을 더 자주 실행해야 합니다.

트랜잭션 성능 최적화 지침

성능을 최적화하려면 단일 키 또는 작은 키 범위에서 높은 경합을 최소화합니다. 이 목표를 달성하려면 다음 지침에 따라 클러스터 키 범위에 업데이트를 분산하도록 스키마를 설계하세요.

- 테이블에 대한 임의 프라이머리 키를 선택합니다.
- 단일 키에 대한 경합을 높이는 패턴을 피합니다. 이 접근 방식은 트랜잭션 볼륨이 증가하더라도 최적의 성능을 보장합니다.

Aurora DSQL의 DDL 및 분산 트랜잭션

데이터 정의 언어(DDL)는 Aurora DSQL에서 PostgreSQL과 다르게 작동합니다. Aurora DSQL에는 다중 테넌트 컴퓨팅 및 스토리지 플릿을 기반으로 구축된 다중 AZ 분산 및 비공유 데이터베이스 계층이 있습니다. 단일 프라이머리 데이터베이스 노드 또는 리더가 없으므로 데이터베이스 카탈로그가 배포됩니다. 따라서 Aurora DSQL은 DDL 스키마 변경을 분산 트랜잭션으로 관리합니다.

특히 DDL은 Aurora DSQL에서 다음과 같이 다르게 작동합니다.

동시성 제어 오류

Aurora DSQL은 다른 트랜잭션이 리소스를 업데이트하는 동안 한 트랜잭션을 실행하는 경우 동시성 제어 위반 오류를 반환합니다. 예를 들어 다음과 같은 일련의 작업을 고려합니다.

1. 세션 1에서 사용자는 `mytable` 테이블에 열을 추가합니다.
2. 세션 2에서 사용자는 `mytable`에 행을 삽입하려고 시도합니다.

   Aurora DSQL에서 `SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (OC001).` 오류를 반환합니다.

동일한 트랜잭션의 DDL 및 DML

Aurora DSQL의 트랜잭션은 하나의 DDL 문만 포함할 수 있으며 DDL 문과 DML 문을 모두 가질 수는 없습니다. 이 제한은 테이블을 생성하고 동일한 트랜잭션 내에서 동일한 테이블에 데이터를 삽입할 수 없음을 의미합니다. 예를 들어 Aurora DSQL은 다음과 같은 순차 트랜잭션을 지원합니다.

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

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

Aurora DSQL은 CREATE 및 INSERT 문을 모두 포함하는 다음 트랜잭션을 지원하지 않습니다.

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

비동기 DDL

표준 PostgreSQL에서 CREATE INDEX와 같은 DDL 작업은 영향을 받는 테이블을 잠그므로 다른 세션의 읽기 및 쓰기에 사용할 수 없습니다. Aurora DSQL에서 이러한 DDL 문은 백그라운드 관리자를 사용하여 비동기적으로 실행됩니다. 영향을 받는 테이블에 대한 액세스는 차단되지 않습니다. 따라서 대규모 테이블의 DDL은 가동 중지 시간이나 성능 영향 없이 실행할 수 있습니다. Aurora DSQL의 비동기 작업 관리자에 대한 자세한 내용은 Aurora DSQL의 비동기 인덱스 섹션을 참조하세요.

Aurora DSQL의 프라이머리 키

Aurora DSQL에서 프라이머리 키는 테이블 데이터를 물리적으로 구성하는 기능입니다. PostgreSQL의 CLUSTER 작업 또는 다른 데이터베이스의 클러스터링된 인덱스와 유사합니다. 프라이머리 키를 정의하면 Aurora DSQL은 테이블의 모든 열을 포함하는 인덱스를 생성합니다. Aurora DSQL의 프라이머리 키 구조는 효율적인 데이터 액세스 및 관리를 보장합니다.

데이터 구조 및 스토리지

프라이머리 키를 정의하면 Aurora DSQL은 프라이머리 키 순서로 테이블 데이터를 저장합니다. 이 인덱스 구성 구조를 사용하면 프라이머리 키 조회가 기존 B-트리 인덱스에서처럼 데이터에 대한 포인터를 따르는 대신 모든 열 값을 직접 검색할 수 있습니다. 데이터를 한 번만 재구성하는 PostgreSQL의 CLUSTER 작업과 달리 Aurora DSQL은 이 순서를 자동으로 지속적으로 유지합니다. 이 접근 방식은 프라이머리 키 액세스에 의존하는 쿼리의 성능을 개선합니다.

또한 Aurora DSQL은 프라이머리 키를 사용하여 테이블 및 인덱스의 각 행에 대한 클러스터 전체 고유 키를 생성합니다. 이 고유 키는 분산 데이터 관리도 뒷받침합니다. 이를 통해 여러 노드에서 데이터를 자동으로 파티셔닝하여 확장 가능한 스토리지와 높은 동시성을 지원할 수 있습니다. 따라서 프라이머리 키 구조는 Aurora DSQL이 자동으로 규모 조정되고 동시 워크로드를 효율적으로 관리하는 데 도움이 됩니다.

프라이머리 키 선택 지침

Aurora DSQL에서 프라이머리 키를 선택하고 사용할 때는 다음 지침을 고려하세요.

- 테이블을 생성할 때 프라이머리 키를 정의합니다. 나중에 이 키를 변경하거나 새 프라이머리 키를 추가할 수 없습니다. 프라이머리 키는 데이터 파티셔닝 및 쓰기 처리량의 자동 규모 조정에 사용되는 클러스터 전체 키의 일부가 됩니다. 프라이머리 키를 지정하지 않으면 Aurora DSQL이 합성 숨김 ID를 할당합니다.
- 쓰기 볼륨이 많은 테이블의 경우 단조롭게 증가하는 정수를 프라이머리 키로 사용하지 마세요. 그럴 경우 모든 새 삽입을 단일 파티션으로 전달하여 성능 문제가 발생할 수 있습니다. 대신 임의의 분산이 있는 프라이머리 키를 사용하여 스토리지 파티션 간에 쓰기를 고르게 분산할 수 있습니다.
- 자주 변경되지 않거나 읽기 전용인 테이블의 경우 오름차순 키를 사용할 수 있습니다. 오름차순 키의 예로는 타임스탬프 또는 시퀀스 번호가 있습니다. 밀집 키에는 밀접하게 이격되거나 중복된 값이 많습니다. 쓰기 성능이 덜 중요하기 때문에 밀도가 높더라도 오름차순 키를 사용할 수 있습니다.
- 전체 테이블 스캔이 성능 요구 사항을 충족하지 않는 경우 더 효율적인 액세스 방법을 선택합니다. 대부분의 경우 이는 쿼리에서 가장 일반적인 조인 및 조회 키와 일치하는 프라이머리 키를 사용하는 것을 의미합니다.
- 프라이머리 키에 있는 열의 최대 결합 크기는 1KiB입니다. 자세한 내용은 Aurora DSQL의 데이터베이스 한도 및 Aurora DSQL에서 지원되는 데이터 유형을 참조하세요.
- 프라이머리 키 또는 보조 인덱스에 최대 8개의 열을 포함할 수 있습니다. 자세한 내용은 Aurora DSQL의 데이터베이스 한도 및 Aurora DSQL에서 지원되는 데이터 유형을 참조하세요.

Aurora DSQL의 비동기 인덱스

`CREATE INDEX ASYNC` 명령은 지정된 테이블의 하나 이상의 열에 인덱스를 생성합니다. 이 명령은 다른 트랜잭션을 차단하지 않는 비동기 DDL 작업입니다. `CREATE INDEX ASYNC`를 실행하면 Aurora DSQL이 즉시 `job_id`를 반환합니다.

`sys.jobs` 시스템 뷰를 사용하여 이 비동기 작업의 상태를 모니터링할 수 있습니다. 인덱스 생성 작업이 진행 중일 때 다음 절차와 명령을 사용할 수 있습니다.

sys.wait_for_job(job_id)*'your_index_creation_job_id'*

지정된 작업이 완료되거나 실패할 때까지 현재 세션을 차단합니다. 성공 또는 실패를 나타내는 부울 값을 반환합니다.

DROP INDEX

진행 중인 인덱스 빌드 작업을 취소합니다.

비동기 인덱스 생성이 완료되면 Aurora DSQL은 시스템 카탈로그를 업데이트하여 인덱스를 활성으로 표시합니다.

> **Note**
>
> 이 업데이트 중에 동일한 네임스페이스의 객체에 액세스하는 동시 트랜잭션에는 동시성 오류가 발생할 수 있습니다.

Aurora DSQL이 비동기 인덱스 작업을 완료하면 시스템 카탈로그를 업데이트하여 인덱스가 활성 상태임을 표시합니다. 현재 다른 트랜잭션이 동일한 네임스페이스의 객체를 참조하는 경우 동시성 오류가 표시될 수 있습니다.

구문

`CREATE INDEX ASYNC`는 다음 구문을 사용합니다.

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

파라미터

UNIQUE

인덱스를 생성할 때와 데이터를 추가할 때마다 테이블에 중복 값이 있는지 확인하도록 Aurora DSQL에 알립니다. 이 파라미터를 지정하는 경우 중복 항목이 발생하는 작업을 삽입 및 업데이트하면 오류가 발생합니다.

`IF NOT EXISTS`

동일한 이름의 인덱스가 이미 있는 경우 Aurora DSQL에서 예외가 발생하지 않아야 함을 나타냅니다. 이 경우 Aurora DSQL은 새 인덱스를 생성하지 않습니다. 생성하려는 인덱스는 존재하는 인덱스와 구조가 매우 다를 수 있습니다. 이 파라미터를 지정하면 인덱스 이름이 필요합니다.

`name`

인덱스의 이름입니다. 이 파라미터에는 스키마 이름을 포함할 수 없습니다.

Aurora DSQL은 상위 테이블과 동일한 스키마에 인덱스를 생성합니다. 인덱스의 이름은 스키마의 테이블 또는 인덱스와 같은 다른 객체의 이름과 달라야 합니다.

이름을 지정하지 않으면 Aurora DSQL은 상위 테이블 및 인덱싱된 열의 이름을 기반으로 이름을 자동으로 생성합니다. 예를 들어 `CREATE INDEX ASYNC on table1 (col1, col2)`를 실행하면 Aurora DSQL이 인덱스의 이름을 `table1_col1_col2_idx`로 자동으로 지정합니다.

`NULLS FIRST | LAST`

Null 열과 Null이 아닌 열의 정렬 순서입니다. `FIRST`는 Aurora DSQL이 Null이 아닌 열 앞에 Null 열을 정렬해야 함을 나타냅니다. `LAST`는 Aurora DSQL이 Null이 아닌 열 뒤에 Null 열을 정렬해야 함을 나타냅니다.

`INCLUDE`

인덱스에 키가 아닌 열로 포함할 열 목록입니다. 인덱스 스캔 검색 검증에는 키가 아닌 열을 사용할 수 없습니다. Aurora DSQL은 인덱스의 고유성 측면에서 이 열을 무시합니다.

`NULLS DISTINCT | NULLS NOT DISTINCT`

Aurora DSQL이 Null 값을 고유 인덱스에서 구별되는 것으로 간주해야 하는지를 지정합니다. 기본값은 `DISTINCT`입니다. 즉, 고유 인덱스는 열에 여러 Null 값을 포함할 수 있습니다. `NOT DISTINCT`는 인덱스가 열에 여러 Null 값을 포함할 수 없음을 나타냅니다.

사용 노트

다음 지침을 참고하세요.

- `CREATE INDEX ASYNC` 명령은 잠금을 도입하지 않습니다. 또한 Aurora DSQL이 인덱스를 생성하는 데 사용하는 기본 테이블에도 영향을 주지 않습니다.
- 스키마 마이그레이션 작업 중에는 `sys.wait_for_job(job_id)`*`'your_index_creation_job_id'`* 프로시저가 유용합니다. 이를 통해 후속 DDL 및 DML 작업이 새로 생성된 인덱스를 대상으로 합니다.

- Aurora DSQL은 새 비동기 작업을 실행할 때마다 `sys.jobs` 뷰를 확인하고 30분 이상 `completed` 또는 `failed` 상태인 작업을 삭제합니다. 따라서 `sys.jobs`은 주로 진행 중인 작업을 표시하고 이전 작업에 대한 정보는 포함하지 않습니다.
- Aurora DSQL이 비동기 인덱스를 빌드하지 못하면 인덱스는 `INVALID`를 유지합니다. 고유 인덱스의 경우 인덱스를 삭제할 때까지 DML 작업에는 고유성 제약이 적용됩니다. 잘못된 인덱스를 삭제하고 다시 생성하는 것이 좋습니다.

인덱스 생성: 예시

다음 예시에서는 스키마, 테이블, 인덱스를 생성하는 방법을 보여줍니다.

1. `test.departments`이라는 테이블을 생성합니다.

```
CREATE SCHEMA test;

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

2. 테이블에 행을 삽입합니다.

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

3. 비동기 인덱스를 생성합니다.

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

   `CREATE INDEX` 명령은 아래와 같이 작업 ID를 반환합니다.

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

   `job_id`는 Aurora DSQL이 인덱스를 생성하기 위해 새 작업을 제출했음을 나타냅니다. `sys.wait_for_job(job_id)` *'your_index_creation_job_id'* 프로시저를 사용하여 작업이 완료되거나 제한 시간이 초과될 때까지 세션에서 다른 작업을 차단할 수 있습니다.

인덱스 생성 상태 쿼리: 예

다음 예시와 같이 sys.jobs 시스템 뷰를 쿼리하여 인덱스의 생성 상태를 확인합니다.

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

Aurora DSQL은 다음과 유사한 응답을 반환합니다.

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

상태 열은 다음 값 중 하나일 수 있습니다.

submitted	**processing**	**failed**	**completed**
작업이 제출되었지만 Aurora DSQL에서 아직 처리를 시작하지 않았습니다.	Aurora DSQL이 작업을 처리하고 있습니다.	작업이 실패했습니다. 자세한 내용은 세부 정보 열을 확인하세요. Aurora DSQL이 인덱스를 빌드하지 못한 경우 Aurora DSQL은 인덱스 정의를 자동으로 제거하지 않습니다. DROP INDEX 명령을 사용하여 인덱스를 수동으로 제거해야 합니다.	Aurora DSQL

카탈로그 테이블 pg_index 및 pg_class를 통해 인덱스의 상태를 쿼리할 수도 있습니다. 특히 indisvalid 및 indisimmediate 속성은 인덱스의 상태를 알려줄 수 있습니다. Aurora DSQL이 인덱스를 생성하는 동안 초기 상태는 INVALID입니다. 인덱스의 indisvalid 플래그는 인덱스가 유효하지 않음을 나타내는 FALSE 또는 f를 반환합니다. 플래그가 TRUE 또는 t를 반환하면 인덱스가 준비된 것입니다.

```
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)
```

고유 인덱스 빌드 실패

비동기 고유 인덱스 빌드 작업에서 `Found duplicate key while validating index for UCVs`라는 세부 정보와 함께 실패 상태가 표시되는 경우 고유성 제약 조건 위반으로 인해 고유 인덱스를 빌드할 수 없음을 나타냅니다.

고유 인덱스 빌드 실패를 해결하는 방법

1. 고유한 보조 인덱스에 지정된 키에 대해 중복 항목이 있는 기본 테이블의 행을 제거합니다.
2. 실패한 인덱스를 삭제합니다.
3. 인덱스 생성 명령을 새로 실행합니다.

기본 테이블에서 고유성 위반 감지

다음 SQL 쿼리는 테이블의 지정된 열에서 중복 값을 식별하는 데 도움이 됩니다. 이는 현재 프라이머리 키로 설정되지 않았거나 사용자 테이블의 이메일 주소와 같은 고유한 제약 조건이 없는 열에 고유성을 적용해야 하는 경우에 특히 유용합니다.

아래 예시에서는 샘플 사용자 테이블을 생성하고, 알려진 중복이 포함된 테스트 데이터로 채우고, 감지 쿼리를 실행하는 방법을 보여줍니다.

테이블 스키마 정의

```
-- 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
);
```

중복 이메일 주소 세트가 포함된 샘플 데이터 삽입

```
-- 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');
```

중복 탐지 쿼리 실행

```
-- 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;
```

이메일 주소가 중복된 모든 레코드 보기

```
 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)
```

인덱스 생성 문을 지금 시도하면 실패합니다.

```
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 DSQL의 시스템 테이블 및 명령

Aurora DSQL에서 지원되는 시스템 테이블 및 카탈로그에 대해 알아보려면 이어지는 섹션을 참조하세요.

시스템 테이블

Aurora DSQL은 PostgreSQL과 호환되므로 PostgreSQL의 많은 시스템 카탈로그 테이블과 뷰가 Aurora DSQL에도 있습니다.

중요한 PostgreSQL 카탈로그 테이블 및 뷰

다음 표에서는 Aurora DSQL에서 사용할 수 있는 가장 일반적인 테이블과 뷰를 설명합니다.

명칭	설명
pg_namespace	모든 스키마에 대한 정보
pg_tables	모든 테이블에 대한 정보
pg_attribute	모든 속성에 대한 정보
pg_views	(미리) 정의된 뷰에 대한 정보
pg_class	모든 테이블, 열, 인덱스 및 유사한 객체 설명
pg_stats	플래너 통계에 대한 뷰
pg_user	사용자에 대한 정보
pg_roles	사용자 및 그룹에 대한 정보
pg_indexes	모든 인덱스 나열
pg_constraint	테이블에 대한 제약 조건 나열

지원되는 카탈로그 테이블과 지원되지 않는 카탈로그 테이블

다음 표에는 Aurora DSQL에서 지원되는 테이블과 지원되지 않는 테이블이 나와 있습니다.

명칭	Aurora DSQL에 적용 가능
pg_aggregate	아니요
pg_am	예
pg_amop	아니요
pg_amproc	아니요

명칭	Aurora DSQL에 적용 가능
pg_attrdef	예
pg_attribute	예
pg_authid	아니요(pg_roles 사용)
pg_auth_members	예
pg_cast	예
pg_class	예
pg_collation	예
pg_constraint	예
pg_conversion	아니요
pg_database	아니요
pg_db_role_setting	예
pg_default_acl	예
pg_depend	예
pg_description	예
pg_enum	아니요
pg_event_trigger	아니요
pg_extension	아니요
pg_foreign_data_wrapper	아니요
pg_foreign_server	아니요
pg_foreign_table	아니요

명칭	Aurora DSQL에 적용 가능
pg_index	예
pg_inherits	예
pg_init_privs	아니요
pg_language	아니요
pg_largeobject	아니요
pg_largeobject_metadata	예
pg_namespace	예
pg_opclass	아니요
pg_operator	예
pg_opfamily	아니요
pg_parameter_acl	예
pg_partitioned_table	아니요
pg_policy	아니요
pg_proc	아니요
pg_publication	아니요
pg_publication_namespace	아니요
pg_publication_rel	아니요
pg_range	예
pg_replication_origin	아니요
pg_rewrite	아니요

명칭	Aurora DSQL에 적용 가능
pg_seclabel	아니요
pg_sequence	아니요
pg_shdepend	예
pg_shdescription	예
pg_shseclabel	아니요
pg_statistic	예
pg_statistic_ext	아니요
pg_statistic_ext_data	아니요
pg_subscription	아니요
pg_subscription_rel	아니요
pg_tablespace	아니요
pg_transform	아니요
pg_trigger	아니요
pg_ts_config	예
pg_ts_config_map	예
pg_ts_dict	예
pg_ts_parser	예
pg_ts_template	예
pg_type	예
pg_user_mapping	아니요

지원되는 시스템 뷰와 지원되지 않는 시스템 뷰

다음 표에는 Aurora DSQL에서 지원되는 뷰와 지원되지 않는 뷰가 나와 있습니다.

명칭	Aurora DSQL에 적용 가능
pg_available_extensions	아니요
pg_available_extension_versions	아니요
pg_backend_memory_contexts	예
pg_config	아니요
pg_cursors	아니요
pg_file_settings	아니요
pg_group	예
pg_hba_file_rules	아니요
pg_ident_file_mappings	아니요
pg_indexes	예
pg_locks	아니요
pg_matviews	아니요
pg_policies	아니요
pg_prepared_statements	아니요
pg_prepared_xacts	아니요
pg_publication_tables	아니요
pg_replication_origin_status	아니요
pg_replication_slots	아니요

명칭	Aurora DSQL에 적용 가능
pg_roles	예
pg_rules	아니요
pg_seclabels	아니요
pg_sequences	아니요
pg_settings	예
pg_shadow	예
pg_shmem_allocations	예
pg_stats	예
pg_stats_ext	아니요
pg_stats_ext_exprs	아니요
pg_tables	예
pg_timezone_abbrevs	예
pg_timezone_names	예
pg_user	예
pg_user_mappings	아니요
pg_views	예
pg_stat_activity	아니요
pg_stat_replication	아니요
pg_stat_replication_slots	아니요
pg_stat_wal_receiver	아니요

명칭	Aurora DSQL에 적용 가능
pg_stat_recovery_prefetch	아니요
pg_stat_subscription	아니요
pg_stat_subscription_stats	아니요
pg_stat_ssl	예
pg_stat_gssapi	아니요
pg_stat_archiver	아니요
pg_stat_io	아니요
pg_stat_bgwriter	아니요
pg_stat_wal	아니요
pg_stat_database	아니요
pg_stat_database_conflicts	아니요
pg_stat_all_tables	아니요
pg_stat_all_indexes	아니요
pg_statio_all_tables	아니요
pg_statio_all_indexes	아니요
pg_statio_all_sequences	아니요
pg_stat_slru	아니요
pg_statio_user_tables	아니요
pg_statio_user_sequences	아니요
pg_stat_user_functions	아니요

명칭	Aurora DSQL에 적용 가능
pg_stat_user_indexes	아니요
pg_stat_progress_analyze	아니요
pg_stat_progress_basebackup	아니요
pg_stat_progress_cluster	아니요
pg_stat_progress_create_index	아니요
pg_stat_progress_vacuum	아니요
pg_stat_sys_indexes	아니요
pg_stat_sys_tables	아니요
pg_stat_xact_all_tables	아니요
pg_stat_xact_sys_tables	아니요
pg_stat_xact_user_functions	아니요
pg_stat_xact_user_tables	아니요
pg_statio_sys_indexes	아니요
pg_statio_sys_sequences	아니요
pg_statio_sys_tables	아니요
pg_statio_user_indexes	아니요

sys.jobs 및 sys.iam_pg_role_mappings 뷰

Aurora DSQL은 다음 시스템 뷰를 지원합니다.

sys.jobs

sys.jobs은 비동기 작업에 대한 상태 정보를 제공합니다. 예를 들어 사용자가 비동기 인덱스를 생성한 후 Aurora DSQL은 job_uuid를 반환합니다. 이 job_uuid와 sys.jobs을 함께 사용하여 작업 상태를 조회할 수 있습니다.

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

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

sys.iam_pg_role_mappings

sys.iam_pg_role_mappings 뷰는 IAM 사용자에게 부여된 권한에 대한 정보를 제공합니다. 예를 들어 DQSLDBConnect가 관리자가 아닌 사용자에게 Aurora DSQL 액세스 권한을 부여하는 IAM 역할이고 testuser라는 사용자에게 DQSLDBConnect 역할 및 해당 권한이 부여된 경우 sys.iam_pg_role_mappings 뷰를 쿼리하여 어떤 사용자에게 어떤 권한이 부여되는지 확인할 수 있습니다.

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

pg_class 테이블

pg_class 테이블은 데이터베이스 객체에 대한 메타데이터를 저장합니다. 테이블에 있는 행 수의 대략적인 개수를 가져오려면 다음 명령을 실행합니다.

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

이 명령은 다음과 비슷한 출력을 반환합니다.

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

ANALYZE 명령

ANALYZE 명령은 데이터베이스의 테이블 내용에 대한 통계를 수집하고 결과를 pg_stats 시스템 뷰에 저장합니다. 이후 쿼리 플래너는 이러한 통계를 사용하여 쿼리에 대한 가장 효율적인 실행 계획을 결정하는 데 도움을 줍니다.

Aurora DSQL에서는 명시적 트랜잭션 내에서 ANALYZE 명령을 실행할 수 없습니다. ANALYZE에는 데이터베이스 트랜잭션 제한 시간이 적용되지 않습니다.

수동 개입의 필요성을 줄이고 통계를 지속적으로 최신 상태로 유지하기 위해 Aurora DSQL은 ANALYZE를 백그라운드 프로세스로 자동으로 실행합니다. 이 백그라운드 작업은 테이블에서 관찰된 변경 속도에 따라 자동으로 트리거됩니다. 마지막 분석 이후 삽입, 업데이트 또는 삭제된 행(튜플) 수와 연결됩니다.

ANALYZE는 백그라운드에서 비동기적으로 실행되며 다음 쿼리를 사용하여 시스템 뷰 sys.jobs에서 활동을 모니터링할 수 있습니다.

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

주요 고려 사항

> **Note**
>
> ANALYZE 작업은 Aurora DSQL의 다른 비동기 작업과 마찬가지로 청구됩니다. 테이블을 수정하면 자동 백그라운드 통계 수집 작업이 간접적으로 트리거되어 연결된 시스템 수준 활동으로 인해 사용량 측정에 따른 요금이 발생할 수 있습니다.

자동으로 트리거되는 백그라운드 ANALYZE 작업은 수동 ANALYZE와 동일한 유형의 통계를 수집하여 기본적으로 사용자 테이블에 적용합니다. 시스템 및 카탈로그 테이블은 이 자동 프로세스에서 제외됩니다.

Aurora DSQL 클러스터 관리

Aurora DSQL은 필요에 맞는 데이터베이스 인프라를 설정하는 데 도움이 되도록 몇 가지 구성 옵션을 제공합니다. Aurora DSQL 클러스터 인프라를 설정하려면 아래 섹션을 검토하세요.

주제

이 설명서에서 설명하는 기능을 사용하면 애플리케이션이 성장하고 발전함에 따라 Aurora DSQL 환경의 복원력과 응답성이 향상되며 애플리케이션을 지원할 수 있습니다.

단일 리전 클러스터 구성

클러스터 생성

create-cluster 명령을 사용하여 클러스터를 생성합니다.

> **Note**
>
> 클러스터 생성은 비동기 작업입니다. 상태가 `ACTIVE`로 변경될 때까지 GetCluster API를 직접 호출합니다. 클러스터가 활성화되면 클러스터에 연결할 수 있습니다.

Example Command

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

> **Note**
>
> 생성 중에 삭제 방지를 비활성화하려면 `--no-deletion-protection-enabled` 플래그를 포함합니다.

Example 응답

```
{
```

```
    "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"
    }
}
```

클러스터 설명

get-cluster 명령을 사용하여 클러스터에 대한 정보를 가져옵니다.

Example Command

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

Example 응답

```
{
    "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"
    }
}
```

클러스터 업데이트

update-cluster 명령을 사용하여 기존 클러스터를 업데이트합니다.

> **Note**
>
> 업데이트는 비동기 작업입니다. 상태가 ACTIVE로 변경될 때까지 GetCluster API를 직접 호출하여 변경 사항을 확인합니다.

Example Command

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

Example 응답

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

클러스터 삭제

delete-cluster 명령을 사용하여 기존 클러스터를 삭제합니다.

> **Note**
>
> 삭제 방지가 비활성화된 클러스터만 삭제할 수 있습니다. 새로운 클러스터를 생성할 때 기본적으로 삭제 방지가 활성화됩니다.

Example Command

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

Example 응답

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

클러스터 나열

list-clusters 명령을 사용하여 클러스터를 나열합니다.

Example Command

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

Example 응답

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

다중 리전 클러스터 구성

이 장에서는 여러 AWS 리전에 걸쳐 클러스터를 구성하고 관리하는 방법을 설명합니다.

다중 리전 클러스터에 연결

다중 리전 피어링된 클러스터는 피어링된 클러스터 AWS 리전마다 하나씩, 총 두 개의 리전 엔드포인트를 제공합니다. 두 엔드포인트 모두 강력한 데이터 일관성으로 동시 읽기 및 쓰기 작업을 지원하는 단일 논리적 데이터베이스를 제공합니다. 피어링된 클러스터 외에도 다중 리전 클러스터에는 다중 리전 내구성 및 가용성을 개선하는 데 사용되는 암호화된 트랜잭션 로그의 제한된 기간을 저장하는 감시 리전도 있습니다. 다중 리전 감시 리전에는 엔드포인트가 없습니다.

다중 리전 클러스터 생성

다중 리전 클러스터를 생성하려면 먼저 감시 리전이 있는 클러스터를 생성합니다. 그런 다음, 이 클러스터를 첫 번째 클러스터와 감시 리전이 동일한 두 번째 클러스터와 피어링합니다. 다음 예시에서는 미국 동부(버지니아 북부) 및 미국 동부(오하이오)에서 미국 서부(오리건)를 감시 리전으로 사용하여 클러스터를 생성하는 방법을 보여줍니다.

1단계: 미국 동부(버지니아 북부)에서 클러스터 1 생성

다중 리전 속성을 사용하여 미국 동부(버지니아 북부) AWS 리전에서 클러스터를 생성하려면 아래 명령을 사용합니다.

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

Example 응답:

```
{
    "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 작업이 성공하면 클러스터가 PENDING_SETUP 상태로 전환됩니다. 클러스터를 피어 클러스터의 ARN으로 업데이트할 때까지 클러스터 생성은 PENDING_SETUP으로 유지됩니다.

2단계: 미국 동부(오하이오)에서 클러스터 2 생성

다중 리전 속성을 사용하여 미국 동부(오하이오) AWS 리전에서 클러스터를 생성하려면 아래 명령을 사용합니다.

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

Example 응답:

```
{
    "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 작업이 성공하면 클러스터가 PENDING_SETUP 상태로 전환됩니다. 피어링을 위해 클러스터를 다른 클러스터의 ARN으로 업데이트할 때까지 클러스터 생성은 PENDING_SETUP 상태로 유지됩니다.

3단계: 미국 동부(버지니아 북부)의 클러스터를 미국 동부(오하이오)와 피어링

미국 동부(버지니아 북부) 클러스터를 미국 동부(오하이오) 클러스터와 피어링하려면 update-cluster 명령을 사용합니다. 미국 동부(버지니아 북부) 클러스터 이름과 JSON 문자열을 미국 동부(오하이오) 클러스터의 ARN으로 지정합니다.

```
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 응답

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

4단계: 미국 동부(오하이오)의 클러스터와 미국 동부(버지니아 북부) 피어링

미국 동부(오하이오) 클러스터를 미국 동부(버지니아 북부) 클러스터와 피어링하려면 `update-cluster` 명령을 사용합니다. 미국 동부(오하이오) 클러스터 이름과 JSON 문자열을 미국 동부(버지니아 북부) 클러스터의 ARN으로 지정합니다.

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 응답

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

> **Note**
>
> 피어링에 성공하면 두 클러스터 모두 'PENDING_SETUP'에서 'CREATING'으로 전환되며, 사용할 준비가 되면 마침내 'ACTIVE' 상태로 전환됩니다.

다중 리전 클러스터 속성 보기

클러스터를 설명할 때 다른 AWS 리전의 클러스터에 대한 다중 리전 속성을 볼 수 있습니다.

Example

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

Example 응답

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

생성 중 클러스터 피어링

클러스터 생성 중에 피어링 정보를 포함하여 단계 수를 줄일 수 있습니다. 미국 동부(버지니아 북부)에서 첫 번째 클러스터를 생성(1단계)한 후 미국 동부(오하이오)에서 두 번째 클러스터를 생성하는 동시에 첫 번째 클러스터의 ARN을 포함하여 피어링 프로세스를 시작할 수 있습니다.

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"]}'
```

이렇게 하면 2단계와 4단계가 결합되지만 피어링 관계를 설정하려면 3단계(첫 번째 클러스터를 두 번째 클러스터의 ARN으로 업데이트)를 완료해야 합니다. 모든 단계가 완료되면 두 클러스터 모두 표준 프로세스와 동일한 상태, 즉 PENDING_SETUP에서 CREATING으로, 사용할 준비가 되면 마침내 ACTIVE로 전환됩니다.

다중 리전 클러스터 삭제

다중 리전 클러스터를 삭제하려면 두 단계를 완료해야 합니다.

1. 각 클러스터에 대해 삭제 방지 기능을 끕니다.
2. 각 피어링된 클러스터를 해당 AWS 리전에서 개별적으로 삭제합니다.

미국 동부(버지니아 북부)에서 클러스터 업데이트 및 삭제

1. `update-cluster` 명령을 사용하여 삭제 방지 기능을 끕니다.

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

2. `delete-cluster` 명령을 사용하여 클러스터를 삭제합니다.

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

   이 명령은 다음 응답을 반환합니다.

   ```
   {
       "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**
>
> 클러스터가 PENDING_DELETE 상태로 전환됩니다. 미국 동부(오하이오)에서 피어링된 클러스터를 삭제할 때까지 삭제가 완료되지 않습니다.

미국 동부(오하이오)에서 클러스터 업데이트 및 삭제

1. `update-cluster` 명령을 사용하여 삭제 방지 기능을 끕니다.

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

2. `delete-cluster` 명령을 사용하여 클러스터를 삭제합니다.

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

이 명령은 다음 응답을 반환합니다.

```
{
    "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**
>
> 클러스터가 PENDING_DELETE 상태로 전환됩니다. 몇 초가 지나면 시스템은 검증 후 피어링된 클러스터를 모두 DELETING 상태로 자동 전환합니다.

AWS CloudFormation을 사용하여 다중 리전 클러스터 구성

동일한 AWS CloudFormation 리소스 `AWS::DSQL::Cluster`를 사용하여 단일 리전 및 다중 리전 Aurora DSQL 클러스터를 배포하고 관리할 수 있습니다.

`AWS::DSQL::Cluster` 리소스를 사용하여 클러스터를 생성, 수정 및 관리하는 방법에 대한 자세한 내용은 Amazon Aurora DSQL resource type reference를 참조하세요.

첫 번째 클러스터 구성 생성

먼저 AWS CloudFormation 템플릿을 생성하여 다중 리전 클러스터를 정의합니다.

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

다음 AWS CLI 명령을 사용하여 두 리전 모두에서 스택을 생성합니다.

```
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
```

클러스터 식별자 찾기

클러스터의 물리적 리소스 ID를 검색합니다.

```
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"
]
```

클러스터 구성 업데이트

두 클러스터 ARN을 모두 포함하도록 AWS CloudFormation 템플릿을 업데이트합니다.

```
---
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
```

업데이트된 구성을 두 리전에 모두 적용합니다.

```
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 DSQL을 사용한 프로그래밍

Aurora DSQL은 프로그래밍 방식으로 Aurora DSQL 리소스를 관리할 수 있는 다음 도구를 제공합니다.

AWS Command Line Interface (AWS CLI)

명령줄 쉘에서 AWS CLI를 사용하여 리소스를 생성하고 관리할 수 있습니다. AWS CLI는 Aurora DSQL과 같은 AWS 서비스의 API에 대한 직접 액세스를 제공합니다. Aurora DSQL 명령의 구문 및 예시는 AWS CLI 명령 참조의 dsql을 참조하세요.

AWS 소프트웨어 개발 키트(SDK)

AWS는 널리 사용되는 많은 기술 및 프로그래밍 언어를 위한 SDK를 제공합니다. 이를 통해 애플리케이션 내에서 해당 언어 또는 기술로 AWS 서비스를 더 쉽게 직접적으로 호출할 수 있습니다. 이러한 SDK에 대한 자세한 내용은 AWS에서 애플리케이션 개발 및 관리를 위한 도구를 참조하세요.

Aurora DSQL API

이 API는 Aurora DSQL의 또 다른 프로그래밍 인터페이스입니다. 이 API를 사용할 때는 모든 HTTPS 요청을 올바르게 포맷하고 모든 요청에 유효한 디지털 서명을 추가해야 합니다. 자세한 내용은 API 참조 섹션을 참조하세요.

AWS CloudFormation

AWS::DSQL::Cluster는 코드형 인프라의 일부로 Aurora DSQL 클러스터를 생성하고 관리할 수 있는 AWS CloudFormation 리소스입니다. AWS CloudFormation은 코드로 전체 AWS 환경을 정의하여 일관되고 신뢰할 수 있는 방식으로 인프라를 더 쉽게 프로비저닝, 업데이트 및 복제할 수 있도록 지원합니다.

AWS CloudFormation 템플릿에서 AWS::DSQL::Cluster 리소스를 사용하는 경우 다른 클라우드 리소스와 함께 Aurora DSQL 클러스터를 선언적으로 프로비저닝할 수 있습니다. 이렇게 하면 데이터 인프라가 나머지 애플리케이션 스택과 함께 배포 및 관리할 수 있습니다.

Amazon Aurora DSQL AWS SDK 및 샘플 코드

다양한 프로그래밍 언어에 대해 AWS 소프트웨어 개발 키트(SDK)을 사용할 수 있습니다. 각 SDK는 개발자가 선호하는 언어로 애플리케이션을 쉽게 구축할 수 있도록 하는 API, 코드 예시 및 설명서를 제공합니다.

SDK	GitHub 샘플	객체 관계형 매핑 (ORMs)
[AWS SDK for C++](#)	[aws-samples/aurora-dsql-samples/cpp](#)	
[Golang](#)	[aws-samples/aurora-dsql-samples/go](#)	[pgx](#)
[Java](#)	[aws-samples/aurora-dsql-samples/](#)	[pgjdbc](#)

SDK Hub
(ORMs)

java

aws-dsql-samples/aurora-dsql-samples/javascript

aws-dsql-samples/aurora-dsql-samples/dotnet

SDK Hub

(ORMs)

psycopg2

Django

Ruby pg

SDK Hub
(ORMs)

Rust sample: aurora-dsql-samples/aurora-dsql-samples/rust

TypeScript sample: aurora-dsql-samples/type-orm/aurora-dsql-samples/typescript

AWS CLI를 사용한 Aurora DSQL

AWS CLI를 사용하여 클러스터를 관리하는 방법을 알아보려면 이어지는 섹션을 참조하세요.

CreateCluster

클러스터를 생성하려면 `create-cluster` 명령을 사용합니다.

> **Note**
>
> 클러스터 생성은 비동기적으로 수행됩니다. 상태가 ACTIVE가 될 때까지 GetCluster API를 직접 호출합니다. 클러스터가 ACTIVE가 되면 클러스터에 연결할 수 있습니다.

샘플 명령

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

> **Note**
>
> 생성 시 삭제 방지를 비활성화하려면 --no-deletion-protection-enabled 플래그를 포함합니다.

샘플 응답

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

GetCluster

클러스터를 설명하려면 get-cluster 명령을 사용합니다.

샘플 명령

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

샘플 응답

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

UpdateCluster

기존 클러스터를 업데이트하려면 `update-cluster` 명령을 사용합니다.

> **Note**
>
> 업데이트는 비동기적으로 수행됩니다. 상태가 `ACTIVE`가 될 때까지 `GetCluster` API를 직접 호출하면 변경 사항을 확인할 수 있습니다.

샘플 명령

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

샘플 응답

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

DeleteCluster

기존 클러스터를 삭제하려면 `delete-cluster` 명령을 사용합니다.

> **Note**
>
> 삭제 방지 기능이 비활성화된 클러스터만 삭제할 수 있습니다. 새로운 클러스터를 생성할 때 기본적으로 삭제 방지가 활성화됩니다.

샘플 명령

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

샘플 응답

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

ListClusters

클러스터를 가져오려면 `list-clusters` 명령을 사용합니다.

샘플 명령

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

샘플 응답

```
{
    "clusters": [
        {
```

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

다중 리전 클러스터에 대한 GetCluster

다중 리전 클러스터에 대한 정보를 가져오려면 `get-cluster` 명령을 사용합니다. 다중 리전 클러스터의 경우 응답에는 연결된 클러스터 ARN이 포함됩니다.

샘플 명령

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

샘플 응답

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

```
}
```

Aurora DSQL 클러스터 생성, 읽기, 업데이트, 삭제

단일 리전 및 다중 리전 배포 모두에 대해 생성, 읽기, 업데이트, 삭제(CRUD) 예시가 제공됩니다. 이러한 키 관리 작업을 보여주는 각 프로그래밍 언어에 대한 전용 `cluster_management` 섹션이 포함되어 있습니다.

단일 리전 배포는 특정 지리적 영역의 사용자에게 서비스를 제공하는 애플리케이션에 적합하며 관리를 간소화하고 지연 시간을 줄입니다. 다중 리전 배포를 사용하면 데이터베이스를 여러 AWS 리전에 분산하여 가용성과 재해 복구 기능을 향상할 수 있습니다.

가용성, 성능 및 지리적 배포에 대한 애플리케이션의 요구 사항에 맞는 배포 유형을 선택합니다.

주제

- 클러스터 생성
- 클러스터 가져오기
- 클러스터 업데이트
- 클러스터 삭제

클러스터 생성

Aurora DSQL에서 단일 리전 및 다중 리전 클러스터를 생성하는 방법을 알아보려면 다음 정보를 참조하세요.

Python

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

```
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()
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

```
import boto3


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

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

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

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

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

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

C++

다음 예시를 사용하면 단일 AWS 리전에서 클러스터를 생성할 수 있습니다.

```
#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;
}
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

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

using namespace Aws;
```

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

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

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

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

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

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

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

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

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

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

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

    CreateClusterRequest createClusterRequest2;
    createClusterRequest2.SetDeletionProtectionEnabled(true);

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

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

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

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

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

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

    MultiRegionProperties updatedProps;
    updatedProps.SetWitnessRegion(witnessRegion);

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

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

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

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

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

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

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

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

JavaScript

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

```
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();
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

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

async function createMultiRegionCluster(region1, region2, witnessRegion) {

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

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

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

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

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

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

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

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

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

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

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

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

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();
```

Java

다음 예시를 사용하여 단일 AWS 리전에서 클러스터를 생성합니다.

```
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);
        }
    }
}
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

```
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

다음 예시를 사용하여 단일 AWS 리전에서 클러스터를 생성합니다.

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

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

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

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

    Client::from_conf(config)
}

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

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

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

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

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

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

```
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

다음 예시를 사용하여 단일 AWS 리전에서 클러스터를 생성합니다.

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

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

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

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

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

main if $PROGRAM_NAME == __FILE__
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

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

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

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

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

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

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

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

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

```
    w.delay = 10
  end

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

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

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

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

main if $PROGRAM_NAME == __FILE__
```

.NET

다음 예시를 사용하여 단일 AWS 리전에서 클러스터를 생성합니다.

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

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

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

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

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

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

                return response;
            }
        }

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

            await Create(region);
```

```
        }
    }
}
```

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

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

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

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

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

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

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

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

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

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

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

                return (cluster1, cluster2);
            }
        }

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

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

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

Golang

다음 예시를 사용하여 단일 AWS 리전에서 클러스터를 생성합니다.

```
package main

import (
```

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

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

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

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

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

 deleteProtect := true

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

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

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

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

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

```
 id := clusterProperties.Identifier

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

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

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

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

 region := "us-east-1"

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

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

다중 리전 클러스터를 생성하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 생성하는 데 시간이 걸릴 수 있습니다.

```
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)
 }

}
```

클러스터 가져오기

Aurora DSQL에서 클러스터에 대한 정보를 반환하는 방법을 알아보려면 다음 정보를 참조하세요.

Python

단일 또는 다중 리전 클러스터에 대한 정보를 가져오려면 다음 예시를 사용합니다.

```
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++

다음 예시를 사용하여 단일 또는 다중 리전 클러스터에 대한 정보를 가져옵니다.

```
#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

단일 또는 다중 리전 클러스터에 대한 정보를 가져오려면 다음 예시를 사용합니다.

```
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

다음 예시를 사용하면 단일 또는 다중 리전 클러스터에 대한 정보를 가져올 수 있습니다.

```
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

다음 예시를 사용하면 단일 또는 다중 리전 클러스터에 대한 정보를 가져올 수 있습니다.

```
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

다음 예시를 사용하면 단일 또는 다중 리전 클러스터에 대한 정보를 가져올 수 있습니다.

```
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

다음 예시를 사용하면 단일 또는 다중 리전 클러스터에 대한 정보를 가져올 수 있습니다.

```
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

다음 예시를 사용하면 단일 또는 다중 리전 클러스터에 대한 정보를 가져올 수 있습니다.

```
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)
 }
}
```

클러스터 업데이트

Aurora DSQL에서 클러스터를 업데이트하는 방법을 알아보려면 다음 정보를 참조하세요. 클러스터를 업데이트하는 데 1~2분이 걸릴 수 있습니다. 잠시 기다린 다음 get cluster를 실행하여 클러스터의 상태를 가져오는 것이 좋습니다.

Python

단일 또는 다중 리전 클러스터를 업데이트하려면 다음 예시를 사용합니다.

```
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++

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
#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

단일 또는 다중 리전 클러스터를 업데이트하려면 다음 예시를 사용합니다.

```
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

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
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

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
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

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
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

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
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

다음 예시를 사용하여 단일 또는 다중 리전 클러스터를 업데이트합니다.

```
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)
 }
}
```

클러스터 삭제

Aurora DSQL에서 클러스터를 삭제하는 방법을 알아보려면 다음 정보를 참조하세요.

Python

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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()
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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++

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
#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;
}
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
#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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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();
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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);
        }
    }
}
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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(())
}
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
equire "aws-sdk-dsql"

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

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

def main
```

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

main if $PROGRAM_NAME == __FILE__
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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);
        }
    }
}
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
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)
 }

}
```

다중 리전 클러스터를 삭제하려면 다음 예시를 사용합니다. 다중 리전 클러스터를 삭제하는 데 시간이 걸릴 수 있습니다.

```
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)
 }
}
```

자습서

다음 자습서 및 GitHub의 샘플 코드는 Aurora DSQL에서 일반적인 작업을 수행하는 데 도움이 됩니다.

- Aurora DSQL에서 Benchbase 사용 - Aurora DSQL에서 작동하는 것으로 확인된 Benchbase 오픈 소스 벤치마킹 유틸리티의 브랜치입니다.
- Aurora DSQL 로더 -이 오픈 소스 Python 스크립트를 사용하면 테스트를 위해 테이블을 채우거나 Aurora DSQL로 데이터를 전송하는 등 사용 사례에 맞게 Aurora DSQL로 데이터를 더 쉽게 로드할 수 있습니다.
- Aurora DSQL 샘플 - GitHub의 `aws-samples/aurora-dsql-samples` 리포지토리에는 AWS SDK, 객체 관계형 매퍼(ORM) 및 웹 프레임워크를 사용하여 다양한 프로그래밍 언어로 Aurora DSQL을 연결하고 사용하는 방법에 대한 코드 예시가 포함되어 있습니다. 이 예시에서는 클라이언트 설치, 인증 처리, CRUD 작업 수행과 같은 일반적인 작업을 수행하는 방법을 보여줍니다.

Amazon Aurora DSQL에서 AWS Lambda 사용

다음 자습서에서는 Aurora DSQL에서 Lambda를 사용하는 방법을 설명합니다.

사전 조건

- Lambda 함수 생성 권한. 자세한 내용은 Getting started with Lambda를 참조하세요.
- Lambda가 생성한 IAM 정책을 생성하거나 수정할 수 있는 권한. `iam:CreatePolicy` 및 `iam:AttachRolePolicy`에 대한 권한이 필요합니다. 자세한 내용은 Actions, resources, and condition keys for IAM을 참조하세요.
- npm v8.5.3 이상이 설치되어 있어야 합니다.
- zip v3.0 이상이 설치되어 있어야 합니다.

AWS Lambda에서 새로운 함수 생성

1. AWS Management Console에 로그인하고 https://console.aws.amazon.com/lambda/에서 AWS Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. `dsql-sample`과 같은 이름을 입력합니다.
4. Lambda가 기본 Lambda 권한으로 새 역할을 생성하도록 기본 설정을 편집하지 마세요.
5. 함수 생성을 선택합니다.

Lambda 실행 역할이 클러스터에 연결하도록 권한 부여

1. Lambda 함수에서 구성 > 권한을 선택합니다.

2. 역할 이름을 선택하여 IAM 콘솔에서 실행 역할을 엽니다.
3. 권한 추가 > 인라인 정책 생성을 선택하고 JSON 편집기를 사용합니다.
4. 작업에서 다음 작업을 붙여 넣어 IAM ID가 관리자 데이터베이스 역할을 사용하여 연결하도록 승인합니다.

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

> **Note**
>
> 시작하기 위한 사전 조건 단계를 최소화하기 위해 관리자 역할을 사용합니다. 프로덕션 애플리케이션에 관리자 데이터베이스 역할을 사용해서는 안 됩니다. 데이터베이스에 대한 최소한의 권한 부여를 사용하여 사용자 지정 데이터베이스 역할을 생성하는 방법은 데이터베이스 역할 및 IAM 인증 사용 섹션을 참조하세요.

5. 리소스에서 클러스터의 Amazon 리소스 이름(ARN)을 추가합니다. 와일드카드를 사용할 수도 있습니다.

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

6. 다음을 선택합니다.
7. 정책의 이름을 입력합니다(예: `dsql-sample-dbconnect`).
8. 정책 생성을 선택합니다.

Lambda에 업로드할 패키지 생성

1. `myfunction`이라는 폴더를 생성합니다.
2. 폴더 안에 다음 내용이 담긴 `package.json`이라는 파일을 새로 생성합니다.

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

3. 폴더 안에서 디렉터리에 다음 내용이 담긴 `index.mjs`라는 파일을 생성합니다.

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


async function dsql_sample(clusterEndpoint, region) {
  let client;
  try {
    // The token expiration time is optional, and the default value 900 seconds
    const signer = new DsqlSigner({
      hostname: clusterEndpoint,
      region,
    });
    const token = await signer.getDbConnectAdminAuthToken();
    // <https://node-postgres.com/apis/client>
    // By default `rejectUnauthorized` is true in TLS options
    // <https://nodejs.org/api/tls.html#tls_tls_connect_options_callback>
    // The config does not offer any specific parameter to set sslmode to verify-
full
    // Settings are controlled either via connection string or by setting
    // rejectUnauthorized to false in ssl options
    client = new Client({
      host: clusterEndpoint,
      user: "admin",
      password: token,
      database: "postgres",
      port: 5432,
      // <https://node-postgres.com/announcements> for version 8.0
      ssl: true,
      rejectUnauthorized: false
    });

    // Connect
    await client.connect();

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

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

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

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

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


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

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

4. 다음과 같은 명령을 사용하여 패키지를 생성합니다.

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

코드 패키지 업로드 및 Lambda 함수 테스트

1. Lambda 함수의 코드 탭에서 에서 업로드 > .zip 파일을 선택합니다.
2. 생성한 `pkg.zip`을 업로드합니다. 자세한 내용은 .zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포를 참조하세요.
3. Lambda 함수의 테스트 탭에서 다음 JSON 페이로드를 붙여 넣고 클러스터 ID를 사용하도록 수정합니다.
4. Lambda 함수의 테스트 탭에서 수정된 다음 이벤트 JSON을 사용하여 클러스터의 엔드포인트를 지정합니다.

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

5. 이벤트 이름을 입력합니다(예: `dsql-sample-test`). 저장을 선택합니다.
6. 테스트를 선택합니다.
7. 세부 정보를 선택하여 실행 응답 및 로그 출력을 확장합니다.
8. 성공한 경우 Lambda 함수 실행 응답은 200 상태 코드를 반환할 것입니다.

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

데이터베이스가 오류를 반환하거나 데이터베이스에 대한 연결이 실패하면 Lambda 함수 실행 응답은 500 상태 코드를 반환합니다.

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

Amazon Aurora DSQL의 백업 및 복원

Amazon Aurora DSQL은 클라우드 및 온프레미스에서 AWS 서비스 전반에 걸쳐 백업을 쉽게 중앙 집중화하고 자동화할 수 있는 완전관리형 데이터 보호 서비스인 AWS Backup과의 통합을 통해 규정 준수 및 비즈니스 연속성 요구 사항을 충족할 수 있도록 지원합니다. 이 서비스는 단일 리전 및 다중 리전 Aurora DSQL 클러스터에 대한 백업 생성, 관리 및 복원을 간소화합니다.

주요 기능은 다음과 같습니다.

- AWS Management Console, SDK 또는 AWS CLI를 통한 중앙 집중식 백업 관리
- 전체 클러스터 백업
- 자동 백업 일정 및 보존 정책
- 교차 리전 및 교차 계정 기능
- 저장하는 모든 백업을 위한 Write Once Read Many(WORM) 구성

AWS Backup Vault Lock의 기능과 Aurora DSQL에 사용할 수 있는 다양한 AWS Backup 기능에 대한 자세한 내용은 AWS Backup 개발자 안내서의 Vault lock benefits 및 AWS Backup feature availability를 참조하세요.

AWS Backup 시작

AWS Backup은 Aurora DSQL 클러스터의 전체 사본을 생성합니다. Getting started with AWS Backup의 단계에 따라 Aurora DSQL에 AWS Backup 사용을 시작할 수 있습니다.

1. 즉각적인 보호를 위해 온디맨드 백업을 생성합니다.
2. 예약된 자동 백업을 위한 백업 계획을 수립합니다.
3. 보존 기간 및 리전 간 복사를 구성합니다.
4. 백업 활동에 대한 모니터링 및 알림을 설정합니다.

백업 복원

Aurora DSQL 클러스터를 복원할 때 AWS Backup이 항상 소스 데이터를 보존하기 위해 새 클러스터를 생성합니다. Aurora DSQL 단일 리전 클러스터를 복원하려면 https://console.aws.amazon.com/backup

또는 CLI를 사용하여 복원할 복구 시점(백업)을 선택합니다. 백업에서 생성할 새 클러스터의 설정을 구성합니다.

Aurora DSQL 다중 리전 클러스터 복원은 AWS CLI를 통해서만 지원됩니다. Aurora DSQL 다중 리전 클러스터를 복원하려면 AWS Backup 및 Aurora DSQL CLI를 모두 사용해야 합니다.

Aurora DSQL 다중 리전 클러스터를 복원하는 방법

1. 다중 리전 클러스터의 복구 시점을 선택합니다.
2. 복구 시점을 다중 리전 클러스터를 지원하는 다른 AWS 리전에 복사합니다.

> **Note**
>
> 다중 리전 클러스터를 지원하지 않는 리전은 복원 작업에 실패합니다.

3. AWS Backup CLI를 사용하여 각 클러스터에 대한 복원 작업을 시작합니다.
4. 다중 리전 클러스터 구성 설명서를 사용하여 새로 생성된 Aurora DSQL 클러스터를 피어링합니다.

이러한 단계에 대한 자세한 지침은 Amazon Aurora DSQL restore 설명서를 참조하세요.

다중 리전 Aurora DSQL 클러스터로 복원하기 위해 단일 AWS 리전에서 가져온 백업을 사용할 수 있습니다. 그러나 복원 프로세스를 시작하기 전에 먼저 다중 리전 클러스터를 지원하는 다른 AWS 리전에 백업을 복사해야 합니다. 이 단계를 수행하면 복원 작업이 성공적으로 완료될 수 있습니다. 강력한 재해 복구 옵션을 활성화하고 규정 준수 요구 사항을 충족하려면 미국 동부(버지니아 북부), 미국 동부(오하이오) 또는 미국 서부(오리건)와 같은 주요 AWS 리전에 백업 복사본을 생성하는 것이 좋습니다.

모니터링 및 규정 준수

AWS Backup은 다음 리소스를 사용하여 백업 및 복원 작업에 대한 포괄적인 가시성을 제공합니다.

- 백업 및 복원 작업을 추적하기 위한 중앙 집중식 대시보드
- CloudWatch 및 CloudTrail과 통합
- 규정 준수 보고 및 감사를 위한 AWS Backup Audit Manager

Aurora DSQL을 사용하는 동안 사용자, 역할 또는 AWS 서비스가 수행한 작업의 레코드 로깅에 대한 자세한 내용은 AWS CloudTrail을 사용하여 Aurora DSQL 작업 로깅 섹션을 참조하세요.

추가 리소스

AWS Backup 기능 및 Aurora DSQL과 함께 사용하는 방법에 대해 자세히 알아보려면 다음 리소스를 참조하세요.

- Managed policies for AWS Backup
- Amazon Aurora DSQL restore
- Supported services by AWS 리전
- Encryption for backups in AWS Backup

Aurora DSQL에 AWS Backup을 사용하면 관리 오버헤드를 최소화하면서 중요한 데이터베이스 리소스를 보호하는 강력하고 규정을 준수하는 자동 백업 전략을 구현할 수 있습니다. 단일 클러스터를 관리하든 복잡한 다중 리전 배포를 관리하든 AWS Backup은 데이터를 안전하고 복구 가능하게 유지하는 데 필요한 도구를 제공합니다.

Aurora DSQL 모니터링 및 로깅

모니터링 및 로깅은 Amazon Aurora DSQL 리소스의 신뢰성, 가용성 및 성능을 유지하는 데 중요한 부분입니다. 다중 지점 장애를 쉽게 디버깅할 수 있도록 Aurora DSQL 리소스의 모든 부분에서 모니터링하고 로깅 데이터를 수집해야 합니다.

- Amazon CloudWatch는 AWS에서 실행하는 AWS 리소스와 애플리케이션을 실시간으로 모니터링합니다. 지표를 수집 및 추적하고, 사용자 지정 대시보드를 생성할 수 있으며, 지정된 지표가 지정한 임곗값에 도달하면 사용자에게 알리거나 조치를 취하도록 경보를 설정할 수 있습니다. 예를 들어 CloudWatch에서 Amazon EC2 인스턴스의 CPU 사용량 또는 기타 지표를 추적하고 필요할 때 자동으로 새 인스턴스를 시작할 수 있습니다. 자세한 내용은 Amazon CloudWatch 사용 설명서를 참조하세요.
- AWS CloudTrail은 직접 수행하거나 AWS 계정을 대신하여 수행한 API 직접 호출 및 관련 이벤트를 캡처하고 지정한 Amazon S3 버킷에 로그 파일을 전송합니다. 어떤 사용자 및 계정이 AWS를 호출했는지 어떤 소스 IP 주소에 호출이 이루어졌는지 언제 호출이 발생했는지 확인할 수 있습니다. 자세한 내용은 AWS CloudTrail 사용 설명서를 참조하세요.

Aurora DSQL 클러스터 상태 보기

Aurora DSQL 클러스터 상태는 클러스터 상태 및 연결에 대한 중요한 정보를 제공합니다. AWS Management Console, AWS CLI 또는 Aurora DSQL API를 사용하여 클러스터 및 클러스터 인스턴스의 상태를 볼 수 있습니다.

Aurora DSQL 클러스터 상태 및 정의

다음 표에서는 Aurora DSQL 클러스터에서 가능한 각 상태와 각 상태가 의미하는 바를 설명합니다.

상태 표시기	설명
[생성 중]	Aurora DSQL이 클러스터에 대한 리소스를 생성하거나 구성하려고 합니다. 클러스터가 이 상태에 있는 동안에는 모든 연결 시도가 실패합니다.
활성	클러스터가 작동 중이며 사용할 준비가 되었습니다.
유휴	Aurora DSQL이 클러스터에 대해 구성된 리소스를 회수할 수 있을 만큼 충분히 오랫동안 유휴 상태이면 클러스터는 유휴 상태가 됩니다. 유휴 클러스터에 연결하면 Aurora DSQL이 클러스터를 다시 활성 상태로 전환합니다.

상태 표시기	설명
비활성	장기간 클러스터에 활동이 없으면 클러스터가 비활성화됩니다. 비활성 클러스터에 연결하려고 하면 Aurora DSQL이 자동으로 클러스터를 다시 활성 상태로 전환합니다.
업데이트 중	클러스터 구성을 변경하면 클러스터가 업데이트 중 상태로 전환됩니다.
[삭제 중]	삭제 요청을 제출하면 클러스터가 삭제 중 상태로 전환됩니다.
Deleted	클러스터가 성공적으로 삭제되었습니다.
실패	Aurora DSQL에서 오류가 발생하여 클러스터를 생성할 수 없습니다.
설정 보류 중	다중 리전 클러스터에만 해당됩니다. 감시 리전이 있는 첫 번째 리전에서 다중 리전 클러스터를 생성하면 다중 리전 클러스터가 설정 보류 중 상태가 됩니다. 클러스터 생성은 보조 리전에 다른 클러스터를 생성하고 두 클러스터를 함께 피어링할 때까지 일시 중지됩니다.
삭제 보류 중	다중 리전 클러스터에만 해당됩니다. 다중 리전 클러스터는 다중 리전 클러스터에서 클러스터를 삭제할 때 삭제 보류 중 상태가 됩니다. 마지막 피어 클러스터를 삭제하면 클러스터가 삭제 중 상태로 전환됩니다.

Aurora DSQL 클러스터 상태 보기

클러스터의 상태를 보려면 AWS Management Console, AWS CLI 또는 Aurora DSQL API를 사용합니다.

콘솔

다음 단계에 따라 AWS Management Console에서 클러스터 상태를 확인합니다.

콘솔에서 클러스터 상태를 보는 방법

1. https://console.aws.amazon.com/dsql에서 Aurora DSQL 콘솔을 엽니다.
2. 탐색 창에서 클러스터를 선택합니다.
3. 대시보드에서 각 클러스터의 상태를 확인합니다.

AWS CLI

AWS CLI 명령을 사용하여 단일 클러스터의 상태를 확인합니다.

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

다음 명령을 실행하여 모든 클러스터의 상태를 나열합니다.

```
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
```

이 샘플 출력은 두 개의 활성 클러스터와 삭제 중인 클러스터 한 개를 보여줍니다.

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

Amazon CloudWatch로 Aurora DSQL 모니터링

원시 데이터를 수집하여 읽기 가능한 실시간에 가까운 지표로 처리하는 CloudWatch를 사용하여 Aurora DSQL을 모니터링할 수 있습니다. CloudWatch는 이러한 통계를 15개월간 보관하므로 웹 애플리케이션 또는 서비스의 성능에 대해 더 잘 파악할 수 있습니다. 특정 임곗값을 감시하고 임곗값에 도달하면 알림을 보내거나 조치를 취하도록 경보를 설정합니다. Aurora DSQL에 사용할 수 있는 다음 사용 및 관찰성 지표를 검토합니다.

자세한 내용은 Amazon CloudWatch 사용 설명서를 참조하세요.

관찰성 및 성능

이 표에는 Aurora DSQL에 대한 관찰성 지표가 요약되어 있습니다. 여기에는 전체 워크로드 특성화를 제공하기 위해 읽기 전용 및 총 트랜잭션을 추적하기 위한 지표가 포함됩니다. 성능 문제 및 동시성 충돌을 식별하는 데 도움이 되도록 쿼리 제한 시간 및 OCC 충돌률과 같은 실행 가능한 지표가 포함됩니다. 활성 및 총계라는 세션 관련 지표는 시스템의 현재 로드에 대한 인사이트를 제공합니다.

CloudWatch 지표 이름	지표	단위	설명
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

사용량 지표

Aurora DSQL은 분산 처리 단위(DPU)라는 정규화된 단일 결제 단위를 사용하여 쿼리 처리, 읽기 및 쓰기와 같은 모든 요청 기반 활동을 측정합니다.

CloudWatch 지표 이름	지표	차원: ResourceId	단위	설명
WriteDPU	Write Units	<cluster-id>	DPU	Approximates the write active-use component of your Aurora DSQL cluster DPU usage.
MultiRegionWriteDPU	Multi-Region Write Units	<cluster-id>	DPU	Applicable for Multi-Region clusters: Approximates the multi-Region write active-use component of your Aurora DSQL cluster DPU usage.
ReadDPU	Read Units	<cluster-id>	DPU	Approximates the read active-use component of your Aurora DSQL cluster DPU usage.
ComputeDPU	Compute Units	<cluster-id>	DPU	Approximates the compute active-use component of your Aurora

CloudWatch 지표 이름	지표	차원: ResourceId	단위	설명
				DSQL cluster DPU usage.
TotalDPU	Total Units	<cluster-id>	DPU	Approximates the total active-use component of your Aurora DSQL cluster DPU usage.

AWS CloudTrail을 사용하여 Aurora DSQL 작업 로깅

Amazon Aurora DSQL은 사용자, 역할 또는 AWS 서비스가 수행한 작업에 대한 레코드를 제공하는 서비스인 AWS CloudTrail과 통합됩니다. CloudTrail에는 두 가지 유형의 이벤트, 즉 관리 이벤트와 데이터 이벤트가 있습니다. 관리 이벤트는 AWS 리소스 구성 변경을 감사하기 위해 내보내집니다. 데이터 이벤트는 일반적으로 서비스 데이터 플레인에서 AWS 리소스 사용량을 캡처합니다.

CloudTrail은 Aurora DSQL에 대한 모든 API 직접 호출을 이벤트로 캡처합니다. Aurora DSQL은 콘솔 활동을 관리 이벤트로 기록합니다. 또한 클러스터에 대한 인증된 연결 시도를 데이터 이벤트로 캡처합니다.

CloudTrail에서 수집한 정보를 사용하여 Aurora DSQL에 수행된 요청, 요청이 수행된 IP 주소, 요청을 수행한 시점, 요청을 수행한 사용자 ID 및 추가 세부 정보를 확인할 수 있습니다.

계정을 생성할 때 AWS 계정에서 CloudTrail이 기본적으로 활성화되며, CloudTrail 이벤트 기록에 자동으로 액세스할 수 있습니다. CloudTrail 이벤트 기록은 지난 90일 간 AWS 리전의 관리 이벤트에 대해 보기, 검색 및 다운로드가 가능하고, 수정이 불가능한 레코드를 제공합니다. 자세한 설명은 AWS CloudTrail 사용 설명서의 CloudTrail 이벤트 기록 작업을 참조하세요. 이벤트 기록 레코드 생성에는 CloudTrail 요금이 부과되지 않습니다.

Aurora DSQL에 대한 이벤트를 포함하여 AWS 계정에 이벤트에 대한 지속적인 레코드를 생성하려면 추적 또는 AWS CloudTrail Lake 이벤트 데이터 저장소(AWS CloudTrail 이벤트에 대한 중앙 집중식 스토리지 및 분석 솔루션)를 생성합니다. 추적 생성에 대한 자세한 내용은 Working with CloudTrail trails을 참조하세요. 이벤트 데이터 저장소 설정 및 관리에 대한 자세한 내용은 CloudTrail Lake event data stores를 참조하세요.

CloudTrail의 Aurora DSQL 관리 이벤트

CloudTrail 관리 이벤트는 AWS 계정의 리소스에 대해 수행되는 관리 작업에 대한 정보를 제공합니다. 이를 컨트롤 플레인 작업이라고도 합니다. 기본적으로 CloudTrail은 이벤트 기록에 관리 이벤트를 캡처합니다.

Amazon Aurora DSQL은 모든 Aurora DSQL 컨트롤 플레인 작업을 관리 이벤트로 로깅합니다. Aurora DSQL이 CloudTrail에 로깅하는 Amazon Aurora DSQL 컨트롤 플레인 작업의 목록은 Aurora DSQL API reference를 참조하세요.

컨트롤 플레인 로그

Amazon Aurora DSQL은 다음 Aurora DSQL 컨트롤 플레인 작업을 CloudTrail에 관리 이벤트로 로깅합니다.

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

백업 및 복원 로그

Amazon Aurora DSQL은 다음 Aurora DSQL 백업 및 복원 작업을 CloudTrail에 관리 이벤트로 로깅합니다.

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

AWS Backup을 사용하여 Aurora DSQL 클러스터를 보호하는 방법에 대한 자세한 내용은 Amazon Aurora DSQL의 백업 및 복원 섹션을 참조하세요.

AWS KMS 로그

Amazon Aurora DSQL은 다음 AWS KMS 작업을 CloudTrail에 관리 이벤트로 로깅합니다.

- `GenerateDataKey`
- `Decrypt`

CloudTrail 로그가 Aurora DSQL이 사용자를 대신하여 AWS KMS에 보내는 요청을 추적하는 방법에 대한 자세한 내용은 AWS KMS와 Aurora DSQL 상호 작용 모니터링 섹션을 참조하세요.

CloudTrail의 Aurora DSQL 데이터 이벤트

CloudTrail 데이터 이벤트는 일반적으로 리소스상에서 또는 리소스 내에서 수행되는 리소스 작업에 대한 정보를 제공합니다. 또한 서비스의 데이터 플레인 작업을 캡처하는 데에도 사용됩니다. 데이터 이벤트가 대량 활동인 경우도 있습니다. 기본적으로 CloudTrail은 데이터 이벤트를 로깅하지 않습니다. CloudTrail 이벤트 기록은 데이터 이벤트를 기록하지 않습니다.

데이터 이벤트를 로깅하는 방법에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 Logging data events with the AWS Management Console 및 Logging data events with the AWS Command Line Interface를 참조하세요.

데이터 이벤트에는 추가 요금이 적용됩니다. CloudTrail 요금에 대한 자세한 내용은 AWS CloudTrail 요금을 참조하세요.

Aurora DSQL의 경우 CloudTrail은 Aurora DSQL 클러스터에 대한 연결 시도를 데이터 이벤트로 캡처합니다. 다음 표에는 데이터 이벤트를 로깅할 수 있는 Aurora DSQL 리소스 유형이 나와 있습니다. 리소스 유형(콘솔) 열에는 CloudTrail 콘솔의 리소스 유형 목록에서 선택할 값이 표시됩니다. resources.type 값 열에는 AWS CLI 또는 CloudTrail API를 사용하여 고급 이벤트 선택기를 구성할 때 지정하는 `resources.type` 값이 표시됩니다. CloudTrail에 로깅되는 데이터 API 열에는 리소스 유형에 대해 CloudTrail에 로깅된 API 호출이 표시됩니다.

리소스 유형(콘솔)	resources.type 값	CloudTrail에 로깅되는 데이터 API
Amazon Aurora DSQL	`AWS::DSQL::Cluster`	• `DbConnect` • `DbConnectAdmin`

고급 이벤트 선택기를 구성하여 필터링된 이벤트만 로깅하도록 `eventName` 및 `resources.ARN` 필드를 기준으로 필터링할 수 있습니다. 이러한 필드에 대한 자세한 내용은 AWS CloudTrail API 참조의 AdvancedFieldSelector 섹션을 참조하세요.

다음 예시에서는 AWS CLI를 사용하여 Aurora DSQL에 대한 데이터 이벤트를 수신하도록 `dsql-data-events-trail`을 구성하는 방법을 보여줍니다.

```
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 DSQL의 보안

AWS에서 클라우드 보안은 가장 중요합니다. AWS 고객은 보안에 가장 보안에 민감한 조직의 요구 사항에 부합하도록 빌드된 데이터 센터 및 네트워크 아키텍처의 혜택을 누릴 수 있습니다.

보안은 AWS와 사용자의 공동 책임입니다. 공동 책임 모델은 이 사항을 클라우드의 보안 및 클라우드 내 보안으로 설명합니다.

- 클라우드의 보안 - AWS는 AWS 클라우드에서 AWS 서비스를 실행하는 인프라를 보호합니다. AWS는 또한 안전하게 사용할 수 있는 서비스를 제공합니다. 타사 감사자는 AWS규정 준수 프로그램의 일환으로 보안 효과를 정기적으로 테스트하고 검증합니다. Amazon Aurora DSQL에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 규정 준수 프로그램에 따른 AWS 서비스 범위를 참조하세요.
- 클라우드 내 보안 – 귀하의 책임은 귀하가 사용하는 AWS서비스에 의해 결정됩니다. 또한 귀하는 귀사의 데이터 민감도, 귀사의 요구 사항, 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 Aurora DSQL 사용 시 공동 책임 모델을 적용하는 방법을 이해하는 데 도움이 됩니다. 이어지는 주제에서는 보안 및 규정 준수 목표를 충족하도록 Aurora DSQL을 구성하는 방법을 보여줍니다. 또한 Aurora DSQL 리소스를 모니터링하고 보호하는 데 도움이 되는 다른 AWS 서비스를 사용하는 방법을 알아봅니다.

주제

- Aurora DSQL의 보안 모범 사례

Amazon Aurora DSQL의 AWS 관리형 정책

AWS 관리형 정책은 AWS에 의해 생성되고 관리되는 독립 실행형 정책입니다. AWS 관리형 정책은 사용자, 그룹 및 역할에 권한 할당을 시작할 수 있도록 많은 일반 사용 사례에 대한 권한을 제공하도록 설계되었습니다.

AWS 관리형 정책은 모든 AWS 고객이 사용할 수 있기 때문에 특정 사용 사례에 대해 최소 권한을 부여하지 않을 수 있습니다. 사용 사례에 고유한 고객 관리형 정책을 정의하여 권한을 줄이는 것이 좋습니다.

AWS 관리형 정책에서 정의한 권한은 변경할 수 없습니다. 만약 AWS가 AWS 관리형 정책에 정의된 권한을 업데이트할 경우 정책이 연결되어 있는 모든 위탁자 ID(사용자, 그룹 및 역할)에도 업데이트가 적용됩니다. 새 AWS 서비스을(를) 시작하거나 새 API 작업을 기존 서비스에 이용하는 경우, AWS가 AWS 관리형 정책을 업데이트할 가능성이 높습니다.

자세한 내용은 IAM 사용 설명서의 AWS 관리형 정책을 참조하세요.

AWS 관리형 정책: AmazonAuroraDSQLFullAccess

사용자, 그룹 및 역할에 `AmazonAuroraDSQLFullAccess`를 연결할 수 있습니다.

이 정책은 Aurora DSQL에 대한 전체 관리 액세스를 허용하는 권한을 부여합니다. 이러한 권한이 있는 위탁자는 다중 리전 클러스터를 포함하여 Aurora DSQL 클러스터를 생성, 삭제 및 업데이트할 수 있습니다. 클러스터에서 태그를 추가하고 제거할 수 있습니다. 클러스터를 나열하고 개별 클러스터에 대한 정보를 볼 수 있습니다. Aurora DSQL 클러스터에 연결된 태그를 볼 수 있습니다. 관리자를 포함한 어느 사용자로든 데이터베이스에 연결할 수 있습니다. 백업 및 복원 작업 시작, 중지 및 모니터링을 포함하여 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행할 수 있습니다. 이 정책에는 클러스터 암호화에 사용되는 고객 관리형 키에 대한 작업을 허용하는 AWS KMS 권한이 포함되어 있습니다. 계정의 CloudWatch에서 어느 지표든 볼 수 있습니다. 또한 클러스터를 생성하는 데 필요한 `dsql.amazonaws.com` 서비스에 대한 서비스 연결 역할을 생성할 수 있는 권한도 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `dsql` - 위탁자에게 Aurora DSQL에 대한 전체 액세스 권한을 부여합니다.
- `cloudwatch` - Amazon CloudWatch에 지표 데이터 포인트를 게시할 수 있는 권한을 부여합니다.
- `iam` – 서비스 연결 역할을 생성할 수 있는 권한을 부여합니다.
- `backup and restore` - Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 시작, 중지 및 모니터링할 수 있는 권한을 부여합니다.
- `kms` - 클러스터를 생성, 업데이트 또는 연결할 때 Aurora DSQL 클러스터 암호화에 사용되는 고객 관리형 키에 대한 액세스를 검증하는 데 필요한 권한을 부여합니다.

`AmazonAuroraDSQLFullAccess` 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 AmazonAuroraDSQLFullAccess에서 찾을 수 있습니다.

AWS 관리형 정책: AmazonAuroraDSQLReadOnlyAccess

사용자, 그룹 및 역할에 `AmazonAuroraDSQLReadOnlyAccess`를 연결할 수 있습니다.

Aurora DSQL에 대한 읽기 액세스를 허용합니다. 이러한 권한이 있는 위탁자는 클러스터를 나열하고 개별 클러스터에 대한 정보를 볼 수 있습니다. Aurora DSQL 클러스터에 연결된 태그를 볼 수 있습니다. 계정의 CloudWatch에서 어느 지표든 검색하고 볼 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `dsql` - Aurora DSQL의 모든 리소스에 읽기 전용 권한을 부여합니다.
- `cloudwatch` - CloudWatch 지표 데이터의 배치 양을 검색하고 검색된 데이터에 대해 지표 수식을 계산할 수 있는 권한을 부여합니다.

`AmazonAuroraDSQLReadOnlyAccess` 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 AmazonAuroraDSQLReadOnlyAccess에서 찾을 수 있습니다.

AWS 관리형 정책: AmazonAuroraDSQLConsoleFullAccess

사용자, 그룹 및 역할에 `AmazonAuroraDSQLConsoleFullAccess`를 연결할 수 있습니다.

AWS Management Console을 통해 Amazon Aurora DSQL에 대한 전체 관리 액세스를 허용합니다. 이러한 권한이 있는 위탁자는 콘솔을 사용하여 다중 리전 클러스터를 포함하여 Aurora DSQL 클러스터를 생성, 삭제 및 업데이트할 수 있습니다. 클러스터를 나열하고 개별 클러스터에 대한 정보를 볼 수 있습니다. 계정의 어느 리소스에 대해서든 태그를 볼 수 있습니다. 관리자를 포함한 어느 사용자로든 데이터베이스에 연결할 수 있습니다. 백업 및 복원 작업 시작, 중지 및 모니터링을 포함하여 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행할 수 있습니다. 이 정책에는 클러스터 암호화에 사용되는 고객 관리형 키에 대한 작업을 허용하는 AWS KMS 권한이 포함되어 있습니다. 이들은 AWS Management Console에서 AWS CloudShell을 시작할 수 있습니다. 계정의 CloudWatch에서 어느 지표든 볼 수 있습니다. 또한 클러스터를 생성하는 데 필요한 `dsql.amazonaws.com` 서비스에 대한 서비스 연결 역할을 생성할 수 있는 권한도 있습니다.

`AmazonAuroraDSQLConsoleFullAccess` 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 AmazonAuroraDSQLConsoleFullAccess에서 찾을 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `dsql` - AWS Management Console을 통해 Aurora DSQL의 모든 리소스에 전체 관리 권한을 부여합니다.
- `cloudwatch` - CloudWatch 지표 데이터의 배치 양을 검색하고 검색된 데이터에 대해 지표 수식을 계산할 수 있는 권한을 부여합니다.
- `tag` - 직접 호출하는 계정에 대해 지정된 AWS 리전에서 현재 사용 중인 태그 키와 값을 반환할 수 있는 권한을 부여합니다.
- `backup and restore` - Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 시작, 중지 및 모니터링할 수 있는 권한을 부여합니다.
- `kms` - 클러스터를 생성, 업데이트 또는 연결할 때 Aurora DSQL 클러스터 암호화에 사용되는 고객 관리형 키에 대한 액세스를 검증하는 데 필요한 권한을 부여합니다.
- `cloudshell` - Aurora DSQL과 상호 작용하기 위해 AWS CloudShell을 시작할 수 있는 권한을 부여합니다.
- `ec2` - Aurora DSQL 연결에 필요한 Amazon VPC 엔드포인트 정보를 볼 수 있는 권한을 부여합니다.

`AmazonAuroraDSQLReadOnlyAccess` 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 AmazonAuroraDSQLReadOnlyAccess에서 찾을 수 있습니다.

AWS 관리형 정책: AuroraDSQLServiceRolePolicy

AuroraDSQLServiceRolePolicy를 IAM 엔터티에 연결할 수 없습니다. 이 정책은 Aurora DSQL이 계정 리소스에 액세스할 수 있도록 허용하는 서비스 연결 역할에 연결됩니다.

`AuroraDSQLServiceRolePolicy` 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 AuroraDSQLServiceRolePolicy에서 찾을 수 있습니다.

AWS 관리형 정책에 대한 Aurora DSQL 업데이트

이 서비스가 이러한 변경 내용을 추적하기 시작한 이후부터 Aurora DSQL의 AWS 관리형 정책 업데이트에 대한 세부 정보를 봅니다. 이 페이지의 변경 사항에 대한 자동 알림을 받으려면 Aurora DSQL 문서 기록 페이지에서 RSS 피드를 구독하세요.

변경 사항	설명	날짜
AmazonAuroraDSQLFullAccess 업데이트	작업 시작, 중지 및 모니터링을 포함하여 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행하는 기능을 추가합니다. 또한 클러스터 암호화에 고객 관리형 KMS 키를 사용할 수 있는 기능도 추가합니다. 자세한 내용은 AmazonAuroraDSQLFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 21일
AmazonAuroraDSQLConsoleFullAccess 업데이트	AWS Console Home을 통해 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행하는 기능을 추가합니다. 여기에는 작업 시작, 중지 및 모니터링이 포함됩니다. 또한 클러스터 암	2025년 5월 21일

변경 사항	설명	날짜
	호화 및 AWS CloudShell 시작을 위해 고객 관리형 KMS 키를 사용하는 것을 지원합니다. 자세한 내용은 AmazonAuroraDSQLConsoleFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	
AmazonAuroraDSQLFullAccess 업데이트	이 정책은 여러 AWS 리전, 즉, `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , `RemovePeerCluster` 에서 데이터베이스 클러스터를 생성하고 관리할 수 있는 네 가지 새로운 권한을 추가합니다. 이러한 권한에는 리소스 수준 제어 및 조건 키가 포함되어 있어 수정할 수 있는 클러스터 사용자를 제어할 수 있습니다. 또한 이 정책은 AWS PrivateLink를 통해 Aurora DSQL 클러스터에 연결하는 데 도움이 되는 `GetVpcEndpointServiceName` 권한을 추가합니다. 자세한 내용은 AmazonAuroraDSQLFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일

변경 사항	설명	날짜
AmazonAuroraDSQLReadOnlyAccess 업데이트	AWS PrivateLink를 통해 Aurora DSQL 클러스터에 연결할 때 올바른 VPC 엔드포인트 서비스 이름을 결정하는 기능을 포함합니다. Aurora DSQL은 셀마다 고유한 엔드포인트를 생성하므로 이 API를 사용하면 클러스터에 대한 올바른 엔드포인트를 식별하고 연결 오류를 방지할 수 있습니다. 자세한 내용은 AmazonAuroraDSQLReadOnlyAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일
AmazonAuroraDSQLConsoleFullAccess 업데이트	다중 리전 클러스터 관리 및 VPC 엔드포인트 연결을 지원하기 위해 Aurora DSQL에 새 권한을 추가합니다. 새 권한은 `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , `RemovePeerCluster` , `GetVpcEndpointServiceName` 입니다. 자세한 내용은 AmazonAuroraDSQLConsoleFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일

변경 사항	설명	날짜
AuroraDsqlServiceLinkedRolePolicy 업데이트	`AWS/AuroraDSQL` 에 지표를 게시하고 정책에 `AWS/Usage` `CloudWatch` 네임스페이스를 게시하는 기능을 추가합니다. 이 기능을 사용하면 연결된 서비스 또는 역할이 CloudWatch 환경에 보다 포괄적인 용도 및 성능 데이터를 내보낼 수 있습니다. 자세한 내용은 AuroraDsqlServiceLinkedRolePolicy 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 8일
페이지 생성됨	Amazon Aurora DSQL과 관련된 AWS 관리형 정책 추적 시작	2024년 12월 3일

Amazon Aurora DSQL의 데이터 보호

공동 책임 모델은 의 데이터 보호에 적용됩니다. 이 모델에서 설명하는 것처럼 은 모든 AWS 클라우드를 실행하는 글로벌 인프라를 보호할 책임이 있습니다. 사용자는 인프라에서 호스팅되는 콘텐츠를 관리해야 합니다. 사용하는 의 보안 구성과 관리 태스크에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 대한 자세한 내용은 데이터 프라이버시 FAQ를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 보안 블로그의 공동 책임 모델 및 GDPR 블로그 게시물을 참조하세요.

데이터를 보호하려면 자격 증명을 보호하고 AWS IAM Identity Center 또는 AWS Identity and Access Management를 통해 개별 사용자를 설정하는 것이 좋습니다. 이렇게 하면 개별 사용자에게 자신의 직무를 충실히 이행하는 데 필요한 권한만 부여됩니다. 또한 다음과 같은 방법으로 데이터를 보호하는 것이 좋습니다.

- 각 계정에 다중 인증(MFA)을 사용하세요.
- SSL/TLS를 사용하여 리소스와 통신하세요. TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- AWS CloudTrail로 API 및 사용자 활동 로깅을 설정하세요. 활동 캡처에 추적을 사용하는 방법에 대한 자세한 내용은 사용 설명서의 Working with trails을 참조하세요.

- 암호화 솔루션을 AWS 서비스 내의 모든 기본 보안 제어 기능과 함께 사용하세요.
- Amazon S3에 저장된 민감한 데이터를 검색하고 보호하는 데 도움이 되는 Amazon Macie와 같은 고급 관리형 보안 서비스를 사용하세요.

고객의 이메일 주소와 같은 기밀 정보나 중요한 정보는 태그나 이름 필드와 같은 자유 형식 텍스트 필드에 입력하지 않는 것이 좋습니다. 여기에는 콘솔, API, AWS CLI 또는 AWS SDK를 사용하여 작업하는 경우가 포함됩니다. 이름에 사용되는 태그 또는 자유 형식 텍스트 필드에 입력하는 모든 데이터는 청구 또는 진단 로그에 사용될 수 있습니다. 외부 서버에 URL을 제공할 때 해당 서버에 대한 요청을 검증하기 위해 자격 증명을 URL에 포함해서는 안 됩니다.

데이터 암호화

Amazon Aurora DSQL에서는 미션 크리티컬 및 기본 데이터 스토리지에 적합하게 설계된, 내구성이 뛰어난 스토리지 인프라를 제공합니다. 데이터는 Aurora DSQL 리전의 여러 시설에 걸쳐 여러 디바이스에 중복 저장됩니다.

전송 중 암호화

기본적으로 전송 중 암호화가 구성됩니다. Aurora DSQL은 TLS를 사용하여 SQL 클라이언트와 Aurora DSQL 간의 모든 트래픽을 암호화합니다.

AWS CLI, SDK 또는 API 클라이언트와 Aurora DSQL 엔드포인트 간에 전송 중 데이터를 암호화하고 서명하는 방법은 다음과 같습니다.

- Aurora DSQL이 전송 중 데이터를 암호화하기 위한 HTTPS 엔드포인트를 제공합니다.
- Aurora DSQL에 대한 API 요청의 무결성을 보호하려면 호출자가 API 직접 호출에 서명해야 합니다. 호출은 서명 버전 4 서명 프로세스(Sigv4)에 따라 X.509 인증서 또는 고객의 AWS 비밀 액세스 키로 서명됩니다. 자세한 정보는 AWS 일반 참조의 서명 버전 4 서명 프로세스를 참조하세요.
- AWS CLI 또는 AWS SDK 중 하나를 사용하여 AWS에 요청합니다. 이들 도구는 도구를 구성할 때 지정한 액세스 키를 사용하여 사용자의 요청에 자동으로 서명합니다.

저장 시 암호화는 Aurora DSQL의 저장 시 암호화 섹션을 참조하세요.

인터네트워크 트래픽 개인 정보 보호

Aurora DSQL과 온프레미스 애플리케이션 간의 연결 및 Aurora DSQL과 동일한 AWS 리전 내 다른 AWS 리소스 간의 연결은 둘 다 보호됩니다.

프라이빗 네트워크와 AWS 사이에 두 연결 옵션이 있습니다.

- AWS Site-to-Site VPN 연결. 자세한 내용은 AWS Site-to-Site VPN란 무엇입니까?를 참조하십시오.
- AWS Direct Connect 연결. 자세한 내용은 AWS Direct Connect란 무엇인가요?를 참조하세요.

AWS 게시 API 작업을 사용하면 네트워크를 통해 Aurora DSQL에 액세스할 수 있습니다. 고객은 다음을 지원해야 합니다.

- Transport Layer Security(TLS) TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군 Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

Aurora DSQL 연결을 위한 SSL/TLS 인증서 구성

Aurora DSQL은 Transport Layer Security(TLS) 암호화를 사용하려면 모든 연결이 필요합니다. 보안 연결을 설정하려면 클라이언트 시스템이 Amazon Root Certificate Authority(Amazon Root CA 1)를 신뢰해야 합니다. 이 인증서는 많은 운영 체제에 사전 설치되어 있습니다. 이 섹션에서는 다양한 운영 체제에서 사전 설치된 Amazon Root CA 1 인증서를 확인하는 지침을 제공하고 인증서가 아직 없는 경우 수동으로 설치하는 프로세스를 안내합니다.

PostgreSQL 버전 17을 사용하는 것이 좋습니다.

> **Important**
>
> 프로덕션 환경의 경우 `verify-full` SSL 모드를 사용하여 최고 수준의 연결 보안을 보장하는 것이 좋습니다. 이 모드는 서버 인증서가 신뢰할 수 있는 인증 기관에서 서명되었고 서버 호스트 이름이 인증서와 일치하는지 확인합니다.

사전 설치된 인증서 확인

대부분의 운영 체제에서 Amazon Root CA 1은 이미 사전 설치되어 있습니다. 이를 검증하려면 아래 단계를 따르세요.

Linux(RedHat/CentOS/Fedora)

터미널에서 다음 명령을 실행합니다.

```
trust list | grep "Amazon Root CA 1"
```

인증서가 설치된 경우 다음 출력이 표시됩니다.

```
label: Amazon Root CA 1
```

macOS

1. Spotlight Search를 엽니다(Command + 스페이스).
2. Keychain Access를 검색합니다.
3. System Keychains에서 System Roots를 선택합니다.
4. 인증서 목록에서 Amazon Root CA 1을 찾습니다.

Windows

> **Note**
>
> psql Windows 클라이언트에서 알려진 문제로 인해 시스템 루트 인증서(`sslrootcert=system`)를 사용하면 `SSL error: unregistered scheme` 오류가 반환될 수 있습니다. SSL을 사용하여 클러스터에 연결하는 다른 방법으로 Windows에서 연결를 따를 수 있습니다.

운영 체제에 Amazon Root CA 1이 설치되어 있지 않은 경우 아래 단계를 따릅니다.

인증서 설치

운영 체제에 `Amazon Root CA 1` 인증서가 사전 설치되어 있지 않은 경우 Aurora DSQL 클러스터에 대한 보안 연결을 설정하려면 인증서를 수동으로 설치해야 합니다.

Linux 인증서 설치

다음 단계에 따라 Linux 시스템에 Amazon Root CA 인증서를 설치합니다.

1. 루트 인증서를 다운로드합니다.

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 인증서를 트러스트 저장소에 추가합니다.

```
sudo cp ./AmazonRootCA1.pem /etc/pki/ca-trust/source/anchors/
```

3. CA 트러스트 저장소를 업데이트합니다.

```
sudo update-ca-trust
```

4. 설치를 확인합니다.

```
trust list | grep "Amazon Root CA 1"
```

macOS 인증서 설치

이러한 인증서 설치 단계는 선택 사항입니다. macOS에서도 Linux 인증서 설치가 적용됩니다.

1. 루트 인증서를 다운로드합니다.

```
wget https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

2. 시스템 키체인에 인증서를 추가합니다.

```
sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain
 AmazonRootCA1.pem
```

3. 설치를 확인합니다.

```
security find-certificate -a -c "Amazon Root CA 1" -p /Library/Keychains/
System.keychain
```

SSL/TLS 연결 확인

Aurora DSQL 클러스터에 대한 보안 연결을 위해 SSL/TLS 인증서를 구성하기 전에 다음 사전 조건을 충족하는지 확인합니다.

- PostgreSQL 버전 17 설치됨
- 적절한 자격 증명으로 AWS CLI가 구성됨
- Aurora DSQL 클러스터 엔드포인트 정보

Linux에서 연결

1. 인증 토큰을 생성하고 설정합니다.

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-cluster-region --hostname your-cluster-endpoint)
```

2. 시스템 인증서(미리 설치된 경우)를 사용하여 연결합니다.

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. 또는 다운로드한 인증서를 사용하여 연결합니다.

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

> **Note**
>
> PGSSLMODE 설정에 대한 자세한 내용은 PostgresQL 17 Database Connection Control Functions 설명서의 sslmode를 참조하세요.

macOS에서 연결

1. 인증 토큰을 생성하고 설정합니다.

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --region=your-cluster-region --hostname your-cluster-endpoint)
```

2. 시스템 인증서(미리 설치된 경우)를 사용하여 연결합니다.

```
PGSSLROOTCERT=system \
PGSSLMODE=verify-full \
```

```
psql --dbname postgres \
--username admin \
--host your-cluster-endpoint
```

3. 또는 루트 인증서를 다운로드하여 `root.pem`으로 저장합니다(인증서가 사전 설치되어 있지 않은 경우).

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

4. psql을 사용하여 연결합니다.

```
PGSSLROOTCERT=/full/path/to/root.pem \
PGSSLMODE=verify-full \
psql –dbname postgres \
--username admin \
--host your_cluster_endpoint
```

Windows에서 연결

명령 프롬프트 사용

1. 인증 토큰을 생성합니다.

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

2. 암호 환경 변수를 설정합니다.

```
set "PGPASSWORD=token-from-above"
```

3. SSL 구성을 설정합니다.

```
set PGSSLROOTCERT=C:\full\path\to\root.pem
set PGSSLMODE=verify-full
```

4. 데이터베이스에 연결합니다.

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres ^
--username admin ^
--host your-cluster-endpoint
```

PowerShell 사용

1. 인증 토큰을 생성하고 설정합니다.

```
$env:PGPASSWORD = (aws dsql generate-db-connect-admin-auth-token --region=your-
cluster-region --expires-in=3600 --hostname=your-cluster-endpoint)
```

2. SSL 구성을 설정합니다.

```
$env:PGSSLROOTCERT='C:\full\path\to\root.pem'
$env:PGSSLMODE='verify-full'
```

3. 데이터베이스에 연결합니다.

```
 "C:\Program Files\PostgreSQL\17\bin\psql.exe" --dbname postgres `
--username admin `
--host your-cluster-endpoint
```

추가 리소스

- PostgreSQL SSL 설명서
- Amazon Trust Services

Amazon Aurora DSQL에 대한 데이터 암호화

Amazon Aurora DSQL은 저장된 모든 사용자 데이터를 암호화합니다. 보안 강화를 위해 이 암호화는 AWS Key Management Service(AWS KMS)를 사용합니다. 이 기능을 사용하면 중요한 데이터 보호와 관련된 운영 부담 및 복잡성을 줄일 수 있습니다. 저장 시 암호화는 다음과 같은 이점을 제공합니다.

- 민감한 데이터 보호의 운영 부담 감소
- 엄격한 암호화 규정 준수 및 규제 요구 사항을 충족하는 보안에 민감한 애플리케이션 구축

- 암호화된 클러스터에서 데이터를 항상 보호하여 데이터 보호 계층 추가
- 조직 정책, 업계 또는 정부 규제, 규정 준수 요구 사항 충족

Aurora DSQL을 사용하면 엄격한 암호화 규정 준수 및 규제 요구 사항을 충족하는 보안에 민감한 애플리케이션을 구축할 수 있습니다. 이어지는 섹션에서는 신규 및 기존 Aurora DSQL 데이터베이스에 대한 암호화를 구성하고 암호화 키를 관리하는 방법을 설명합니다.

주제

Aurora DSQL용 KMS 키 유형

Aurora DSQL은 AWS KMS와 통합되어 클러스터의 암호화 키를 관리합니다. 키 유형 및 상태에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 AWS Key Management Service concepts를 참조하세요. 새 클러스터를 생성할 때 다음 KMS 키 유형 중에서 선택하여 클러스터를 암호화할 수 있습니다.

AWS 소유 키

기본 암호화 유형. Aurora DSQL은 추가 비용 없이 이 키를 소유합니다. Amazon Aurora DSQL은 암호화된 클러스터에 액세스할 때 클러스터 데이터를 투명하게 복호화합니다. 암호화된 클러스터를 사용하거나 관리하기 위해 코드 또는 애플리케이션을 변경할 필요가 없으며 모든 Aurora DSQL 쿼리는 암호화된 데이터에서 작동합니다.

고객 관리형 키

AWS 계정에서 고객이 직접 키를 생성하고 소유하고 관리합니다. 고객이 KMS 키에 대해 전체 제어 권한을 가지며 AWS KMS 비용이 부과됩니다.

AWS 소유 키를 사용한 저장 시 암호화는 추가 비용 없이 제공됩니다. 그러나 고객 관리형 키에 대해서는 AWS KMS 비용이 부과됩니다. 자세한 내용은 AWS KMS 요금 페이지를 참조하세요.

언제든지 이러한 키 유형 간에 전환할 수 있습니다. 키 유형에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 Customer managed keys 및 AWS 소유 키 섹션을 참조하세요.

> **Note**
>
> Aurora DSQL 저장 시 암호화는 Aurora DSQL을 사용할 수 있는 모든 AWS 리전에서 사용할 수 있습니다.

Aurora DSQL의 저장 시 암호화

Amazon Aurora DSQL에서는 256비트 고급 암호화 표준(AES-256)을 사용하여 저장 데이터를 암호화합니다. 이 암호화는 기본 스토리지에 대한 무단 액세스로부터 데이터를 보호하는 데 도움이 됩니다. AWS KMS는 클러스터의 암호화 키를 관리합니다. 기본 AWS 소유 키를 사용하거나 자체 AWS KMS 고객 관리형 키를 사용할 수 있습니다. Aurora DSQL 클러스터의 키 지정 및 관리에 대한 자세한 내용은 암호화된 Aurora DSQL 클러스터 생성 및 Aurora DSQL 클러스터의 키 제거 또는 업데이트 섹션을 참조하세요.

주제

- AWS 소유 키
- 고객 관리형 키

AWS 소유 키

Aurora DSQL은 기본적으로 AWS 소유 키를 사용하여 모든 클러스터를 암호화합니다. 이러한 키는 무료로 사용 가능하며 계정 리소스를 보호하기 위해 매년 교체할 수 있습니다. 이러한 키를 보거나 관리하거나 사용하거나 감사할 필요가 없으므로 데이터 보호를 위해 별도로 작업할 필요가 없습니다. AWS 소유 키에 대한 자세한 내용은 AWS Key Management Service 개발자 가이드의 AWS 소유 키를 참조하십시오.

고객 관리형 키

고객은 AWS 계정에서 고객 관리형 키를 만들고 소유하고 관리합니다. 고객은 정책, 암호화 자료, 태그 및 별칭을 포함하여 이러한 KMS 키를 완전히 제어할 수 있습니다. 권한 관리에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 Customer managed keys를 참조하세요.

클러스터 수준 암호화를 위해 고객 관리형 키를 지정하면 Aurora DSQL은 해당 키를 사용하여 클러스터와 클러스터의 모든 리전 데이터를 암호화합니다. 데이터 손실을 방지하고 클러스터 액세스를 유지하려면 Aurora DSQL이 암호화 키에 액세스할 수 있어야 합니다. 고객 관리형 키를 비활성화하거나, 키 삭제를 예약하거나, 서비스 액세스를 제한하는 정책이 있는 경우 클러스터의 암호화 상태가 `KMS_KEY_INACCESSIBLE`로 변경됩니다. Aurora DSQL이 키에 액세스할 수 없는 경우 사용자는 클러스터에 연결할 수 없으며 클러스터의 암호화 상태가 `KMS_KEY_INACCESSIBLE`로 변경되고 서비스가 클러스터 데이터에 대한 액세스 권한을 상실합니다.

다중 리전 클러스터의 경우 고객은 각 리전의 AWS KMS 암호화 키를 별도로 구성할 수 있으며, 각 리전 클러스터는 자체 클러스터 수준 암호화 키를 사용합니다. Aurora DSQL이 다중 리전 클러스터의 피어에 대한 암호화 키에 액세스할 수 없는 경우 해당 피어의 상태가 `KMS_KEY_INACCESSIBLE`이 되고 읽기 및 쓰기 작업에 사용할 수 없게 됩니다. 다른 피어는 계속해서 정상적으로 작동합니다.

> **Note**
>
> Aurora DSQL이 고객 관리형 키에 액세스할 수 없는 경우 클러스터 암호화 상태가 `KMS_KEY_INACCESSIBLE`로 변경됩니다. 키 액세스를 복원하면 서비스가 15분 이내에 복원을 자동으로 감지합니다. 자세한 내용은 Cluster idling을 참조하세요.
> 다중 리전 클러스터의 경우 키 액세스가 장기간 상실된 경우 클러스터 복원 시간은 키에 액세스할 수 없는 동안 기록된 데이터의 양에 따라 달라집니다.

Aurora DSQL에서 AWS KMS 및 데이터 키 사용

Aurora DSQL 저장 시 암호화 기능은 AWS KMS key와 데이터 키 계층 구조를 사용하여 클러스터 데이터를 보호합니다.

Aurora DSQL에서 클러스터를 구현하기 전에 암호화 전략을 계획하는 것이 좋습니다. 민감한 데이터나 기밀 데이터를 Aurora DSQL에 저장하는 경우 계획에 클라이언트 측 암호화를 포함하는 것을 고려해 보세요. 이렇게 하면 데이터를 원본에 최대한 가깝게 암호화하고 전체 수명 주기 동안 데이터를 보호할 수 있습니다.

주제

- Aurora DSQL에서 AWS KMS key 사용
- Aurora DSQL에서 클러스터 키 사용
- 클러스터 키 캐싱

Aurora DSQL에서 AWS KMS key 사용

저장 시 암호화는 AWS KMS key를 사용해 Aurora DSQL 클러스터를 보호합니다. 기본적으로 Aurora DSQL은 Aurora DSQL 서비스 계정에서 생성 및 관리되는 멀티테넌트 암호화 키인 AWS 소유 키를 사용합니다. 그러나 AWS 계정의 고객 관리형 키로 Aurora DSQL 클러스터를 암호화할 수 있습니다. 다중 리전 설정에 포함되어 있더라도 각 클러스터에 대해 다른 KMS 키를 선택할 수 있습니다.

클러스터를 생성 또는 업데이트할 때 클러스터에 대해 KMS 키를 선택합니다. 언제든지 Aurora DSQL 콘솔에서 또는 `UpdateCluster` 작업을 사용하여 클러스터에 대한 KMS 키를 변경할 수 있습니다. 키 교체 프로세스 중에 가동 중지 시간 또는 서비스 저하가 발생하지 않습니다.

> **Important**
>
> Aurora DSQL은 대칭 KMS 키만 지원합니다. 비대칭 KMS 키를 사용하여 Aurora DSQL 클러스터를 암호화할 수 없습니다.

고객 관리형 키는 다음과 같은 이점을 제공합니다.

- KMS 키에 대한 액세스를 제어하기 위한 키 정책 및 IAM 정책 설정을 포함하여 KMS 키를 생성하고 관리합니다. KMS 키를 활성화 및 비활성화하고, 자동 키 교체를 활성화 및 비활성화하고, KMS 키가 더 이상 사용되지 않을 때 KMS 키를 삭제할 수 있습니다.
- 가져온 키 구성 요소가 있는 고객 관리형 키를 사용하거나 고객이 소유하고 관리하는 사용자 지정 키 스토어에서 고객 관리형 키를 사용할 수 있습니다.
- AWS CloudTrail 로그에서 AWS KMS에 대한 Aurora DSQL API 직접 호출을 검사하여 Aurora DSQL 클러스터에 대한 암호화 및 복호화를 감사할 수 있습니다.

그러나 AWS 소유 키는 무료이며, 사용 시 AWS KMS 리소스 또는 요청 할당량에 포함되지 않습니다. 고객 관리형 키는 각 API 직접 호출에 대해 요금이 부과되며 이러한 키에 AWS KMS 할당량이 적용됩니다.

Aurora DSQL에서 클러스터 키 사용

Aurora DSQL은 클러스터에 AWS KMS key를 사용하여 클러스터 키라고 하는 클러스터의 고유한 데이터 키를 생성하고 암호화합니다.

클러스터 키는 키 암호화 키로 사용됩니다. Aurora DSQL은 이 클러스터 키를 사용하여 클러스터 데이터를 암호화하는 데 사용되는 데이터 암호화 키를 보호합니다. Aurora DSQL은 클러스터에 있는 각 기

본 구조의 고유한 데이터 암호화 키를 생성하지만, 동일한 데이터 암호화 키로 여러 클러스터 항목을 보호할 수 없습니다.

클러스터 키를 복호화하기 위해 Aurora DSQL은 암호화된 클러스터에 처음 액세스할 때 AWS KMS에 요청을 보냅니다. 클러스터를 계속 사용할 수 있도록 하기 위해, 클러스터에 적극적으로 액세스하지 않는 경우에도 Aurora DSQL은 KMS 키에 대한 복호화 액세스를 주기적으로 확인합니다.

Aurora DSQL은 클러스터 키와 데이터 암호화 키를 AWS KMS 외부에 저장하고 사용합니다. 고급 암호화 표준(AES) 암호화 및 256비트 암호화 키로 모든 키를 보호합니다. 그런 다음 필요에 따라 클러스터 데이터를 복호화할 때 사용할 수 있도록 암호화된 데이터와 함께 암호화된 키를 저장합니다.

클러스터의 KMS 키를 변경하면 Aurora DSQL은 기존 클러스터 키를 새 KMS 키로 다시 암호화합니다.

클러스터 키 캐싱

각 Aurora DSQL 작업마다 AWS KMS를 직접 호출하지 않도록 Aurora DSQL은 메모리에 있는 각 호출자의 일반 텍스트 클러스터 키를 캐싱합니다. 15분의 비활성 시간이 경과하여 Aurora DSQL이 캐시 클러스터 키를 요청할 경우 클러스터 키를 복호화하라는 새로운 요청을 AWS KMS에 보냅니다. 이 직접 호출은 마지막 클러스터 키 복호화 요청 이후 AWS KMS 또는 AWS Identity and Access Management(IAM)에서 AWS KMS key의 액세스 정책에 적용된 모든 변경 사항을 캡처합니다.

Aurora DSQL에 AWS KMS key 사용 권한 부여

계정에서 고객 관리형 키를 사용하여 Aurora DSQL 클러스터를 보호할 경우 해당 키에 대한 정책이 Aurora DSQL에 고객 대신 키를 사용할 권한을 부여해야 합니다.

고객은 고객 관리형 키에 대한 정책을 완벽하게 제어할 수 있습니다. 기본 AWS 소유 키를 사용하여 AWS 계정의 Aurora DSQL 클러스터를 보호하려면 Aurora DSQL에 추가 권한이 필요하지 않습니다.

고객 관리형 키에 대한 키 정책

Aurora DSQL 클러스터를 보호하기 위해 고객 관리형 키를 선택하면 Aurora DSQL은 선택한 위탁자를 대신하여 AWS KMS key를 사용할 수 있는 권한이 필요합니다. 해당 위탁자, 사용자 또는 역할은 Aurora DSQL이 요구하는 AWS KMS key에 대한 권한이 있어야 합니다. 키 정책 또는 IAM 정책에서 이러한 권한을 제공할 수 있습니다.

Aurora DSQL은 고객 관리형 키에 대해 최소한 다음 권한이 있어야 합니다.

- `kms:Encrypt`
- `kms:Decrypt`
- `kms:ReEncrypt*`(kms:ReEncryptFrom 및 kms:ReEncryptTo의 경우)

- kms:GenerateDataKey
- kms:DescribeKey

예를 들어 다음 예제 키 정책은 필수 권한만 제공합니다. 이 정책에는 다음과 같은 효과가 있습니다.

- Aurora DSQL이 암호화 작업에 AWS KMS key를 사용하도록 허용합니다. 단, 계정에서 Aurora DSQL을 사용할 권한이 있는 위탁자를 대신해 작업하는 경우에 한합니다. 정책 문에 지정된 위탁자가 보안 Aurora DSQL을 사용할 권한이 없는 경우 Aurora DSQL 서비스에서 오는 경우에도 직접 호출이 실패합니다.
- kms:ViaService 조건 키는 Aurora DSQL이 정책 문에 나열된 위탁자를 대신하여 요청을 보낸 경우에만 권한을 허용합니다. 이러한 보안 주체는 이러한 작업을 직접 호출 할 수 없습니다.
- AWS KMS key 관리자(db-team 역할을 수임할 수 있는 사용자)에게 AWS KMS key에 대한 읽기 전용 액세스 권한을 부여합니다.

예제 키 정책을 사용하기 전에 예제 보안 주체를 AWS 계정의 실제 보안 주체로 바꿉니다.

```
{
  "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 암호화 컨텍스트

암호화 컨텍스트는 보안되지 않은 임의의 데이터를 포함하는 키-값 페어 세트입니다. 데이터 암호화 요청에 암호화 컨텍스트를 포함하는 경우 AWS KMS는 암호화된 데이터에 암호화 컨텍스트를 암호 방식으로 바인딩합니다. 따라서 동일한 암호화 컨텍스트로 전달해야 이 데이터를 해독할 수 있습니다.

Aurora DSQL은 모든 AWS KMS 암호화 작업에서 동일한 암호화 컨텍스트를 사용합니다. 고객 관리형 키를 사용하여 Aurora DSQL 클러스터를 보호하는 경우, 암호화 컨텍스트를 사용하여 감사 레코드 및 로그에서 AWS KMS key 사용을 식별할 수 있습니다. 또한 AWS CloudTrail과 같은 로그에서 일반 텍스트로 나타나기도 합니다.

암호화 컨텍스트를 정책에서 권한 부여의 조건으로 사용할 수도 있습니다.

AWS KMS에 요청할 때 Aurora DSQL은 키-값 쌍이 있는 암호화 컨텍스트를 사용합니다.

```
"encryptionContext": {
  "aws:dsql:ClusterId": "w4abucpbwuxx"
},
```

키-값 쌍은 Aurora DSQL이 암호화 중인 클러스터를 식별합니다. 키는 `aws:dsql:ClusterId`입니다. 값은 클러스터의 식별자입니다.

AWS KMS와 Aurora DSQL 상호 작용 모니터링

고객 관리형 키를 사용하여 Aurora DSQL 클러스터를 보호할 경우, AWS CloudTrail 로그를 사용하여 Aurora DSQL이 고객 대신 AWS KMS로 전송하는 요청을 추적할 수 있습니다.

다음 섹션을 확장하여 Aurora DSQL이 AWS KMS 작업 GenerateDataKey 및 Decrypt를 어떻게 사용하는지 알아보세요.

GenerateDataKey

클러스터에서 저장 시 암호화를 활성화하면 Aurora DSQL이 고유한 클러스터 키를 생성합니다. 이는 클러스터에 AWS KMS key를 지정하는 GenerateDataKey 요청을 AWS KMS에 보냅니다.

GenerateDataKey 작업을 기록하는 이벤트는 다음 예시 이벤트와 유사합니다. 사용자는 Aurora DSQL 서비스 계정입니다. 파라미터에는 AWS KMS key의 Amazon 리소스 이름(ARN), 256비트 키가 필요한 키 지정자, 클러스터를 식별하는 암호화 컨텍스트가 포함됩니다.


```
{
    "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

암호화된 Aurora DSQL 클러스터에 액세스하면 Aurora DSQL은 클러스터 키를 복호화해야만 계층 구조에서 그 아래에 있는 키를 복호화할 수 있습니다. 그런 다음, 클러스터에 있는 데이터를 복호화합니다. 클러스터 키를 복호화하기 위해 Aurora DSQL은 클러스터에 AWS KMS key를 지정하는 `Decrypt` 요청을 AWS KMS에 보냅니다.

`Decrypt` 작업을 기록하는 이벤트는 다음 예시 이벤트와 유사합니다. 사용자는 AWS 계정에서 클러스터에 액세스 중인 위탁자입니다. 파라미터에는 암호화된 클러트서 키(사이퍼텍스트 blob), 그리고 클러스터를 식별하는 암호화 컨텍스트가 포함됩니다. AWS KMS는 사이퍼텍스트에서 AWS KMS key의 ID를 파생합니다.

```
{
  "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"
}
```

암호화된 Aurora DSQL 클러스터 생성

모든 Aurora DSQL 클러스터는 저장 시 암호화됩니다. 기본적으로 클러스터에서 무료로 AWS 소유 키를 사용하거나 고객이 사용자 지정 AWS KMS 키를 지정할 수 있습니다. 다음 단계에 따라 AWS Management Console 또는 AWS CLI에서 암호화된 클러스터를 생성합니다.

Console

AWS Management Console에서 암호화된 클러스터를 생성하는 방법

1. AWS Management Console에 로그인하고 https://console.aws.amazon.com/dsql/에서 Aurora DSQL 콘솔을 엽니다.
2. 콘솔 왼쪽의 탐색 창에서 클러스터를 선택합니다.
3. 오른쪽 상단에서 클러스터 생성을 선택하고 단일 리전을 선택합니다.
4. 클러스터 암호화 설정에서 다음 옵션 중 하나를 선택합니다.
   - 기본 설정을 수락하여 추가 비용 없이 AWS 소유 키로 암호화합니다.

- 암호화 설정 사용자 지정(고급)을 선택하여 사용자 지정 KMS 키를 지정합니다. 그런 다음, KMS 키의 ID 또는 별칭을 검색하거나 입력합니다. AWS KMS 콘솔에서 새로운 키를 생성하려면 AWS KMS 키 생성을 선택합니다.

5. 클러스터 생성을 선택합니다.

클러스터의 암호화 유형을 확인하려면 클러스터 페이지로 이동하여 클러스터의 ID를 선택하여 클러스터 세부 정보를 확인합니다. 클러스터 설정 탭을 검토합니다. 클러스터 KMS 키 설정에는 AWS 소유 키를 사용하는 클러스터의 Aurora DSQL 기본 키 또는 다른 암호화 유형의 키 ID가 표시됩니다.

> **Note**
>
> 자체 키 소유 및 관리를 선택하는 경우 KMS 키 정책을 적절하게 설정합니다. 예제와 추가 정보는 the section called “고객 관리형 키에 대한 키 정책” 섹션을 참조하세요.

CLI

기본 AWS 소유 키로 암호화된 클러스터를 생성하는 방법

- 다음 명령을 사용하여 Aurora DSQL 클러스터를 생성합니다.

```
aws dsql create-cluster
```

다음 암호화 세부 정보에 표시된 것처럼 클러스터의 암호화 상태는 기본적으로 활성화되어 있으며 기본 암호화 유형은 AWS 소유 키입니다. 이제 클러스터가 Aurora DSQL 서비스 계정의 기본 AWS 소유 키로 암호화됩니다.

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

고객 관리형 키로 암호화된 클러스터를 생성하는 방법

- 다음 명령을 사용하여 Aurora DSQL 클러스터를 생성하되, 빨간색 텍스트의 키 ID를 고객 관리형 키의 ID로 바꿉니다.

```
aws dsql create-cluster \
--kms-encryption-key d41d8cd98f00b204e9800998ecf8427e
```

다음 암호화 세부 정보에서 볼 수 있듯이 클러스터의 암호화 상태는 기본적으로 활성화되어 있으며 암호화 유형은 고객 관리형 KMS 키입니다. 이제 클러스터가 고객의 키로 암호화됩니다.

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:kms:us-east-1:111122223333:key/
d41d8cd98f00b204e9800998ecf8427e",
  "encryptionStatus" : "ENABLED"
}
```

Aurora DSQL 클러스터의 키 제거 또는 업데이트

AWS Management Console 또는 AWS CLI를 사용하여 Amazon Aurora DSQL의 기존 클러스터에서 암호화 키를 업데이트하거나 제거할 수 있습니다. 키를 교체하지 않고 제거하면 Aurora DSQL은 기본 AWS 소유 키를 사용합니다. 다음 단계에 따라 Aurora DSQL 콘솔 또는 AWS CLI에서 기존 클러스터의 암호화 키를 업데이트합니다.

Console

AWS Management Console에서 암호화 키를 업데이트하거나 제거하는 방법

1. AWS Management Console에 로그인하고 https://console.aws.amazon.com/dsql/에서 Aurora DSQL 콘솔을 엽니다.
2. 콘솔 왼쪽의 탐색 창에서 클러스터를 선택합니다.
3. 목록 뷰에서 업데이트하려는 클러스터의 행을 찾아 선택합니다.
4. 작업 메뉴를 선택하고 수정을 선택합니다.
5. 클러스터 암호화 설정에서 다음 옵션 중 하나를 선택하여 암호화 설정을 수정합니다.

   - 사용자 지정 키에서 AWS 소유 키로 전환하려면 암호화 설정 사용자 지정(고급) 옵션을 선택 취소합니다. 기본 설정이 적용되고 AWS 소유 키를 사용하여 무료로 클러스터를 암호화합니다.
   - 사용자 지정 KMS 키에서 다른 키로 교체하거나 AWS 소유 키에서 KMS 키로 전환하려면 암호화 설정 사용자 지정(고급) 옵션을 아직 선택하지 않은 경우 선택합니다. 그런 다음, 사용

하려는 키의 ID 또는 별칭을 검색하고 선택합니다. AWS KMS 콘솔에서 새로운 키를 생성하려면 AWS KMS 키 생성을 선택합니다.

6. 저장을 선택합니다.

CLI

다음 예에서는 AWS CLI를 사용해 암호화된 클러스터를 업데이트하는 방법을 보여줍니다.

기본 AWS 소유 키를 사용하여 암호화된 클러스터를 업데이트하는 방법

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key "AWS_OWNED_KMS_KEY"
```

클러스터 설명의 EncryptionStatus는 ENABLED로 설정되고 EncryptionType은 AWS_OWNED_KMS_KEY입니다.

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

이 클러스터는 이제 Aurora DSQL 서비스 계정에서 기본 AWS 소유 키를 사용하여 암호화됩니다.

Aurora DSQL에서 고객 관리형 키로 암호화된 클러스터를 업데이트하는 방법

암호화된 클러스터를 다음 예와 같이 업데이트합니다.

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-ab1234a1b234
```

클러스터 설명의 `EncryptionStatus`는 `UPDATING`으로 전환되고 `EncryptionType`은 `CUSTOMER_MANAGED_KMS_KEY`입니다. Aurora DSQL이 플랫폼을 전체에 새 키 전파를 완료하면 암호화 상태가 `ENABLED`로 전환됩니다.

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:us-east-1:kms:key/abcd1234-abcd-1234-a123-ab1234a1b234",
  "encryptionStatus" : "ENABLED"
}
```

> **Note**
>
> 자체 키 소유 및 관리를 선택하는 경우 KMS 키 정책을 적절하게 설정합니다. 예제와 추가 정보는 the section called “고객 관리형 키에 대한 키 정책” 섹션을 참조하세요.

Aurora DSQL에서 암호화 고려 사항

- Aurora DSQL은 모든 클러스터 저장 데이터를 암호화합니다. 이 암호화를 비활성화하거나 클러스터의 일부 항목만 암호화할 수 없습니다.
- AWS Backup은 백업과 이러한 백업에서 복원된 모든 클러스터를 암호화합니다. AWS 소유 키 또는 고객 관리형 키를 사용하여 AWS Backup에서 백업 데이터를 암호화할 수 있습니다.
- Aurora DSQL에서 다음 데이터 보호 상태가 활성화됩니다.
  - 저장 데이터 - Aurora DSQL은 영구 스토리지 미디어의 모든 정적 데이터를 암호화합니다.
  - 전송 중 데이터 - Aurora DSQL은 기본적으로 Transport Layer Security(TLS)를 사용하여 모든 통신을 암호화합니다.
- 다른 키로 전환할 때는 전환이 완료될 때까지 원본 키를 활성화된 상태로 유지하는 것이 좋습니다. AWS는 새 키로 데이터를 암호화하기 전에 데이터를 복호화하기 위해 원본 키가 필요합니다. 클러스터의 `encryptionStatus`가 `ENABLED`이고 새 고객 관리형 키의 `kmsKeyArn`이 표시되면 프로세스가 완료된 것입니다.
- 고객 관리형 키를 비활성화하거나 Aurora DSQL의 키 사용 액세스 권한을 취소하면 클러스터가 `IDLE` 상태로 전환됩니다.
- AWS Management Console과 Amazon Aurora DSQL API는 암호화 유형에 대해 서로 다른 용어를 사용합니다.

  - AWS Console - 고객 관리형 키를 사용할 때는 콘솔에 KMS가 표시되고, AWS 소유 키를 사용할 때는 DEFAULT가 표시됩니다.
  - API - Amazon Aurora DSQL API는 고객 관리형 키에 CUSTOMER_MANAGED_KMS_KEY를 사용하고, AWS 소유 키에 AWS_OWNED_KMS_KEY를 사용합니다.
- 클러스터 생성 중에 암호화 키를 지정하지 않으면 Aurora DSQL은 AWS 소유 키를 사용하여 데이터를 자동으로 암호화합니다.
- 언제든지 AWS 소유 키와 고객 관리형 키 간에 전환할 수 있습니다. AWS Management Console, AWS CLI 또는 Amazon Aurora DSQL API를 사용하여 이 변경을 수행합니다.

Aurora DSQL의 ID 및 액세스 관리

AWS Identity and Access Management(IAM)은 관리자가 AWS 리소스에 대한 액세스를 안전하게 제어할 수 있도록 지원하는 AWS 서비스입니다. IAM 관리자는 어떤 사용자가 Aurora DSQL을 사용할 수 있도록 인증(로그인)되고 권한이 부여(권한 있음)될 수 있는지 제어합니다. IAM은 추가 비용 없이 사용할 수 있는 AWS 서비스입니다.

주제

고객

AWS Identity and Access Management(IAM)를 사용하는 방법은 Aurora DSQL에서 수행하는 작업에 따라 달라집니다.

서비스 사용자 - Aurora DSQL 서비스를 사용하여 작업을 수행하는 경우 필요한 자격 증명과 권한을 관리자가 제공합니다. 더 많은 Aurora DSQL 기능을 사용하여 작업을 수행하게 되면 추가 권한이 필요할 수 있습니다. 액세스 권한 관리 방법을 이해하면 관리자에게 올바른 권한을 요청하는 데 도움이 됩니다. Aurora DSQL의 기능에 액세스할 수 없는 경우 Amazon Aurora DSQL ID 및 액세스 문제 해결 섹션을 참조하세요.

서비스 관리자 - 회사에서 Aurora DSQL 리소스를 책임지고 있는 경우 Aurora DSQL에 대한 전체 액세스 권한을 가지고 있을 것입니다. 서비스 관리자는 서비스 사용자가 액세스해야 하는 Aurora DSQL 기능과 리소스를 결정합니다. 그런 다음 IAM 관리자에게 요청을 제출하여 서비스 사용자의 권한을 변경해야 합니다. 이 페이지의 정보를 검토하여 IAM의 기본 개념을 이해하세요. 회사가 Aurora DSQL에서 IAM을 사용하는 방법에 대해 자세히 알아보려면 Amazon Aurora DSQL에서 IAM을 사용하는 방법 섹션을 참조하세요.

IAM 관리자 - IAM 관리자라면 Aurora DSQL에 대한 액세스 권한 관리 정책을 작성하는 방법을 자세히 알고 싶을 것입니다. IAM에서 사용할 수 있는 Aurora DSQL ID 기반 정책 예시를 보려면 Amazon Aurora DSQL의 ID 기반 정책 예시 섹션을 참조하세요.

ID를 통한 인증

인증은 ID 자격 증명을 사용하여 AWS에 로그인하는 방식입니다. AWS 계정 루트 사용자이나 IAM 사용자로, 또는 IAM 역할을 수임하여 인증(AWS에 로그인)받아야 합니다.

ID 소스를 통해 제공된 자격 증명을 사용하여 페더레이션 ID로 AWS에 로그인할 수 있습니다. AWS IAM Identity Center (IAM Identity Center) 사용자, 회사의 Single Sign-On 인증, Google 또는 Facebook 보안 인증이 페더레이션 ID의 예제입니다. 페더레이션형 ID로 로그인할 때 관리자가 이전에 IAM 역할을 사용하여 ID 페더레이션을 설정했습니다. 연동을 사용하여 AWS에 액세스하면 간접적으로 역할을 수임합니다.

사용자 유형에 따라AWS Management Console 또는 AWS 액세스 포털에 로그인할 수 있습니다. AWS에 로그인하는 방법에 대한 자세한 내용은 AWS 로그인 사용 설명서의 AWS 계정에 로그인하는 방법을 참조하세요.

AWS에 프로그래밍 방식으로 액세스하는 경우, AWS에서는 자격 증명을 사용하여 요청에 암호화 방식으로 서명할 수 있는 소프트웨어 개발 키트(SDK) 및 명령줄 인터페이스(CLI)를 제공합니다. AWS 도구를 사용하지 않는 경우, 요청에 직접 서명해야 합니다. 권장 방법을 사용하여 요청에 직접 서명하는 자세한 방법은 IAM 사용 설명서에서 API 요청용 AWS Signature Version 4를 참조하세요.

사용하는 인증 방법에 상관없이 추가 보안 정보를 제공해야 할 수도 있습니다. 예를 들어, AWS은(는) 다중 인증(MFA)을 사용하여 계정의 보안을 강화하는 것을 권장합니다. 자세한 내용은 AWS IAM Identity Center 사용 설명서에서 다중 인증 및 IAM 사용 설명서에서 IAM의 AWS 다중 인증을 참조하세요.

AWS 계정 루트 사용자

AWS 계정을(를) 생성할 때는 해당 계정의 모든 AWS 서비스 및 리소스에 대한 완전한 액세스 권한이 있는 단일 로그인 ID로 시작합니다. 이 ID는 AWS 계정루트 사용자라고 하며, 계정을 생성할 때 사용한

이메일 주소와 암호로 로그인하여 액세스합니다. 일상적인 작업에 루트 사용자를 사용하지 않을 것을 강력히 권장합니다. 루트 사용자 자격 증명을 보호하고 루트 사용자만 수행할 수 있는 작업을 수행하는 데 사용합니다. 루트 사용자로 로그인해야 하는 전체 작업 목록은 IAM 사용 설명서의 루트 사용자 보안 인증이 필요한 작업을 참조하세요.

페더레이션 자격 증명

가장 좋은 방법은 관리자 액세스가 필요한 사용자를 포함한 사용자가 ID 공급자와의 페더레이션을 사용하여 임시 자격 증명을 사용하여 AWS 서비스에 액세스하도록 요구합니다.

페더레이션 ID는 엔터프라이즈 사용자 디렉터리, 웹 ID 공급자, AWS Directory Service, Identity Center 디렉터리의 사용자 또는 자격 증명 소스를 통해 제공된 자격 증명을 사용하여 AWS 서비스에 액세스하는 모든 사용자입니다. 페더레이션 ID는 AWS 계정에 액세스할 때 역할을 수임하고 역할은 임시 자격 증명을 제공합니다.

중앙 집중식 액세스 관리를 위해 AWS IAM Identity Center을(를) 사용하는 것이 좋습니다. IAM Identity Center에서 사용자 및 그룹을 생성하거나 모든 AWS 계정 및 애플리케이션에서 사용하기 위해 고유한 ID 소스의 사용자 및 그룹 집합에 연결하고 동기화할 수 있습니다. IAM Identity Center에 대한 자세한 내용은 AWS IAM Identity Center 사용 설명서에서 IAM Identity Center란 무엇인가요?를 참조하세요.

IAM 사용자 및 그룹

IAM 사용자는 단일 개인 또는 애플리케이션에 대한 특정 권한을 가지고 있는 AWS 계정 내 ID입니다. 가능하면 암호 및 액세스 키와 같은 장기 자격 증명이 있는 IAM 사용자를 생성하는 대신 임시 자격 증명을 사용하는 것이 좋습니다. 하지만 IAM 사용자의 장기 자격 증명이 필요한 특정 사용 사례가 있는 경우, 액세스 키를 교체하는 것이 좋습니다. 자세한 내용은 IAM 사용 설명서의 장기 보안 인증이 필요한 사용 사례의 경우, 정기적으로 액세스 키 교체를 참조하세요.

IAM 그룹은 IAM 사용자 컬렉션을 지정하는 자격 증명입니다. 사용자는 그룹으로 로그인할 수 없습니다. 그룹을 사용하여 여러 사용자의 권한을 한 번에 지정할 수 있습니다. 그룹을 사용하면 대규모 사용자 집합의 권한을 더 쉽게 관리할 수 있습니다. 예를 들어, IAMAdmins라는 그룹이 있고 이 그룹에 IAM 리소스를 관리할 권한을 부여할 수 있습니다.

사용자는 역할과 다릅니다. 사용자는 한 사람 또는 애플리케이션과 고유하게 연결되지만, 역할은 해당 역할이 필요한 사람이라면 누구나 수임할 수 있습니다. 사용자는 영구적인 장기 자격 증명을 가지고 있지만, 역할은 임시 보안 인증만 제공합니다. 자세한 내용은 IAM 사용 설명서에서 IAM 사용자 사용 사례를 참조하세요.

IAM 역할

IAM 역할은 특정 권한을 가지고 있는 AWS 계정 계정 내 ID입니다. IAM 사용자와 유사하지만, 특정 개인과 연결되지 않습니다. AWS Management Console에서 일시적으로 IAM 역할을 수임하려면 사용자에서 IAM 역할로 전환(콘솔)하면 됩니다. AWS CLI 또는 AWS API 작업을 직접적으로 호출하거나 사용자 지정 URL을 사용하여 역할을 수임할 수 있습니다. 역할 사용 방법에 대한 자세한 내용은 IAM 사용 설명서의 역할 수임 방법을 참조하세요.

임시 보안 인증이 있는 IAM 역할은 다음과 같은 상황에서 유용합니다.

- 페더레이션 사용자 액세스 - 페더레이션 ID에 권한을 부여하려면 역할을 생성하고 해당 역할의 권한을 정의합니다. 페더레이션 ID가 인증되면 역할이 연결되고 역할에 정의된 권한이 부여됩니다. 페더레이션 관련 역할에 대한 자세한 내용은 IAM 사용 설명서의 Create a role for a third-party identity provider (federation)를 참조하세요. IAM Identity Center를 사용하는 경우, 권한 집합을 구성합니다. 인증 후 ID가 액세스할 수 있는 항목을 제어하기 위해 IAM Identity Center는 권한 집합을 IAM의 역할과 연관짓습니다. 권한 집합에 대한 자세한 내용은 AWS IAM Identity Center 사용 설명서의 권한 집합을 참조하세요.
- 임시 IAM 사용자 권한 - IAM 사용자 또는 역할은 IAM 역할을 수임하여 특정 작업에 대한 다양한 권한을 임시로 받을 수 있습니다.
- 교차 계정 액세스 - IAM 역할을 사용하여 다른 계정의 사용자(신뢰할 수 있는 보안 주체)가 내 계정의 리소스에 액세스하도록 허용할 수 있습니다. 역할은 계정 간 액세스를 부여하는 기본적인 방법입니다. 하지만 일부 AWS 서비스을(를) 사용하면 리소스에 정책을 직접 연결할 수 있습니다(역할을 프록시로서 사용하는 대신) . 교차 계정 액세스에 대한 역할과 리소스 기반 정책의 차이점을 알아보려면 IAM 사용 설명서의 IAM의 교차 계정 리소스 액세스를 참조하세요.
- 교차 서비스 액세스 - 일부 AWS 서비스는 다른 AWS 서비스의 특성을 사용합니다. 예를 들어, 서비스에서 호출하면 일반적으로 해당 서비스는 Amazon EC2에서 애플리케이션을 실행하거나 Amazon S3에 객체를 저장합니다. 서비스는 직접적으로 호출하는 위탁자의 권한을 사용하거나, 서비스 역할을 사용하거나, 또는 서비스 연결 역할을 사용하여 이 작업을 수행할 수 있습니다.
  - 전달 액세스 세션(FAS) - IAM 사용자 또는 역할을 사용하여 AWS에서 작업을 수행하는 사람은 보안 주체로 간주됩니다. 일부 서비스를 사용하는 경우, 다른 서비스에서 다른 작업을 시작하는 작업을 수행할 수 있습니다. FAS는 AWS 서비스를 직접 호출하는 위탁자의 권한과 요청하는 AWS 서비스를 함께 사용하여 다운스트림 서비스에 대한 요청을 수행합니다. FAS 요청은 서비스에서 완료를 위해 다른 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 받은 경우에만 이루어집니다. 이 경우, 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하세요.

  - 서비스 역할 - 서비스 역할은 서비스가 사용자를 대신하여 작업을 수행하기 위해 맡는 IAM 역할입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 Create a role to delegate permissions to an AWS 서비스를 참조하세요.
  - 서비스 연결 역할 – 서비스 연결 역할은 AWS 서비스에 연결된 서비스 역할의 한 유형입니다. 서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 링크 역할은 AWS 계정에 나타나고, 서비스가 소유합니다. IAM 관리자는 서비스 연결 역할의 권한을 볼 수 있지만 편집할 수는 없습니다.
- Amazon EC2에서 실행 중인 애플리케이션 – IAM 역할을 사용하여 EC2 인스턴스에서 실행되고 AWS CLI 또는 AWS API 요청을 수행하는 애플리케이션의 임시 자격 증명을 관리할 수 있습니다. 이는 EC2 인스턴스 내에 액세스 키를 저장할 때 권장되는 방법입니다. EC2 인스턴스에 AWS역할을 할당하고 해당 역할을 모든 애플리케이션에서 사용할 수 있도록 하려면 인스턴스에 연결된 인스턴스 프로필을 생성합니다. 인스턴스 프로필에는 역할이 포함되어 있으며 EC2 인스턴스에서 실행되는 프로그램이 임시 보안 인증을 얻을 수 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM 역할을 사용하여 Amazon EC2 인스턴스에서 실행되는 애플리케이션에 권한 부여를 참조하세요.

정책을 사용하여 액세스 관리

정책을 생성하고 AWS ID 또는 리소스에 연결하여 AWS에서 내 액세스를 제어합니다. 정책은 ID 또는 리소스와 연결될 때 해당 권한을 정의하는 AWS의 객체입니다. AWS는 보안 주체(사용자, 루트 사용자 또는 역할 세션)가 요청을 보낼 때 이러한 정책을 평가합니다. 정책에서 권한은 요청이 허용되거나 거부되는 지를 결정합니다. 대부분의 정책은 AWS에 JSON 문서로 저장됩니다. JSON 정책 문서의 구조와 콘텐츠에 대한 자세한 정보는 IAM 사용 설명서의 JSON 정책 개요를 참조하세요.

관리자는 AWSJSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

기본적으로, 사용자 및 역할에는 어떠한 권한도 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 수임할 수 있습니다.

IAM 정책은 작업을 수행하기 위해 사용하는 방법과 상관없이 작업에 대한 권한을 정의합니다. 예를 들어, `iam:GetRole` 작업을 허용하는 정책이 있다고 가정합니다. 해당 정책이 있는 사용자는 AWS Management Console, AWS CLI 또는 AWS API에서 역할 정보를 가져올 수 있습니다.

ID 기반 정책

ID 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 ID에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자 및 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지

를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서에서 고객 관리형 정책으로 사용자 지정 IAM 권한 정의를 참조하세요.

ID 기반 정책은 인라인 정책 또는 관리형 정책으로 한층 더 분류할 수 있습니다. 인라인 정책은 단일 사용자, 그룹 또는 역할에 직접 포함됩니다. 관리형 정책은 AWS 계정에 속한 다수의 사용자, 그룹 및 역할에 독립적으로 추가할 수 있는 정책입니다. 관리형 정책에는 AWS 관리형 정책과 고객 관리형 정책이 포함되어 있습니다. 관리형 정책 또는 인라인 정책을 선택하는 방법을 알아보려면 IAM 사용 설명서의 관리형 정책 및 인라인 정책 중에서 선택을 참조하세요.

리소스 기반 정책

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 리소스 기반 정책의 예제는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 위탁자가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 정의합니다. 리소스 기반 정책에서 위탁자를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 페더레이션 사용자 또는 AWS 서비스가 포함될 수 있습니다.

리소스 기반 정책은 해당 서비스에 있는 인라인 정책입니다. 리소스 기반 정책에서는 IAM의 AWS 관리형 정책을 사용할 수 없습니다.

액세스 제어 목록(ACL)

액세스 제어 목록(ACL)은 어떤 보안 주체(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는지를 제어합니다. ACLs는 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

Amazon S3, AWS WAF 및 Amazon VPC는 ACL을 지원하는 대표적인 서비스입니다. ACL에 관한 자세한 내용은 Amazon Simple Storage Service 개발자 가이드의 액세스 제어 목록(ACL) 개요를 참조하세요.

기타 정책 타입

AWS는 비교적 일반적이지 않은 추가 정책 유형을 지원합니다. 이러한 정책 타입은 더 일반적인 정책 유형에 따라 사용자에게 부여되는 최대 권한을 설정할 수 있습니다.

- 권한 경계 – 권한 경계는 ID 기반 정책에 따라 IAM 엔터티(IAM 사용자 또는 역할)에 부여할 수 있는 최대 권한을 설정하는 고급 기능입니다. 개체에 대한 권한 경계를 설정할 수 있습니다. 그 결과로 얻는 권한은 객체의 ID 기반 정책과 그 권한 경계의 교집합입니다. `Principal` 필드에서 사용자나 역할을 지정하는 리소스 기반 정책은 권한 경계를 통해 제한되지 않습니다. 이러한 정책 중 하나에 포

함된 명시적 거부는 허용을 재정의합니다. 권한 경계에 대한 자세한 정보는 IAM 사용 설명서의 IAM 엔티티에 대한 권한 경계를 참조하세요.

- 서비스 제어 정책(SCP) – SCP는 AWS Organizations에서 조직 또는 조직 단위(OU)에 최대 권한을 지정하는 JSON 정책입니다. AWS Organizations는 기업이 소유하는 여러 개의 AWS 계정을 그룹화하고 중앙에서 관리하기 위한 서비스입니다. 조직에서 모든 특성을 활성화할 경우, 서비스 제어 정책(SCP)을 임의의 또는 모든 계정에 적용할 수 있습니다. SCP는 각 AWS 계정 루트 사용자을 비롯하여 멤버 계정의 엔티티에 대한 권한을 제한합니다. 조직 및 SCP에 대한 자세한 내용은 AWS Organizations 사용 설명서에서 Service control policies을 참조하세요.
- 리소스 제어 정책(RCP) - RCP는 소유한 각 리소스에 연결된 IAM 정책을 업데이트하지 않고 계정의 리소스에 대해 사용 가능한 최대 권한을 설정하는 데 사용할 수 있는 JSON 정책입니다. RCP는 멤버 계정의 리소스에 대한 권한을 제한하며 조직에 속하는지 여부에 관계없이 AWS 계정 루트 사용자를 포함한 ID에 대한 유효 권한에 영향을 줄 수 있습니다. RCP를 지원하는 AWS 서비스 목록을 포함하여 Organizations 및 RCP에 대한 자세한 내용은 AWS Organizations 사용 설명서의 리소스 제어 정책(RCP)을 참조하세요.
- 세션 정책 – 세션 정책은 역할 또는 페더레이션 사용자에 대해 임시 세션을 프로그래밍 방식으로 생성할 때 파라미터로 전달하는 고급 정책입니다. 결과적으로 얻는 세션의 권한은 사용자 또는 역할의 ID 기반 정책의 교차와 세션 정책입니다. 또한 권한을 리소스 기반 정책에서 가져올 수도 있습니다. 이러한 정책 중 하나에 포함된 명시적 거부는 허용을 재정의합니다. 자세한 정보는 IAM 사용 설명서의 세션 정책을 참조하세요.

여러 정책 유형

여러 정책 유형이 요청에 적용되는 경우, 결과 권한은 이해하기가 더 복잡합니다. 여러 정책 유형이 관련될 때 AWS가 요청을 허용할지를 결정하는 방법을 알아보려면 IAM 사용 설명서의 정책 평가 로직을 참조하세요.

Amazon Aurora DSQL에서 IAM을 사용하는 방법

IAM을 사용하여 Aurora DSQL에 대한 액세스를 관리하기 전에 Aurora DSQL에서 사용할 수 있는 IAM 기능에 대해 알아봅니다.

Amazon Aurora DSQL에서 사용할 수 있는 IAM 기능

IAM 기능	Aurora DSQL 지원
ID 기반 정책	예

IAM 기능	Aurora DSQL 지원
리소스 기반 정책	아니요
정책 작업	예
정책 리소스	예
정책 조건 키	예
ACLs	아니요
ABAC(정책의 태그)	예
임시 보안 인증	예
보안 주체 권한	예
서비스 역할	예
서비스 연결 역할	예

Aurora DSQL 및 기타 AWS 서비스가 대부분의 IAM 기능과 작동하는 방법을 개괄적으로 알아보려면 IAM 사용 설명서의 IAM으로 작업하는 AWS 서비스를 참조하세요.

Aurora DSQL의 ID 기반 정책

ID 기반 정책 지원: 예

ID 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 ID에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자 및 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서에서 고객 관리형 정책으로 사용자 지정 IAM 권한 정의를 참조하세요.

IAM ID 기반 정책을 사용하면 허용되거나 거부되는 작업과 리소스뿐 아니라 작업이 허용되거나 거부되는 조건을 지정할 수 있습니다. ID 기반 정책에서는 위탁자가 연결된 사용자 또는 역할에 적용되므로 위탁자를 지정할 수 없습니다. JSON 정책에서 사용하는 모든 요소에 대해 알아보려면 IAM 사용 설명서의 IAM JSON 정책 요소 참조를 참조하세요.

Aurora DSQL의 ID 기반 정책 예시

Aurora DSQL ID 기반 정책의 예를 보려면 Amazon Aurora DSQL의 ID 기반 정책 예시 섹션을 참조하세요.

Aurora DSQL 내의 리소스 기반 정책

리소스 기반 정책 지원: 아니요

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 리소스 기반 정책의 예제는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 위탁자가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 정의합니다. 리소스 기반 정책에서 위탁자를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 페더레이션 사용자 또는 AWS 서비스가 포함될 수 있습니다.

교차 계정 액세스를 활성화하려는 경우, 전체 계정이나 다른 계정의 IAM 개체를 리소스 기반 정책의 위탁자로 지정할 수 있습니다. 리소스 기반 정책에 크로스 계정 보안 주체를 추가하는 것은 트러스트 관계 설정의 절반밖에 되지 않는다는 것을 유념하세요. 보안 주체와 리소스가 서로 다른 AWS 계정에 있는 경우, 신뢰할 수 있는 계정의 IAM 관리자는 보안 주체 엔터티(사용자 또는 역할)에도 리소스 액세스 권한을 부여해야 합니다. 엔터티에 ID 기반 정책을 연결하여 권한을 부여합니다. 하지만 리소스 기반 정책이 동일 계정의 위탁자에 액세스를 부여하는 경우, 추가 자격 증명 기반 정책이 필요하지 않습니다. 자세한 내용은 IAM 사용 설명서의 교차 계정 리소스 액세스를 참조하세요.

Aurora DSQL의 정책 작업

정책 작업 지원: 예

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 위탁자가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

JSON 정책의 `Action` 요소는 정책에서 액세스를 허용하거나 거부하는 데 사용할 수 있는 작업을 설명합니다. 일반적으로 정책 작업의 이름은 연결된 AWSAPI 작업의 이름과 동일합니다. 일치하는 API 작업이 없는 권한 전용 작업 같은 몇 가지 예외도 있습니다. 정책에서 여러 작업이 필요한 몇 가지 작업도 있습니다. 이러한 추가 작업을 일컬어 종속 작업이라고 합니다.

연결된 작업을 수행할 수 있는 권한을 부여하기 위한 정책에 작업을 포함하세요.

Aurora DSQL 작업 목록을 보려면 서비스 권한 부여 참조에서 Actions Defined by Amazon Aurora DSQL을 참조하세요.

Aurora DSQL의 정책 작업은 작업 앞에 다음 접두사를 사용합니다.

```
dsql
```

단일 문에서 여러 작업을 지정하려면 다음과 같이 쉼표로 구분합니다.

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

Aurora DSQL ID 기반 정책의 예를 보려면 Amazon Aurora DSQL의 ID 기반 정책 예시 섹션을 참조하세요.

Aurora DSQL의 정책 리소스

정책 리소스 지원: 예

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

`Resource` JSON 정책 요소는 작업이 적용되는 하나 이상의 객체를 지정합니다. 문에는 `Resource`또는 `NotResource`요소가 반드시 추가되어야 합니다. 모범 사례에 따라 Amazon 리소스 이름(ARN)을 사용하여 리소스를 지정합니다. 리소스 수준 권한이라고 하는 특정 리소스 유형을 지원하는 작업에 대해 이를 수행할 수 있습니다.

작업 나열과 같이 리소스 수준 권한을 지원하지 않는 작업의 경우, 와일드카드(*)를 사용하여 해당 문이 모든 리소스에 적용됨을 나타냅니다.

```
"Resource": "*"
```

Aurora DSQL 리소스 유형 및 해당 ARN 목록을 보려면 서비스 권한 부여 참조에서 Resources Defined by Amazon Aurora DSQL을 참조하세요. 각 리소스의 ARN을 지정할 수 있는 작업을 알아보려면 Actions Defined by Amazon Aurora DSQL을 참조하세요.

Aurora DSQL ID 기반 정책의 예를 보려면 Amazon Aurora DSQL의 ID 기반 정책 예시 섹션을 참조하세요.

Aurora DSQL의 정책 조건 키

서비스별 정책 조건 키 지원: 예

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

`Condition` 요소(또는 `Condition` 블록)를 사용하면 정책이 발효되는 조건을 지정할 수 있습니다. `Condition` 요소는 옵션입니다. 같거나 작음과 같은 조건 연산자를 사용하여 정책의 조건을 요청의 값과 일치시키는 조건식을 생성할 수 있습니다.

한 문에서 여러 `Condition` 요소를 지정하거나 단일 `Condition` 요소에서 여러 키를 지정하는 경우, AWS는 논리적 `AND` 작업을 사용하여 평가합니다. 단일 조건 키의 여러 값을 지정하는 경우, AWS는 논리적 `OR` 작업을 사용하여 조건을 평가합니다. 문의 권한을 부여하기 전에 모든 조건을 충족해야 합니다.

조건을 지정할 때 자리 표시자 변수를 사용할 수도 있습니다. 예를 들어, IAM 사용자에게 IAM 사용자 이름으로 태그가 지정된 경우에만 리소스에 액세스할 수 있는 권한을 부여할 수 있습니다. 자세한 내용은 IAM 사용 설명서의 IAM 정책 요소: 변수 및 태그를 참조하세요.

AWS는 전역 조건 키와 서비스별 조건 키를 지원합니다. 모든 AWS 전역 조건 키를 보려면 IAM 사용 설명서의 AWS 전역 조건 컨텍스트 키를 참조하세요.

Aurora DSQL 조건 키 목록을 보려면 서비스 승인 참조의 Condition keys for Amazon Aurora DSQL을 참조하세요. 조건 키를 사용할 수 있는 작업과 리소스를 알아보려면 Actions defined by Amazon Aurora DSQL을 참조하세요.

Aurora DSQL ID 기반 정책의 예를 보려면 Amazon Aurora DSQL의 ID 기반 정책 예시 섹션을 참조하세요.

Aurora DSQL의 ACL

ACL 지원: 아니요

액세스 제어 목록(ACL)은 어떤 위탁자(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는지를 제어합니다. ACL은 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

Aurora DSQL의 ABAC

ABAC 지원(정책의 태그): 예

속성 기반 액세스 제어(ABAC)는 속성에 근거하여 권한을 정의하는 권한 부여 전략입니다. AWS에서는 이러한 속성을 태그라고 합니다. IAM 엔티티(사용자 또는 역할) 및 많은 AWS리소스에 태그를 연결할 수 있습니다. ABAC의 첫 번째 단계로 개체 및 리소스에 태그를 지정합니다. 그런 다음 위탁자의 태그가 액세스하려는 리소스의 태그와 일치할 때 작업을 허용하도록 ABAC 정책을 설계합니다.

ABAC는 빠르게 성장하는 환경에서 유용하며 정책 관리가 번거로운 상황에 도움이 됩니다.

태그에 근거하여 액세스를 제어하려면 `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` 또는 `aws:TagKeys` 조건 키를 사용하여 정책의 조건 요소에 태그 정보를 제공합니다.

서비스가 모든 리소스 유형에 대해 세 가지 조건 키를 모두 지원하는 경우, 값은 서비스에 대해 예입니다. 서비스가 일부 리소스 유형에 대해서만 세 가지 조건 키를 모두 지원하는 경우, 값은 부분적입니다.

ABAC에 대한 자세한 내용은 IAM 사용 설명서의 ABAC 권한 부여를 통한 권한 정의를 참조하세요. ABAC 설정 단계가 포함된 자습서를 보려면 IAM 사용 설명서의 속성 기반 액세스 제어(ABAC) 사용을 참조하세요.

Aurora DSQL에서 임시 자격 증명 사용

임시 자격 증명 지원: 예

일부 AWS 서비스는 임시 자격 증명을 사용하여 로그인할 때 작동하지 않습니다. 임시 보안 인증으로 작동하는 AWS 서비스를 비롯한 추가 정보는 IAM 사용 설명서의 IAM으로 작업하는 AWS 서비스를 참조하세요.

사용자 이름과 암호를 제외한 다른 방법을 사용하여 AWS Management Console에 로그인하면 임시 자격 증명을 사용하는 것입니다. 예를 들어, 회사의 Single Sign-On(SSO) 링크를 사용하여 AWS에 액세스하면 해당 프로세스에서 자동으로 임시 자격 증명을 생성합니다. 또한 콘솔에 사용자로 로그인한 다음 역할을 전환할 때 임시 자격 증명을 자동으로 생성합니다. 역할 전환에 대한 자세한 내용은 IAM 사용 설명서의 사용자에서 IAM 역할로 전환(콘솔)을 참조하세요.

AWS CLI 또는 AWS API를 사용하여 임시 자격 증명을 수동으로 만들 수 있습니다 그런 다음 이러한 임시 자격 증명을 사용하여 AWS에 액세스할 수 있습니다. AWS에서는 장기 액세스 키를 사용하는 대신 임시 자격 증명을 동적으로 생성할 것을 권장합니다. 자세한 정보는 IAM의 임시 보안 자격 증명 섹션을 참조하세요.

Aurora DSQL의 서비스 간 위탁자 권한

전달 액세스 세션(FAS) 지원: 예

IAM 사용자 또는 역할을 사용하여 AWS에서 작업을 수행하는 사람은 위탁자로 간주됩니다. 일부 서비스를 사용하는 경우, 다른 서비스에서 다른 작업을 시작하는 작업을 수행할 수 있습니다. FAS는 AWS 서비스를 직접 호출하는 위탁자의 권한과 요청하는 AWS 서비스를 함께 사용하여 다운스트림 서비스에 대한 요청을 수행합니다. FAS 요청은 서비스에서 완료를 위해 다른 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 받은 경우에만 이루어집니다. 이 경우, 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하세요.

Aurora DSQL의 서비스 역할

서비스 역할 지원: 예

서비스 역할은 서비스가 사용자를 대신하여 작업을 수행하는 것으로 가정하는 IAM 역할입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 Create a role to delegate permissions to an AWS 서비스를 참조하세요.

> **Warning**
>
> 서비스 역할에 대한 권한을 변경하면 Aurora DSQL 기능이 중단될 수 있습니다. Aurora DSQL이 편집하라는 지침을 제공하는 경우에만 서비스 역할을 편집하세요.

Aurora DSQL의 서비스 연결 역할

서비스 링크 역할 지원: 예

서비스 연결 역할은 AWS 서비스에 연결된 서비스 역할의 한 유형입니다. 서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 링크 역할은 AWS 계정에 나타나고, 서비스가 소유합니다. IAM 관리자는 서비스 링크 역할의 권한을 볼 수 있지만 편집은 할 수 없습니다.

Aurora DSQL의 서비스 연결 역할을 생성 또는 관리하는 방법에 대한 자세한 내용은 Aurora DSQL에서 서비스 연결 역할 사용 섹션을 참조하세요.

Amazon Aurora DSQL의 ID 기반 정책 예시

기본적으로 사용자 및 역할에는 Aurora DSQL 리소스를 생성하거나 수정할 수 있는 권한이 없습니다. 또한 AWS Management Console, AWS Command Line Interface(AWS CLI) 또는 AWS API를 사용해 태스크를 수행할 수 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 맡을 수 있습니다.

이러한 예제 JSON 정책 문서를 사용하여 IAM 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성(콘솔)을 참조하세요.

각 리소스 유형에 대한 ARN 형식을 포함하여 Aurora DSQL에서 정의한 작업 및 리소스 유형에 대한 자세한 내용은 서비스 권한 부여 참조에서 Actions, Resources, and Condition Keys for Amazon Aurora DSQL을 참조하세요.

주제

- 정책 모범 사례
- Aurora DSQL 콘솔 사용
- 사용자가 자신의 고유한 권한을 볼 수 있도록 허용

정책 모범 사례

ID 기반 정책에 따라 계정에서 누군가가 Aurora DSQL 리소스를 생성, 액세스 또는 삭제할 수 있는지가 결정됩니다. 이 작업으로 인해 AWS 계정에 비용이 발생할 수 있습니다. ID 기반 정책을 생성하거나 편집할 때는 다음 지침과 권장 사항을 따릅니다.

- AWS 관리형 정책으로 시작하고 최소 권한을 향해 나아가기 - 사용자 및 워크로드에 권한 부여를 시작하려면 많은 일반 사용 사례에 대한 권한을 부여하는 AWS 관리형 정책을 사용합니다. AWS 계정에서 사용할 수 있습니다. 사용 사례에 고유한 AWS 고객 관리형 정책을 정의하여 권한을 줄이는 것이 좋습니다. 자세한 정보는 IAM 사용 설명서의 AWS 관리형 정책 또는 AWS직무에 대한 관리형 정책을 참조하세요.
- 최소 권한 적용 – IAM 정책을 사용하여 권한을 설정하는 경우, 작업을 수행하는 데 필요한 권한만 부여합니다. 이렇게 하려면 최소 권한으로 알려진 특정 조건에서 특정 리소스에 대해 수행할 수 있는 작업을 정의합니다. IAM을 사용하여 권한을 적용하는 방법에 대한 자세한 정보는 IAM 사용 설명서에 있는 IAM의 정책 및 권한을 참조하세요.
- IAM 정책의 조건을 사용하여 액세스 추가 제한 – 정책에 조건을 추가하여 작업 및 리소스에 대한 액세스를 제한할 수 있습니다. 예를 들어, SSL을 사용하여 모든 요청을 전송해야 한다고 지정하는 정책 조건을 작성할 수 있습니다. AWS CloudFormation와 같이, 특정 AWS 서비스를 통해 사용되는 경우에만 서비스 작업에 대한 액세스 권한을 부여할 수도 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM JSON 정책 요소: 조건을 참조하세요.
- IAM Access Analyzer를 통해 IAM 정책을 확인하여 안전하고 기능적인 권한 보장 - IAM Access Analyzer에서는 IAM 정책 언어(JSON)와 모범 사례가 정책에서 준수되도록 새로운 및 기존 정책을 확인합니다. IAM Access Analyzer는 100개 이상의 정책 확인 항목과 실행 가능한 추천을 제공하

여 안전하고 기능적인 정책을 작성하도록 돕습니다. 자세한 내용은 IAM 사용 설명서의 IAM Access Analyzer에서 정책 검증을 참조하세요.

- 다중 인증(MFA) 필요 – AWS 계정에 IAM 사용자 또는 루트 사용자가 필요한 시나리오가 있는 경우, 추가 보안을 위해 MFA를 설정합니다. API 작업을 직접 호출할 때 MFA가 필요하면 정책에 MFA 조건을 추가합니다. 자세한 내용은 IAM 사용 설명서의 MFA를 통한 보안 API 액세스를 참조하세요.

IAM의 모범 사례에 대한 자세한 내용은 IAM 사용 설명서의 IAM의 보안 모범 사례를 참조하세요.

Aurora DSQL 콘솔 사용

Amazon Aurora DSQL 콘솔에 액세스하려면 최소한의 권한 세트가 있어야 합니다. 이러한 권한은 AWS 계정에서 Aurora DSQL 리소스에 대한 세부 정보를 나열하고 볼 수 있도록 허용해야 합니다. 최소 필수 권한보다 더 제한적인 ID 기반 정책을 생성하는 경우, 콘솔이 해당 정책에 연결된 엔티티(사용자 또는 역할)에 대해 의도대로 작동하지 않습니다.

AWS CLI 또는 AWS API만 직접 호출하는 사용자에게 최소 콘솔 권한을 허용할 필요가 없습니다. 대신 수행하려는 API 작업과 일치하는 작업에만 액세스할 수 있도록 합니다.

사용자와 역할이 Aurora DSQL 콘솔을 여전히 사용할 수 있도록 하려면 Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess` 또는 `AmazonAuroraDSQLReadOnlyAccess` AWS 관리형 정책을 엔터티에 연결합니다. 자세한 내용은 IAM 사용 설명서의 사용자에게 권한 추가를 참조하세요.

사용자가 자신의 고유한 권한을 볼 수 있도록 허용

이 예제는 IAM 사용자가 자신의 사용자 ID에 연결된 인라인 및 관리형 정책을 볼 수 있도록 허용하는 정책을 생성하는 방법을 보여줍니다. 이 정책에는 콘솔에서 또는 AWS CLI 또는 AWS API를 사용하여 프로그래밍 방식으로 이 태스크를 완료할 수 있는 권한이 포함됩니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
```

```
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Aurora DSQL ID 및 액세스 문제 해결

다음 정보를 사용하여 Aurora DSQL 및 IAM에서 발생할 수 있는 일반적인 문제를 진단하고 수정할 수 있습니다.

주제

- Aurora DSQL에서 작업을 수행할 권한이 없음
- iam:PassRole을 수행하도록 인증되지 않음
- AWS 계정 외부인이 내 Aurora DSQL 리소스에 액세스할 수 있게 허용하려고 함

Aurora DSQL에서 작업을 수행할 권한이 없음

작업을 수행할 수 있는 권한이 없다는 오류가 수신되면 작업을 수행할 수 있도록 정책을 업데이트해야 합니다.

다음의 예시 오류는 `mateojackson`이 콘솔을 사용하여 *`my-dsql-cluster`* 리소스에 대한 세부 정보를 보려고 하지만 *`GetCluster`* 권한이 없을 때 발생합니다.

```
User: iam:::user/mateojackson is not authorized to perform: GetCluster on resource: my-dsql-cluster
```

이 경우, *GetCluster* 작업을 사용하여 *my-dsql-cluster* 리소스에 액세스할 수 있도록 mateojackson 사용자 정책을 업데이트해야 합니다.

도움이 필요한 경우 관리자에게 문의하세요. 관리자는 로그인 자격 증명을 제공한 사람입니다.

iam:PassRole을 수행하도록 인증되지 않음

iam:PassRole 작업을 수행할 수 있는 권한이 없다는 오류가 수신되면 Aurora DSQL에 역할을 전달할 수 있도록 정책을 업데이트해야 합니다.

일부 AWS 서비스에서는 새로운 서비스 역할 또는 서비스 연결 역할을 생성하는 대신 해당 서비스에 기존 역할을 전달할 수 있습니다. 이렇게 하려면 사용자가 서비스에 역할을 전달할 수 있는 권한을 가지고 있어야 합니다.

다음 예시 오류는 marymajor라는 IAM 사용자가 콘솔을 사용하여 Aurora DSQL에서 작업을 수행하려고 하는 경우에 발생합니다. 하지만 작업을 수행하려면 서비스 역할이 부여한 권한이 서비스에 있어야 합니다. Mary는 서비스에 역할을 전달할 수 있는 권한을 가지고 있지 않습니다.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

이 경우, Mary가 iam:PassRole 작업을 수행할 수 있도록 Mary의 정책을 업데이트해야 합니다.

도움이 필요한 경우 AWS 관리자에게 문의하세요. 관리자는 로그인 자격 증명을 제공한 사람입니다.

AWS 계정 외부인이 내 Aurora DSQL 리소스에 액세스할 수 있게 허용하려고 함

다른 계정의 사용자 또는 조직 외부의 사람이 리소스에 액세스할 때 사용할 수 있는 역할을 생성할 수 있습니다. 역할을 수임할 신뢰할 수 있는 사람을 지정할 수 있습니다. 리소스 기반 정책 또는 액세스 제어 목록(ACL)을 지원하는 서비스의 경우, 이러한 정책을 사용하여 다른 사람에게 리소스에 대한 액세스 권한을 부여할 수 있습니다.

자세히 알아보려면 다음을 참조하세요.

- Aurora DSQL에서 이러한 기능을 지원하는지 알아보려면 Amazon Aurora DSQL에서 IAM을 사용하는 방법 섹션을 참조하세요.

- 소유하고 있는 AWS 계정의 리소스에 대한 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 자신이 소유한 다른 AWS 계정의 IAM 사용자에 대한 액세스 권한 제공을 참조하세요.
- 리소스에 대한 액세스 권한을 서드 파티 AWS 계정에게 제공하는 방법을 알아보려면 IAM 사용 설명서의 서드 파티가 소유한 AWS 계정에 대한 액세스 제공을 참조하세요.
- ID 페더레이션을 통해 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 외부에서 인증된 사용자에게 액세스 권한 제공(ID 페더레이션)을 참조하세요.
- 크로스 계정 액세스에 대한 역할과 리소스 기반 정책 사용의 차이점을 알아보려면 IAM 사용 설명서의 IAM의 크로스 계정 리소스 액세스를 참조하세요.

Aurora DSQL에서 서비스 연결 역할 사용

Aurora DSQL은 AWS Identity and Access Management(IAM) 서비스 연결 역할을 사용합니다. 서비스 연결 역할은 Aurora DSQL에 직접 연결된 고유한 유형의 IAM 역할입니다. 서비스 연결 역할은 Aurora DSQL에서 사전 정의하며, 해당 서비스가 Aurora DSQL 클러스터를 대신해 AWS 서비스 서비스를 직접 호출하기 위해 필요한 모든 권한을 가지고 있습니다.

서비스 연결 역할을 사용하면 Aurora DSQL을 사용하는 데 필요한 권한을 수동으로 추가할 필요가 없으므로 더 쉽게 설정할 수 있습니다. 클러스터를 생성할 때 Aurora DSQL이 자동으로 서비스 연결 역할을 생성합니다. 클러스터를 모두 삭제한 후에만 서비스 연결 역할을 삭제할 수 있습니다. 리소스에 대한 액세스에 필요한 권한을 부주의하게 제거할 수 없기 때문에 Aurora DSQL 리소스가 보호됩니다.

서비스 연결 역할을 지원하는 기타 서비스에 대한 자세한 내용은 IAM으로 작업하는 AWS 서비스을(를) 참조하고 서비스 연결 역할 열에 예가 있는 서비스를 찾아보세요. 해당 서비스에 대한 서비스 연결 역할 설명서를 보려면 예 링크를 선택합니다.

서비스 연결 역할은 지원되는 모든 Aurora DSQL 리전에서 사용할 수 있습니다.

Aurora DSQL의 서비스 연결 역할 권한

Aurora DSQL은 `AWSServiceRoleForAuroraDsql`이라는 서비스 연결 역할을 사용하는데 이는 Amazon Aurora DSQL이 사용자를 대신하여 AWS 리소스를 생성하고 관리하도록 허용합니다. 이 서비스 연결 역할은 AuroraDsqlServiceLinkedRolePolicy 관리형 정책에 연결됩니다.

> **Note**
>
> IAM 엔터티(사용자, 그룹, 역할 등)가 서비스 링크 역할을 생성하고 편집하거나 삭제할 수 있도록 권한을 구성할 수 있습니다. `You don't have the permissions to create an`

Amazon Aurora DSQL service-linked role이라는 오류 메시지가 표시될 수 있습니다. 이 메시지가 표시되면 다음 권한이 활성화되어 있는지 확인합니다.

```
{
"Sid" : "CreateDsqlServiceLinkedRole",
    "Effect" : "Allow",
    "Action" : "iam:CreateServiceLinkedRole",
    "Resource" : "*",
    "Condition" : {
      "StringEquals" : {
        "iam:AWSServiceName" : "dsql.amazonaws.com"
      }
    }
  }
```

자세한 내용은 서비스 연결 역할 권한을 참조하세요.

서비스 연결 역할 생성

AuroraDSQLServiceLinkedRolePolicy 서비스 연결 역할은 수동으로 생성할 필요가 없습니다. Aurora DSQL이 대신해서 이 서비스 연결 역할을 생성합니다. 사용자 계정에서 AuroraDSQLServiceLinkedRolePolicy 서비스 연결 역할이 삭제되면 사용자가 새 Aurora DSQL이 클러스터를 생성할 때 Aurora DSQL이 해당 역할을 생성합니다.

서비스 연결 역할 편집

Aurora DSQL은 AuroraDSQLServiceLinkedRolePolicy 서비스 연결 역할을 편집하도록 허용하지 않습니다. 서비스 연결 역할을 생성한 후에는 다양한 엔터티가 역할을 참조할 수 있기 때문에 역할 이름을 변경할 수 없습니다. 그러나 IAM 콘솔, AWS Command Line Interface(AWS CLI) 또는 IAM API를 사용하여 역할에 대한 설명을 편집할 수 있습니다.

서비스 연결 역할 삭제

서비스 연결 역할이 필요한 기능 또는 서비스가 더 이상 필요 없는 경우에는 해당 역할을 삭제하는 것이 좋습니다. 이로 인해 적극적으로 모니터링하거나 유지 관리하지 않는 미사용 엔터티가 없습니다.

계정에 대한 서비스 연결 역할을 삭제하려면 계정에서 클러스터를 삭제해야 합니다.

IAM 콘솔, AWS CLI 또는 IAM API를 사용하여 서비스 연결 역할을 삭제할 수 있습니다. 자세한 내용은 IAM 사용 설명서의 서비스 연결 역할 생성을 참조하세요.

Aurora DSQL 서비스 연결 역할이 지원되는 리전

Aurora DSQL에서는 서비스를 사용할 수 있는 모든 리전에서 서비스 연결 역할 사용을 지원합니다. 자세한 내용은 AWS 리전 및 엔드포인트 섹션을 참조하세요.

Amazon Aurora DSQL에서 IAM 조건 키 사용

Aurora DSQL에서 권한을 부여할 때 권한 정책이 적용되는 방식을 결정하는 조건을 지정할 수 있습니다. 다음은 Aurora DSQL 권한 정책에서 조건 키를 사용할 수 있는 방법의 예입니다.

예시 1: 특정 AWS 리전에 클러스터를 생성할 수 있는 권한 부여

다음 정책은 미국 동부(버지니아 북부) 및 미국 동부(오하이오) 리전에서 클러스터를 생성할 수 있는 권한을 부여합니다. 이 정책은 리소스 ARN을 사용하여 허용된 리전을 제한하므로 Aurora DSQL은 정책의 `Resource` 섹션에 해당 ARN이 지정된 경우에만 클러스터를 생성할 수 있습니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            # Control where clusters can be created
            "Action": ["CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

예시 2: 특정 AWS 리전에 다중 리전 클러스터를 생성할 수 있는 권한 부여

다음 정책은 미국 동부(버지니아 북부) 및 미국 동부(오하이오) 리전에서 다중 리전 클러스터를 생성할 수 있는 권한을 부여합니다. 이 정책은 리소스 ARN을 사용하여 허용된 리전을 제한하므로 Aurora DSQL은 정책의 `Resource` 섹션에 해당 ARN이 지정된 경우에만 다중 리전 클러스터를 생성할 수 있습니다. 다중 리전 클러스터를 생성하려면 지정된 각 리전에서도 `PutMultiRegionProperties`, `PutWitnessRegion`, `AddPeerCluster` 권한이 필요합니다.

```
{
```

```
    "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/*"
        ]
      }
    ]
}
```

예시 3: 특정 감시 리전을 사용하여 다중 리전 클러스터를 생성할 수 있는 권한 부여

다음 정책은 Aurora DSQL `dsql:WitnessRegion` 조건 키를 사용하고 사용자가 미국 서부(오리건)에 감시 리전을 사용하여 다중 리전 클러스터를 생성하도록 허용합니다. `dsql:WitnessRegion` 조건을 지정하지 않으면 모든 리전을 감시 리전으로 사용할 수 있습니다.

```
{
    "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 DSQL의 사고 대응

AWS에서는 보안을 가장 중요하게 생각합니다. AWS 클라우드 공동 책임 모델의 일환으로 AWS는 보안에 매우 민감한 조직의 요구 사항에 부합하는 데이터 센터, 네트워크 및 소프트웨어 아키텍처를 관리합니다. AWS는 Amazon Aurora DSQL 서비스 내에서 발생한 사고에 대응할 책임이 있습니다. 사용자는 AWS 고객으로서 클라우드의 보안을 유지할 책임을 공유합니다. 즉, 액세스 가능한 AWS 도구 및 기능을 통해 구현하기로 선택한 보안을 제어한다는 의미입니다. 또한 사용자는 공유 책임 모델에 따라 사고 대응에 대한 책임이 있습니다.

클라우드에서 실행되는 애플리케이션의 목표를 충족하는 보안 기준을 설정하면 대응할 수 있는 편차를 감지할 수 있습니다. 인시던트 대응과 사용자의 선택이 회사 목표에 미치는 영향을 이해하는 데 도움이 되도록 다음 리소스를 검토하는 것이 좋습니다.

- AWS Security Incident Response Guide
- AWS 보안, 자격 증명 및 규정 준수 모범 사례
- AWS Cloud Adoption Framework(CAF) 보안 관점 백서

Amazon GuardDuty는 고객이 AWS 계정과 워크로드를 보호하고 의심스러운 활동이 사고로 확대되기 전에 잠재적으로 식별할 수 있도록 악의적이거나 승인되지 않은 행동을 지속적으로 모니터링하는 관리형 위협 감지 서비스입니다. 이 서비스는 비정상적인 API 직접 호출이나 잠재적으로 승인되지 않은 배포와 같은 활동을 모니터링하여 악의적인 행위자에 의한 계정 또는 리소스 손상이나 정찰 가능성을 보여줍니다. 예를 들어 Amazon GuardDuty는 Amazon Aurora DSQL API에서 사용자가 새 위치에서 로그인하고 새 클러스터를 생성하는 등의 의심스러운 활동을 감지할 수 있습니다.

Amazon Aurora DSQL의 규정 준수 검증

AWS 서비스(이)가 특정 규정 준수 프로그램의 범위에 포함되는지 알아보려면 규정 준수 프로그램 제공 범위 내 AWS 서비스를 참조하고 관심 있는 규정 준수 프로그램을 선택합니다. 일반적인 정보는 AWS 규정 준수 프로그램을 참조하세요.

AWS Artifact(을)를 사용하여 타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 AWS Artifact에서 보고서 다운로드를 참조하세요.

AWS 서비스 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률과 규정에 따라 결정됩니다.AWS에서는 규정 준수를 지원할 다음과 같은 리소스를 제공합니다.

- 보안 규정 준수 및 거버넌스 - 이러한 솔루션 구현 가이드에서는 아키텍처 고려 사항을 설명하고 보안 및 규정 준수 기능을 배포하는 단계를 제공합니다.
- HIPAA 적격 서비스 참조 - HIPAA 적격 서비스가 나열되어 있습니다. 모든 AWS 서비스에 HIPAA 자격이 있는 것은 아닙니다.
- AWS 규정 준수 리소스 - 고객 조직이 속한 산업 및 위치에 적용될 수 있는 워크북 및 가이드 컬렉션입니다.
- AWS 고객 규정 준수 가이드 - 규정 준수의 관점에서 공동 책임 모델을 이해합니다. 이 가이드에서는 AWS 서비스를 보호하기 위한 모범 사례를 요약하고 여러 프레임워크(미국 표준 기술 연구소(NIST), 결제 카드 산업 보안 표준 위원회(PCI), 국제 표준화기구(ISO) 등)에서 보안 컨트롤에 대한 지침을 매핑합니다.
- AWS Config 개발자 가이드의 규칙을 사용하여 리소스 평가 – AWS Config 서비스는 내부 사례, 산업 지침 및 규제에 대한 리소스 구성의 준수 상태를 평가합니다.
- AWS Security Hub – 이 AWS 서비스(은)는 AWS 내 보안 상태에 대한 포괄적인 보기를 제공합니다. Security Hub는 보안 컨트롤을 사용하여 AWS리소스를 평가하고 보안 업계 표준 및 모범 사례에 대한 규정 준수를 확인합니다. 지원되는 서비스 및 제어 목록은 Security Hub 제어 참조를 참조하세요.
- Amazon GuardDuty - 이 AWS 서비스는 의심스럽고 악의적인 활동이 있는지 환경을 모니터링하여 AWS 계정, 워크로드, 컨테이너 및 데이터에 대한 잠재적 위협을 탐지합니다. GuardDuty는 특정 규정 준수 프레임워크에서 요구하는 침입 탐지 요구 사항을 충족하여 PCI DSS와 같은 다양한 규정 준수 요구 사항을 따르는 데 도움을 줄 수 있습니다.
- AWS Audit Manager - 이 AWS 서비스은(는) AWS 사용을 지속해서 감사하여 위험을 관리하고 규정 및 업계 표준을 준수하는 방법을 간소화할 수 있도록 지원합니다.

Amazon Aurora DSQL의 복원력

AWS 글로벌 인프라는 AWS 리전 및 가용 영역(AZ)을 중심으로 구축됩니다. AWS 리전에서는 물리적으로 분리되고 격리된 다수의 가용 영역을 제공하며 이러한 가용 영역은 짧은 지연 시간, 높은 처리량 및 높은 중복성을 갖춘 네트워크에 연결되어 있습니다. 가용 영역을 사용하면 중단 없이 영역 간에 자동으로 장애 극복 조치가 이루어지는 애플리케이션 및 데이터베이스를 설계하고 운영할 수 있습니다. 가용 영역은 기존의 단일 또는 다중 데이터 센터 인프라보다 가용성, 내결함성, 확장성이 뛰어납니다. Aurora DSQL은 최고의 데이터베이스 가용성을 제공하면서 AWS 리전 인프라를 활용할 수 있도록 설계되었습니다. 기본적으로 Aurora DSQL의 단일 리전 클러스터에는 다중 AZ 가용성이 있어 전체 AZ에 대한 액세스에 영향을 미칠 수 있는 주요 구성 요소 장애 및 인프라 중단에 대한 허용성을 제공합니다. 다중 리전 클러스터는 AWS 리전이 애플리케이션 클라이언트에 액세스할 수 없는 경우에도 강력히 일관된 데이터베이스 가용성을 제공하면서 다중 AZ 복원력의 모든 이점을 제공합니다.

AWS 리전 및 가용 영역에 대한 자세한 정보는 AWS 글로벌 인프라를 참조하세요.

AWS 글로벌 인프라 외에도 Aurora DSQL은 데이터 복원력과 백업 요구 사항을 지원하는 다양한 기능을 제공합니다.

백업 및 복원

Aurora DSQL은 AWS Backup 콘솔을 사용한 백업 및 복원을 지원합니다. 단일 리전 및 다중 리전 클러스터에 대해 전체 백업 및 복원을 수행할 수 있습니다. 자세한 내용은 Amazon Aurora DSQL의 백업 및 복원 섹션을 참조하세요.

복제

설계상 Aurora DSQL은 모든 쓰기 트랜잭션을 분산 트랜잭션 로그에 커밋하고 모든 커밋된 로그 데이터를 세 개의 AZ에 있는 사용자 스토리지 복제본에 동기적으로 복제합니다. 다중 리전 클러스터는 읽기 및 쓰기 리전 간에 전체 리전 간 복제 기능을 제공합니다.

지정된 감시 리전은 트랜잭션 로그 전용 쓰기를 지원하며 스토리지를 사용하지 않습니다. 감시 리전에는 엔드포인트가 없습니다. 즉, 감시 리전은 암호화된 트랜잭션 로그만 저장하고 관리 또는 구성이 필요하지 않으며 사용자가 액세스할 수 없습니다.

Aurora DSQL 트랜잭션 로그 및 사용자 스토리지는 Aurora DSQL 쿼리 프로세서에 단일 논리 볼륨으로 제공되는 모든 데이터와 함께 분산됩니다. Aurora DSQL은 데이터베이스 프라이머리 키 범위 및 액세스 패턴을 기반으로 데이터를 자동으로 분할, 병합 및 복제합니다. Aurora DSQL은 읽기 액세스 빈도에 따라 읽기 전용 복제본을 자동으로 스케일 업 및 스케일 다운합니다.

클러스터 스토리지 복제본은 다중 테넌트 스토리지 플릿에 분산됩니다. 구성 요소 또는 AZ가 손상된 경우 Aurora DSQL은 액세스 권한을 살아남은 구성 요소로 자동 리디렉션하고 누락된 복제본을 비동기적으로 복구합니다. Aurora DSQL이 손상된 복제본을 수정하면 Aurora DSQL은 자동으로 복제본을 스토리지 쿼럼에 다시 추가하고 클러스터에서 사용할 수 있도록 합니다.

높은 가용성

기본적으로 Aurora DSQL의 단일 리전 및 다중 리전 클러스터는 액티브-액티브이므로 클러스터를 수동으로 프로비저닝, 구성 또는 재구성할 필요가 없습니다. Aurora DSQL은 클러스터 복구를 완전히 자동화하므로 기존의 프라이머리-세컨더리 장애 조치 작업이 필요하지 않습니다. 복제는 항상 동기식이며 여러 AZ에서 수행되므로 장애 복구 중에 복제 지연 또는 비동기 세컨더리 데이터베이스로의 장애 조치로 인해 데이터가 손실될 위험이 없습니다.

단일 리전 클러스터는 세 개의 AZ에서 강력한 데이터 일관성으로 동시 액세스를 자동으로 활성화하는 다중 AZ 중복 엔드포인트를 제공합니다. 즉, 이 세 AZ 중 하나의 사용자 스토리지 복제본은 항상 하나 이상의 리더에게 동일한 결과를 반환하고 항상 쓰기를 수신할 수 있습니다. 이러한 강력한 일관성과 다중 AZ 복원력은 Aurora DSQL 다중 리전 클러스터의 모든 리전에서 사용할 수 있습니다. 즉, 다중 리전 클러스터는 강력히 일관된 두 개의 리전 엔드포인트를 제공하므로 클라이언트는 커밋 시 복제 지연 없이 두 리전 중 하나에 무차별적으로 읽거나 쓸 수 있습니다.

Aurora DSQL은 단일 리전 클러스터의 경우 99.99%, 다중 리전 클러스터의 경우 99.999%의 가용성을 제공합니다.

Amazon Aurora DSQL의 인프라 보안

관리형 서비스인 Amazon Aurora DSQL은 보안, 자격 증명 및 규정 준수 모범 사례에 설명된 AWS 글로벌 네트워크 보안 절차에 의해 보호됩니다.

AWS에서 게시한 API 직접 호출을 사용하여 네트워크를 통해 Aurora DSQL에 액세스합니다. 클라이언트가 전송 계층 보안(TLS) 1.2 이상을 지원해야 합니다. 클라이언트는 DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군도 지원해야 합니다. Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

또한 요청은 액세스 키 ID 및 IAM 위탁자와 관련된 보안 암호 액세스 키를 사용하여 서명해야 합니다. 또는 AWS Security Token Service(AWS STS)를 사용하여 임시 자격 증명을 생성하여 요청에 서명할 수 있습니다.

AWS PrivateLink를 사용하여 Amazon Aurora DSQL 클러스터 관리 및 연결

AWS PrivateLink for Amazon Aurora DSQL을 사용하면 Amazon Virtual Private Cloud에서 인터페이스 Amazon VPC 엔드포인트(인터페이스 엔드포인트)를 프로비저닝할 수 있습니다. 이러한 엔드포인트는 Amazon VPC 및 AWS Direct Connect를 통해 온프레미스에 있는 애플리케이션에서 또는 Amazon VPC 피어링을 통해 다른 AWS 리전에 있는 애플리케이션에서 직접 액세스할 수 있습니다. AWS PrivateLink 및 인터페이스 엔드포인트를 사용하면 애플리케이션에서 Aurora DSQL로의 프라이빗 네트워크 연결을 단순화할 수 있습니다.

Amazon VPC 내의 애플리케이션은 퍼블릭 IP 주소 없이 Amazon VPC 인터페이스 엔드포인트를 사용하여 Aurora DSQL에 액세스할 수 있습니다.

인터페이스 엔드포인트는 Amazon VPC의 서브넷에서 프라이빗 IP 주소가 할당된 하나 이상의 탄력적 네트워크 인터페이스(ENI)로 표시됩니다. 인터페이스 엔드포인트를 통해 Aurora DSQL로 보내는 요청은 AWS 네트워크에 유지됩니다. Amazon VPC를 온프레미스 네트워크에 연결하는 방법에 대한 자세한 내용은 AWS Direct Connect 사용 설명서 및 AWS Site-to-Site VPN VPN 사용 설명서를 참조하세요.

인터페이스 엔드포인트에 대한 일반적인 내용은 AWS PrivateLink 사용 설명서의 Access an AWS service using an interface Amazon VPC endpoint를 참조하세요.

Aurora DSQL용 Amazon VPC 엔드포인트의 유형

Aurora DSQL에는 두 가지 유형의 AWS PrivateLink 엔드포인트가 필요합니다.

1. 관리 엔드포인트 -이 엔드포인트는 Aurora DSQL 클러스터의 `get`, `create`, `update`, `delete`, `list`와 같은 관리 작업에 사용됩니다. AWS PrivateLink를 사용하여 Aurora DSQL 클러스터 관리을(를) 참조하세요.
2. 연결 엔드포인트 -이 엔드포인트는 PostgreSQL 클라이언트를 통해 Aurora DSQL 클러스터에 연결하는 데 사용됩니다. AWS PrivateLink를 사용하여 Aurora DSQL 클러스터에 연결을(를) 참조하세요.

AWS PrivateLink for Aurora DSQL 사용 시 고려 사항

Amazon VPC 고려 사항은 AWS PrivateLink for Aurora DSQL에 적용됩니다. 자세한 내용은 AWS PrivateLink 안내서의 Access an AWS service using an interface VPC endpoint 및 AWS PrivateLink quotas를 참조하세요.

AWS PrivateLink를 사용하여 Aurora DSQL 클러스터 관리

AWS Command Line Interface 또는 AWS 소프트웨어 개발 키트(SDK)를 사용하여 Aurora DSQL 인터페이스 엔드포인트를 통해 Aurora DSQL 클러스터를 관리할 수 있습니다.

Amazon VPC 엔드포인트 생성

Amazon VPC 인터페이스 엔드포인트를 생성하려면 AWS PrivateLink 안내서의 Create an Amazon VPC endpoint를 참조하세요.

```
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 요청에 기본 리전 DNS 이름을 사용하려면 Aurora DSQL 인터페이스 엔드포인트를 생성할 때 프라이빗 DNS를 비활성화하지 마세요. 프라이빗 DNS가 활성화되면 Amazon VPC 내에서 이루어진 Aurora DSQL 서비스에 대한 요청은 퍼블릭 DNS 이름이 아닌 Amazon VPC 엔드포인트의 프라이빗 IP 주소로 자동 확인됩니다. 프라이빗 DNS가 활성화되면 Amazon VPC 내에서 이루어진 Aurora DSQL 요청은 Amazon VPC 엔드포인트로 자동으로 확인됩니다.

프라이빗 DNS가 활성화되지 않은 경우 AWS CLI 명령과 함께 `--region` 및 `--endpoint-url` 파라미터를 사용하여 Aurora DSQL 인터페이스 엔드포인트를 통해 Aurora DSQL 클러스터를 관리합니다.

엔드포인트 URL을 사용하여 클러스터 나열

다음 예시에서 AWS 리전 `us-east-1` 및 VPC 엔드포인트 ID의 DNS 이름(`vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com`)을 실제 정보로 바꿉니다.

```
aws dsql --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsql.us-
east-1.vpce.amazonaws.com list-clusters
```

API 작업

Aurora DSQL의 리소스 관리에 대한 설명서는 Aurora DSQL API 참조를 참조하세요.

엔드포인트 정책 관리

Amazon VPC 엔드포인트 정책을 철저히 테스트하고 구성하면 Aurora DSQL 클러스터가 안전하고 규정을 준수하며 조직의 특정 액세스 제어 및 거버넌스 요구 사항에 부합하도록 할 수 있습니다.

예: 전체 Aurora DSQL 액세스 정책

다음 정책은 지정된 Amazon VPC 엔드포인트를 통해 모든 Aurora DSQL 작업 및 리소스에 대한 전체 액세스 권한을 부여합니다.

```
aws ec2 modify-vpc-endpoint \
    --vpc-endpoint-id vpce-xxxxxxxxxxxxxxxxx \
    --region region \
    --policy-document '{
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": "*",
          "Action": "dsql:*",
          "Resource": "*"
        }
      ]
    }'
```

예: 제한된 Aurora DSQL 액세스 정책

다음 정책은 이러한 Aurora DSQL 작업만 허용합니다.

- `CreateCluster`
- `GetCluster`
- `ListClusters`

다른 Aurora DSQL 작업은 모두 거부됩니다.

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

```
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "*"
    }
  ]
}
```

AWS PrivateLink를 사용하여 Aurora DSQL 클러스터에 연결

AWS PrivateLink 엔드포인트가 설정되고 활성화되면 PostgreSQL 클라이언트를 사용하여 Aurora DSQL 클러스터에 연결할 수 있습니다. 아래 연결 지침은 AWS PrivateLink 엔드포인트를 통해 연결하기 위한 적절한 호스트 이름을 구성하는 단계를 간략하게 설명합니다.

AWS PrivateLink 연결 엔드포인트 설정

1단계: 클러스터의 서비스 이름 가져오기

클러스터에 연결하기 위한 AWS PrivateLink 엔드포인트를 생성할 때 먼저 해당 클러스터의 서비스 이름을 가져와야 합니다.

AWS CLI

```
aws dsql get-vpc-endpoint-service-name \
--region us-east-1 \
--identifier your-cluster-id
```

응답의 예

```
{
    "serviceName": "com.amazonaws.us-east-1.dsql-fnh4"
}
```

서비스 이름에는 예시의 `dsql-fnh4`와 같은 식별자가 포함됩니다. 이 식별자는 클러스터에 연결하기 위한 호스트 이름을 구성할 때도 필요합니다.

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);
```

2단계: Amazon VPC 엔드포인트 생성

이전 단계에서 얻은 서비스 이름을 사용하여 Amazon VPC 엔드포인트를 생성합니다.

> **Important**
>
> 아래 연결 지침은 프라이빗이 DNS를 활성화한 경우에만 클러스터에 연결할 수 있습니다. 엔드포인트를 생성할 때 `--no-private-dns-enabled` 플래그를 사용하지 마세요. 사용하면 아래 연결 지침이 제대로 작동하지 않습니다. 프라이빗 DNS를 비활성화하는 경우 생성된 엔드포인트를 가리키는 자체 와일드카드 프라이빗 DNS 레코드를 생성해야 합니다.

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
```

응답의 예

```
{
    "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 API에 엔드포인트 URL 사용

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ec2.Ec2Client;
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointRequest;
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointResponse;
import software.amazon.awssdk.services.ec2.model.VpcEndpointType;

String region = "us-east-1";
String serviceName = "com.amazonaws.us-east-1.dsql-fnh4";  // Use the service name
 from previous step
String vpcId = "your-vpc-id";

Ec2Client ec2Client = Ec2Client.builder()
    .region(Region.of(region))
```

```
    .credentialsProvider(DefaultCredentialsProvider.create())
    .build();

CreateVpcEndpointRequest request = CreateVpcEndpointRequest.builder()
    .vpcId(vpcId)
    .serviceName(serviceName)
    .vpcEndpointType(VpcEndpointType.INTERFACE)
    .subnetIds("subnet-id-1", "subnet-id-2")
    .securityGroupIds("security-group-id")
    .build();

CreateVpcEndpointResponse response = ec2Client.createVpcEndpoint(request);
String vpcEndpointId = response.vpcEndpoint().vpcEndpointId();
System.out.println("VPC Endpoint created with ID: " + vpcEndpointId);
```

AWS PrivateLink 연결 엔드포인트를 사용하여 Aurora DSQL 클러스터에 연결

AWS PrivateLink 엔드포인트가 설정되고 활성화되면(`State`가 `available`인지 확인) PostgreSQL 클라이언트를 사용하여 Aurora DSQL 클러스터에 연결할 수 있습니다. AWS SDK 사용에 대한 지침은 Programming with Aurora DSQL의 가이드를 참조하세요. 호스트 이름 형식과 일치하도록 클러스터 엔드포인트를 변경해야 합니다.

호스트 이름 구성

AWS PrivateLink를 통해 연결하기 위한 호스트 이름은 퍼블릭 DNS 호스트 이름과 다릅니다. 다음 구성 요소를 사용하여 구성해야 합니다.

1. `Your-cluster-id`
2. 서비스 이름의 서비스 식별자입니다. 예: `dsql-fnh4`
3. AWS 리전은

다음 형식을 사용합니다. *`cluster-id`*`.`*`service-identifier`*`.`*`region`*`.on.aws`

예: PostgreSQL을 사용하여 연결

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export SERVICE_IDENTIFIER=dsql-fnh4  # This should match the identifier in your service
 name
```

```
# Construct the hostname
export HOSTNAME="$CLUSTERID.$SERVICE_IDENTIFIER.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $HOSTNAME)

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

AWS PrivateLink 문제 해결

일반적인 문제 및 해결 방법

다음 표에는 Aurora DSQL과 AWS PrivateLink를 함께 사용할 때와 관련된 일반적인 문제와 해결 방법이 나와 있습니다.

문제	가능한 원인	Solution
연결 제한 시간	보안 그룹이 제대로 구성되지 않음	Amazon VPC Reachability Analyzer를 사용하여 네트워킹 설정이 포트 5432에서 트래픽을 허용하는지 확인합니다.
DNS 확인 실패	프라이빗 DNS가 활성화되지 않음	프라이빗 DNS가 활성화된 상태에서 Amazon VPC 엔드포인트가 생성되었는지 확인합니다.
인증 실패	자격 증명이 잘못됨 또는 토큰이 만료됨	새 인증 토큰을 생성하고 사용자 이름을 확인합니다.
서비스 이름을 찾을 수 없음	클러스터 ID가 잘못됨	서비스 이름을 가져올 때 클러스터 ID와 AWS 리전을 다시 확인합니다.

관련 리소스

자세한 정보는 다음 자료를 참조하세요.

- Amazon Aurora DSQL 사용 설명서
- AWS PrivateLink 설명서

- AWS PrivateLink를 통해 AWS 서비스에 액세스

Amazon Aurora DSQL의 구성 및 취약성 분석

AWS가 게스트 운영 체제(OS), 데이터베이스 패치, 방화벽 구성, 재해 복구 등의 기본 보안 작업을 처리합니다. 적합한 제3자가 이 절차를 검토하고 인증하였습니다. 자세한 내용은 다음 리소스를 참조하세요.

- 공동 책임 모델
- Amazon Web Services: 보안 프로세스의 개요(백서)

교차 서비스 혼동된 대리인 방지

혼동된 대리자 문제는 작업을 수행할 권한이 없는 엔터티가 권한이 더 많은 엔터티에게 작업을 수행하도록 강요할 수 있는 보안 문제입니다. AWS에서는 교차 서비스 가장으로 인해 혼동된 대리자 문제가 발생할 수 있습니다. 교차 서비스 가장은 한 서비스(호출하는 서비스)가 다른 서비스(호출되는 서비스)를 직접적으로 호출할 때 발생할 수 있습니다. 직접적으로 호출하는 서비스는 다른 고객의 리소스에 대해 액세스 권한이 없는 방식으로 작동하게 권한을 사용하도록 조작될 수 있습니다. 이를 방지하기 위해 AWS에서는 계정의 리소스에 대한 액세스 권한이 부여된 서비스 보안 주체를 사용하여 모든 서비스에 대한 데이터를 보호하는 데 도움이 되는 도구를 제공합니다.

Amazon Aurora DSQL이 리소스에 다른 서비스를 제공하는 권한을 제한하려면 리소스 정책에서 `aws:SourceArn` 및 `aws:SourceAccount` 전역 조건 컨텍스트 키를 사용하는 것이 좋습니다. 하나의 리소스만 교차 서비스 액세스와 연결되도록 허용하려는 경우 `aws:SourceArn`를 사용하세요. 해당 계정의 모든 리소스가 교차 서비스 사용과 연결되도록 허용하려는 경우 `aws:SourceAccount`을(를) 사용합니다.

혼동된 대리자 문제로부터 보호하는 가장 효과적인 방법은 리소스의 전체 ARN이 포함된 `aws:SourceArn` 전역 조건 컨텍스트 키를 사용하는 것입니다. 리소스의 전체 ARN을 모르거나 여러 리소스를 지정하는 경우, ARN의 알 수 없는 부분에 대해 와일드카드 문자(*)를 포함한 `aws:SourceArn` 글로벌 조건 컨텍스트 키를 사용합니다. 예를 들어 `arn:aws:`*`servicename`*`:*:`*`123456789012`*`:*`입니다.

만약 `aws:SourceArn` 값에 Amazon S3 버킷 ARN과 같은 계정 ID가 포함되어 있지 않은 경우, 권한을 제한하려면 두 글로벌 조건 컨텍스트 키를 모두 사용해야 합니다.

`aws:SourceArn`의 값은 ResourceDescription이어야 합니다.

다음 예는 Aurora DSQL에서 aws:SourceArn 및 aws:SourceAccount 전역 조건 컨텍스트 키를 사용하여 혼동된 대리자 문제를 방지하는 방법을 보여줍니다.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "servicename.amazonaws.com"
    },
    "Action": "servicename:ActionName",
    "Resource": [
      "arn:aws:servicename:::ResourceName/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:servicename:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

Aurora DSQL의 보안 모범 사례

Aurora DSQL은 자체 보안 정책을 개발하고 구현할 때 고려해야 할 여러 보안 기능을 제공합니다. 다음 모범 사례는 일반적인 지침이며 완벽한 보안 솔루션을 나타내지는 않습니다. 이러한 모범 사례는 환경에 적절하지 않거나 충분하지 않을 수 있으므로 참고용으로만 사용해 주세요.

주제

- Aurora DSQL의 탐지 보안 모범 사례
- Aurora DSQL의 예방 보안 모범 사례

Aurora DSQL의 탐지 보안 모범 사례

Aurora DSQL을 안전하게 사용하는 다음 방법 외에도 클라우드 기술이 보안을 어떻게 개선하는지 알아보려면 AWS Well-Architected Tool의 보안을 참조하세요.

Amazon CloudWatch 경보

Amazon CloudWatch 경보를 사용하면 지정한 기간 동안 단일 지표를 감시합니다. 지표가 지정된 임계값을 초과하면 Amazon SNS 주제 또는 AWS Auto Scaling 정책으로 알림이 전송됩니다. CloudWatch 경보는 단순히 특정 상태에 있다고 해서 작업을 호출하지 않습니다. 대신, 상태가 변경되어 지정된 기간 동안 유지되어야 합니다.

식별 및 자동화를 위해 Aurora DSQL 리소스에 태그 지정

AWS 리소스에 태그 형태로 메타데이터를 지정할 수 있습니다. 각 태그는 리소스 관리, 검색 및 필터링을 더 수월하게 해줄 수 있는 옵션 값과 고객이 정의한 키로 구성된 간단한 레이블입니다.

태그를 지정하면 그룹화 제어를 구현할 수 있습니다. 고유한 태그 유형은 없지만 목적, 소유자, 환경 또는 기타 기준에 따라 리소스를 분류할 수 있습니다. 다음은 몇 가지 예입니다.

- 보안 - 암호화와 같은 요구 사항을 결정하는 데 사용됩니다.
- 기밀성 – 리소스가 지원하는 특정 데이터 기밀성 수준에 대한 식별자입니다.
- 환경 - 개발, 테스트 및 프로덕션 인프라 간 구별에 사용됩니다.

AWS 리소스에 태그 형태로 메타데이터를 지정할 수 있습니다. 각 태그는 리소스 관리, 검색 및 필터링을 더 수월하게 해줄 수 있는 옵션 값과 고객이 정의한 키로 구성된 간단한 레이블입니다.

태그를 지정하면 그룹화 제어를 구현할 수 있습니다. 고유한 태그 유형은 없지만 목적, 소유자, 환경 또는 기타 기준에 따라 리소스를 분류할 수 있습니다. 다음은 몇 가지 예입니다.

- 보안 - 암호화와 같은 요구 사항을 결정하는 데 사용됩니다.
- 기밀성 – 리소스가 지원하는 특정 데이터 기밀성 수준에 대한 식별자입니다.
- 환경 - 개발, 테스트 및 프로덕션 인프라 간 구별에 사용됩니다.

자세한 내용은 Best Practices for Tagging AWS Resources를 참조하세요.

Aurora DSQL의 예방 보안 모범 사례

Aurora DSQL을 안전하게 사용하는 다음 방법 외에도 클라우드 기술이 보안을 어떻게 개선하는지 알아보려면 AWS Well-Architected Tool의 보안을 참조하세요.

IAM 역할을 사용해 Aurora DSQL에 대한 액세스 인증

Aurora DSQL에 액세스하는 사용자, 애플리케이션 및 기타 AWS 서비스는 AWS API 및 AWS CLI 요청에 유효한 AWS 자격 증명을 포함해야 합니다. AWS 자격 증명을 애플리케이션이나 EC2 인스턴스에 직접 저장해서는 안 됩니다. 이는 자동으로 교체되지 않는 장기 자격 증명입니다. 이러한 자

격 증명이 손상되면 비즈니스에 상당한 영향을 미칩니다. IAM 역할을 사용하면 AWS 서비스 및 리소스에 액세스하는 데 사용할 수 있는 임시 액세스 키를 얻을 수 있습니다.

자세한 내용은 Aurora DSQL에 대한 인증 및 권한 부여 섹션을 참조하세요.

Aurora DSQL 기본 권한 부여에 IAM 정책 사용

권한을 부여할 때 권한을 부여받을 대상, 권한을 행사할 수 있는 Aurora DSQL API 작업, 해당 리소스에 허용하고자 하는 특정 작업을 결정합니다. 최소 권한을 구현하는 것이 오류 또는 악의적인 의도로 인해 발생할 수 있는 보안 위험과 영향을 줄일 수 있는 비결입니다.

권한 정책을 IAM 역할에 연결하고 Aurora DSQL 리소스에 대한 작업을 수행할 수 있는 권한을 부여합니다. 또한 IAM 엔터티에 대한 권한 경계를 사용할 수 있으며, 이를 통해 ID 기반 정책이 IAM 엔터티에 부여할 수 있는 최대 권한을 설정할 수 있습니다.

AWS 계정에 대한 루트 사용자 모범 사례와 마찬가지로 Aurora DSQL의 `admin` 역할을 사용하여 일상적인 작업을 수행하지 마세요. 대신 사용자 지정 데이터베이스 역할을 생성하여 클러스터를 관리하고 연결하는 것이 좋습니다. 자세한 내용은 Accessing Aurora DSQL 및 Understanding authentication and authorization for Aurora DSQL을 참조하세요.

프로덕션 환경에서 **`verify-full`** 사용

이 설정은 서버 인증서가 신뢰할 수 있는 인증 기관에서 서명되었고 서버 호스트 이름이 인증서와 일치하는지 확인합니다.

PostgreSQL 클라이언트 업데이트

보안 개선 사항을 활용하려면 PostgreSQL 클라이언트를 최신 버전으로 정기적으로 업데이트하세요. PostgreSQL 버전 17을 사용하는 것이 좋습니다.

Aurora DSQL에서 리소스 태그 지정

AWS에서 태그는 사용자가 정의하고 클러스터와 같은 Aurora DSQL 리소스와 연결하는 사용자 정의 키-값 페어입니다. 태그는 선택 사항입니다. 키를 제공하는 경우 값은 선택 사항입니다.

AWS Management Console, AWS CLI 또는 AWS SDK를 사용하여 Aurora DSQL 클러스터에서 태그를 추가, 나열 및 삭제할 수 있습니다. AWS Console을 사용하여 클러스터 생성 중 및 생성 후에 태그를 추가할 수 있습니다. AWS CLI를 사용하여 클러스터 생성 후에 태그를 지정하려면 `TagResource` 작업을 사용합니다.

이름으로 클러스터 태그 지정

Aurora DSQL은 전역적으로 고유한 식별자가 Amazon 리소스 이름(ARN)으로 할당된 클러스터를 생성합니다. 클러스터에 사용자 친화적 이름을 할당하려면 태그를 사용하는 것이 좋습니다.

Aurora DSQL 콘솔을 사용하여 콘솔을 생성하면 Aurora DSQL이 자동으로 태그를 생성합니다. 이 태그에는 이름 키와 클러스터의 이름을 나타내는 자동 생성된 값이 있습니다. 이 값은 구성 가능하므로 클러스터에 보다 친숙한 이름을 할당할 수 있습니다. 클러스터에 값이 연결된 이름 태그가 있는 경우 Aurora DSQL 콘솔 전체에서 값을 볼 수 있습니다.

태그 지정 요구 사항

태그를 지정할 때 요건은 다음과 같습니다.

- 키는 접두사로 `aws:`를 사용할 수 없습니다.
- 키는 태그 집합에 대해 고유해야 합니다.
- 키는 1~128자 사이의 허용된 문자이어야 합니다.
- 값은 0~256자 사이의 허용된 문자이어야 합니다.
- 값은 태그 집합마다 고유할 필요는 없습니다.
- 키와 값의 문자로는 문자, 숫자, 공백 그리고 _ . : / = + - @ 기호가 허용됩니다.
- 키와 값은 대/소문자를 구분합니다.

태그 지정 사용 참고 사항

Aurora DSQL에서 태그를 사용할 때는 다음 사항을 고려하세요.

- AWS CLI 또는 Aurora DSQL API 작업을 사용할 때 작업할 Aurora DSQL 리소스에 대한 Amazon 리소스 이름(ARN)을 제공해야 합니다. 자세한 내용은 Amazon Resource Name (ARNs) format for Aurora DSQL resources를 참조하세요.
- 각 리소스마다 태그 집합이 하나씩 있습니다. 여기에서 태그 집합이란 리소스에 할당되는 태그들이 하나 이상 모인 집합을 말합니다.
- 각 리소스는 태그 집합당 최대 50개의 태그를 가질 수 있습니다.
- 리소스를 삭제하는 경우에는 연결된 태그까지 모두 삭제됩니다.
- 리소스를 생성할 때 태그를 추가할 수 있으며, `TagResource`, `UntagResource` 및 `ListTagsForResource` API 작업을 사용하여 태그를 보고 수정할 수 있습니다.
- IAM 정책에 태그를 사용할 수 있습니다. 태그를 사용하여 Aurora DSQL 클러스터에 대한 액세스를 관리하고 해당 리소스에 적용 가능한 작업을 제어할 수 있습니다. 자세한 내용은 태그를 사용한 AWS 리소스 액세스 제어를 참조하세요.
- AWS에서 다양한 다른 활동에 태그를 사용할 수 있습니다. 자세한 내용은 Common tagging strategies를 참조하세요.

Amazon Aurora DSQL 사용에 대한 고려 사항

Amazon Aurora DSQL을 사용할 때는 다음 동작을 고려하세요. PostgreSQL 호환성 및 지원에 대한 자세한 내용은 Aurora DSQL의 SQL 기능 호환성 섹션을 참조하세요. 할당량 및 한도는 Amazon Aurora DSQL의 클러스터 할당량 및 데이터베이스 한도 섹션을 참조하세요.

- Aurora DSQL은 대규모 테이블의 트랜잭션 제한 시간 전에 `COUNT(*)` 작업을 완료하지 않습니다. 시스템 카탈로그에서 테이블 행 수를 검색하려면 Using systems tables and commands in Aurora DSQL을 참조하세요.
- `PG_PREPARED_STATEMENTS`를 직접 호출하는 드라이버는 클러스터에 대해 캐시된 준비된 문에 일관되지 않은 뷰를 제공할 수 있습니다. 동일한 클러스터 및 IAM 역할에 대해 연결당 준비된 문 수가 예상보다 많을 수 있습니다. Aurora DSQL은 사용자가 준비한 문 이름을 보존하지 않습니다.
- 드문 경우지만 다중 리전 연결 클러스터 장애 시나리오에서는 트랜잭션 커밋 가용성이 재개되는 데 예상보다 오래 걸릴 수 있습니다. 일반적으로 자동화된 클러스터 복구 작업으로 인해 일시적인 동시성 제어 또는 연결 오류가 발생할 수 있습니다. 대부분의 경우 워크로드의 일부에 대해서만 효과가 적용됩니다. 이러한 전송 오류가 표시되면 트랜잭션을 다시 시도하거나 클라이언트와 다시 연결합니다.
- Datagrip과 같은 일부 SQL 클라이언트는 시스템 메타데이터를 광범위하게 직접 호출하여 스키마 정보를 채웁니다. Aurora DSQL은 이 정보를 모두 지원하지 않으며 오류를 반환합니다. 이 문제는 SQL 쿼리 기능에 영향을 주지 않지만 스키마 표시에는 영향을 미칠 수 있습니다.
- 관리자 역할에는 데이터베이스 관리 작업과 관련된 권한 집합이 있습니다. 기본적으로 이러한 권한은 다른 사용자가 생성하는 객체로 확장되지 않습니다. 관리자 역할은 이러한 사용자 생성 객체에 대한 권한을 다른 사용자에게 부여하거나 취소할 수 없습니다. 관리자 사용자는 이러한 객체에 필요한 권한을 얻기 위해 그 외에 다른 역할을 자신에게 부여할 수 있습니다.

Amazon Aurora DSQL의 클러스터 할당량 및 데이터베이스 한도

이어지는 섹션에서는 Aurora DSQL의 클러스터 할당량 및 데이터베이스 한도에 대해 설명합니다.

클러스터 할당량

사용자의 AWS 계정에는 Aurora DSQL에 대해 다음과 같은 클러스터 할당량이 있습니다. 특정 AWS 리전 내의 단일 리전 및 다중 리전 클러스터에 대한 서비스 할당량 증가를 요청하려면 Service Quotas 콘솔 페이지를 사용하세요. 기타 할당량 증가는 AWS Support에 문의하세요.

설명	기본 한도	구성 가능성	Aurora DSQL 오류 코드
AWS 계정당 최대 단일 리전 클러스터 수	클러스터 20개	예	API 오류 코드 `ServiceQuotaExceededException`
AWS 계정당 최대 다중 리전 클러스터 수	클러스터 5개	예	API 오류 코드 `ServiceQuotaExceededException`
클러스터당 최대 스토리지	기본 한도 10TiB, 한도 증가 승인 시 최대 128TiB	예	`DISK_FULL(53100)`
클러스터당 최대 연결 수	연결 1만 개	예	`TOO_MANY_CONNECTIONS(53300)`
클러스터당 최대 연결 속도	초당 연결 100개	아니요	`CONFIGURED_LIMIT_EXCEEDED(53400)`
클러스터당 최대 연결 버스트 용량	연결 1,000개	아니요	오류 코드 없음

설명	기본 한도	구성 가능성	Aurora DSQL 오류 코드
최대 동시 복원 작업 수	4	아니요	오류 코드 없음
연결 채우기 속도	초당 연결 100개	아니요	오류 코드 없음

Aurora DSQL의 데이터베이스 한도

다음 표에서는 Aurora DSQL의 데이터베이스 한도를 설명합니다.

설명	기본 한도	구성 가능성	Aurora DSQL 오류 코드	오류 메시지
프라이머리 키에 사용되는 열의 최대 결합 크기	1KiB	아니요	54000	ERROR: key size too large
보조 인덱스에 있는 열의 최대 결합 크기	1KiB	아니요	54000	ERROR: key size too large
테이블 내 행의 최대 크기	2MiB	아니요	54000	ERROR: maximum row size excee
인덱스의 일부가 아닌 열의 최대 크기	1MiB	아니요	54000	ERROR: maximum column size ex
프라이머리 키 또는 보조 인덱스의 최대 열 수	8	아니요	54011	ERROR: more than 8 column key are not supported

설명	기본 한도	구성 가능성	Aurora DSQL 오류 코드	오류 메시지
테이블 내 최대 열 수	255	아니요	54011	ERROR: tables can have at mos
테이블의 최대 열 수	24	아니요	54000	ERROR: more than 24 indexes p allowed
한 쓰기 트랜잭션에서 수정된 모든 데이터의 최대 크기	10MiB	아니요	54000	ERROR: transaction size limit DETAIL: Current transaction s 10mb
트랜잭션 블록에서 변형할 수 있는 최대 테이블 및 인덱스 행 수	트랜잭션당 행 3,000개. PostgreSQL 호환성을 위한 Aurora DSQL 고려 사항을(를) 참조하세요.	아니요	54000	ERROR: transaction row limit
한 쿼리 작업에서 사용할 수 있는 최대 기본 메모리 양	트랜잭션당 128MiB	아니요	53200	ERROR: query requires too muc out of memory.
데이터베이스에 정의된 최대 스키마 수	10	아니요	54000	ERROR: more than 10 schemas n
데이터베이스의 최대 테이블 수	테이블 1,000개	아니요	54000	ERROR: creating more than 100 allowed

설명	기본 한도	구성 가능성	Aurora DSQL 오류 코드	오류 메시지
클러스터의 최대 데이터베이스 수	1	아니요	오류 코드 없음	`ERROR: unsupported statement`
최대 트랜잭션 시간	5분	아니요	`54000`	`ERROR: transaction age limit exceeded`
최대 연결 기간	60분	아니요	오류 코드 없음	오류 메시지 없음
데이터세트의 최대 뷰 수	5,000	아니요	`54000`	`ERROR: creating more than 500 allowed`
최대 뷰 정의 크기	2MiB	아니요	`54000`	`ERROR: view definition too la`

Aurora DSQL의 데이터 유형 제한은 Aurora DSQL에서 지원되는 데이터 유형 섹션을 참조하세요.

Aurora DSQL API 참조

AWS Management Console 및 AWS Command Line Interface(AWS CLI) 외에도 Aurora DSQL은 API 인터페이스도 제공합니다. API 작업을 사용하여 Aurora DSQL에서 리소스를 관리할 수 있습니다.

API 작업의 알파벳순 목록은 작업을 참조하십시오.

데이터 형식에 대한 알파벳순 목록은 데이터 형식을 참조하십시오.

공통 쿼리 파라미터 목록은 공통 파라미터를 참조하십시오.

오류 코드에 대한 설명은 공통 오류를 참조하십시오.

AWS CLI에 대한 자세한 내용은 Aurora DSQL에 대한 AWS Command Line Interface 참조를 참조하세요.

Aurora DSQL의 문제 해결

> **Note**
>
> 이어지는 주제에서는 Aurora DSQL을 사용할 때 발생할 수 있는 오류 및 문제에 대한 해결 방법을 조언합니다. 여기에 나열되지 않은 문제가 발견되면 AWS Support에 문의하세요.

주제

연결 오류 문제 해결

오류: 인식할 수 없는 SSL 오류 코드: 6

원인: 버전 14 이전의 psql 버전, 즉 서버 이름 표시(SNI)를 지원하지 않는 버전을 사용하고 있을 수 있습니다. Aurora DSQL에 연결할 때 SNI가 필요합니다.

`psql --version`을 사용하여 클라이언트 버전을 확인할 수 있습니다.

오류: NetworkUnreachable

연결 시도 중 `NetworkUnreachable` 오류가 발생하면 실제 네트워크에 문제가 있다는 신호라기보다는 클라이언트가 IPv6 연결을 지원하지 않는 것일 수 있습니다. 이 오류는 일반적으로 PostgreSQL 클라이언트가 듀얼 스택 연결을 처리하는 방식 때문에 IPv4 전용 인스턴스에서 발생합니다. 서버가 듀얼 스택 모드를 지원하는 경우 이러한 클라이언트는 먼저 호스트 이름을 IPv4 및 IPv6 주소로 확인합니다. 먼저 IPv4 연결을 시도한 다음 초기 연결이 실패하면 IPv6를 시도합니다. 시스템에서 IPv6를 지원하지 않는 경우 명확한 'IPv6 지원되지 않음' 메시지 대신 일반적인 `NetworkUnreachable` 오류가 표시됩니다.

인증 오류 문제 해결

사용자 '...'에 대한 IAM 인증 실패

Aurora DSQL IAM 인증 토큰을 생성할 때 설정할 수 있는 최대 기간은 1주일입니다. 1주일 후에는 해당 토큰으로 인증할 수 없습니다.

또한 Aurora DSQL은 위임된 역할이 만료된 경우 연결 요청을 거부합니다. 예를 들어 인증 토큰이 만료되지 않았더라도 임시 IAM 역할로 연결하려고 하면 Aurora DSQL은 연결 요청을 거부합니다.

IAM이 Aurora DSQL에서 작동하는 방식에 대한 자세한 내용은 Understanding authentication and authorization for Aurora DSQL 및 AWS Identity and Access Management in Aurora DSQL을 참조하세요.

GetObject 작업을 호출할 때 오류(InvalidAccessKeyId)가 발생했습니다. 제공한 AWS 액세스 키 ID가 레코드에 없습니다.

IAM이 요청을 거부했습니다. 자세한 내용은 요청에 서명하는 이유를 참조하세요.

<역할> IAM 역할이 존재하지 않음

Aurora DSQL에서 IAM 역할을 찾을 수 없습니다. 자세한 내용은 IAM 역할을 참조하세요.

IAM 역할은 IAM ARN과 같아야 합니다.

자세한 내용은 IAM 식별자 - IAM ARN을 참조하세요.

권한 부여 오류 문제 해결

<역할> 역할이 지원되지 않음

Aurora DSQL은 GRANT 작업을 지원하지 않습니다. Supported subsets of PostgreSQL commands in Aurora DSQL을 참조하세요.

<역할> 역할을 사용하여 신뢰를 구축할 수 없음

Aurora DSQL은 GRANT 작업을 지원하지 않습니다. Supported subsets of PostgreSQL commands in Aurora DSQL을 참조하세요.

<역할> 역할이 존재하지 않음

Aurora DSQL에서 지정된 데이터베이스 사용자를 찾을 수 없습니다. Authorize custom database roles to connect to a cluster를 참조하세요.

오류: <역할> 역할을 사용하여 IAM 신뢰를 부여하는 권한이 거부됨

데이터베이스 역할에 대한 액세스 권한을 부여하려면 관리자 역할을 사용하여 클러스터에 연결되어 있어야 합니다. 자세한 내용은 Authorize database roles to use SQL in a database를 참조하세요.

오류: <역할> 역할에 LOGIN 속성이 있어야 함

생성하는 데이터베이스 역할에 `LOGIN` 권한이 있어야 합니다.

이 오류를 해결하려면 `LOGIN` 권한이 있는 PostgreSQL 역할을 생성했는지 확인합니다. 자세한 내용은 PostgreSQL 설명서에서 CREATE ROLE 및 ALTER ROLE을 참조하세요.

오류: 일부 객체가 해당 역할에 의존하기 때문에 <역할> 역할을 삭제할 수 없음

Aurora DSQL은 `AWS IAM REVOKE`를 사용하여 관계를 취소할 때까지 IAM 관계가 있는 데이터베이스 역할을 삭제하면 오류를 반환합니다. 자세한 내용은 Revoking authorization을 참조하세요.

SQL 오류 문제 해결

오류: 지원되지 않음

Aurora DSQL은 모든 PostgreSQL 기반 언어를 지원하지 않습니다. 지원되는 항목에 대한 자세한 내용은 Supported PostgreSQL features in Aurora DSQL을 참조하세요.

오류: 읽기 전용 트랜잭션의 SELECT FOR UPDATE가 no-op임

읽기 전용 트랜잭션에서 허용되지 않는 작업을 시도하고 있습니다. 자세한 내용은 Understanding concurrency control in Aurora DSQL을 참조하세요.

오류: 대신 **`CREATE INDEX ASYNC`** 사용

기존 행이 있는 테이블에 인덱스를 생성하려면 `CREATE INDEX ASYNC` 명령을 사용해야 합니다. 자세한 내용은 Creating indexes asynchronously in Aurora DSQL을 참조하세요.

OCC 오류 문제 해결

OC000 '오류: 변형이 다른 트랜잭션과 충돌합니다. 필요에 따라 다시 시도하세요.'

OC001 '오류: 스키마가 다른 트랜잭션에 의해 업데이트되었습니다. 필요에 따라 다시 시도하세요.'

PostgreSQL 세션에 스키마 카탈로그의 캐시된 복사본이 있었습니다. 캐시된 복사본은 로드 당시 유효했습니다. 이것을 T1 시점과 버전 V1이라고 하겠습니다.

또 다른 트랜잭션은 T2 시점에 카탈로그를 업데이트합니다. 이것을 V2라고 하겠습니다.

원래 세션이 T2 시점에 스토리지에서 읽기를 시도하면 카탈로그 버전 V1을 계속 사용하고 있는 것입니다. T2 시점의 최신 카탈로그 버전이 V2이므로 Aurora DSQL의 스토리지 계층은 요청을 거부합니다.

원래 세션에서 T3 시점에 다시 시도하면 Aurora DSQL이 카탈로그 캐시를 새로 고칩니다. T3 시점의 트랜잭션은 카탈로그 V2를 사용합니다. Aurora DSQL은 T2 시점 이후 다른 카탈로그 변경 사항이 없는 한 트랜잭션을 완료합니다.

SSL/TLS 연결 문제 해결

SSL 오류: 인증서 확인 실패

이 오류는 클라이언트가 서버의 인증서를 확인할 수 없음을 나타냅니다. 다음을 확인하세요.

1. Amazon Root CA 1 인증서가 올바르게 설치됨. 이 인증서를 검증하고 설치하는 방법에 대한 지침은 Aurora DSQL 연결을 위한 SSL/TLS 인증서 구성 섹션을 참조하세요.
2. PGSSLROOTCERT 환경 변수가 올바른 인증서 파일을 가리킴
3. 인증서 파일에 올바른 권한이 있음

인식할 수 없는 SSL 오류 코드: 6

이 오류는 버전 14 미만의 PostgreSQL 클라이언트에서 발생합니다. PostgreSQL 클라이언트를 버전 17로 업그레이드하여 이 문제를 해결합니다.

SSL 오류: 등록되지 않은 체계(Windows)

이는 시스템 인증서를 사용할 때 Windows psql 클라이언트에서 알려진 문제입니다. Windows에서 연결 지침에 설명된 다운로드된 인증서 파일 메서드를 사용합니다.

Amazon Aurora DSQL 사용 설명서의 문서 기록

다음 표에서는 Aurora DSQL에 대한 문서 릴리스를 소개합니다.

변경 사항	설명	날짜
Amazon Aurora DSQL의 정식 출시(GA)	Amazon Aurora DSQL은 이제 CloudWatch 모니터링, 향상된 데이터 보호 기능 및 AWS Backup 통합에 대한 추가 지원과 함께 정식 출시되었습니다. 자세한 내용은 Monitoring Aurora DSQL with CloudWatch, Backup and restore for Amazon Aurora DSQL, Data encryption for Amazon Aurora DSQL을 참조하세요.	2025년 5월 27일
AmazonAuroraDSQLFullAccess 업데이트	작업 시작, 중지 및 모니터링을 포함하여 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행하는 기능을 추가합니다. 또한 클러스터 암호화에 고객 관리형 KMS 키를 사용할 수 있는 기능도 추가합니다. 자세한 내용은 AmazonAuroraDSQLFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 21일
AmazonAuroraDSQLConsoleFullAccess 업데이트	AWS Console Home을 통해 Aurora DSQL 클러스터에 대한 백업 및 복원 작업을 수행하는 기능을 추가합니다. 여기에는 작업 시작, 중지 및 모니터링이 포함됩니다. 또한 클러스터 암호화 및 AWS CloudShell	2025년 5월 21일

	시작을 위해 고객 관리형 KMS 키를 사용하는 것을 지원합니다. 자세한 내용은 AmazonAuroraDSQLConsoleFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	
AmazonAuroraDSQLReadOnlyAccess 업데이트	AWS PrivateLink를 통해 Aurora DSQL 클러스터에 연결할 때 올바른 VPC 엔드포인트 서비스 이름을 결정하는 기능을 포함합니다. Aurora DSQL은 셀마다 고유한 엔드포인트를 생성하므로 이 API를 사용하면 클러스터에 대한 올바른 엔드포인트를 식별하고 연결 오류를 방지할 수 있습니다. 자세한 내용은 AmazonAuroraDSQLReadOnlyAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일

AmazonAuroraDSQLFullAccess 업데이트	이 정책은 여러 AWS 리전, 즉, `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , `RemovePeerCluster` 에서 데이터베이스 클러스터를 생성하고 관리할 수 있는 네 가지 새로운 권한을 추가합니다. 이러한 권한에는 리소스 수준 제어 및 조건 키가 포함되어 있어 수정할 수 있는 클러스터 사용자를 제어할 수 있습니다. 또한 이 정책은 AWS PrivateLink를 통해 Aurora DSQL 클러스터에 연결하는 데 도움이 되는 `GetVpcEndpointServiceName` 권한을 추가합니다. 자세한 내용은 AmazonAuroraDSQLConsoleFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일

AmazonAuroraDSQLConsoleFullAccess 업데이트	다중 리전 클러스터 관리 및 VPC 엔드포인트 연결을 지원하기 위해 Aurora DSQL에 새 권한을 추가합니다. 새 권한은 `PutMultiRegionProperties` , `PutWitnessRegion` , `AddPeerCluster` , `RemovePeerCluster` , `GetVpcEndpointServiceName` 입니다. AmazonAuroraDSQLConsoleFullAccess 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 13일
AuroraDsqlServiceLinkedRolePolicy 업데이트	`AWS/AuroraDSQL` 에 지표를 게시하고 정책에 `AWS/Usage` `CloudWatch` 네임스페이스를 게시하는 기능을 추가합니다. 이 기능을 사용하면 연결된 서비스 또는 역할이 CloudWatch 환경에 보다 포괄적인 용도 및 성능 데이터를 내보낼 수 있습니다. 자세한 내용은 AuroraDsqlServiceLinkedRolePolicy 및 Using service-linked roles in Aurora DSQL을 참조하세요.	2025년 5월 8일

AWS PrivateLink for Amazon Aurora DSQL	Aurora DSQL은 이제 AWS PrivateLink를 지원합니다. AWS PrivateLink를 사용하면 인터페이스 Amazon VPC 엔드포인트와 프라이빗 IP 주소를 사용하여 Virtual Private Cloud(VPC), Aurora DSQL 및 온프레미스 데이터 센터 간의 프라이빗 네트워크 연결을 단순화할 수 있습니다. 자세한 내용은 Managing and connecting to Amazon Aurora DSQL clusters using AWS PrivateLink를 참조하세요.	2025년 5월 8일
최초 릴리스	Amazon Aurora DSQL 사용 설명서의 최초 릴리스입니다.	2024년 12월 3일