



사용 설명서

# Amazon Aurora DSQL



# Amazon Aurora DSQL: 사용 설명서

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

Amazon의 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께, Amazon 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

# Table of Contents

|                                       |    |
|---------------------------------------|----|
| Amazon Aurora DSQL이란 무엇인가요? .....     | 1  |
| 사용해야 하는 경우 .....                      | 1  |
| 주요 기능 .....                           | 1  |
| AWS 리전 가용성 .....                      | 2  |
| 다중 리전 클러스터 .....                      | 4  |
| 가격 책정 .....                           | 5  |
| 다음 단계 .....                           | 5  |
| 시작하기 .....                            | 6  |
| 사전 조건 .....                           | 6  |
| 단일 리전 클러스터 생성 .....                   | 6  |
| 클러스터에 연결 .....                        | 7  |
| SQL 명령 실행 .....                       | 8  |
| 다중 리전 클러스터 생성 .....                   | 8  |
| 문제 해결 .....                           | 11 |
| 인증 및 권한 부여 .....                      | 12 |
| 클러스터 관리 .....                         | 12 |
| 클러스터에 연결 .....                        | 12 |
| PostgreSQL 및 IAM 역할 .....             | 13 |
| Aurora DSQL에서 IAM 정책 작업 사용 .....      | 14 |
| IAM 정책 작업을 사용하여 클러스터에 연결 .....        | 14 |
| IAM 정책 작업을 사용하여 클러스터 관리 .....         | 15 |
| IAM 및 PostgreSQL을 사용하여 권한 부여 취소 ..... | 16 |
| 인증 토큰 생성 .....                        | 16 |
| 콘솔 .....                              | 17 |
| AWS CloudShell .....                  | 18 |
| AWS CLI .....                         | 19 |
| Aurora DSQL SDK .....                 | 20 |
| 데이터베이스 역할 및 IAM 인증 .....              | 29 |
| IAM 역할 .....                          | 29 |
| IAM 사용자 .....                         | 30 |
| 연결 .....                              | 30 |
| 쿼리 .....                              | 30 |
| 매핑 보기 .....                           | 31 |
| 취소 .....                              | 31 |

|                                |     |
|--------------------------------|-----|
| Aurora DSQL 및 PostgreSQL ..... | 32  |
| 호환성 하이라이트 .....                | 32  |
| 분산 아키텍처의 이점 .....              | 33  |
| SQL 호환성 .....                  | 33  |
| 지원되는 데이터 유형 .....              | 34  |
| 지원되는 SQL 기능 .....              | 39  |
| 지원되는 SQL 명령 하위 집합 .....        | 43  |
| 마이그레이션 가이드 .....               | 62  |
| 동시성 제어 .....                   | 69  |
| 트랜잭션 충돌 .....                  | 70  |
| 트랜잭션 성능 최적화 지침 .....           | 70  |
| DDL 및 분산 트랜잭션 .....            | 70  |
| 프라이머리 키 .....                  | 71  |
| 데이터 구조 및 스토리지 .....            | 72  |
| 프라이머리 키 선택 지침 .....            | 72  |
| 시퀀스 및 자격 증명 열 .....            | 73  |
| 시퀀스 조작 함수 .....                | 73  |
| ID 열 .....                     | 75  |
| 시퀀스 및 자격 증명 열 작업 .....         | 77  |
| 비동기 인덱스 .....                  | 78  |
| 구문 .....                       | 79  |
| 파라미터 .....                     | 79  |
| 사용 노트 .....                    | 80  |
| 인덱스 생성 .....                   | 81  |
| 인덱스 쿼리 .....                   | 82  |
| 고유 인덱스 빌드 실패 .....             | 83  |
| 고유성 위반 .....                   | 83  |
| 시스템 테이블 및 명령 .....             | 85  |
| 시스템 테이블 .....                  | 85  |
| 유용한 시스템 쿼리 .....               | 96  |
| ANALYZE 명령 .....               | 97  |
| EXPLAIN 계획 .....               | 98  |
| PostgreSQL EXPLAIN 계획 .....    | 98  |
| 핵심 요소 .....                    | 99  |
| 필터링 .....                      | 99  |
| EXPLAIN 계획 읽기 .....            | 100 |

|                              |     |
|------------------------------|-----|
| EXPLAIN ANALYZE의 DPU .....   | 104 |
| Aurora DSQL 클러스터 관리 .....    | 107 |
| 단일 리전 클러스터 .....             | 107 |
| AWS SDK 사용 .....             | 107 |
| AWS CLI 사용 .....             | 146 |
| 다중 리전 클러스터 .....             | 149 |
| AWS SDK 사용 .....             | 149 |
| AWS CLI 사용 .....             | 203 |
| CloudFormation .....         | 209 |
| 초기 구성 .....                  | 209 |
| 클러스터 찾기 .....                | 210 |
| 구성 업데이트 .....                | 210 |
| Aurora DSQL 클러스터 수명 주기 ..... | 211 |
| 클러스터 상태 .....                | 211 |
| 클러스터 상태 보기 .....             | 214 |
| Aurora DSQL을 사용한 프로그래밍 ..... | 215 |
| Connectors .....             | 215 |
| JDBC 커넥터 .....               | 216 |
| Python 커넥터 .....             | 220 |
| Go 커넥터 .....                 | 232 |
| Node.js 커넥터 .....            | 238 |
| Ruby 커넥터 .....               | 247 |
| .NET 커넥터 .....               | 253 |
| Aurora DSQL 액세스 .....        | 257 |
| SQL 클라이언트 .....              | 259 |
| DBeaver .....                | 259 |
| JetBrains DataGrip .....     | 262 |
| psql .....                   | 263 |
| VSCode .....                 | 265 |
| 문제 해결 .....                  | 266 |
| 데이터베이스 연결 도구 .....           | 266 |
| Aurora DSQL 어댑터 .....        | 267 |
| 데이터베이스 드라이버 샘플 .....         | 267 |
| ORM 및 프레임워크 샘플 .....         | 269 |
| 데이터 로딩 .....                 | 270 |
| 로드 접근 방식 선택 .....            | 270 |

|                                             |     |
|---------------------------------------------|-----|
| Aurora DSQL 로더 .....                        | 271 |
| 마이그레이션 경로 .....                             | 276 |
| PostgreSQL \copy 사용 .....                   | 278 |
| 추가 리소스 .....                                | 279 |
| 생성형 AI .....                                | 279 |
| AWS Labs Aurora DSQL MCP 서버 .....           | 279 |
| Aurora DSQL 운영: 스킬 및 파워 .....               | 287 |
| 쿼리 편집기 .....                                | 293 |
| 사전 조건 .....                                 | 293 |
| 쿼리 편집기 사용하기 .....                           | 293 |
| 쿼리 편집기: Aurora DSQL과 함께 JupyterLab 사용 ..... | 295 |
| 시작하기 .....                                  | 295 |
| 예제 노트북 .....                                | 298 |
| 참조 자료 .....                                 | 298 |
| 백업 및 복원 .....                               | 299 |
| AWS Backup 시작 .....                         | 299 |
| 백업 복원 .....                                 | 299 |
| 단일 리전 클러스터 복원 .....                         | 300 |
| 다중 리전 클러스터 복원 .....                         | 300 |
| 모니터링 및 규정 준수 .....                          | 300 |
| 추가 리소스 .....                                | 300 |
| 모니터링 및 로깅 .....                             | 302 |
| CloudWatch를 사용하여 모니터링 .....                 | 302 |
| 관찰성 .....                                   | 302 |
| 사용법 .....                                   | 304 |
| CloudTrail을 사용하여 로깅 .....                   | 305 |
| 관리 이벤트 .....                                | 306 |
| 데이터 이벤트 .....                               | 307 |
| 보안 .....                                    | 309 |
| AWS 관리형 정책 .....                            | 310 |
| AmazonAuroraDSQLFullAccess .....            | 310 |
| AmazonAuroraDSQLReadOnlyAccess .....        | 311 |
| AmazonAuroraDSQLConsoleFullAccess .....     | 312 |
| AuroraDSQLServiceRolePolicy .....           | 313 |
| 정책 업데이트 .....                               | 313 |
| 데이터 보호 .....                                | 318 |

|                                      |     |
|--------------------------------------|-----|
| 데이터 암호화 .....                        | 319 |
| 감시 리전의 데이터 보호 .....                  | 320 |
| SSL/TLS 인증서 .....                    | 321 |
| 데이터 암호화 .....                        | 319 |
| KMS 키 유형 .....                       | 327 |
| 저장 시 암호화 .....                       | 328 |
| KMS 및 데이터 키 사용 .....                 | 329 |
| KMS 키 권한 부여 .....                    | 331 |
| 암호화 컨텍스트 .....                       | 333 |
| AWS KMS 모니터링 .....                   | 333 |
| 암호화된 클러스터 생성 .....                   | 336 |
| 키 제거 또는 업데이트 .....                   | 338 |
| 고려 사항 .....                          | 340 |
| ID 및 액세스 관리 .....                    | 341 |
| 고객 .....                             | 341 |
| ID를 통한 인증 .....                      | 342 |
| 정책을 사용하여 액세스 관리 .....                | 343 |
| Aurora DSQL에서 IAM을 사용하는 방법 .....     | 344 |
| ID 기반 정책 예시 .....                    | 349 |
| 문제 해결 .....                          | 355 |
| 리소스 기반 정책 .....                      | 356 |
| 사용해야 하는 경우 .....                     | 357 |
| 정책을 사용하여 생성 .....                    | 358 |
| 정책 추가 및 편집 .....                     | 361 |
| 정책 보기 .....                          | 364 |
| 정책 제거 .....                          | 365 |
| 정책 예시 .....                          | 366 |
| 퍼블릭 액세스 차단 .....                     | 371 |
| API 작업 .....                         | 374 |
| 서비스 연결 역할 사용 .....                   | 376 |
| Aurora DSQL의 서비스 연결 역할 권한 .....      | 376 |
| 서비스 연결 역할 생성 .....                   | 377 |
| 서비스 연결 역할 편집 .....                   | 377 |
| 서비스 연결 역할 삭제 .....                   | 378 |
| Aurora DSQL 서비스 연결 역할이 지원되는 리전 ..... | 378 |
| IAM 조건 키 사용 .....                    | 378 |

|                                     |     |
|-------------------------------------|-----|
| 지정된 리전에 클러스터 생성 .....               | 378 |
| 특정 리전에 다중 리전 클러스터 생성 .....          | 379 |
| 특정 감시 리전을 사용하여 다중 리전 클러스터 생성 .....  | 379 |
| 사고 대응 .....                         | 380 |
| 규정 준수 확인 .....                      | 381 |
| 복원력 .....                           | 381 |
| 백업 및 복원 .....                       | 382 |
| 복제 .....                            | 382 |
| 높은 가용성 .....                        | 382 |
| 결함 주입 테스트 .....                     | 383 |
| 인프라 보안 .....                        | 384 |
| AWS PrivateLink를 사용하여 클러스터 관리 ..... | 384 |
| 구성 및 취약성 분석 .....                   | 395 |
| 교차 서비스 혼동된 대리인 방지 .....             | 395 |
| 보안 모범 사례 .....                      | 396 |
| 탐지 보안 모범 사례 .....                   | 397 |
| 예방적 보안 모범 사례 .....                  | 397 |
| 리소스에 태그 지정 .....                    | 399 |
| [Name tag] .....                    | 399 |
| 태그 지정 요구 사항 .....                   | 399 |
| 태그 지정 사용 참고 사항 .....                | 399 |
| 고려 사항 .....                         | 401 |
| 할당량 및 제한 .....                      | 403 |
| 클러스터 할당량 .....                      | 403 |
| 데이터베이스 한도 .....                     | 404 |
| API 참조 .....                        | 407 |
| 문제 해결 .....                         | 261 |
| 연결 오류 .....                         | 408 |
| 인증 오류 .....                         | 409 |
| 권한 부여 오류 .....                      | 410 |
| SQL 오류 .....                        | 410 |
| VPC 오류 .....                        | 411 |
| SSL/TLS 연결 .....                    | 411 |
| 피드백 제공 .....                        | 413 |
| 피드백 채널 .....                        | 413 |
| 효과적인 기능 요청 .....                    | 413 |



문서 이력 ..... 414

# 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.

## PostgreSQL와의 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 다양한 PostgreSQL 기능, 표현식 및 데이터 유형을 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.

| 리전 이름              | 리전             | 엔드포인트                       | 프로토콜  |
|--------------------|----------------|-----------------------------|-------|
| 미국 동부<br>(오하이오)    | us-east-2      | dsql.us-east-2.api.aws      | HTTPS |
|                    |                | dsql-fips.us-east-2.api.aws | HTTPS |
| 미국 동부<br>(버지니아 북부) | us-east-1      | dsql.us-east-1.api.aws      | HTTPS |
|                    |                | dsql-fips.us-east-1.api.aws | HTTPS |
| 미국 서부<br>(오리곤)     | us-west-2      | dsql.us-west-2.api.aws      | HTTPS |
|                    |                | dsql-fips.us-west-2.api.aws | HTTPS |
| 아시아 태평양(멜버른)       | ap-southeast-4 | dsql.ap-southeast-4.api.aws | HTTPS |
| 아시아 태평양(오사카)       | ap-northeast-3 | dsql.ap-northeast-3.api.aws | HTTPS |
| 아시아 태평양(서울)        | ap-northeast-2 | dsql.ap-northeast-2.api.aws | HTTPS |
| 아시아 태평양(시드니)       | ap-southeast-2 | dsql.ap-southeast-2.api.aws | HTTPS |
| 아시아 태평양(도쿄)        | ap-northeast-1 | dsql.ap-northeast-1.api.aws | HTTPS |

| 리전 이름       | 리전           | 엔드포인트                          | 프로토콜  |
|-------------|--------------|--------------------------------|-------|
| 캐나다(중부)     | ca-central-1 | dsql.ca-central-1.api.aws      | HTTPS |
|             |              | dsql-fips.ca-central-1.api.aws | HTTPS |
| 캐나다 서부(캘거리) | ca-west-1    | dsql.ca-west-1.api.aws         | HTTPS |
|             |              | dsql-fips.ca-west-1.api.aws    | HTTPS |
| 유럽(프랑크푸르트)  | eu-central-1 | dsql.eu-central-1.api.aws      | HTTPS |
| 유럽(아일랜드)    | eu-west-1    | dsql.eu-west-1.api.aws         | HTTPS |
| 유럽(런던)      | eu-west-2    | dsql.eu-west-2.api.aws         | HTTPS |
| 유럽(파리)      | eu-west-3    | dsql.eu-west-3.api.aws         | HTTPS |

## Aurora DSQL의 다중 리전 클러스터 가용성

특정 AWS 리전 세트 내에서 Aurora DSQL 다중 리전 클러스터를 생성할 수 있습니다. 각 리전 세트는 다중 리전 클러스터에서 함께 작동할 수 있는 지리적으로 관련된 리전을 그룹화한 것입니다.

### 미국 리전

- 미국 동부(버지니아 북부)
- 미국 동부(오하이오)
- 미국 서부(오리건)

### 아시아 태평양 리전

- 아시아 태평양(오사카)
- 아시아 태평양(서울)

- 아시아 태평양(도쿄)

## 유럽 리전

- 유럽(프랑크푸르트)
- 유럽(아일랜드)
- 유럽(런던)
- 유럽(파리)

## 중요 제한 사항

다중 리전 클러스터는 단일 리전 세트 내에서 생성해야 합니다. 예를 들어 미국 동부(버지니아 북부)와 유럽(아일랜드) 리전을 모두 포함하는 클러스터는 생성할 수 없습니다.

### Important

Aurora DSQL은 현재 교차 대륙 다중 리전 클러스터를 지원하지 않습니다.

## Aurora DSQL 요금

비용 정보는 [Aurora DSQL pricing](#)을 참조하세요.

## 다음 단계

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

- [Aurora DSQL 시작하기](#)
- [Aurora DSQL의 SQL 기능 호환성](#)
- [PostgreSQL 호환 클라이언트를 사용하여 Aurora DSQL에 액세스](#)
- [Aurora DSQL 및 PostgreSQL](#)

# Aurora DSQL 시작하기

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

## 주제

- [사전 조건](#)
- [1단계: Aurora DSQL 단일 리전 클러스터 생성](#)
- [2단계: Aurora DSQL 클러스터에 연결](#)
- [3단계: Aurora DSQL에서 샘플 SQL 명령 실행](#)
- [4단계 \(선택 사항\): 다중 리전 클러스터 생성](#)
- [문제 해결](#)

## 사전 조건

Aurora DSQL 사용을 시작하기 전에 다음 사전 조건을 충족하는지 확인하세요.

- IAM Identity에 [콘솔에 로그인](#)할 수 있는 권한이 있어야 합니다.
- IAM Identity는 다음 기준을 충족해야 합니다.
  - AWS 계정의 어떤 리소스에서나 작업을 수행할 수 있는 액세스 권한
  - AmazonAuroraDSQLConsoleFullAccess AWS 관리형 정책이 [연결됩니다](#).

## 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/dsql>에서 Aurora DSQL 콘솔을 엽니다.

2. 클러스터 생성을 선택한 다음 단일 리전을 선택합니다.
3. (선택 사항) 기본 이름 태그의 값을 변경합니다.
4. (선택 사항) 이 클러스터에 대한 태그를 추가합니다.
5. (선택 사항) 클러스터 설정에서 다음 옵션 중 하나를 선택합니다.
  - 암호화 설정 사용자 지정(고급)을 선택하여 AWS KMS key를 선택하거나 생성합니다. 고객 관리형 키를 사용하는 경우 키 정책이 Aurora DSQL에 필요한 권한을 부여하는지 확인합니다. 자세한 내용은 [고객 관리형 키에 대한 키 정책](#) 섹션을 참조하세요.
  - 삭제 작업으로 인해 클러스터가 삭제되지 않도록 방지하려면, 삭제 방지 활성화를 선택합니다. 삭제 방지 기능은 기본적으로 선택됩니다.
  - 리소스 기반 정책(고급)을 선택하여 이 클러스터에 대한 액세스 제어 정책을 지정합니다.
6. 클러스터 생성을 선택합니다.
7. 콘솔에서 클러스터 페이지로 돌아갑니다. 클러스터가 생성 중임을 나타내는 알림 배너가 나타납니다. 클러스터 ID를 선택하여 클러스터 세부 정보 보기를 엽니다.

## 2단계: Aurora DSQL 클러스터에 연결

Aurora DSQL은 DSQL 쿼리 편집기, AWS CloudShell, 로컬 psql 클라이언트 및 기타 PostgreSQL 호환 도구를 포함하여 클러스터에 연결하는 여러 가지 방법을 지원합니다. 이 단계에서는 새 클러스터와 상호 작용을 시작하는 빠른 방법을 제공하는 [Aurora DSQL 쿼리 편집기](#)를 사용하여 연결합니다.

쿼리 편집기를 사용하여 연결하는 방법

1. Aurora DSQL 콘솔(<https://console.aws.amazon.com/dsql>)에서 클러스터 페이지를 열고 클러스터 생성이 완료되었고 상태가 활성인지 확인합니다.
2. 목록에서 클러스터를 선택하거나 클러스터 ID를 선택하여 클러스터 세부 정보 페이지를 엽니다.
3. 쿼리 편집기로 연결을 선택합니다.
4. 방금 생성된 클러스터의 관리자 연결을 선택합니다.
  - 선택적으로 사용자 지정 역할에 연결할 수 있습니다. [데이터베이스 역할 및 IAM 인증 사용](#) 섹션을 참조하세요.



## 3단계: Aurora DSQL에서 샘플 SQL 명령 실행

SQL 문을 실행하여 Aurora DSQL 클러스터를 테스트합니다. 쿼리 편집기에서 클러스터를 연 후 각 샘플 쿼리를 단계별로 선택하고 실행합니다.

### Aurora DSQL에서 샘플 SQL 명령 실행

1. test이라는 스키마를 생성합니다.

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

2. 자동으로 생성된 UUID를 프라이머리 키로 사용하는 hello\_world 테이블을 생성합니다.

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

3. 샘플 행을 삽입합니다.

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

4. 삽입된 값을 읽습니다.

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

5. (선택 사항) 정리

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

## 4단계 (선택 사항): 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 쿼리를 지원합니다.

다음 절차에서는 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. [Aurora DSQL 콘솔](#)에 로그인합니다.
2. 탐색 창에서 클러스터를 선택합니다.
3. 클러스터 생성을 선택한 다음 다중 리전을 선택합니다.
4. (선택 사항) 기본 이름 태그의 값을 변경합니다.
5. (선택 사항)이 클러스터에 대한 태그를 추가합니다.
6. 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.
7. (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. 고객 관리형 키를 사용하는 경우 키 정책이 Aurora DSQL에 필요한 권한을 부여하는지 확인합니다. 자세한 내용은 [고객 관리형 키에 대한 키 정책](#) 섹션을 참조하세요.
  - 삭제 작업으로 인해 클러스터가 삭제되지 않도록 방지하려면, 삭제 방지 활성화를 선택합니다. 삭제 방지 기능은 기본적으로 선택됩니다.

- 리소스 기반 정책(고급)을 선택하여 이 클러스터에 대한 액세스 제어 정책을 지정합니다.
8. 클러스터 생성을 선택하여 첫 번째 클러스터를 생성합니다. 이전 단계에서 ARN을 입력하지 않은 경우 콘솔에 클러스터 설정 보류 중 알림이 표시됩니다.
  9. 클러스터 설정 보류 중 알림에서 다중 리전 클러스터 설정 완료를 선택합니다. 이 작업은 다른 리전에서 두 번째 클러스터 생성을 시작합니다.
  10. 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.
  11. ***your-second-region***에서 클러스터 생성을 선택합니다. 여기서 ***your-second-region***은 두 번째 클러스터의 위치입니다. 콘솔이 두 번째 리전에서 열립니다.
  12. (선택 사항) 두 번째 클러스터의 클러스터 설정을 선택합니다. 예를 들어 AWS KMS key를 선택할 수 있습니다. 고객 관리형 키를 사용하는 경우 키 정책이 Aurora DSQL에 필요한 권한을 부여하는지 확인합니다. 자세한 내용은 [고객 관리형 키에 대한 키 정책](#) 섹션을 참조하세요.
  13. 클러스터 생성을 선택하여 두 번째 클러스터를 생성합니다.
  14. ***initial-cluster-region***에서 피어링을 선택합니다. 여기서 ***initial-cluster-region***은 생성한 첫 번째 클러스터를 호스팅하는 리전입니다.
  15. 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. 쿼리 편집기로 연결을 선택합니다.
5. 관리자로 연결을 선택합니다.
6. [3단계: Aurora DSQL에서 샘플 SQL 명령 실행](#)의 단계에 따라 샘플 스키마와 테이블을 생성하고 데이터를 삽입합니다.

## 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. 쿼리 편집기로 연결을 선택합니다.
5. 관리자 연결을 선택합니다.
6. 두 번째 클러스터에 삽입한 데이터를 쿼리합니다.

#### Example

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

## 문제 해결

Aurora DSQL 설명서의 [문제 해결](#) 섹션을 참조하세요.

# 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 역할](#)과 연결합니다. 이 접근 방식은 [IAM의 이점](#)과 [PostgreSQL 권한](#)을 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 SDK](#)를 사용하여 인증을 제공할 수 있습니다.

### IAM 권한 부여

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

```
{
  "Effect": "Allow",
  "Action": "dsql:GetCluster",
  "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 생성](#)를 참조하세요.

## IAM 권한 부여

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

- admin 역할을 사용하는 경우 `dsql:DbConnectAdmin`을 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:DbConnect`를 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.

## Authentication

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 부여](#)를 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 privileges](#)를 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.

JSON

```
{
  "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.

JSON

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



```
    ],
    "Resource" : "*"
  }
]
}
```

## IAM 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:DbConnectAdmin`에 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 limits](#)를 참조하세요.

### 클러스터에 연결하는 사용자 지정 권한 부여 취소

admin 외의 데이터베이스 역할에 대한 액세스를 취소하려면 `dsql:DbConnect`에 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 privileges](#)를 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 Console, AWS CLI 또는 SDK를 사용하여 인증 토큰을 생성하면 토큰은 기본적으로 15분 후에 자동으로 만료됩니다. 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 DSQL](#)에서 [IAM 정책과 클러스터를 생성](#)합니다. 그런 다음 AWS Console, AWS CLI 또는 AWS SDK를 사용하여 토큰을 생성합니다.

연결하는 데 사용하는 데이터베이스 역할에 따라 최소한 [IAM을 사용하여 클러스터에 연결](#)에 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/dsql>에서 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 \(선택 사항\): 다중 리전 클러스터 생성](#)의 단계를 따릅니다.
3. 연결을 선택한 다음 토큰 가져오기를 선택합니다.
4. admin으로 연결할지, 아니면 [사용자 지정 데이터베이스 역할](#)로 연결할지를 선택합니다.
5. 생성된 인증 토큰을 복사하여 [SQL 클라이언트를 사용하여 Aurora DSQL에 액세스](#)에 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 CloudShell를 사용하여 인증 토큰을 생성하려면 먼저 [Aurora DSQL 클러스터를 생성](#)해야 합니다.

AWS CloudShell을 사용하여 인증 토큰을 생성하는 방법

1. AWS Management Console에 로그인한 다음, <https://console.aws.amazon.com/dsql>에서 Aurora DSQL 콘솔을 엽니다.
2. AWS Console 왼쪽 하단에서 AWS CloudShell을 선택합니다.
3. 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 dsq1 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 하나요?](#)를 참조하세요.

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

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

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

```
Password for user admin:
```

6. 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 하나요?](#)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 ACTIVE인 경우 `aws dsql` 명령을 사용하여 CLI에서 인증 토큰을 생성할 수 있습니다. 다음 중 한 가지 방법을 사용합니다.

### Note

토큰 생성은 현재 IAM 자격 증명을 사용하여 요청에 서명하는 로컬 작업입니다. 자격 증명을 검증하기 위해 AWS에 연결하지 않습니다. 자격 증명만료되었거나 유효하지 않은 경우에도 토큰 생성은 성공하지만 연결 시도는 실패합니다. 토큰을 생성하기 전에 IAM 자격 증명 유효한지 확인합니다.

- `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 `01abc21defg3hijklmnopqrstu.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 dsq1 generate-db-connect-admin-auth-token \  
  --region region \  
  --expires-in 3600 \  
  --hostname your_cluster_endpoint
```

## Windows

```
aws dsq1 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 01abc2ldefg3hijklmnopqrstu.dsq1.us-east-1.on.aws와 같이 이름 지정 형식은 *your\_cluster\_identifier*.dsq1.*region*.on.aws입니다.
- *region*(또는 *RegionEndpoint*) - us-east-2 또는 us-east-1과 같이 클러스터가 위치한 AWS 리전입니다.

## Python SDK

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [Python용 Aurora DSQL 커넥터](#) 사용을 권장합니다.

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.

```
import boto3

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

## C++ SDK

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 `GenerateDBConnectAdminAuthToken`을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 `GenerateDBConnectAuthToken`을 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) {
    DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQLClient client{clientConfig};
    std::string token = "";

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

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

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

## JavaScript SDK

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [Node.js용 Aurora DSQL 커넥터](#) 사용을 권장합니다.

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

- admin 역할로 연결하는 경우 `getDbConnectAdminAuthToken`을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 `getDbConnectAuthToken`을 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

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [JDBC 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#) 사용을 권장합니다.

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.dsdl.DsdlUtilities;
import software.amazon.awssdk.regions.Region;

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

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

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

## Rust SDK

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

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [Ruby 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#) 사용을 권장합니다.

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::CredentialProviderChain.new.resolve

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

```

```

}))

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

```

## PHP SDK

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

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

```

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

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

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

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

    echo $token . PHP_EOL;
    return $token;
}

```

## .NET

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [.NET 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#) 사용을 권장합니다.

### Note

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

다음과 같은 방법으로 토큰을 생성할 수 있습니다.

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

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

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

Console.WriteLine(token);
```

## Go

### Tip

AWS에서는 토큰 생성을 자동으로 처리하는 [Go 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#) 사용을 권장합니다.

Go용 AWS SDK v2는 [github.com/aws/aws-sdk-go-v2/feature/dsql/auth](https://github.com/aws/aws-sdk-go-v2/tree/main/feature/dsql/auth) 패키지에서 인증 토큰을 생성하기 위한 기본 제공 방법을 제공합니다.

- admin 역할로 연결하는 경우 `auth.GenerateDBConnectAdminAuthToken`을 사용합니다.
- 사용자 지정 데이터베이스 역할로 연결하는 경우 `auth.GenerateDbConnectAuthToken`을 사용합니다.

```

package main

import (
    "context"
    "fmt"

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

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

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

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

    fmt.Println(token)
}

```

## 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 수입할 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 증명](#)을 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 사용](#)의 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 Privileges](#)를 참조하세요.

## IAM 역할과 데이터베이스 역할의 매핑 보기

IAM 역할과 데이터베이스 역할 간의 매핑을 보려면 sys.iam\_pg\_role\_mappings 시스템 테이블을 쿼리합니다.

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

출력 예시:

| iam_oid             | arn                                    | pg_role_oid | pg_role_name |
|---------------------|----------------------------------------|-------------|--------------|
| grantor_pg_role_oid | grantor_pg_role_name                   |             |              |
| 26398               | arn:aws:iam::012345678912:role/example | 26396       | example      |
| 15579               | admin                                  |             |              |

(1 row)

이 표에는 IAM 역할(ARN으로 식별됨)과 PostgreSQL 데이터베이스 역할 간의 모든 매핑이 나와 있습니다.

## 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 애플리케이션의 성능과 확장성을 활용하면서 친숙한 PostgreSQL 도구를 사용할 수 있습니다.

## PostgreSQL 호환성 하이라이트

Aurora DSQL은 현재 PostgreSQL 버전 16을 기반으로 합니다. 주요 내용은 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.

### DDL 구문

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.

## 분산 아키텍처의 이점

Aurora DSQL의 분산된 공유 없음 설계는 기존 단일 노드 데이터베이스보다 뛰어난 성능과 확장성 이점을 제공합니다. 주요 기능은 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.

PostgreSQL에서 EXPLAIN 명령을 사용하는 방법에 대한 자세한 내용은 [Aurora DSQL의 DDL 및 분산 트랜잭션](#) 섹션을 참조하세요.

## Aurora DSQL의 SQL 기능 호환성

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](#)
- [PostgreSQL에서 Aurora DSQL로 마이그레이션](#)

## 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.

| 이름               | 에일리어스        | 범위 및 정밀도                                              | 스토리지 크기 | 인덱스 지원 |
|------------------|--------------|-------------------------------------------------------|---------|--------|
| smallint         | int2         | 2 bytes                                               | 2 bytes | 예      |
| integer          | int,<br>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      |
| double precision | float8       | 소수 자릿수 15까지의 정밀도                                      | 8 bytes | 예      |

| 이름                 | 에일리어스                                | 범위 및 정밀도                                                                            | 스토리지 크기                                | 인덱스 지원 |
|--------------------|--------------------------------------|-------------------------------------------------------------------------------------|----------------------------------------|--------|
| numeric [ (p, s) ] | decimal [ (p, s) ]<br>dec [ (p, s) ] | 정밀도를 선택할 수 있는 정확한 숫자. 최대 정밀도는 38이며 최대 범위는 37이고 <sup>1</sup> 기본값은 numeric (18,6)입니다. | 정밀도 숫자당 8 바이트 + 2바이트. 최대 크기는 27바이트입니다. | 예      |

<sup>1</sup> - 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.

## 문자 데이터 유형

Aurora DSQL은 다음 PostgreSQL 문자 데이터 유형을 지원합니다.

| 이름                        | 에일리어스           | 설명                                                                     | Aurora DSQL 제한       | 스토리지 크기            | 인덱스 지원 |
|---------------------------|-----------------|------------------------------------------------------------------------|----------------------|--------------------|--------|
| character [ (n) ]         | char [ (n) ]    | 고정 길이 문자열                                                              | 4,096B <sup>1</sup>  | 최대 4,100바이트까지 가변   | 예      |
| character varying [ (n) ] | varchar [ (n) ] | 가변 길이 문자열                                                              | 65,535B <sup>1</sup> | 최대 65,539 바이트까지 가변 | 예      |
| bpchar [ (n) ]            |                 | 고정 길이인 경우 char의 별칭입니다. 가변 길이인 경우 이는 후행 공백이 의미상 유의하지 않은 varchar의 별칭입니다. | 4,096B <sup>1</sup>  | 최대 4,100바이트까지 가변   | 예      |

| 이름   | 에일리어스 | 설명        | Aurora DSQL 제한    | 스토리지 크기      | 인덱스 지원 |
|------|-------|-----------|-------------------|--------------|--------|
| text |       | 가변 길이 문자열 | 1MiB <sup>1</sup> | 최대 1MiB까지 가변 | 예      |

<sup>1</sup> - 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.

| 이름                                 | 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      |

| 이름                             | 에일리어스    | 설명               | Range                      | 해결 방법  | 스토리지 크기 | 인덱스 지원 |
|--------------------------------|----------|------------------|----------------------------|--------|---------|--------|
| timestamp [(p)] with time zone | time: tz | 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.

| 이름      | 에일리어스 | 설명                  | Aurora DSQL 제한    | 스토리지 크기      | 인덱스 지원 |
|---------|-------|---------------------|-------------------|--------------|--------|
| boolean | bool  | 논리적 부울 (true/false) |                   | 1바이트         | 예      |
| bytea   |       | 바이너리 데이터('바이트 배열')  | 1MiB <sup>1</sup> | 최대 1MiB까지 가변 | 아니요    |
| UUID    |       | 범용 고유 식별자           |                   | 16바이트        | 예      |

<sup>1</sup> - 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.

## 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 [inet](#)을 참조하세요.

### JSON 런타임 함수

Aurora DSQL은 쿼리 처리를 위한 런타임 데이터 유형으로 JSON 및 JSONB를 지원합니다. JSON 데이터를 `text` 열로 저장할 수 있으며 쿼리 실행 시 JSON으로 캐스팅하여 PostgreSQL JSON 함수와 연산자를 사용할 수 있습니다.

Aurora DSQL은 [9.1.6 섹션 JSON 함수 및 연산자](#)에 명시된 대부분의 PostgreSQL JSON 함수를 동일한 동작으로 지원합니다.

JSON 또는 JSONB 유형을 반환하는 함수는 올바른 표시를 위해 `text`로 추가 형변환이 필요할 수 있습니다.

```
SELECT json_build_array(1, 2, 'foo', 4, 5)::text;
```

이 함수는 다음과 유사한 응답을 반환합니다.

```
json_build_array
-----
[1, 2, "foo", 4, 5]
(1 row)
```

## 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 |



| 기본 절       | 지원 절                  |
|------------|-----------------------|
| 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 <a href="#">CREATE TABLE</a> 섹션을 참조하세요.                                                                              |
| ALTER   | TABLE                   | ALTER TABLE 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">ALTER TABLE</a> 섹션을 참조하세요.                                                                                |
| DROP    | TABLE                   |                                                                                                                                                           |
| CREATE  | [UNIQUE] INDEX<br>ASYNC | 이 명령과 ON, NULLS FIRST, NULLS LAST 파라미터를 함께 사용할 수 있습니다.<br><br>CREATE INDEX ASYNC 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">Aurora DSQL의 비 동기 인덱스</a> 섹션을 참조하세요. |
| DROP    | INDEX                   |                                                                                                                                                           |

| Command | 기본 절       | 지원 절                                                                               |
|---------|------------|------------------------------------------------------------------------------------|
| CREATE  | VIEW       | CREATE VIEW 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">CREATE VIEW</a> 섹션을 참조하세요.         |
| ALTER   | VIEW       | ALTER VIEW 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">ALTER VIEW</a> 섹션을 참조하세요.           |
| DROP    | VIEW       | DROP VIEW 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">DROP VIEW</a> 섹션을 참조하세요.             |
| CREATE  | SEQUENCE   | CREATE SEQUENCE 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">CREATE SEQUENCE</a> 섹션을 참조하세요. |
| ALTER   | SEQUENCE   | ALTER SEQUENCE 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">ALTER SEQUENCE</a> 섹션을 참조하세요.   |
| DROP    | SEQUENCE   | DROP SEQUENCE 명령의 지원되는 구문에 대한 자세한 내용은 <a href="#">DROP SEQUENCE</a> 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 |

| Command | 기본 절 | 지원 절                         |
|---------|------|------------------------------|
|         |      | SELECT                       |
| UPDATE  | SET  | WHERE (SELECT)<br>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                                                                                      | 별칭  |
|-------------------|-------------------------------------------------------------------------------------------|-----|
| COMMIT            | [WORK   TRANSACTION ]<br>[AND NO CHAIN]                                                   | END |
| BEGIN             | [WORK   TRANSACTION ]<br>[ISOLATION LEVEL REPEATABLE<br>READ]<br>[READ WRITE   READ ONLY] |     |
| START TRANSACTION | [ISOLATION LEVEL REPEATABLE<br>READ]                                                      |     |

| Command  | 지원 절                                    | 별칭    |
|----------|-----------------------------------------|-------|
|          | [READ WRITE   READ ONLY]                |       |
| ROLLBACK | [WORK   TRANSACTION ]<br>[AND NO CHAIN] | ABORT |

## 유틸리티 명령

Aurora DSQL은 다음과 같은 PostgreSQL 유틸리티 명령을 지원합니다.

- EXPLAIN
- ANALYZE(관계 이름만 해당)

## Aurora DSQL에서 지원되는 SQL 명령 하위 집합

이 섹션에서는 광범위한 파라미터 세트 및 하위 명령이 있는 명령에 중점을 두고 지원되는 SQL 명령에 대한 자세한 정보를 제공합니다. 예를 들어 PostgreSQL의 CREATE TABLE은 Aurora DSQL에서 지원하는 하위 집합인 여러 절과 파라미터를 제공합니다. 이 섹션에서는 Aurora DSQL이 지원하는 익숙한 PostgreSQL 구문 요소를 사용하여 지원되는 일반적인 SQL 명령의 하위 집합을 설명합니다.

### 주제

- [CREATE TABLE](#)
- [ALTER TABLE](#)
- [CREATE SEQUENCE](#)
- [ALTER SEQUENCE](#)
- [DROP SEQUENCE](#)
- [CREATE VIEW](#)
- [ALTER VIEW](#)
- [DROP VIEW](#)

## CREATE TABLE

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

and table\_constraint is:

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

and like\_option is:

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

index\_parameters in UNIQUE, and PRIMARY KEY constraints are:

```
[ INCLUDE ( column_name [, ... ] ) ]
```

## ID 열

### Note

자격 증명 열을 사용할 때는 캐시 값을 신중하게 고려해야 합니다. 자세한 내용은 [CREATE SEQUENCE](#) 페이지의 중요 안내를 참조하세요.

워크로드 패턴을 기반으로 자격 증명 열을 가장 잘 사용하는 방법에 대한 지침은 [시퀀스 및 자격 증명 열 작업](#) 섹션을 참조하세요.

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY ( *sequence\_options* ) 절은 열을 자격 증명 열로 만듭니다. 암시적 시퀀스가 연결되고 새로 삽입된 행에서 열은 할당된 시퀀스의 값을 자동으로 갖습니다. 이러한 열은 암시적으로 NOT NULL입니다.

ALWAYS 및 BY DEFAULT 절은 INSERT 및 UPDATE 명령에서 사용자 지정 값을 명시적으로 처리하는 방법을 결정합니다.

INSERT 명령에서 ALWAYS가 선택된 경우 INSERT 문이 OVERRIDING SYSTEM VALUE를 지정하는 경우에만 사용자 지정 값이 허용됩니다. BY DEFAULT를 선택하면 사용자 지정 값이 우선합니다.

UPDATE 명령에서 ALWAYS가 선택된 경우 열을 이외의 값으로 업데이트하면 DEFAULT가 거부됩니다. BY DEFAULT를 선택하면 열을 정상적으로 업데이트할 수 있습니다. (UPDATE명령에는 OVERRIDING 절이 없습니다.)

*sequence\_options* 절을 사용하여 시퀀스의 파라미터를 재정의할 수 있습니다. 사용 가능한 옵션에는 [CREATE SEQUENCE](#)에 대해 표시된 옵션과 SEQUENCE NAME *name*이 포함됩니다. SEQUENCE NAME이 없으면 시스템에서 시퀀스에 사용되지 않는 이름을 선택합니다.

## 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
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET
sequence_option | RESTART [ [ WITH ] restart ] } [...]
```

```
ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

## 자격 증명 열 작업

### SET GENERATED { ALWAYS | BY DEFAULT } / SET *sequence\_option* / RESTART

이러한 양식은 열이 자격 증명 열인지 아니면 기존 자격 증명 열의 생성 속성을 변경하는지 여부를 변경합니다. 세부 정보는 [CREATE TABLE](#) 섹션을 참조하세요. SET DEFAULT와 마찬가지로 이러한 형식은 후속 INSERT 및 UPDATE 명령의 동작에만 영향을 미치며 테이블에 이미 있는 행은 변경되지 않습니다.

*sequence\_option*은 INCREMENT BY와 같이 [ALTER SEQUENCE](#)에서 지원하는 옵션입니다. 이러한 형식은 기존 자격 증명 열의 기반이 되는 시퀀스를 변경합니다.

### DROP IDENTITY [ IF EXISTS ]

이 양식은 열에서 자격 증명 속성을 제거합니다. DROP IDENTITY IF EXISTS가 지정되어 있고 열이 자격 증명 열이 아닌 경우 오류가 발생하지 않습니다. 이 경우 대신 알림이 발행됩니다.

## CREATE SEQUENCE

CREATE SEQUENCE - 새 시퀀스 생성기를 정의합니다.

### Important

PostgreSQL에서 CACHE 지정은 선택 사항이며 기본값은 1입니다. Amazon Aurora DSQL과 같은 분산 시스템에서 시퀀스 작업은 조정을 포함하며, 캐시 크기가 1이면 동시성이 높을 때 조정 오버헤드가 증가할 수 있습니다. 캐시 값이 크면 로컬에 미리 할당된 범위에서 시퀀스 번호를 제공할 수 있으므로, 처리량이 향상되고 사용하지 않는 예약 값이 손실되어 간격과 순서 지정 효과를 더 명확하게 확인할 수 있습니다. 애플리케이션마다 할당 순서와 처리량에 대한 민감도가 다르기 때문에 Amazon Aurora DSQL은 CACHE를 명시적으로 지정하도록 요구하며, 현재 CACHE = 1 또는 CACHE >= 65536을 지원하므로 엄격한 순차 생성에 더 가까운 할당 동작과 높은 동시성 워크로드에 최적화된 할당을 명확히 구분하도록 해 줍니다.

CACHE >= 65536인 경우 시퀀스 값은 고유하게 유지되지만 세션 간에 엄격하게 증가하는 순서로 생성되지 않을 수 있으며, 특히 캐시된 값이 완전히 사용되지 않는 경우 간격이 발생할 수 있습니다. 이러한 특성은 동시 사용 시 캐시된 시퀀스에 대한 PostgreSQL 의미 체계와 일치하며, 두 시스템 모두 고유한 값을 보장하지만 세션 간에 엄격하게 순차적인 순서 지정을 보장하지는 않습니다.

단일 클라이언트 세션 내에서 시퀀스 값은 특히 명시적 트랜잭션 외부에서 항상 엄격하게 증가하는 것처럼 보이지 않을 수 있습니다. 이 동작은 연결 풀링을 사용하는 PostgreSQL 배포와 유사합니다. 단일 세션 PostgreSQL 환경에 더 가까운 할당 동작은 `CACHE = 1`을 사용하거나 명시적 트랜잭션 내에서 시퀀스 값을 가져와 달성할 수 있습니다.

`CACHE = 1`을 사용하면 시퀀스 할당이 PostgreSQL의 캐시되지 않은 시퀀스 동작을 따릅니다.

워크로드 패턴을 기반으로 시퀀스를 가장 잘 사용하는 방법에 대한 지침은 [시퀀스 및 자격 증명 열 작업](#) 섹션을 참조하세요.

## 지원되는 구문

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

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

## 설명

`CREATE SEQUENCE`는 새 시퀀스 번호 생성기를 생성합니다. 여기에는 이름이 *name*인 새 특수 단일 행 테이블을 생성하고 초기화하는 작업이 포함됩니다. 명령을 실행하는 사용자가 생성기를 소유합니다.

스키마 이름이 제공되면 지정된 스키마에 시퀀스가 생성됩니다. 그러지 않으면 현재 스키마에서 생성됩니다. 시퀀스 이름은 동일한 스키마 내의 다른 관계(테이블, 시퀀스, 인덱스, 뷰, 구체화된 뷰 또는 외부 테이블)의 이름과 달라야 합니다.

시퀀스가 생성되면 함수 `nextval`, `currval` 및 `setval`을 사용하여 시퀀스에서 작업합니다. 이러한 함수는 [시퀀스 조작 함수](#)에 설명되어 있습니다.

시퀀스를 직접 업데이트할 수는 없지만 다음과 같은 쿼리를 사용할 수 있습니다.

```
SELECT * FROM name;
```



시퀀스의 일부 파라미터와 현재 상태를 검사합니다. 특히 시퀀스의 `last_value` 필드에는 세션에서 할당한 마지막 값이 표시됩니다. (물론 다른 세션이 적극적으로 `nextval` 직접 호출을 수행하는 경우 이 값은 출력되는 시점에 더 이상 사용되지 않을 수 있습니다.) `pg_sequences` 뷰에서 *increment* 및 *maxvalue*와 같은 다른 파라미터를 관찰할 수 있습니다.

## 파라미터

### IF NOT EXISTS

동일한 이름의 관계가 이미 있는 경우 오류를 발생시키지 않습니다. 이 경우 알림이 발행됩니다. 기존 관계가 생성되었을 시퀀스와 비슷하다는 보장은 없으며 시퀀스가 아닐 수도 있습니다.

### ##

생성할 시퀀스의 이름(선택적으로 스키마 한정자 포함)입니다.

### *data\_type*

선택적 절 AS *data\_type*은 시퀀스의 데이터 형식을 지정합니다. 유효한 값은 `bigint`이며, `bigint`가 기본값입니다. 데이터 형식에 따라 시퀀스의 기본 최솟값과 최댓값이 결정됩니다.

### *increment*

선택적 절 INCREMENT BY *increment*은 새 값을 생성하기 위해 현재 시퀀스 값에 추가할 값을 지정합니다. 양수 값은 오름차순 시퀀스를 만들고, 음수 값은 내림차순 시퀀스를 만듭니다. 기본값은 1입니다.

### *minvalue* / NO MINVALUE

선택적 절 MINVALUE *minvalue*는 시퀀스가 생성할 수 있는 최솟값을 결정합니다. 이 절이 제공되지 않거나 NO MINVALUE가 지정된 경우 기본값이 사용됩니다. 오름차순 시퀀스의 기본값은 1입니다. 내림차순 시퀀스의 기본값은 데이터 형식의 최솟값입니다.

### *maxvalue* / NO MAXVALUE

선택적 절 MAXVALUE *maxvalue*는 시퀀스의 최댓값을 결정합니다. 이 절이 제공되지 않거나 NO MAXVALUE가 지정된 경우 기본값이 사용됩니다. 오름차순 시퀀스의 기본값은 데이터 형식의 최댓값입니다. 내림차순 시퀀스의 기본값은 -1입니다.

### CYCLE / NO CYCLE

CYCLE 옵션을 사용하면 각각 오름차순 또는 내림차순 시퀀스에 의해 *maxvalue* 또는 *minvalue*에 도달했을 때 시퀀스를 래핑할 수 있습니다. 한도에 도달하면 생성되는 다음 숫자는 각각 *minvalue* 또는 *maxvalue*가 됩니다.

NO CYCLE이 지정된 경우 시퀀스가 최댓값에 도달한 후 `nextval`을 직접적으로 호출하면 오류가 반환됩니다. CYCLE 및 NO CYCLE을 둘 다 지정하지 않으면 NO CYCLE이 기본값입니다.

## ##

선택적 절 START WITH `start`를 사용하면 어디서나 시퀀스를 시작할 수 있습니다. 기본 시작 값은 오름차순 시퀀스의 경우 `minvalue`이고 내림차순 시퀀스의 경우 `maxvalue`입니다.

## cache

절 CACHE `cache`는 더 빠른 액세스를 위해 사전 할당되고 메모리에 저장될 시퀀스 번호의 수를 지정합니다. Aurora DSQL에서 CACHE에 허용되는 값은 1 또는 65,536 이상의 모든 숫자입니다. 최솟값은 1입니다(한 번에 하나의 값만 생성할 수 있으므로 캐시가 없음).

## OWNED BY `table_name.column_name` / OWNED BY NONE

OWNED BY 옵션을 사용하면 시퀀스가 특정 테이블 열과 연결되므로 해당 열(또는 전체 테이블)이 삭제되면 시퀀스도 자동으로 삭제됩니다. 지정된 테이블은 소유자가 동일해야 하며 시퀀스와 동일한 스키마에 있어야 합니다. 기본값인 OWNED BY NONE은 이러한 연결이 없음을 지정합니다.

## 참고

[DROP SEQUENCE](#)를 사용하여 시퀀스를 제거합니다.

시퀀스는 bigint 산술을 기반으로 하므로 범위는 8바이트 정수 (-9,223,372,036,854,775,808~9,223,372,036,854,775,807)의 범위를 초과할 수 없습니다.

`nextval` 및 `setval` 직접 호출은 롤백되지 않으므로 시퀀스 번호의 '간격 없는' 할당이 필요한 경우 시퀀스 객체를 사용할 수 없습니다.

각 세션은 시퀀스 객체에 대한 한 번의 액세스 중에 연속 시퀀스 값을 할당 및 캐싱하고 그에 따라 시퀀스 객체의 `last_value`를 늘립니다. 그런 다음 해당 세션 내에서 다음번에 `nextval`을 `cache`-1번 사용하면 시퀀스 객체를 터치하지 않고 미리 할당된 값을 반환하기만 합니다. 따라서 세션 내에서 할당되었지만 사용되지 않은 모든 숫자는 해당 세션이 종료될 때 손실되어 시퀀스에서 '구멍'이 됩니다.

또한 여러 세션이 고유한 시퀀스 값을 할당하도록 보장되지만 모든 세션이 고려되면 값이 시퀀스를 벗어나 생성될 수 있습니다. 예를 들어 `cache` 설정이 10인 경우 세션 A가 값 1..10을 예약하고 `nextval`=1을 반환한 다음, 세션 B는 세션 A가 `nextval`=2를 생성하기 전에 값 11..20을 예약하고 `nextval`=11을 반환할 수 있습니다. 따라서 `cache` 설정이 1이면 `nextval` 값이 순차적으로 생성된다고 가정하는 것이 안전합니다. `cache` 설정이 1보다 크면 `nextval` 값이 순차적으로만 생성되지 않고 모두 고유하다고 가정해야 합니다. 또한 `last_value`는 `nextval`에서 아직 반환했는지 여부에 관계없이 모든 세션에서 예약한 최신 값을 반영합니다.

또 다른 고려 사항은 이러한 시퀀스에서 실행된 `setval`은 캐싱된 미리 할당된 값을 모두 사용할 때까지 다른 세션에서 인식되지 않는다는 것입니다.

## 예제

101부터 시작하여 `serial`이라는 오름차순 시퀀스를 생성합니다.

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

이 시퀀스에서 다음 번호를 선택합니다.

```
SELECT nextval('serial');
```

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

이 시퀀스에서 다음 번호를 선택합니다.

```
SELECT nextval('serial');
```

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

`INSERT` 명령에서 이 시퀀스를 사용합니다.

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

`setval`을 사용하여 시퀀스를 특정 값으로 재설정합니다.

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

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

## 호환성

`CREATE SEQUENCE`는 SQL 표준을 준수합니다. 단, 다음은 예외입니다.

- 다음 값 가져오기는 표준 NEXT VALUE FOR 표현식 대신 nextval() 함수를 사용하여 수행됩니다.
- OWNED BY 절은 PostgreSQL 확장입니다.

## ALTER SEQUENCE

ALTER SEQUENCE - 시퀀스 생성기의 정의를 변경합니다.

### Important

시퀀스를 사용할 때는 캐시 값을 신중하게 고려해야 합니다. 자세한 내용은 [CREATE SEQUENCE](#) 페이지의 중요 안내를 참조하세요.

워크로드 패턴을 기반으로 시퀀스를 가장 잘 사용하는 방법에 대한 지침은 [시퀀스 및 자격 증명 열 작업](#) 섹션을 참조하세요.

### 지원되는 구문

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

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

### 설명

ALTER SEQUENCE는 기존 시퀀스 생성기의 파라미터를 변경합니다. ALTER SEQUENCE 명령에 구체적으로 설정되지 않은 파라미터는 이전 설정을 유지합니다.

ALTER SEQUENCE를 사용하려면 시퀀스를 소유해야 합니다. 시퀀스의 스키마를 변경하려면 새 스키마에 대한 CREATE 권한도 있어야 합니다. 소유자를 변경하려면 새 소유 역할에 SET ROLE을 수행할 수 있어야 하며 해당 역할에 시퀀스의 스키마에 대한 CREATE 권한이 있어야 합니다. (이러한 제한은

소유자를 변경해도 시퀀스를 삭제하고 다시 생성하여 수행할 수 없는 작업은 수행하지 않도록 강제합니다. 그러나 수퍼유저는 어떤 시퀀스의 소유권도 변경할 수 있습니다.)

파라미터

##

변경할 시퀀스의 이름(선택적으로 스키마 한정자 포함)입니다.

## IF EXISTS

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

## increment

INCREMENT BY *increment* 절은 선택 사항입니다. 양수 값은 오름차순 시퀀스를 만들고, 음수 값은 내림차순 시퀀스를 만듭니다. 지정하지 않으면 이전 증분 값이 유지됩니다.

## minvalue / NO MINVALUE

선택적 절 MINVALUE *minvalue*는 시퀀스가 생성할 수 있는 최솟값을 결정합니다. NO MINVALUE를 지정하면 오름차순 및 내림차순 시퀀스에 대해 1 및 해당 데이터 형식의 최솟값이 기본값으로 각각 사용됩니다. 두 옵션을 모두 지정하지 않으면 현재 최솟값이 유지됩니다.

## maxvalue / NO MAXVALUE

선택적 절 MAXVALUE *maxvalue*는 시퀀스의 최댓값을 결정합니다. NO MAXVALUE를 지정하면 오름차순 및 내림차순 시퀀스에 대해 해당 데이터 형식의 최댓값과 -1이 기본값으로 각각 사용됩니다. 두 옵션을 모두 지정하지 않으면 현재 최댓값이 유지됩니다.

## CYCLE

선택적 CYCLE 키워드를 사용하여 각각 오름차순 또는 내림차순 시퀀스에 의해 *maxvalue* 또는 *minvalue*에 도달했을 때 시퀀스를 래핑할 수 있습니다. 한도에 도달하면 생성되는 다음 숫자는 각각 *minvalue* 또는 *maxvalue*가 됩니다.

## NO CYCLE

선택적 NO CYCLE 키워드가 지정된 경우 시퀀스가 최댓값에 도달한 후 *nextval*을 직접적으로 호출하면 오류가 반환됩니다. CYCLE 및 NO CYCLE를 모두 지정하지 않으면 이전 주기 동작이 유지됩니다.

##

선택적 절 START WITH *start*는 시퀀스의 기록된 시작 값을 변경합니다. 이는 현재 시퀀스 값에 영향을 주지 않으며, 향후 ALTER SEQUENCE RESTART 명령에서 사용할 값을 설정하기만 합니다.

**###**

선택적 절 RESTART [ WITH *restart* ]는 시퀀스의 현재 값을 변경합니다. 이는 is\_called = false로 setval 함수를 직접적으로 호출하는 것과 유사합니다. 지정된 값은 nextval의 다음 직접 호출에 의해 반환됩니다. *restart* 값 없이 RESTART를 쓰는 것은 CREATE SEQUENCE에서 기록했거나 ALTER SEQUENCE START WITH에서 마지막으로 설정한 시작 값을 제공하는 것과 같습니다.

setval 직접 호출과 달리 시퀀스에 대한 RESTART 작업은 트랜잭션이며 동시 트랜잭션이 동일한 시퀀스에서 번호를 얻는 것을 차단합니다. 원하는 작업 모드가 아닌 경우 setval을 사용해야 합니다.

**cache**

절 CACHE *cache*를 사용하면 더 빠른 액세스를 위해 시퀀스 번호를 미리 할당하고 메모리에 저장할 수 있습니다. 값은 1이거나 65,536 이상이어야 합니다. 지정하지 않으면 이전 캐시 값이 유지됩니다. 캐시 동작에 대한 자세한 내용은 [CREATE SEQUENCE](#) 아래의 지침을 참조하세요.

**OWNED BY *table\_name.column\_name* / OWNED BY NONE**

OWNED BY 옵션을 사용하면 시퀀스가 특정 테이블 열과 연결되므로 해당 열(또는 전체 테이블)이 삭제되면 시퀀스도 자동으로 삭제됩니다. 지정된 경우 이 연결은 시퀀스에 대해 이전에 지정된 연결을 대체합니다. 지정된 테이블은 소유자가 동일해야 하며 시퀀스와 동일한 스키마에 있어야 합니다. OWNED BY NONE를 지정하면 기존 연결이 제거되어 시퀀스가 '독립적으로 존재하는 상태'가 됩니다.

***new\_owner***

새 시퀀스 소유자의 사용자 이름입니다.

***new\_name***

시퀀스의 새 이름입니다.

***new\_schema***

시퀀스의 새 스키마입니다.

**참고**

ALTER SEQUENCE는 사전 할당된(캐싱된) 시퀀스 값이 있는 현재 백엔드를 제외하면 백엔드의 nextval 결과에 즉시 영향을 미치지 않습니다. 변경된 시퀀스 생성 파라미터를 확인하기 전에 캐시된 모든 값을 사용합니다. 현재 백엔드는 즉시 영향을 받습니다.

ALTER SEQUENCE는 시퀀스의 currval 상태에 영향을 주지 않습니다.

ALTER SEQUENCE는 다른 트랜잭션을 OCC로 만들 수 있습니다.

기록상 이유로 시퀀스에도 ALTER TABLE을 사용할 수 있지만, 시퀀스에서 허용되는 ALTER TABLE의 유일한 변형은 위에 표시된 형식과 동일합니다.

### 예제

105에서 serial이라는 시퀀스를 다시 시작합니다.

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

### 호환성

ALTER SEQUENCE는 PostgreSQL 확장인 AS, START WITH, OWNED BY, OWNER TO, RENAME TO, SET SCHEMA 절을 제외하고 SQL 표준을 준수합니다.

## DROP SEQUENCE

DROP SEQUENCE - 시퀀스를 제거합니다.

### 지원되는 구문

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

### 설명

DROP SEQUENCE는 시퀀스 번호 생성기를 제거합니다. 시퀀스는 소유자 또는 수퍼유저만 삭제할 수 있습니다.

### 파라미터

#### IF EXISTS

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

**##**

시퀀스의 이름(선택적으로 스키마 한정자 포함)입니다.

#### CASCADE

확장에 종속된 객체와 해당 객체에 종속된 모든 객체가 자동으로 삭제됩니다.

## RESTRICT

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

### 예제

seq 시퀀스 제거:

```
DROP SEQUENCE seq;
```

### 호환성

DROP SEQUENCE는 SQL 표준을 준수합니다. 그러나 표준에서는 명령당 하나의 시퀀스만 삭제할 수 있으며 PostgreSQL 확장 기능인 IF EXISTS 옵션이 제외되어 있다는 점이 다릅니다.

## 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 `cascaded`일 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.

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.

## 참고

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.
- 뷰를 생성하거나 교체하는 사용자는 스키마에서 참조된 객체를 조회하기 위해 뷰 쿼리에서 참조된 스키마에 대한 USAGE 권한이 있어야 합니다.
- 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.

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.

니다. 그러나 뷰에 `security_invoker`가 `true`로 설정된 경우 뷰 소유자가 아닌 업데이트를 수행하는 사용자에게 기본 관계에 대한 관련 권한이 있어야 합니다.

## 예제

모든 코미디 영화로 구성된 뷰를 생성합니다.

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

LOCAL CHECK OPTION을 포함하여 뷰를 생성합니다.

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

재귀 뷰를 생성합니다.

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

## 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 사용합니다.) 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.

## 파라미터

### name

기존 뷰의 이름(선택적으로 스키마 지정)입니다.

### column\_name

기존 열의 이름 또는 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.

### new\_owner

뷰의 새 소유자의 사용자 이름입니다.

### new\_name

새 뷰의 이름입니다.

### new\_schema

뷰의 새 스키마입니다.

**SET ( view\_option\_name [= view\_option\_value] [, ... ] )**

뷰 옵션을 설정합니다. 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.
- `security_invoker` (boolean) - 뷰의 `security-invoker` 속성을 변경합니다.

**RESET ( view\_option\_name [, ... ] )**

뷰 옵션을 기본값으로 재설정합니다.

**예제**

foo 뷰의 이름을 bar로 변경:

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

업데이트 가능한 뷰에 기본 열 값을 연결:

```
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 VIEW는 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.

## Parameters

### 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.

## 예제

```
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.

## PostgreSQL에서 Aurora DSQL로 마이그레이션

Aurora DSQL은 [PostgreSQL과 호환](#)되도록 설계되었으며 ACID 트랜잭션, 보조 인덱스, 조인, 표준 DML 작업과 같은 핵심 관계형 기능을 지원합니다. 대부분의 기존 PostgreSQL 애플리케이션은 최소한의 변경으로 Aurora DSQL로 마이그레이션할 수 있습니다.

이 섹션에서는 프레임워크 호환성, 마이그레이션 패턴, 아키텍처 고려 사항을 포함하여 애플리케이션을 Aurora DSQL로 마이그레이션하기 위한 실용적인 지침을 제공합니다.

### 프레임워크 및 ORM 호환성

Aurora DSQL은 표준 PostgreSQL 유선 프로토콜을 사용하므로 PostgreSQL 드라이버 및 프레임워크와 호환됩니다. 대부분의 인기 ORM은 변경 없이 또는 최소한의 변경만으로 Aurora DSQL과 호환됩니다.

다. 참조 구현 및 사용 가능한 ORM 통합은 [the section called “Aurora DSQL 어댑터”](#) 섹션을 참조하세요.

## 일반적인 마이그레이션 패턴

PostgreSQL에서 Aurora DSQL로 마이그레이션할 때 일부 기능은 다르게 작동하거나 대체 구문이 있습니다. 이 섹션에서는 일반적인 마이그레이션 시나리오에 대한 지침을 제공합니다.

### DDL 작업 대안

Aurora DSQL은 기존 PostgreSQL DDL 작업에 대한 최신 대안을 제공합니다.

#### 인덱스 생성

비차단 인덱스를 생성하는 데 `CREATE INDEX` 대신 `CREATE INDEX ASYNC`를 사용합니다.

이점: 큰 테이블에서 가동 중지 시간 없이 인덱스를 생성할 수 있습니다.

#### 데이터 제거

`TRUNCATE` 대신 `DELETE FROM table_name`를 사용합니다.

대안: 테이블을 완전히 다시 생성하려면 `DROP TABLE`을 사용한 후 `CREATE TABLE`을 사용합니다.

#### 시스템 구성

Aurora DSQL은 완전 관리형이므로 구성이 워크로드 패턴에 따라 자동으로 처리됩니다. AWS Management Console 또는 API를 사용하여 클러스터 설정을 관리합니다.

이점: 데이터베이스 튜닝이나 파라미터 관리가 필요하지 않습니다.

### 스키마 설계 패턴

Aurora DSQL과의 호환성을 위해 다음과 같은 일반적인 PostgreSQL 패턴을 조정합니다.

#### 참조 무결성 패턴

Aurora DSQL은 테이블 관계 및 `JOIN` 작업을 지원합니다. 참조 무결성을 위해 애플리케이션 계층에서 검증을 구현합니다. 이러한 설계는 애플리케이션 계층 검증을 통해 유연성을 높이고 연쇄적 작업으로 인한 성능 병목 현상을 방지하는 최신 분산 데이터베이스 패턴과 일맥상통합니다.

패턴: 일관된 이름 지정 규칙, 검증 로직 및 트랜잭션 경계를 사용하여 애플리케이션 계층에서 참조 무결성 검사를 구현하세요. 많은 대규모 애플리케이션은 오류 처리 및 성능을 더 잘 제어하기 위해 이 접근 방식을 선호합니다.



## 임시 데이터 처리

임시 테이블 대신 CTE, 하위 쿼리 또는 정리 로직이 포함된 일반 테이블을 사용합니다.

대안: 세션별 이름을 가진 테이블을 생성하고 애플리케이션 내에서 정리하세요.

## 아키텍처 차이점 이해

Aurora DSQL의 분산 서버리스 아키텍처는 여러 면에서 기존 PostgreSQL과 의도적으로 다릅니다. 이러한 차이점을 통해 Aurora DSQL은 단순성과 확장성이라는 핵심적인 이점을 제공합니다.

### 간소화된 데이터베이스 모델

#### 클러스터당 단일 데이터베이스

Aurora DSQL은 클러스터마다 postgres라는 내장 데이터베이스 하나를 제공합니다.

마이그레이션 팁: 애플리케이션에서 여러 데이터베이스를 사용하는 경우 논리적 분리를 위해 별도의 Aurora DSQL 클러스터를 생성하거나 단일 클러스터 내에서 스키마를 사용하세요.

#### 임시 테이블 미지원

임시 데이터 처리를 위해서는 복잡한 쿼리에 대한 유연한 대안을 제공하는 공통 테이블 표현(CTE) 및 하위 쿼리를 사용해야 합니다.

대안: 임시 결과 집합의 경우 WITH 절과 함께 CTE를 사용하고, 세션별 데이터의 경우 고유한 이름이 지정된 일반 테이블을 사용할 수 있습니다.

#### 자동 스토리지 관리

Aurora DSQL을 사용하면 테이블스페이스 및 수동 스토리지 관리가 필요 없습니다. 스토리지는 데이터 패턴에 따라 자동으로 규모가 조정되고 최적화됩니다.

이점: 디스크 공간을 모니터링하거나 스토리지 할당을 계획하거나 테이블스페이스 구성을 관리할 필요가 없습니다.

#### 최신 애플리케이션 패턴

Aurora DSQL은 유지 관리성과 성능을 향상시키는 최신 애플리케이션 개발 패턴을 장려합니다.

#### 데이터베이스 트리거 대신 애플리케이션 수준 로직

트리거와 같은 기능을 사용하려면 애플리케이션 계층에서 이벤트 기반 로직을 구현합니다.

마이그레이션 전략: 트리거 로직을 애플리케이션 코드로 이동하거나, EventBridge와 같은 AWS 서비스와 함께 이벤트 기반 아키텍처를 사용하거나, 애플리케이션 로깅을 통해 감사 추적을 구현하세요.

## 데이터 처리를 위한 SQL 함수

Aurora DSQL은 SQL 기반 함수를 지원하지만 PL/pgSQL과 같은 절차적 언어는 지원하지 않습니다.

대안: 데이터 변환에 SQL 함수를 사용하거나, 복잡한 로직을 애플리케이션 계층 또는 AWS Lambda 함수로 이동하세요.

## 수동적 잠금 대신 낙관적 동시성 제어

Aurora DSQL은 기존 데이터베이스 잠금 메커니즘과 다른, 잠금 없는 방식인 낙관적 동시성 제어(OCC)를 사용합니다. Aurora DSQL은 다른 트랜잭션을 차단하는 잠금을 획득하는 대신 차단 없이 트랜잭션이 진행되도록 하고 커밋 시 충돌을 감지합니다. 이를 통해 교착 상태를 방지하고 느린 트랜잭션이 다른 작업을 차단하는 것을 막습니다.

주요 차이점: Aurora DSQL은 충돌이 발생할 경우 트랜잭션이 잠금을 기다리도록 하는 대신 직렬화 오류를 반환합니다. 이를 위해 애플리케이션이 기존 데이터베이스에서 잠금 제한 시간을 처리하는 것과 유사하게 재시도 로직을 구현해야 하지만, 충돌이 즉시 해결되어 차단 대기를 유발하지 않습니다.

설계 패턴: 재시도 메커니즘을 갖춘 멍등성 트랜잭션 로직을 구현하세요. 무작위 프라이머리 키를 사용하고 키 범위 전체에 걸쳐 업데이트를 분산시켜 경합을 최소화하도록 스키마를 설계합니다. 자세한 내용은 [Aurora DSQL의 동시성 제어](#)를 참조하세요.

## 관계 및 참조 무결성

Aurora DSQL은 JOIN 연산을 포함한 테이블 간의 외래 키 관계를 지원합니다. 참조 무결성을 위해 애플리케이션 계층에서 검증을 구현합니다. 참조 무결성을 적용하는 것은 중요할 수 있지만 연쇄적 작업(예: 연쇄 삭제)은 예기치 않은 성능 문제를 일으킬 수 있습니다. 예를 들어, 1,000개의 품목이 있는 주문을 삭제하는 경우 1,001개의 행을 처리하는 트랜잭션이 실행됩니다. 이러한 이유로 많은 고객이 외래 키 제약 조건을 피합니다.

설계 패턴: 애플리케이션 계층에서 참조 무결성 검사를 구현하거나, 최종 일관성 패턴을 사용하거나, AWS 서비스를 활용하여 데이터를 검증하세요.

## 운영 간소화

Aurora DSQL을 사용하면 기존 데이터베이스 유지 관리 작업이 대부분 필요 없게되므로 운영 오버헤드가 줄어듭니다.

## 수동 유지 관리 불필요

Aurora DSQL은 스토리지 최적화, 통계 수집 및 성능 튜닝을 자동으로 관리합니다. VACUUM과 같은 기존 유지 관리 명령은 시스템에서 처리합니다.

이점: 데이터베이스 유지 관리 기간, vacuum 예약 및 시스템 파라미터 튜닝이 필요 없어집니다.

## 자동 파티셔닝 및 규모 조정

Aurora DSQL은 액세스 패턴에 따라 데이터를 자동으로 파티셔닝하고 배포합니다. 최적의 배포를 위해 UUID 또는 애플리케이션 생성 ID를 사용합니다.

마이그레이션 팁: 수동 파티셔닝 로직을 제거하고 Aurora DSQL이 데이터 배포를 처리하도록 하세요. 최적의 배포를 위해 UUID 또는 애플리케이션 생성 ID를 사용합니다. 애플리케이션에 순차적 식별자가 필요한 경우 [시퀀스 및 자격 증명 열](#) 섹션을 참조하세요.

## AI 도구를 사용한 에이전틱 마이그레이션

AI 코딩 에이전트는 기본 제공 안전 검사를 통해 스키마를 분석하고, 코드를 변환하고, DDL 마이그레이션을 실행하여 Aurora DSQL로의 마이그레이션을 가속화할 수 있습니다.

### 마이그레이션에 Kiro 사용

[Kiro](#)와 같은 코딩 에이전트는 PostgreSQL 코드를 분석하고 Aurora DSQL로 마이그레이션하는 데 도움이 될 수 있습니다.

- 스키마 분석: 기존 스키마 파일을 업로드하고 Kiro에게 잠재적 호환성 문제를 식별하고 대안을 제안하도록 요청하세요.
- 코드 변환: 애플리케이션 코드를 제공하고 Kiro에게 트리거 로직을 리팩터링하거나, 시퀀스를 UUID로 대체하거나, 트랜잭션 패턴을 수정하도록 요청하세요.
- 마이그레이션 계획: Kiro에게 특정 애플리케이션 아키텍처를 기반으로 단계별 마이그레이션 계획을 생성하도록 요청하세요.
- DDL 마이그레이션: 기본 제공 안전 검사 및 사용자 확인과 함께 테이블 다시 만들기 패턴을 사용하여 스키마 수정 실행

**프롬프트 예제:**

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

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

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

"Drop the legacy\_status column from the orders table"

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

**테이블 다시 만들기를 사용한 DDL 마이그레이션**

Aurora DSQL MCP 서버에서 AI 에이전트를 사용하는 경우, 특정 ALTER TABLE 작업은 데이터를 안전하게 마이그레이션하는 테이블 다시 만들기 패턴을 사용합니다. 에이전트는 각 단계에서 정보를 계속 제공하면서 복잡성을 처리합니다.

다음 작업은 테이블 다시 만들기 패턴을 사용합니다.

| 연산                             | 접근 방식                              |
|--------------------------------|------------------------------------|
| DROP COLUMN                    | 새 테이블에서 열 제외                       |
| ALTER COLUMN TYPE              | 마이그레이션 중 데이터 유형 캐스팅                |
| ALTER COLUMN SET/DROP NOT NULL | 새 테이블 정의의 제약 조건 변경                 |
| ALTER COLUMN SET/DROP DEFAULT  | 새 테이블 정의에서 기본값 정의                  |
| ADD/DROP CONSTRAINT            | 새 테이블에 제약 포함 또는 제거                 |
| MODIFY PRIMARY KEY             | 고유성 검증을 사용하여 새 PK 정의               |
| 열 분할/병합                        | SPLIT_PART, SUBSTRING 또는 CONCAT 사용 |

다음 ALTER TABLE 작업은 테이블 다시 만들기 없이 직접 지원됩니다.

- ALTER TABLE ... RENAME COLUMN – 열 이름 바꾸기
- ALTER TABLE ... RENAME TO – 테이블 이름 바꾸기
- ALTER TABLE ... ADD COLUMN – 새 열 추가

안전 기능: DDL 마이그레이션을 실행할 때 AI 에이전트는 마이그레이션 계획을 제시하고, 데이터 호환성을 확인하고, 행 수를 확인하고, DROP TABLE과 같은 파괴적인 작업을 수행하기 전에 명시적 승인을 요청합니다.

배치 마이그레이션: 행이 3,000개를 초과하는 테이블의 경우 에이전트는 트랜잭션 한도 내에서 유지되도록 500~1,000개의 행 증분으로 마이그레이션을 자동으로 배치화합니다.

### Aurora DSQL MCP 서버

Aurora DSQL Model Context Protocol(MCP) 서버를 사용하면 AI 어시스턴트가 Aurora DSQL 클러스터에 직접 연결하고 Aurora DSQL 설명서를 검색할 수 있습니다. 이를 통해 AI는 다음을 수행할 수 있습니다.

- 기존 스키마 분석 및 마이그레이션 변경 사항 제안
- 테이블 다시 만들기 패턴을 사용하여 DDL 마이그레이션 실행
- 마이그레이션 중 쿼리 테스트 및 호환성 확인
- 최신 Aurora DSQL 설명서를 기반으로 정확한 최신 지침 제공

Aurora DSQL MCP 서버를 AI 어시스턴트와 함께 사용하려면 [Aurora DSQL MCP 서버](#)의 설정 지침을 참조하세요.

### PostgreSQL 호환성을 위한 Aurora DSQL 고려 사항

Aurora DSQL은 분산 아키텍처, 서버리스 운영 및 자동 규모 조정을 가능하게 하는 기능 지원 측면에서 자체 관리형 PostgreSQL과 차이가 있습니다. 대부분의 애플리케이션은 이러한 차이점에도 불구하고 수정 없이 작동합니다.

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 클러스터당 postgres라는 단일 내장 데이터베이스를 사용합니다. 논리적 분리를 위해 별도의 Aurora DSQL 클러스터를 만들거나 단일 클러스터 내에서 스키마를 사용합니다.
- postgres 데이터베이스는 광범위한 국제 문자 지원을 제공하는 UTF-8 문자 인코딩을 사용합니다.

- 데이터베이스는 C 데이터 정렬만 사용합니다.
- Aurora DSQL은 UTC를 시스템 시간대로 사용합니다. Postgres는 모든 시간대 인식 날짜 및 시간을 UTC로 내부적으로 저장합니다. TimeZone 구성 파라미터를 설정하여 클라이언트에 표시되는 방식을 변환하고 서버가 UTC로 내부적으로 변환하는 데 사용할 클라이언트 입력의 기본값으로 사용할 수 있습니다.
- 트랜잭션 격리 수준은 PostgreSQL Repeatable Read에서 고정됩니다.
- 트랜잭션의 제약 조건은 다음과 같습니다.
  - DDL 및 DML 작업에는 별도의 트랜잭션이 필요합니다.
  - 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 DSQL은 스키마 수준 권한 부여를 통해 권한을 관리합니다. 관리자 사용자는 CREATE SCHEMA를 사용하여 스키마를 만들고 GRANT USAGE ON SCHEMA를 사용하여 액세스 권한을 부여합니다. 관리자 사용자는 퍼블릭 스키마에서 객체를 관리하는 반면, 관리자가 아닌 사용자는 명확한 소유권 경계를 위해 사용자가 만든 스키마에서 객체를 만듭니다. 자세한 내용은 [데이터베이스 역할이 데이터베이스에서 SQL을 사용하도록 권한 부여](#) 섹션을 참조하세요.

마이그레이션에 도움이 필요하세요?

마이그레이션에 중요하지만 현재 Aurora DSQL에서 지원되지 않는 기능이 있는 경우 [Amazon Aurora DSQL에 대한 피드백 제공](#)에서 AWS와 피드백을 공유하는 방법에 대한 정보를 참조하세요.

## 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 호환성](#)을 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 멍등적이므로 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 임의의 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 mytable에 행을 삽입하려고 시도합니다.

Aurora DSQL에서 SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (0C001). 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 F00 VALUES (1);
COMMIT;
```

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

```
BEGIN;
  CREATE TABLE F00 (ID_col integer);
  INSERT into F00 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 관리자 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 정의



하면 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 유형](#)을 참조하세요.

- 프라이머리 키 또는 보조 인덱스에 최대 8개의 열을 포함할 수 있습니다. 자세한 내용은 [Aurora DSQL의 데이터베이스 한도](#) 및 [Aurora DSQL에서 지원되는 데이터 유형](#)을 참조하세요.

## 시퀀스 및 자격 증명 열

시퀀스 및 자격 증명 열은 정수 값을 생성하며 간단하거나 사람이 읽을 수 있는 식별자가 필요할 때 유용합니다. 이러한 값에는 [CREATE SEQUENCE](#) 설명서에 설명된 할당 및 캐싱 동작이 포함됩니다.

주제

- [시퀀스 조작 함수](#)
- [ID 열](#)
- [시퀀스 및 자격 증명 열 작업](#)

## 시퀀스 조작 함수

이 섹션에서는 시퀀스 객체(시퀀스 생성기 또는 단순히 시퀀스라고도 함)를 조작하기 위한 함수에 대해 설명합니다. 시퀀스 객체는 [CREATE SEQUENCE](#)로 만들어진 특수 단일 행 테이블입니다. 시퀀스 객체는 일반적으로 테이블 행의 고유 식별자를 생성하는 데 사용됩니다. 시퀀스 함수는 시퀀스 객체에서 연속 시퀀스 값을 얻기 위한 간단한 다중 사용자 환경에서 안전한 방법을 제공합니다.

### Important

시퀀스를 사용할 때는 캐시 값을 신중하게 고려해야 합니다. 자세한 내용은 [CREATE SEQUENCE](#) 페이지의 중요 안내를 참조하세요.

워크로드 패턴을 기반으로 시퀀스를 가장 잘 사용하는 방법에 대한 지침은 [시퀀스 및 자격 증명 열 작업](#) 섹션을 참조하세요.

| 함수                                             | 설명                                                                                                                                                                                                                                             |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>nextval ( regclass ) #<br/>bigint</code> | 시퀀스 객체를 다음 값으로 진행시키고 해당 값을 반환합니다. 이 작업은 원자적으로 수행됩니다. 여러 <code>nextval</code> 세션이 동시에 실행되더라도 각 세션은 안전하게 서로 다른 시퀀스 값을 수신합니다. 시퀀스 객체가 기본 파라미터로 만들어진 경우 연속 <code>nextval</code> 직접 호출은 1로 시작하는 증가하는 값을 반환합니다. <a href="#">CREATE SEQUENCE</a> 명령에서 |

| 함수                                                           | 설명                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>setval ( regclass, bigint [, boolean ] ) # bigint</pre> | <p>적절한 파라미터를 사용하여 다른 동작을 가져올 수 있습니다. 이 함수에는 시퀀스에 대한 USAGE 또는 UPDATE 권한이 필요합니다.</p> <p>시퀀스 객체의 현재 값과 선택적으로 <code>is_called</code> 플래그를 설정합니다. 2-파라미터 형식은 시퀀스의 <code>last_value</code> 필드를 지정된 값으로 설정하고 해당 <code>is_called</code> 필드를 <code>true</code>로 설정합니다. 즉, 다음 <code>nextval</code>은 값을 반환하기 전에 시퀀스를 진행합니다. <code>currval</code>에서 보고할 값도 지정된 값으로 설정됩니다. 3-파라미터 양식에서는 <code>is_called</code> 를 <code>true</code> 또는 <code>false</code>로 설정할 수 있습니다. <code>true</code>는 2-파라미터 양식과 동일한 효과를 갖습니다. <code>false</code>로 설정하면 다음 <code>nextval</code>은 지정된 값을 정확히 반환하고 시퀀스 진행은 다음 <code>nextval</code>로 시작됩니다. 또한 <code>currval</code>에서 보고하는 값은 여기에서 변경되지 않습니다. 예제:</p> <pre>SELECT setval('myseq', 42);           -- Next nextval will return 43 SELECT setval('myseq', 42, true);     -- Same as above SELECT setval('myseq', 42, false);    -- Next nextval will return 42</pre> <p><code>setval</code>에서 반환되는 결과는 두 번째 인수의 값일 뿐입니다. 이 함수에는 시퀀스에 대한 UPDATE 권한이 필요합니다.</p> |
| <pre>currval ( regclass ) # bigint</pre>                     | <p>현재 세션에서 이 시퀀스에 대해 <code>nextval</code>에서 가장 최근에 얻은 값을 반환합니다. (이 세션에서 이 시퀀스에 대해 <code>nextval</code>이 직접적으로 호출된 적이 없는 경우 오류가 보고됩니다.) 세션-로컬 값을 반환하기 때문에 현재 세션으로 인해 다른 세션이 <code>nextval</code>을 실행했는지 여부에 대한 예측 가능한 답변을 제공합니다. 이 함수에는 시퀀스에 대한 USAGE 또는 SELECT 권한이 필요합니다.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

| 함수                               | 설명                                                                                                                                                                                                                                                                                                                               |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>lastval () # bigint</code> | 현재 트랜잭션에서 <code>nextval</code> 이 가장 최근에 반환한 값을 반환합니다. 이 함수는 시퀀스 이름을 인수로 취하는 대신 현재 트랜잭션에서 <code>nextval</code> 이 가장 최근에 적용된 시퀀스를 참조한다는 점을 제외하면 <code>currval</code> 과 동일합니다. <code>nextval</code> 이 현재 트랜잭션에서 아직 직접적으로 호출되지 않은 경우 <code>lastval</code> 을 직접적으로 호출하는 것은 오류입니다. 이 함수에는 마지막으로 사용된 시퀀스에 대한 USAGE 또는 SELECT 권한이 필요합니다. |

### Warning

직접 호출 트랜잭션이 나중에 중단되는 경우 `nextval`에서 얻은 값은 재사용을 위해 회수되지 않습니다. 즉, 트랜잭션 중단 또는 데이터베이스 충돌로 인해 할당된 값 시퀀스에 공백이 발생할 수 있습니다. 이는 트랜잭션 중단 없이도 발생할 수 있습니다. 예를 들어 ON CONFLICT 절이 있는 INSERT는 ON CONFLICT 규칙을 대신 따르게 하는 충돌을 감지하기 전에 필요한 `nextval` 직접 호출 수행을 포함하여 삽입할 튜플을 계산합니다. 따라서 Aurora DSQL의 시퀀스 객체는 '간격 없는' 시퀀스를 얻는 데 사용할 수 없습니다.

마찬가지로 `setval`에서 수행한 시퀀스 상태 변경은 다른 트랜잭션에 즉시 표시되며 호출 트랜잭션이 롤백되는 경우 실행 취소되지 않습니다.

시퀀스 함수에 의해 작동할 시퀀스는 `regclass` 인수에 의해 지정됩니다. 인수는 단순히 `pg_class` 시스템 카탈로그에 있는 시퀀스의 OID입니다. 그러나 `regclass` 데이터 형식의 입력 변환기가 작업을 대신 수행하므로 OID를 직접 조회할 필요는 없습니다. 자세한 내용은 [객체 식별자 유형](#)에 대한 PostgreSQL 설명서를 참조하세요.

## ID 열

### Important

자격 증명 열을 사용할 때는 캐시 값을 신중하게 고려해야 합니다. 자세한 내용은 [CREATE SEQUENCE](#) 페이지의 중요 안내를 참조하세요.

워크로드 패턴을 기반으로 자격 증명 열을 가장 잘 사용하는 방법에 대한 지침은 [시퀀스 및 자격 증명 열 작업](#) 섹션을 참조하세요.

자격 증명 열은 암시적 시퀀스에서 자동으로 생성되는 특수 열입니다. 키 값을 생성하는 데 사용할 수 있습니다. 자격 증명 열을 만들려면 [CREATE TABLE](#)의 GENERATED ... AS IDENTITY 절을 사용합니다. 예를 들면 다음과 같습니다.

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

또는 다음을 사용합니다.

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

자세한 내용은 [CREATE TABLE](#) 섹션을 참조하세요.

자격 증명 열이 있는 테이블에서 INSERT 명령이 실행되고 자격 증명 열에 명시적으로 지정된 값이 없는 경우 암시적 시퀀스에서 생성된 값이 삽입됩니다. 예를 들어 앞의 정의를 사용하고 적절한 열을 추가로 가정하여 다음을 작성합니다.

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

1부터 id 열 값을 생성하여 다음 테이블 데이터를 생성합니다.

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

또는 값 대신 DEFAULT 키워드를 지정하여 시퀀스 생성 값을 명시적으로 요청할 수 있습니다.

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

마찬가지로 DEFAULT 키워드는 UPDATE 명령에 사용할 수 있습니다.

따라서 여러 가지 면에서 자격 증명 열은 기본값이 있는 열처럼 동작합니다.

열 정의의 ALWAYS 및 BY DEFAULT 절은 INSERT 및 UPDATE 명령에서 사용자 지정 값을 명시적으로 처리하는 방법을 결정합니다. INSERT 명령에서 ALWAYS가 선택된 경우 INSERT 문이 OVERRIDING SYSTEM VALUE를 지정하는 경우에만 사용자 지정 값이 허용됩니다. BY DEFAULT를 선택하면 사용자 지정 값이 우선합니다. 따라서 BY DEFAULT를 사용하면 기본값과 더 유사한 동작이 발생하며, 기본 값은 명시적 값으로 재정의될 수 있는 반면 ALWAYS는 실수로 명시적 값을 삽입하는 것에 대해 더 많은 보호를 제공합니다.

자격 증명 열의 데이터 형식은 시퀀스에서 지원하는 데이터 형식 중 하나여야 합니다. ([CREATE SEQUENCE](#)를 참조하세요.) 자격 증명 열을 만들 때 연결된 시퀀스의 속성을 지정하거나([CREATE TABLE](#) 참조) 나중에 변경할 수 있습니다([ALTER TABLE](#) 참조).

자격 증명 열은 자동으로 NOT NULL로 표시됩니다. 그러나 자격 증명 열은 고유성을 보장하지 않습니다. (시퀀스는 일반적으로 고유한 값을 반환하지만 앞서 설명한 대로 시퀀스를 재설정하거나 값을 자격 증명 열에 수동으로 삽입할 수 있습니다.) PRIMARY KEY 또는 UNIQUE 제약 조건을 사용하여 고유성을 적용해야 합니다.

## 시퀀스 및 자격 증명 열 작업

이 섹션에서는 워크로드 패턴을 기반으로 시퀀스 및 자격 증명 열을 가장 잘 사용하는 방법을 이해하는데 도움이 됩니다.

### Important

할당 및 캐싱 동작에 대한 자세한 내용은 [CREATE SEQUENCE](#) 페이지의 중요 안내를 참조하세요.

## 식별자 유형 선택

Amazon Aurora DSQL은 시퀀스 또는 자격 증명 열을 사용하여 생성된 UUID 기반 식별자와 정수 값을 모두 지원합니다. 이러한 옵션은 값이 할당되는 방식과 로드 시 조정되는 방식에 따라 다릅니다.

UUID 값은 조정 없이 만들 수 있으며 식별자가 자주 또는 여러 세션에서 만들어지는 워크로드에 적합합니다. Amazon Aurora DSQL은 분산 작업을 위해 설계되었으므로 조정을 피하는 것이 유용한 경우가 많습니다. 따라서 UUID 기본 식별자 유형으로 권장되며, 특히 확장성이 중요하고 식별자의 엄격한 순서 지정이 필요하지 않은 워크로드의 프라이머리 키에 권장됩니다.

시퀀스 및 자격 증명 열은 사람이 읽을 수 있는 식별자, 보고 및 외부 인터페이스에 편리한 작은 정수 값을 생성합니다. 사용성 또는 통합 이유로 숫자 식별자를 선호하는 경우, UUID 기반 식별자와 함께 시퀀

스 또는 자격 증명 열을 사용하는 것이 좋습니다. 정수 시퀀스 또는 자격 증명 값이 필요한 경우, 적절한 캐시 크기를 선택하는 것이 워크로드 설계의 중요한 부분이 됩니다. 캐시 크기 선택에 대한 지침은 다음 섹션을 참조하세요.

## 캐시 크기 선택

적절한 캐시 값을 선택하는 것은 시퀀스 및 자격 증명 열을 효과적으로 사용하는 데 중요한 부분입니다. 캐시 설정은 로드 시 식별자 할당이 작동하는 방식을 결정하여 시스템 처리량과 값이 할당 순서를 반영하는 정도에 영향을 미칩니다.

**CACHE >= 65536**의 캐시 크기가 클수록 다음과 같은 경우에 적합합니다.

- 식별자가 높은 빈도로 생성됨
- 여러 세션이 동시에 삽입됨
- 워크로드가 간격과 눈에 보이는 순서 지정 효과를 허용할 수 있음

예를 들어 대용량 이벤트 수집 워크로드(예: IoT 또는 원격 측정)와 작업 실행 ID, 지원 사례 참조 또는 내부 주문 번호와 같은 운영 식별자는 일반적으로 식별자가 자주 생성되고 엄격한 순서 지정이 필요하지 않은 더 큰 캐시 크기의 이점을 누릴 수 있습니다.

다음과 같은 경우에는 캐시 크기 1이 더 적합합니다.

- 할당률이 비교적 낮음
- 식별자가 시간이 지남에 따라 할당 순서를 더 가깝게 따를 것으로 예상됨
- 간격을 최소화하는 것이 최대 처리량보다 더 중요함

식별자가 덜 자주 생성되고 순서 지정이 더 가까운 계정 또는 참조 번호 할당과 같은 워크로드에는 캐시 크기 1이 더 적합합니다.

## 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 ASYNC를 실행하면 Aurora DSQL이 즉시 job\_id를 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 상태를 표시합니다. 현재 다른 트랜잭션이 동일한 네임스페이스의 객체를 참조하는 경우 동시성 오류가 표시될 수 있습니다.

## 구문

CREATE INDEX ASYNC는 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. FIRST는 Aurora DSQL이 Null이 아닌 열 앞에 Null 열을 정렬해야 함을 나타냅니다. LAST는 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 DISTINCT입니다. 즉, 고유 인덱스는 열에 여러 Null 값을 포함할 수 있습니다. NOT DISTINCT는 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
-----
jh2gbtx4mzhgfkimbmgwn5j45y
```

`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 where job_id = 'wqhu6ewifze5xitg3umt24h5ua';
```

Aurora DSQL은 다음과 유사한 응답을 반환합니다.

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

상태 열은 다음 값 중 하나일 수 있습니다.

| Status     | 설명                                                                                                                                            |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| submitted  | 작업이 제출되었지만 Aurora DSQL에서 아직 처리를 시작하지 않았습니다.                                                                                                   |
| processing | Aurora DSQL이 작업을 처리하고 있습니다.                                                                                                                   |
| failed     | 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. |
| completed  | Aurora DSQL이 작업을 성공적으로 완료했습니다.                                                                                                                |

카탈로그 테이블 `pg_index` 및 `pg_class`를 통해 인덱스의 상태를 쿼리할 수도 있습니다. 특히 `indisvalid` 및 `indisimmediate` 속성은 인덱스의 상태를 알려줄 수 있습니다. Aurora DSQL이 인덱스를 생성하는 동안 초기 상태는 `INVALID`입니다. 인덱스의 `indisvalid` 플래그는 인덱스가 유효하지 않음을 나타내는 `FALSE` 또는 `f`를 반환합니다. 플래그가 `TRUE` 또는 `t`를 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 UCVs라는 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 카탈로그와 시스템 관련 정보(예: 버전)를 가져오는 데 유용한 쿼리를 알아보려면 다음 섹션을 참조하세요.

### 시스템 테이블

Aurora DSQL은 PostgreSQL과 호환되므로 PostgreSQL의 많은 [시스템 카탈로그 테이블](#)과 [뷰](#)가 Aurora DSQL에도 있습니다.

## 중요한 PostgreSQL 카탈로그 테이블 및 뷰

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

| 이름            | 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.

| 이름           | Aurora DSQL에 적용 가능 |
|--------------|--------------------|
| pg_aggregate | 아니요                |
| pg_am        | 예                  |
| pg_amop      | 아니요                |
| pg_amproc    | 아니요                |

| 이름                      | 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                |



| 이름                       | 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                |

| 이름                    | 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.

| 이름                              | 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                |

| 이름                        | 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                |

| 이름                         | 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                |

| 이름                            | 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 보기

sys.jobs은 비동기 작업에 대한 상태 정보를 제공합니다. 예를 들어 사용자가 [비동기 인덱스를 생성](#)한 후 Aurora DSQL은 job\_uuid를 반환합니다. 이 job\_uuid와 sys.jobs을 함께 사용하여 작업 상태를 조회할 수 있습니다.

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

Aurora DSQL은 다음과 유사한 응답을 반환합니다.

```

      job_id      | status | details | job_type | class_id | object_id
| object_name    | start_time         | update_time
-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----
wqhu6ewifze5xitg3umt24h5ua | completed |          | INDEX_BUILD |      1259 |      26433
| public.nt2_c1_idx | 2025-09-25 22:07:31+00 | 2025-09-25 22:07:46+00
kknzgf33dnd13daacxehpx5eba | completed |          | ANALYZE      |      1259 |      26419
| public.nt       | 2025-09-25 21:57:05+00 | 2025-09-25 21:57:27+00
fyopxjb6ovdn7po6l1rkj63cyea | completed |          | DROP         |      1259 |      26422
|                  | 2025-09-25 22:05:57+00 | 2025-09-25 22:06:03+00

```

다음 표는 sys.jobs 보기의 열을 설명합니다.

#### sys.jobs 보기 열

| 열         | Type | 설명                                                                                                                                         |
|-----------|------|--------------------------------------------------------------------------------------------------------------------------------------------|
| job_id    | text | 작업을 나타내는 base-32 UUID입니다.                                                                                                                  |
| status    | text | 작업의 현재 상태입니다. 가능한 값은 submitted , processing , completed 및 failed입니다. 자세한 내용은 <a href="#">sys.jobs 상태 값</a> 섹션을 참조하세요.                      |
| details   | text | 작업에 대한 모든 관련 세부 정보입니다. 작업이 실패하면 자세한 이유가 제공됩니다.                                                                                             |
| job_type  | text | 비동기 작업의 유형입니다. 가능한 값은 다음과 같습니다. INDEX_BUILD - 비동기식 인덱스 구축. ANALYZE - 시스템에서 제출하는 자동 분석 작업. DROP - DROP TABLE 또는 DROP INDEX 작업 후 물리적 데이터 제거. |
| class_id  | oid  | 객체가 포함된 카탈로그 테이블의 OID입니다.                                                                                                                  |
| object_id | oid  | 객체의 OID입니다.                                                                                                                                |

| 열           | Type                     | 설명                                                                                             |
|-------------|--------------------------|------------------------------------------------------------------------------------------------|
| object_name | text                     | 객체의 정규화된 이름입니다. DROP 작업은 이미 삭제된 객체를 참조할 수 없습니다. 참조된 객체가 이미 삭제된 경우는 object_name 은 NULL일 수 있습니다. |
| start_time  | timestamp with time zone | 작업이 제출된 타임스탬프입니다.                                                                              |
| update_time | timestamp with time zone | 작업 행이 마지막으로 업데이트된 타임스탬프입니다.                                                                    |

### sys.jobs 상태 값

| Status     | 설명                                          |
|------------|---------------------------------------------|
| submitted  | 작업이 제출되었지만 Aurora DSQL에서 아직 처리를 시작하지 않았습니다. |
| processing | Aurora DSQL이 작업을 처리하고 있습니다.                 |
| failed     | 작업이 실패했습니다. 자세한 내용은 details 열을 참조하세요.       |
| completed  | Aurora DSQL이 작업을 성공적으로 완료했습니다.              |

### sys.iam\_pg\_role\_mappings 보기

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;
```



## 유용한 시스템 메타데이터 쿼리

이러한 쿼리를 사용하면 전체 테이블 스캔과 같이 비용이 많이 드는 작업을 수행하지 않고도 테이블 통계 및 시스템 메타데이터를 가져올 수 있습니다.

### 테이블의 예상 행 수 가져오기

전체 테이블 스캔을 수행하지 않고 테이블의 대략적인 행 수를 가져오려면 다음 쿼리를 사용합니다.

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

이 명령은 다음과 비슷한 출력을 반환합니다.

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

이 접근 방식은 Aurora DSQL의 SELECT COUNT(\*) 대형 테이블보다 더 효율적입니다.

### 현재 Aurora DSQL 메이저 버전 확인

Aurora DSQL 클러스터의 현재 메이저 버전을 확인하려면 다음 쿼리를 사용합니다.

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

이 명령은 다음과 비슷한 출력을 반환합니다.

```
dsqli_major_version
-----
1
```

이 반환 값은 Aurora DSQL에서 SQL 연결이 활성화된 메이저 버전입니다.

### 현재 PostgreSQL 버전 확인

Aurora DSQL 클러스터의 현재 PostgreSQL 버전을 확인하려면 다음 쿼리를 사용합니다.

```
SHOW server_version;
```

이 명령은 다음과 비슷한 출력을 반환합니다.

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

이 반환 값은 Aurora DSQL에서 SQL 연결이 활성화된 PostgreSQL 버전입니다.

## 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 EXPLAIN 계획 작업

Aurora DSQL은 PostgreSQL과 유사한 EXPLAIN 계획 구조를 사용하지만 분산 아키텍처 및 실행 모델을 반영하는 키가 추가되었습니다.

이 설명서에서는 Aurora DSQL EXPLAIN 계획에 대한 개요를 제공하여 PostgreSQL과 비교한 유사성과 차이점을 강조합니다. Aurora DSQL에서 사용할 수 있는 다양한 유형의 스캔 작업을 다루고 쿼리 실행 비용을 이해하는 데 도움이 됩니다.

### PostgreSQL VS Aurora DSQL EXPLAIN 계획

Aurora DSQL은 PostgreSQL 데이터베이스를 기반으로 구축되었으며 대부분의 계획 구조를 PostgreSQL과 공유하지만 쿼리 실행 및 최적화에 영향을 미치는 주요 아키텍처 차이점이 있습니다.

| 기능     | PostgreSQL                | Aurora DSQL                                |
|--------|---------------------------|--------------------------------------------|
| 데이터 저장 | 힙 스토리지                    | 힙 없음, 모든 행이 고유 식별자로 인덱싱됨                   |
| 기본 키   | 프라이머리 키 인덱스는 테이블 데이터와 별개임 | 프라이머리 키 인덱스는 모든 추가 열을 INCLUDE 열로 포함하는 테이블  |
| 보조 인덱스 | 표준 보조 인덱스                 | 키가 아닌 열을 포함할 수 있는 기능으로 PostgreSQL과 동일하게 작업 |
| 필터링 기능 | 인덱스 조건, 힙 필터              | 인덱스 조건, 스토리지 필터, 쿼리 프로세서 필터                |
| 스캔 유형  | 순차 스캔, 인덱스 스캔, 인덱스 전용 스캔  | 전체 스캔, 인덱스 전용 스캔, 인덱스 스캔                   |
| 쿼리 실행  | 데이터베이스에 로컬                | 분산(컴퓨팅 및 스토리지는 별도)                         |

Aurora DSQL은 테이블 데이터를 별도의 힙이 아닌 프라이머리 키 순서로 직접 저장합니다. 각 행은 일반적으로 프라이머리 키인 고유 키로 식별되므로 데이터베이스가 조회를 보다 효율적으로 최적화할 수 있습니다. 아키텍처 차이는 PostgreSQL이 인덱스 전용 스캔을 자주 사용하는 이유를 설명합니다.

또 다른 주요 차이점은 Aurora DSQL이 컴퓨팅을 스토리지와 분리하여 실행 경로의 앞부분에서 필터를 적용하여 데이터 이동을 줄이고 성능을 개선할 수 있다는 것입니다.

PostgreSQL에서 EXPLAIN 계획을 사용하는 방법에 대한 자세한 내용은 [PostgreSQL EXPLAIN 설명서](#)를 참조하세요.

## Aurora DSQL EXPLAIN 계획의 주요 요소

Aurora DSQL EXPLAIN 계획은 필터링이 발생하는 위치 및 스토리지에서 검색되는 열을 포함하여 쿼리가 실행되는 방법에 대한 자세한 정보를 제공합니다. 이 출력을 이해하면 쿼리 성능을 최적화하는 데 도움이 됩니다.

### 인덱스 조건

인덱스를 탐색하는 데 사용되는 조건입니다. 스캔되는 데이터를 줄이는 가장 효율적인 필터링입니다. Aurora DSQL에서는 실행 계획의 여러 계층에 인덱스 조건을 적용할 수 있습니다.

### 프로젝션

스토리지에서 검색된 열입니다. 프로젝트가 적을수록 성능이 향상됩니다.

### 스토리지 필터

스토리지 수준에서 적용되는 조건입니다. 쿼리 프로세서 필터보다 더 효율적입니다.

### 쿼리 프로세서 필터

쿼리 프로세서 수준에서 적용되는 조건입니다. 필터링하기 전에 모든 데이터를 전송해야 하므로 데이터 이동과 처리 오버헤드가 증가합니다.

## Aurora DSQL의 필터

Aurora DSQL은 컴퓨팅을 스토리지와 분리하므로 쿼리 실행 중에 필터가 적용되는 지점이 성능에 상당한 영향을 미칩니다. 대량의 데이터가 전송되기 전에 적용되는 필터는 지연 시간을 줄이고 효율성을 개선합니다. 필터가 적용되기 전에는 처리, 이동 및 스캔해야 하는 데이터가 적을수록 쿼리 속도가 빨라집니다.

Aurora DSQL은 쿼리 경로의 여러 단계에서 필터를 적용할 수 있습니다. 이러한 단계를 이해하는 것은 쿼리 계획을 해석하고 성능을 최적화하는 데 중요합니다.

| 수준 | 필터 유형      | 설명                                                                                                                                                        |
|----|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | 인덱스 조건     | 인덱스를 스캔하는 동안 적용됩니다. 스토리지에서 읽는 데이터의 양을 제한하고 컴퓨팅 계층으로 전송되는 데이터를 줄입니다.                                                                                       |
| 2  | 스토리지 필터    | 스토리지에서 데이터를 읽은 후 컴퓨팅으로 전송하기 전에 적용됩니다. 다음은 인덱스의 포함 열에 대한 필터의 예입니다. 데이터 전송은 줄이지만 읽기 양은 줄이지 않습니다.                                                            |
| 3  | 쿼리 프로세서 필터 | 데이터가 컴퓨팅 계층에 도달한 후 적용됩니다. 모든 데이터를 먼저 전송해야 하므로 지연 시간과 비용이 증가합니다. 현재 Aurora DSQL은 스토리지에서 모든 필터링 및 프로젝션 작업을 수행할 수 없으므로, 일부 쿼리는 이러한 유형의 필터링으로 강제로 폴백될 수 있습니다. |

## Aurora DSQL EXPLAIN 계획 읽기

EXPLAIN 계획을 읽는 방법을 이해하는 것이 쿼리 성능을 최적화하는 데 중요합니다. 이 섹션에서는 Aurora DSQL 쿼리 계획의 실제 예제를 살펴보고, 다양한 스캔 유형이 작동하는 방식을 보여주고, 필터가 적용되는 위치를 설명하고, 최적화 기회를 강조합니다.

### 이 예제에 사용된 샘플 테이블

아래 예제에서는 두 테이블 `transaction` 및 `account`를 참조합니다.

`transaction` 테이블에는 프라이머리 키가 없으므로 쿼리할 때 Aurora DSQL이 전체 테이블 스캔을 수행합니다.

`account` 테이블에는 `customer_id`에 대한 인덱스가 있습니다. 이 인덱스는 `balance` 및 `status`를 열로 포함하고 있어 기본 테이블에서 읽지 않고도 특정 쿼리를 인덱스에서 직접 처리할 수 있습니다. 그러나 이 인덱스에는 `created_at`이 포함되지 않으므로 이 열을 참조하는 쿼리에서 추가 테이블 액세스가 필요합니다.

```
CREATE TABLE transaction (
  account_id uuid,
```

```

transaction_date timestamp,
description text
);

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

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

```

## 전체 스캔 예제

Aurora DSQL에는 PostgreSQL과 기능적으로 동일한 순차 스캔과 전체 스캔이 모두 있습니다. 이 두 가지의 유일한 차이점은 전체 스캔이 스토리지에서 추가 필터링을 활용할 수 있다는 것입니다. 따라서 거의 항상 순차 스캔보다 먼저 선택됩니다. 유사성으로 인해 더 흥미로운 전체 스캔의 예만 다룹니다.

전체 스캔은 대부분 프라이머리 키가 없는 테이블에서 사용됩니다. Aurora DSQL 프라이머리 키는 기본적으로 전체 커버링 인덱스이므로 Aurora DSQL은 PostgreSQL이 순차 스캔을 사용하는 많은 상황에서 프라이머리 키에 인덱스 전용 스캔을 사용할 가능성이 높습니다. 대부분의 다른 데이터베이스와 마찬가지로 인덱스가 없는 테이블은 잘못 확장됩니다.

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

### QUERY PLAN

```

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

```

이 계획은 서로 다른 단계에 적용된 두 개의 필터를 보여줍니다. `transaction_date > '2025-01-01'` 조건은 스토리지 계층에 적용되므로 반환되는 데이터의 양이 줄어듭니다. `description LIKE '%external%'` 조건은 데이터가 전송된 후 쿼리 프로세서에서 나중에 적용되

므로 효율성이 떨어집니다. 스토리지 또는 인덱스 계층에 더 선택적인 필터를 푸시하면 일반적으로 성능이 향상됩니다.

## 인덱스 전용 스캔 예제

인덱스 전용 스캔은 스토리지 계층으로의 왕복이 가장 적기 때문에 Aurora DSQL에서 가장 최적의 스캔 유형이며 가장 많은 필터링을 수행할 수 있습니다. 하지만 인덱스 전용 스캔이 표시되었다고 해서 최상의 계획이 있다는 의미는 아닙니다. 발생할 수 있는 다양한 수준의 필터링으로 인해 필터링이 발생할 수 있는 다양한 위치에 계속 주의를 기울여야 합니다.

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

### QUERY PLAN

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

이 계획에서는 인덱스 조건인 `customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'`가 인덱스 스캔 중에 먼저 평가되는데, 이는 스토리지에서 읽는 데이터의 양을 제한하기 때문에 가장 효율적인 단계입니다. 스토리지 필터 `status = 'pending'`는 데이터를 읽은 후 컴퓨팅 계층으로 전송하기 전에 적용되므로 전송되는 데이터의 양이 줄어듭니다. 마지막으로 쿼리 프로세서 필터 `balance > 100`은 데이터가 이동된 후 마지막으로 실행되므로 효율성이 가장 낮습니다. 이 중 인덱스 조건은 스캔되는 데이터의 양을 직접 제어하기 때문에 최고의 성능을 제공합니다.

## 인덱스 스캔 예제

인덱스 스캔은 기본 테이블을 직접적으로 호출해야 하는 추가 단계가 있다는 점을 제외하면 인덱스 전용 스캔과 유사합니다. Aurora DSQL은 스토리지 필터를 지정할 수 있으므로 인덱스 직접 호출과 조회 직접 호출 모두에서 지정할 수 있습니다.

이를 명확히 하기 위해 Aurora DSQL은 계획을 두 개의 노드로 제공합니다. 이렇게 하면 스토리지에서 반환되는 행 측면에서 포함 열을 추가하는 것이 얼마나 도움이 되는지 명확하게 확인할 수 있습니다.

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

#### QUERY PLAN

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

이 계획은 여러 단계에서 필터링이 수행되는 방식을 보여 줍니다.

- `customer_id`의 인덱스 조건은 데이터를 조기에 필터링합니다.
- `status`의 스토리지 필터는 컴퓨팅으로 전송되기 전에 결과의 범위를 좁힙니다.
- `balance`의 쿼리 프로세서 필터는 전송 후 나중에 적용됩니다.
- `created_at`의 조회 필터는 기본 테이블에서 추가 열을 가져올 때 평가됩니다.

자주 사용되는 열을 `INCLUDE` 필드로 추가하면이 조회가 제거되고 성능이 향상될 수 있습니다.

#### 모범 사례

- 필터를 인덱싱된 열과 정렬하여 필터링을 이전에 푸시합니다.
- `INCLUDE` 열을 사용하여 인덱스 전용 스캔을 허용하고 조회를 방지합니다.
- 성능 문제를 조사할 때 행 추정치를 검증합니다. Aurora DSQL은 데이터 변경 속도에 따라 백그라운드에서 `ANALYZE`를 실행하여 통계를 자동으로 관리합니다. 추정치가 부정확해 보이면 `ANALYZE`를 수동으로 실행하여 통계를 즉시 새로 고칠 수 있습니다.



- 큰 테이블에서 인덱싱되지 않은 쿼리를 방지하여 비용이 많이 드는 전체 스캔을 방지합니다.

## EXPLAIN ANALYZE의 DPU 이해

Aurora DSQL은 EXPLAIN ANALYZE VERBOSE 계획 출력에 문 수준 분산 처리 장치(DPU) 정보를 제공하므로 개발 중에 쿼리 비용을 심층적으로 파악할 수 있습니다. 이 섹션에서는 DPU가 무엇이고 EXPLAIN ANALYZE VERBOSE 출력에서 해석하는 방법을 설명합니다.

### DPU란?

분산 처리 장치(DPU)는 Aurora DSQL에서 수행하는 작업의 정규화된 치수입니다. 다음과 같이 구성됩니다.

- ComputeDPU - SQL 쿼리를 실행하는 데 소요된 시간
- ReadDPU - 스토리지에서 데이터를 읽는 데 사용되는 리소스
- WriteDPU - 스토리지에 데이터를 쓰는 데 사용되는 리소스
- MultiRegionWriteDPU - 다중 리전 구성에서 피어링된 클러스터에 쓰기를 복제하는 데 사용되는 리소스입니다.

## EXPLAIN ANALYZE VERBOSE에서 DPU 사용량

Aurora DSQL은 출력이 끝날 때까지 문 수준 DPU 사용량 추정치를 EXPLAIN ANALYZE VERBOSE를 포함하도록 확장됩니다. 이를 통해 쿼리 비용을 즉시 파악할 수 있으므로 워크로드 비용 동인을 식별하고 쿼리 성능을 조정하며 리소스 사용량을 더 잘 예측할 수 있습니다.

다음 예제에서는 EXPLAIN ANALYZE VERBOSE 출력에 포함된 문 수준 DPU 추정치를 해석하는 방법을 보여줍니다.

### 예제 1: SELECT 쿼리

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

#### QUERY PLAN

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

```

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

```

이 예제에서 SELECT 문은 인덱스 전용 스캔을 수행하므로 대부분의 비용은 스토리지에서 검색된 데이터와 결과를 처리하고 반환하는 데 사용되는 컴퓨팅 리소스를 반영하는 컴퓨팅 DPU(0.01607)를 나타내는 읽기 DPU(0.04312)에서 발생합니다. 쿼리는 데이터를 수정하지 않으므로 쓰기 DPU가 없습니다. 총 DPU(0.05919)는 컴퓨팅 + 읽기 + 쓰기의 합계입니다.

## 예제 2: 쿼리 삽입

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

### QUERY PLAN

```

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

```

이 문은 주로 쓰기를 수행하므로 대부분의 비용이 쓰기 DPU와 연결됩니다. 컴퓨팅 DPU(0.01550)는 값을 처리하고 삽입하기 위해 수행된 작업을 나타냅니다. 읽기 DPU(0.00307)는 마이너 시스템 읽기(카탈로그 조회 또는 인덱스 검사용)를 반영합니다.

읽기 및 쓰기 DPU 옆에 표시된 트랜잭션 최소값을 확인합니다. 이는 작업에 읽기 또는 쓰기가 포함된 경우에만 적용되는 기준 트랜잭션당 비용을 나타냅니다. 모든 트랜잭션에 0.00375 읽기 DPU 또는 0.05 쓰기 DPU 요금이 자동으로 부과된다는 의미는 아닙니다. 대신 이러한 최소값은 비용 집계 중에 트랜잭션 수준에서 적용되며 해당 트랜잭션 내에서 읽기 또는 쓰기가 발생하는 경우에만 적용됩니다. 이러한 범위 차이로 인해 EXPLAIN ANALYZE VERBOSE의 문 수준 추정치는 CloudWatch 또는 결제 데이터에 보고된 트랜잭션 수준 지표와 정확히 일치하지 않을 수 있습니다.

## 최적화를 위한 DPU 정보 사용

문별 DPU 추정치는 실행 시간 이상으로 쿼리를 최적화할 수 있는 강력한 방법을 제공합니다. 일반적인 사용 사례는 다음과 같습니다.

- 비용 인식: 쿼리가 다른 쿼리에 비해 얼마나 많은지 이해합니다.
- 스키마 최적화: 인덱스 또는 스키마 변경이 성능 및 리소스 효율성에 미치는 영향을 비교합니다.
- 예산 계획: 관찰된 DPU 사용량을 기준으로 워크로드 비용을 추정합니다.
- 쿼리 비교: 상대적 DPU 사용량을 기준으로 대체 쿼리 접근 방식을 평가합니다.

## DPU 정보 해석

EXPLAIN ANALYZE VERBOSE에서 DPU 데이터를 사용할 때는 다음 모범 사례를 염두에 두세요.

- 방향에 따라 사용: 보고된 DPU를 CloudWatch 지표 또는 결제 데이터와 정확히 일치하지 않고 쿼리의 상대 비용을 이해하는 방법으로 취급합니다. EXPLAIN ANALYZE VERBOSE는 문 수준 비용을 보고하는 반면 CloudWatch는 트랜잭션 수준 활동을 집계하기 때문에 차이가 예상됩니다. CloudWatch에는 EXPLAIN ANALYZE VERBOSE에서 의도적으로 제외되는 백그라운드 작업(예: ANALYZE 또는 압축) 및 트랜잭션 오버헤드(BEGIN/COMMIT)도 포함됩니다.
- 실행 간 DPU 변동성은 분산 시스템에서 정상이며 오류를 나타내지 않습니다. 캐싱, 실행 계획 변경, 동시성 또는 데이터 배포의 전환과 같은 요인으로 인해 동일한 쿼리가 한 실행에서 다음 실행으로 서로 다른 리소스를 소비할 수 있습니다.
- 소규모 작업 배치: 워크로드에서 작은 문이 많이 발생하는 경우 대규모 작업(10MB를 초과하지 않음)으로 일괄 처리하는 것이 좋습니다. 이렇게 하면 반올림 오버헤드가 줄어들고 더 의미 있는 비용 추정치가 생성됩니다.
- 결제가 아닌 튜닝에 사용: EXPLAIN ANALYZE VERBOSE의 DPU 데이터는 비용 인식, 쿼리 튜닝 및 최적화를 위해 설계되었습니다. 결제 등급 지표가 아닙니다. 신뢰할 수 있는 비용 및 사용량 데이터는 항상 CloudWatch 지표 또는 월별 결제 보고서에 의존하세요.

# 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

- [단일 리전 클러스터 구성](#)
- [다중 리전 클러스터 구성](#)
- [AWS CloudFormation을 사용하여 Aurora DSQL 클러스터 구성](#)
- [Aurora DSQL 클러스터 수명 주기](#)

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

## 단일 리전 클러스터 구성

AWS CLI 또는 선호하는 프로그래밍 언어(Python, C++, JavaScript, Java, Rust, Ruby, .NET, Golang 포함)를 사용하여 단일 AWS 리전의 클러스터를 구성하고 관리합니다. AWS CLI는 셸 명령어를 통해 빠른 접근을 제공하는 반면, AWS 소프트웨어 개발 키트(SDK)는 네이티브 언어 지원을 통해 프로그래밍 방식으로 제어할 수 있게 합니다.

주제

- [AWS SDK 사용](#)
- [AWS CLI 사용](#)

## AWS SDK 사용

AWS SDK는 선호하는 프로그래밍 언어로 Aurora DSQL에 대한 프로그래밍 방식 액세스를 제공합니다. 다음 섹션에서는 다양한 프로그래밍 언어를 사용하여 일반적인 클러스터 작업을 수행하는 방법을 보여줍니다.

## 클러스터 생성

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 단일 리전 클러스터를 생성하는 방법을 보여줍니다.

## 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()
```

## 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;
}

```

## 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();

```

## 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.dsdl.DsdlClient;
import software.amazon.awssdk.services.dsdl.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsdl.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsdl.model.GetClusterResponse;

import java.time.Duration;
import java.util.Map;

public class CreateCluster {

```



```

public static void main(String[] args) {
    Region region = Region.US_EAST_1;

    try (
        DsqlClient client = DsqlClient.builder()
            .region(region)
            .credentialsProvider(DefaultCredentialsProvider.create())
            .build()
    ) {
        CreateClusterRequest request = CreateClusterRequest.builder()
            .deletionProtectionEnabled(true)
            .tags(Map.of("Name", "java single region cluster"))
            .build();
        CreateClusterResponse cluster = client.createCluster(request);
        System.out.println("Created " + cluster.arn());

        // The DSQL SDK offers a built-in waiter to poll for a cluster's
        // transition to ACTIVE.
        System.out.println("Waiting for cluster to become ACTIVE");
        WaiterResponse<GetClusterResponse> waiterResponse =
client.waiter().waitUntilClusterActive(
            getCluster -> getCluster.identifier(cluster.identifier()),
            config -> config.backoffStrategyV2(

BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                ).waitTimeout(Duration.ofMinutes(5))
            );
        waiterResponse.matched().response().ifPresent(System.out::println);
    }
}

```

## Rust

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

```

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

```

```

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

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

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

    Client::from_conf(config)
}

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

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

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

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

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

    cluster_output
}

```

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

    let cluster = create_cluster(region).await;

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

    Ok(())
}
```

## Ruby

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

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

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

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

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

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

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

  cluster = create_cluster(region)

  puts "Created single region cluster:"
  pp cluster
end

main if $PROGRAM_NAME == __FILE__
```

## .NET

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

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

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

```

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

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

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

            return cluster;
        }
    }

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

        var cluster = await Create(region);

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

```

## Golang

단일 AWS 리전에서 클러스터를 생성하려면 다음 예시를 사용합니다.

```
package main
```

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

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

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

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

    client := dsql.NewFromConfig(cfg)

    deleteProtect := true

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

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

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

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

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

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

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

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

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

## 클러스터 가져오기

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

### Python

단일 리전 클러스터에 대한 정보를 얻으려면 다음 예제를 사용하세요.

```
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 예제를 사용하여 단일 리전 클러스터에 대한 정보를 확인합니다.

```
#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 리전 클러스터에 대한 정보를 얻으려면 다음 예제를 사용하세요.

```
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 예제는 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```
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 예제는 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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 dsq_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 예제는 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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 예제는 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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<AmazonDSQIClient> CreateDSQIClient(RegionEndpoint
region)
        {
            var awsCredentials = await
DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
            var clientConfig = new AmazonDSQLConfig
            {
                RegionEndpoint = region
            };
            return new AmazonDSQIClient(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 예제는 단일 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 단일 리전 클러스터를 업데이트하는 방법을 보여줍니다.

### Python

단일 리전 클러스터를 업데이트하려면 다음 예제를 사용하세요.

```
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 예제를 사용하여 단일 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 리전 클러스터를 업데이트하려면 다음 예제를 사용하세요.

```

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 예제를 사용하여 단일 리전 클러스터를 업데이트합니다.

```

package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsqli.DsqliClient;
import software.amazon.awssdk.services.dsqli.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsqli.model.UpdateClusterResponse;

public class UpdateCluster {

    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        String clusterId = "<your cluster id>";

        try (
            DsqliClient client = DsqliClient.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 예제를 사용하여 단일 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 dsq_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 = dsq_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 예제를 사용하여 단일 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 예제를 사용하여 단일 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 예제를 사용하여 단일 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

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 단일 리전 클러스터를 삭제하는 방법을 보여줍니다.



## 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()
```

## 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;
}

```

## 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();

```

## 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.dsdl.DsdlClient;
import software.amazon.awssdk.services.dsdl.model.DeleteClusterResponse;
import software.amazon.awssdk.services.dsdl.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 (
            DsdlClient client = DsdlClient.builder()
                .region(region)

```

```

        .credentialsProvider(DefaultCredentialsProvider.create())
        .build()
    ) {
        DeleteClusterResponse cluster = client.deleteCluster(r ->
r.identifier(clusterId));
        System.out.println("Initiated delete of " + cluster.arn());

        // The DSQL SDK offers a built-in waiter to poll for deletion.
        System.out.println("Waiting for cluster to finish deletion");
        client.waiter().waitUntilClusterNotExists(
            getCluster -> getCluster.identifier(clusterId),
            config -> config.backoffStrategyV2(
                BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    .waitTimeout(Duration.ofMinutes(5))
            );
        System.out.println("Deleted " + cluster.arn());
    } catch (ResourceNotFoundException e) {
        System.out.printf("Cluster %s not found in %s\n", clusterId, region);
    }
}
}

```

## Rust

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 dsq_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

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

```

```

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

    Client::from_conf(config)
}

/// Delete a DSQL cluster
pub async fn delete_cluster(region: &'static str, identifier: &'static str) {
    let client = dsql_client(region).await;
    let delete_response = client
        .delete_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap();
    println!("Initiated delete of {}", delete_response.arn);

    println!("Waiting for cluster to finish deletion");
    client
        .wait_until_cluster_not_exists()
        .identifier(identifier)
        .wait(std::time::Duration::from_secs(300)) // Wait up to 5 minutes
        .await
        .unwrap();
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";
    let cluster_id = "<cluster to be deleted>";

    delete_cluster(region, cluster_id).await;
    println!("Deleted {cluster_id}");

    Ok(())
}

```

## Ruby

단일 AWS 리전에서 클러스터를 삭제하려면 다음 예시를 사용합니다.

```
require "aws-sdk-dsql"

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

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

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

main if $PROGRAM_NAME == __FILE__
```

## .NET

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<AmazonDSQIClient> CreateDSQIClient(RegionEndpoint
region)
    {
        var awsCredentials = await
DefaultAWSCredentialsIdentityResolver.GetCredentialsAsync();
        var clientConfig = new AmazonDSQLConfig
        {
            RegionEndpoint = region
        };
        return new AmazonDSQIClient(awsCredentials, clientConfig);
    }

    /// <summary>
    /// Delete a DSQL cluster.
    /// </summary>
    public static async Task Delete(RegionEndpoint region, string identifier)
    {
        using (var client = await CreateDSQIClient(region))
        {
            var deleteRequest = new DeleteClusterRequest
            {
                Identifier = identifier
            };

            var deleteResponse = await client.DeleteClusterAsync(deleteRequest);
            Console.WriteLine($"Initiated deletion of {deleteResponse.Arn}");
        }
    }

    private static async Task Main()
    {
        var region = RegionEndpoint.USEast1;
        var clusterId = "<cluster to be deleted>";

        await Delete(region, clusterId);
    }
}

```



## Golang

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)
    }
}

```

더 많은 코드 샘플과 예제를 보려면 [Aurora DSQL 샘플 GitHub 리포지토리](#)를 참조하세요.

## AWS CLI 사용

AWS CLI는 Aurora DSQL 클러스터를 관리하기 위한 명령줄 인터페이스를 제공합니다. 다음 예제에서는 아래와 같은 작업을 포함하여 공통적인 클러스터 관리 작업을 설명합니다.

### 클러스터 생성

create-cluster 명령을 사용하여 클러스터를 생성합니다.

#### Note

클러스터 생성은 비동기 작업입니다. 상태가 ACTIVE로 변경될 때까지 GetCluster API를 직접 호출합니다. 클러스터가 활성화되면 클러스터에 연결할 수 있습니다.

#### Example Command

```
aws dsq1 create-cluster --region us-east-1
```

#### Note

생성 중에 삭제 방지를 비활성화하려면 --no-deletion-protection-enabled 플래그를 포함합니다.

#### Example 응답

```
{
  "identifier": "abc0def1baz2quux3quux4",
  "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4",
  "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 dsq1 get-cluster \  
  --region us-east-1 \  
  --identifier your_cluster_id
```

### Example 응답

```
{  
  "identifier": "abc0def1baz2quux3quux4",  
  "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4",  
  "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 dsq1 update-cluster \  
  --region us-east-1 \  
  --no-deletion-protection-enabled \  
  --identifier your_cluster_id
```

```
--identifier your_cluster_id
```

## Example 응답

```
{
  "identifier": "abc0def1baz2quux3quux4",
  "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4",
  "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 dsq1 delete-cluster \
  --region us-east-1 \
  --identifier your_cluster_id
```

## Example 응답

```
{
  "identifier": "abc0def1baz2quux3quux4",
  "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4",
  "status": "DELETING",
  "creationTime": "2024-05-24T09:16:43.778000-07:00"
}
```

## 클러스터 나열

list-clusters 명령을 사용하여 클러스터를 나열합니다.

## Example Command

```
aws dsq1 list-clusters --region us-east-1
```

## Example 응답

```
{
  "clusters": [
    {
      "identifier": "abc0def1baz2quux3quux4quuux",
      "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4quuux"
    },
    {
      "identifier": "abc0def1baz2quux3quux5quuuux",
      "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux5quuuux"
    }
  ]
}
```

## 다중 리전 클러스터 구성

AWS CLI 또는 선호하는 프로그래밍 언어(Python, C++, JavaScript, Java, Rust, Ruby, .NET, Golang 포함)를 사용하여 여러 AWS 리전에 걸쳐 클러스터를 구성하고 관리합니다. AWS CLI는 셸 명령어를 통해 빠른 접근을 제공하는 반면, AWS SDK는 네이티브 언어 지원을 통해 프로그래밍 방식으로 제어할 수 있게 합니다.

### 주제

- [AWS SDK 사용](#)
- [AWS CLI 사용](#)

## AWS SDK 사용

AWS SDK는 선호하는 프로그래밍 언어로 Aurora DSQL에 대한 프로그래밍 방식 액세스를 제공합니다. 다음 섹션에서는 다양한 프로그래밍 언어를 사용하여 일반적인 클러스터 작업을 수행하는 방법을 보여줍니다.

## 클러스터 생성

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 다중 리전 클러스터를 생성하는 방법을 보여줍니다.

### Python

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

return (cluster_1, cluster_2)

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

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

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



## C++

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);

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

    CreateClusterRequest createClusterRequest1;
    createClusterRequest1.SetDeletionProtectionEnabled(true);

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

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

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

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

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

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

CreateClusterRequest createClusterRequest2;
createClusterRequest2.SetDeletionProtectionEnabled(true);

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

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

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

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

```

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

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

MultiRegionProperties updatedProps;
updatedProps.SetWitnessRegion(witnessRegion);

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

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

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

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

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

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

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

```

```

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

```

## JavaScript

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

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

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

```

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

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

    await createMultiRegionCluster(region1, region2, witnessRegion);
}

main();

```

## Java

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.dsdl.DsdlClient;
import software.amazon.awssdk.services.dsdl.DsdlClientBuilder;
import software.amazon.awssdk.services.dsdl.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsdl.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsdl.model.GetClusterResponse;
import software.amazon.awssdk.services.dsdl.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

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

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"
    }
  )
end
```

```

)
puts "Created #{cluster_2.arn}"

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

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

puts "Waiting for #{cluster_2.arn} to become ACTIVE"
cluster_2 = client_2.wait_until(:cluster_active, identifier: cluster_2.identifier)
do |w|
  w.max_attempts = 30
  w.delay = 10
end

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

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

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

  puts "Created multi region clusters:"

```

```

    pp cluster_1
    pp cluster_2
end

main if $PROGRAM_NAME == __FILE__

```

## Golang

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)
    }
}

```

## .NET

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 {cluster2.Arn} as a peer of
{cluster1.Arn}");

        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}");
}
}
}

```

## 클러스터 가져오기

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 다중 리전 클러스터에 대한 정보를 얻는 방법을 보여줍니다.

### Python

다중 리전 클러스터에 대한 정보를 얻으려면 다음 예제를 사용하세요.

```

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 예제를 사용하여 다중 리전 클러스터에 대한 정보를 확인합니다.

```

#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

다중 리전 클러스터에 대한 정보를 얻으려면 다음 예제를 사용하세요.

```

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 예제는 다중 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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 예제는 다중 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```

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 예제는 다중 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```
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 예제는 다중 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```
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 예제는 다중 리전 클러스터에 대한 정보를 가져오는 방법을 보여줍니다.

```
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

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 다중 리전 클러스터를 업데이트하는 방법을 보여줍니다.

### Python

다중 리전 클러스터를 업데이트하려면 다음 예제를 사용하세요.

```
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 예제를 사용하여 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

다중 리전 클러스터를 업데이트하려면 다음 예제를 사용하세요.

```

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 예제를 사용하여 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.dsqli.DsqliClient;
import software.amazon.awssdk.services.dsqli.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsqli.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 예제를 사용하여 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 예제를 사용하여 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 예제를 사용하여 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 예제를 사용하여 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

다음 예제에서는 다양한 프로그래밍 언어를 사용하여 다중 리전 클러스터를 삭제하는 방법을 보여줍니다.

### Python

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

```
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++

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

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

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.dsdl.DsdlClient;
import software.amazon.awssdk.services.dsdl.DsdlClientBuilder;
import software.amazon.awssdk.services.dsdl.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>";

    DsdlClientBuilder clientBuilder = DsdlClient.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

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

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

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

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)
    }
}

```

더 많은 코드 샘플과 예제를 보려면 [Aurora DSQL 샘플 GitHub 리포지토리](#)를 참조하세요.

## AWS CLI 사용

AWS CLI는 다중 리전 Aurora DSQL 클러스터를 관리하기 위한 명령줄 인터페이스를 제공합니다. 다음 예제에서는 다중 리전 클러스터를 생성, 구성 및 삭제하는 방법을 보여 줍니다.

### 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 dsq1 create-cluster \  
--region us-east-1 \  
--multi-region-properties '{"witnessRegion":"us-west-2"}'
```

Example응답:

```
{  
  "identifier": "abc0def1baz2quux3quux4",  
  "arn": "arn:aws:dsq1:us-east-1:111122223333:cluster/abc0def1baz2quux3quux4",  
  "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 dsq1 create-cluster \  

```

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

Example응답:

```
{
  "identifier": "foo0bar1baz2quux3quuxquux5",
  "arn": "arn:aws:dsq1: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:dsq1: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 dsq1 update-cluster \
--region us-east-1 \
--identifier 'foo0bar1baz2quux3quuxquux4' \
--multi-region-properties '{"witnessRegion": "us-west-2","clusters": ["arn:aws:dsq1:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"]}'
```

Example응답

```
{
  "identifier": "foo0bar1baz2quux3quuxquux4",
  "arn": "arn:aws:dsq1: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 dsq1 update-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5' \
--multi-region-properties '{"witnessRegion": "us-west-2", "clusters":
["arn:aws:dsq1:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuxquux4"]}'
```

### Example응답

```
{
  "identifier": "foo0bar1baz2quux3quuxquux5",
  "arn": "arn:aws:dsq1: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 dsq1 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 dsq1 update-cluster \
  --region us-east-1 \
  --identifier 'foo0bar1baz2quux3quuxquux4' \
  --no-deletion-protection-enabled
```

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

```
aws dsq1 delete-cluster \
  --region us-east-1 \
  --identifier 'foo0bar1baz2quux3quuxquux4'
```

이 명령은 다음 응답을 반환합니다.

```
{
  "identifier": "foo0bar1baz2quux3quuxquux4",
  "arn": "arn:aws:dsq1: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 dsq1 update-cluster \
```

```
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quux4quux' \
--no-deletion-protection-enabled
```

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

```
aws dsq1 delete-cluster \
--region us-east-2 \
--identifier 'foo0bar1baz2quux3quuxquux5'
```

이 명령은 다음 응답을 반환합니다.

```
{
  "identifier": "foo0bar1baz2quux3quuxquux5",
  "arn": "arn:aws:dsq1: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 Aurora DSQL 클러스터 구성

동일한 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 reference](#)를 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 DSQL 클러스터 수명 주기

Aurora DSQL 클러스터 수명 주기를 이해하면 클러스터를 효과적으로 관리하는 데 도움이 됩니다. 이 섹션에서는 클러스터 상태 정의와 비용을 최적화하는 0으로 조정 기능에 대해 설명합니다.

### Aurora DSQL 클러스터 상태 정의

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.

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

| Status | 설명                                                                               |
|--------|----------------------------------------------------------------------------------|
| 생성 중   | Aurora DSQL이 클러스터에 대한 리소스를 생성하거나 구성하려고 합니다. 클러스터가 이 상태에 있는 동안에는 모든 연결 시도가 실패합니다. |

| Status  | 설명                                                                                                                                                                       |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 활성      | 클러스터가 작동 중이며 사용할 준비가 되었습니다.                                                                                                                                              |
| 유휴      | Aurora DSQL이 실행 중인 리소스를 축소하여 용량과 비용을 줄일 수 있을 만큼 충분히 유휴 상태이면 클러스터가 유휴 상태가 됩니다. 유휴 클러스터에 연결하면 Aurora DSQL이 클러스터를 다시 활성 상태로 전환합니다.                                          |
| 비활성     | 장기간 클러스터에 활동이 없으면 유휴 클러스터가 비활성화됩니다. 이 일시 중지 상태에서 실행 중인 리소스는 데이터가 보존되는 동안 0으로 조정됩니다. 비활성 클러스터에 연결하려고 하면 Aurora DSQL이 자동으로 클러스터를 다시 활성 상태로 전환합니다. 복원 시간은 클러스터 크기에 따라 다릅니다. |
| 업데이트 중  | 클러스터 구성을 변경하면 클러스터가 업데이트 중 상태로 전환됩니다.                                                                                                                                    |
| 삭제 중    | 삭제 요청을 제출하면 클러스터가 삭제 중 상태로 전환됩니다.                                                                                                                                        |
| 삭제됨     | 클러스터가 성공적으로 삭제되었습니다.                                                                                                                                                     |
| 실패      | Aurora DSQL에서 오류가 발생하여 클러스터를 만들 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.                                                        |

## 유휴 및 비활성 클러스터 작업

Aurora DSQL은 일정 기간 동안 클러스터에서 연결 활동을 감지하지 못하면 클러스터를 유휴 상태로 전환하여 실행 중인 리소스를 줄여 용량과 비용을 최소화합니다. 연결 활동이 장기간 동안 없는 경우 유휴 클러스터는 데이터가 보존되는 동안 실행 중인 리소스가 0으로 조정되는 비활성 상태로 자동 전환됩니다.

정상 작업을 재개하려면 평소와 같이 클러스터에 연결하기만 하면 됩니다. 클러스터에 성공적으로 연결하면 Aurora DSQL이 클러스터를 자동으로 활성 상태로 전환합니다.

**Note**

유휴 또는 비활성 클러스터에 대한 첫 번째 연결 시도는 평소보다 느립니다.

### 활성 클러스터 상태가 필요한 작업

일부 작업에서는 클러스터가 활성 상태여야 합니다. 유휴 또는 비활성 클러스터에서 이러한 작업을 수행하려면 클러스터에 연결하여 클러스터를 다시 활성으로 전환해야 합니다.

### 백업 작업

백업을 수행하려면 활성 클러스터 상태가 필요합니다. 클러스터가 유휴 또는 비활성 상태인 경우 다음 오류와 함께 백업이 실패합니다.

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

### 백업 진행 방법:

1. 원하는 데이터베이스 클라이언트 또는 Aurora DSQL 콘솔을 사용하여 클러스터에 연결하여 활성화합니다.
2. 활성 상태로 자동 전환될 때까지 기다립니다.
3. 클러스터가 완전히 작동하면 백업을 시작합니다.

**Note**

클러스터가 유휴 또는 비활성이기 전에 수행된 기존 백업은 유효하며 영향을 받지 않습니다. 클러스터의 새 백업 시도는 클러스터가 자동 절전 모드를 해제하기 위해 연결될 때까지 실패합니다.

## 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/dsql>에서 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 dsq1 get-cluster --identifier cluster-id --query status --output text
```

다음 명령을 실행하여 모든 클러스터의 상태를 나열합니다.

```
for id in $(aws dsq1 list-clusters --query 'clusters[*].identifier' --output text); do
    cluster_status=$(aws dsq1 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 |

# Aurora DSQL을 사용한 프로그래밍

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

## AWS Command Line Interface (AWS CLI)

명령줄 셸에서 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 [dsql](#)을 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 도구](#)를 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.

## CloudFormation

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

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.

# Aurora DSQL용 커넥터

Aurora DSQL은 기존 데이터베이스 드라이버를 확장하여 AWS 서비스와의 원활한 IAM 인증 및 통합을 지원하는 전용 커넥터를 제공합니다. 이러한 커넥터는 기존 PostgreSQL 워크플로와의 호환성을 유지하면서 널리 사용되는 프로그래밍 언어 및 프레임워크와 함께 작동하도록 설계되었습니다.

향후 릴리스에 추가 커넥터가 계획되어 있습니다. 커넥터 가용성에 대한 최신 정보는 [Aurora DSQL 샘플 리포지토리](#)를 참조하세요.



## JDBC 커넥터를 사용하여 Aurora DSQL 클러스터에 연결

[JDBC용 Aurora DSQL 커넥터](#)는 애플리케이션이 IAM 자격 증명을 사용하여 Aurora DSQL로 인증할 수 있도록 PostgreSQL JDBC 드라이버의 기능을 확장하는 인증 플러그인으로 설계되었습니다. 커넥터는 데이터베이스에 직접 연결되지 않지만 기본 PostgreSQL JDBC 드라이버를 기반으로 원활한 IAM 인증을 제공합니다.

JDBC용 Aurora DSQL 커넥터는 [PostgreSQL JDBC 드라이버](#)와 함께 작동하도록 빌드되었으며 Aurora DSQL의 IAM 인증 요구 사항과 원활하게 통합됩니다.

JDBC용 Aurora DSQL 커넥터는 PostgreSQL JDBC 드라이버와 함께 사용 시 Aurora DSQL에 대한 IAM 기반 인증을 지원합니다. [AWS Identity and Access Management\(IAM\)](#)과 같은 AWS 인증 서비스와의 심층 통합을 제공합니다.

### 커넥터 정보

Aurora DSQL은 PostgreSQL 호환 애플리케이션에 고가용성과 확장성을 제공하는 분산된 SQL 데이터베이스 서비스입니다. Aurora DSQL에는 기존 JDBC 드라이버가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 기반 인증이 필요합니다.

JDBC용 Aurora DSQL 커넥터의 기본 개념은 IAM 토큰 생성을 처리하는 PostgreSQL JDBC 드라이버 위에 인증 계층을 추가하여 사용자가 기존 JDBC 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 하는 것입니다.

Aurora DSQL 인증이란 무엇입니까?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 인증 토큰은 AWS 자격 증명을 사용하여 생성되며 구성 가능한 수명이 있습니다.

JDBC용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성하도록 설계되었습니다.

### JDBC용 Aurora DSQL 커넥터의 이점

Aurora DSQL은 PostgreSQL 호환 인터페이스를 제공하지만 기존 PostgreSQL 드라이버는 현재 Aurora DSQL의 IAM 인증 요구 사항을 지원하지 않습니다. JDBC용 Aurora DSQL 커넥터를 사용하면 다음을 통해 IAM 인증을 활성화하면서 기존 PostgreSQL 워크플로를 계속 사용할 수 있습니다.

- 자동 토큰 생성: AWS 자격 증명을 사용하여 IAM 토큰이 자동으로 생성됨
- 원활한 통합: 기존 JDBC 연결 패턴과 함께 작동함
- AWS 자격 증명 지원: 다양한 AWS 자격 증명 공급자(기본값, 프로필 기반 등) 지원

## 연결 풀링과 함께 JDBC용 Aurora DSQL 커넥터 사용

JDBC용 Aurora DSQL 커넥터는 HikariCP와 같은 연결 풀링 라이브러리에서 작동합니다. 커넥터는 연결 설정 중에 IAM 토큰 생성을 처리하여 연결 풀이 정상적으로 작동할 수 있도록 합니다.

## 주요 기능

### 자동 토큰 생성

IAM 토큰은 AWS 자격 증명을 사용하여 자동으로 생성됩니다.

### 원활한 통합

기존 JDBC 연결 패턴과 호환되며 워크플로 변경 없이 작동합니다.

### AWS 자격 증명 지원

다양한 AWS 자격 증명 공급자(기본값, 프로필 기반 등)를 지원합니다.

### 연결 풀링 호환성

HikariCP와 같은 연결 풀링 라이브러리와 원활하게 작동합니다.

## 사전 조건

시작하기 전에 다음 사전 조건을 완료했는지 확인합니다.

- [Aurora DSQL에 클러스터 생성](#)
- JDK(Java Development Kit) 생성 버전 17 이상이 있는지 확인
- 애플리케이션이 Aurora DSQL에 연결할 수 있도록 적절한 IAM 권한 설정
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

## 기본 연결에 JDBC용 Aurora DSQL 커넥터 사용

Java 애플리케이션에서 JDBC용 Aurora DSQL 커넥터를 사용하려면 다음 단계를 따르세요.

## 1. 다음 종속성을 Maven 프로젝트에 추가합니다.

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

Gradle 프로젝트의 경우 다음 종속성을 추가합니다.

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

## 2. AWS DSQL PostgreSQL 커넥터 형식을 사용하여 Aurora DSQL 클러스터에 대한 기본 연결을 생성합니다.

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

public class DsdlJdbcConnectorExample {
    public static void main(String[] args) {
        // Using AWS DSQL PostgreSQL Connector prefix
        String jdbcUrl = "jdbc:aws-dsdl:postgresql://your-cluster.dsdl.us-east-1.on.aws/postgres?user=admin";

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

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

                // Query data
```

```

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

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

```

## 구성 속성

JDBC용 Aurora DSQL 커넥터는 다음 연결 속성을 지원합니다.

**user**

연결에 사용될 사용자 및 토큰 생성 방법을 결정합니다. 예시: admin

**token-duration-secs**

토큰 유효 기간(초 단위)입니다. 토큰 제한에 대한 자세한 내용은 [Amazon Aurora DSQL에서 인증 토큰 생성](#)을 참조하세요.

**profile**

제공된 프로필 이름으로 토큰 생성을 위한 ProfileCredentialsProvider를 인스턴스화하는 데 사용됩니다.

**region**

Aurora DSQL 연결을 위한 AWS 리전입니다. 이는 선택 사항입니다. 제공된 경우 URL에서 추출된 리전을 재정의합니다.

**데이터베이스**

연결할 데이터베이스 이름입니다. 기본값은 postgres입니다.

## 로깅

Aurora DSQL JDBC 커넥터를 사용하는 동안 발생할 수 있는 문제를 해결하기 위해 로깅을 활성화합니다.

이 커넥터는 Java의 내장 로깅 시스템(java.util.logging)을 사용합니다. logging.properties 파일을 생성하여 로깅 수준을 구성할 수 있습니다.

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

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

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

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

## 예제

보다 포괄적인 예제 및 사용 사례는 다음을 포함하는 [JDBC용 Aurora DSQL 커넥터 리포지토리](#)를 참조하세요.

## Python용 Aurora DSQL 커넥터

[Python용 Aurora DSQL 커넥터](#)는 Python 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합합니다. 내부적으로는 [psycopg](#), [psycopg2](#) 및 [asyncpg](#) 클라이언트 라이브러리를 활용합니다.

Python용 Aurora DSQL 커넥터는 psycopg, psycopg2, 및 asyncpg 클라이언트 및 라이브러리를 확장하는 인증 플러그인으로 설계되어 애플리케이션이 IAM 자격 증명을 사용하여 Amazon Aurora DSQL과 인증할 수 있도록 합니다. 커넥터는 데이터베이스에 직접 연결되지 않지만 기본 클라이언트 라이브러리를 기반으로 원활한 IAM 인증을 제공합니다.

## 커넥터 정보

Amazon Aurora DSQL은 PostgreSQL 호환 애플리케이션에 고가용성과 확장성을 제공하는 분산된 SQL 데이터베이스 서비스입니다. Aurora DSQL에는 기존 Python 라이브러리가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 기반 인증이 필요합니다.

Python용 Aurora DSQL 커넥터의 개념은 IAM 토큰 생성을 처리하는 psycopg, psycopg2 및 asyncpg 클라이언트 라이브러리 위에 인증 계층을 추가하여 사용자가 기존 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 하는 것입니다.

Aurora DSQL 인증이란 무엇입니까?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 인증 토큰은 AWS 자격 증명을 사용하여 생성되며 구성 가능한 수명이 있습니다.

Python용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성하도록 설계되었습니다.

## 특성

- 자동 IAM 인증 - IAM 토큰은 AWS 자격 증명을 사용하여 자동으로 생성됩니다.
- psycopg, psycopg2 및 asyncpg 기반 - psycopg, psycopg2 및 asyncpg 클라이언트 라이브러리 활용
- 원활한 통합 - 워크플로 변경 없이 기존 psycopg, psycopg2 및 asyncpg 연결 패턴과 함께 작동
- 리전 자동 검색 - DSQL 클러스터 호스트 이름에서 AWS 리전 추출
- AWS 자격 증명 지원 - 다양한 AWS 자격 증명 공급자(기본값, 프로필 기반, 사용자 지정) 지원
- 연결 풀링 호환성 - psycopg, psycopg2 및 asyncpg 내장 연결 풀링과 함께 사용 가능

## 빠른 시작 설명서

### 요구 사항

- Python 3.10 이상
- [Aurora DSQL 클러스터에 대한 액세스](#)
- 애플리케이션이 Aurora DSQL에 연결할 수 있도록 적절한 IAM 권한 설정
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

## 설치

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

psycopg 또는 psycopg2 또는 asyncpg를 별도로 설치

Python용 Aurora DSQL 커넥터 설치 관리자는 기본 라이브러리를 설치하지 않습니다. 별도로 설치해야 합니다. 예:

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

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

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

참고:

필요한 라이브러리만 설치해야 합니다. 따라서 클라이언트가 psycopg를 사용할 경우 psycopg만 설치하면 됩니다. 클라이언트가 psycopg2를 사용하려는 경우 psycopg2만 설치하면 됩니다. 클라이언트가 asyncpg를 사용하려는 경우 asyncpg만 설치하면 됩니다.

클라이언트에 둘 이상 필요한 경우 필요한 모든 라이브러리를 설치해야 합니다.

## 기본 사용법

### psycopg

```
import aurora_dsql_psycopg as dsql

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

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

```
result = cur.fetchone()
print(result)
```

## psycopg2

```
import aurora_dsycopg2 as dsql

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

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

## asyncpg

```
import asyncio
import aurora_dsycopg2_asyncpg as dsql

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

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

## 호스트만 사용

### psycopg

```
import aurora_dsycopg as dsql

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



## psycopg2

```
import aurora_dsycopg2 as dsq1

conn = dsq1.connect("your-cluster.dsq1.us-east-1.on.aws")
```

## asyncpg

```
import asyncio
import aurora_dsycopg2_asyncpg as dsq1

conn = await dsq1.connect("your-cluster.dsq1.us-east-1.on.aws")
```

## 클러스터 ID만 사용

### psycopg

```
import aurora_dsycopg as dsq1

conn = dsq1.connect("your-cluster")
```

### psycopg2

```
import aurora_dsycopg2 as dsq1

conn = dsq1.connect("your-cluster")
```

### asyncpg

```
import asyncio
import aurora_dsycopg2_asyncpg as dsq1

conn = await dsq1.connect("your-cluster")
```

## 참고:

"클러스터 ID만 사용" 시나리오에서는 시스템에 이전에 설정된 리전이 사용됩니다. 예:

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

리전이 설정되지 않았거나 지정된 클러스터 ID가 다른 리전에 있는 경우 연결이 실패합니다. 작동하려면 아래 예제와 같이 리전을 파라미터로 제공합니다.

### psycopg

```
import aurora_dsycopg as dsql

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

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

### psycopg2

```
import aurora_dsycopg2 as dsql

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

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

### asyncpg

```
import asyncio
import aurora_dsycopg as dsql

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

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

### 연결 문자열

#### psycopg

```
import aurora_dsycopg as dsql

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

## psycopg2

```
import aurora_dsycopg2 as dsqldb

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

## asyncpg

```
import asyncpg
import aurora_dsycopg2_asyncpg as dsqldb

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

## 고급 구성

### psycopg2

```
import aurora_dsycopg2 as dsqldb

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

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

### psycopg2

```
import aurora_dsycopg2 as dsqldb

config = {
    'host': "your-cluster.dsql.us-east-1.on.aws",
```

```

    'region': "us-east-1",
    'user': "admin",
    "profile": "default",
    "token_duration_secs": "15",
}

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

```

## asyncpg

```

import asyncio
import aurora_dsql_asyncpg as dsql

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

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

```

## 구성 옵션

| 옵션   | Type   | 필수  | 설명                          |
|------|--------|-----|-----------------------------|
| host | string | 예   | DSQL 클러스터 호스트 이름 또는 클러스터 ID |
| user | string | 아니요 | DSQL 사용자 이름입니다. 기본값: admin  |

| 옵션                          | Type               | 필수  | 설명                                 |
|-----------------------------|--------------------|-----|------------------------------------|
| dbname                      | string             | 아니요 | 데이터베이스 이름. 기본값: postgres           |
| region                      | string             | 아니요 | AWS 리전(제공되지 않은 경우 호스트 이름에서 자동 감지됨) |
| port                        | int                | 아니요 | 기본값: 5432                          |
| custom_credentials_provider | CredentialProvider | 아니요 | 사용자 지정 AWS 자격 증명 공급자               |
| profile                     | string             | 아니요 | IAM 프로파일 이름입니다. 기본값: default       |
| token_duration_secs         | int                | 아니요 | 초 단위의 토큰 만료 시간                     |

DSQL에서 지원되지 않는 `asyncpg` 파라미터 `krbsrvname` 및 `gsslib`를 제외하고 기본 `psycopg`, `psycopg2` 및 `asyncpg` 라이브러리의 모든 표준 연결 옵션도 지원됩니다.

### 연결 풀링과 함께 Python용 Aurora DSQL 커넥터 사용

Python용 Aurora DSQL 커넥터는 `psycopg`, `psycopg2` 및 `asyncpg` 내장 연결 풀링과 함께 작동합니다. 커넥터는 연결 설정 중에 IAM 토큰 생성을 처리하여 연결 풀이 정상적으로 작동할 수 있도록 합니다.

## psycopg

psycopg의 경우 커넥터는 psycopg\_pool.ConnectionPool 생성자로 직접 전달할 수 있는 DSQLConnection ConnectionPool 이라는 연결 클래스를 구현합니다. 비동기 작업의 경우 DSQLAsyncConnection 버전도 있습니다.

```
from psycopg_pool import ConnectionPool as PsycopgPool

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

### 참고: 연결 max\_lifetime 구성

max\_lifetime 파라미터는 Aurora DSQL 데이터베이스에서 허용되는 최대 연결 기간이므로 3600초(1시간) 미만으로 설정해야 합니다. max\_lifetime을 낮게 설정하면 연결 풀이 연결 재활용을 사전에 관리할 수 있으므로 데이터베이스의 연결 제한 시간 오류를 처리하는 것보다 더 효율적입니다.

## psycopg2

psycopg2의 경우 커넥터는 psycopg2.pool.ThreadedConnectionPool에서 상속되는 AuroraDSQLThreadedConnectionPool이라는 클래스를 제공합니다. AuroraDSQLThreadedConnectionPool 클래스는 내부 \_connect 메서드만 재정의합니다. 나머지 구현은 psycopg2.pool.ThreadedConnectionPool에서 변경 없이 제공됩니다.

```
import aurora_dsql_psycopg2 as dsql

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

## asyncpg

asyncpg의 경우 커넥터는 asyncpg.Pool의 인스턴스를 반환하는 create\_pool 함수를 제공합니다.

```
import asyncio
import os

import aurora_dsqli_asyncpg as dsqli

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

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

## Authentication

커넥터는 DSQL 클라이언트 토큰 생성기를 사용하여 토큰을 생성하여 DSQL 인증을 자동으로 처리합니다. AWS 리전이 제공되지 않으면 제공된 호스트 이름에서 자동으로 구문 분석됩니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [사용 설명서](#)를 참조하세요.

### 관리자 및 일반 사용자

- "admin"이라는 사용자가 관리자 인증 토큰을 자동으로 사용
- 다른 모든 사용자는 관리자가 아닌 인증 토큰을 사용합니다.
- 토큰은 각 연결에 대해 동적으로 생성됩니다.

## 예제

전체 예제 코드는 아래 섹션에 표시된 예제를 참조하세요. 예제를 실행하는 방법에 대한 지침은 예제 README 파일을 참조하세요.

### psycopg

#### [예제 README](#)

| 설명                                | 예제                       |
|-----------------------------------|--------------------------|
| 기본 연결에 Python용 Aurora DSQL 커넥터 사용 | <a href="#">기본 연결 예제</a> |

| 설명                                      | 예제                                     |
|-----------------------------------------|----------------------------------------|
| 기본 비동기식 연결에 Python용 Aurora DSQL 커넥터 사용  | <a href="#">기본 비동기 연결 예제</a>           |
| 연결 풀링과 함께 Python용 Aurora DSQL 커넥터 사용    | <a href="#">연결 풀을 사용하는 기본 연결 예제</a>    |
|                                         | <a href="#">연결 풀을 사용하는 동시 연결 예제</a>    |
| 비동기 연결 풀과 함께 Python용 Aurora DSQL 커넥터 사용 | <a href="#">비동기 연결 풀을 사용한 기본 연결 예제</a> |

## psycopg2

### [예제 README](#)

| 설명                                   | 예제                                  |
|--------------------------------------|-------------------------------------|
| 기본 연결에 Python용 Aurora DSQL 커넥터 사용    | <a href="#">기본 연결 예제</a>            |
| 연결 풀링과 함께 Python용 Aurora DSQL 커넥터 사용 | <a href="#">연결 풀을 사용하는 기본 연결 예제</a> |
|                                      | <a href="#">연결 풀을 사용하는 동시 연결 예제</a> |

## asyncpg

### [예제 README](#)

| 설명                                | 예제                       |
|-----------------------------------|--------------------------|
| 기본 연결에 Python용 Aurora DSQL 커넥터 사용 | <a href="#">기본 연결 예제</a> |



| 설명                                   | 예제                                  |
|--------------------------------------|-------------------------------------|
| 연결 풀링과 함께 Python용 Aurora DSQL 커넥터 사용 | <a href="#">연결 풀을 사용하는 기본 연결 예제</a> |
|                                      | <a href="#">연결 풀을 사용하는 동시 연결 예제</a> |

## Go 커넥터를 사용하여 Aurora DSQL 클러스터에 연결

[Go용 Aurora DSQL 커넥터](#)는 [pgx](#)를 자동 IAM 인증으로 래핑합니다. 커넥터는 토큰 생성, SSL 구성 및 연결 관리를 처리하므로 애플리케이션 로직에 집중할 수 있습니다.

### 커넥터 정보

Aurora DSQL에는 기존 Go PostgreSQL 드라이버가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 기반 인증이 필요합니다. Go용 Aurora DSQL 커넥터는 IAM 토큰 생성을 처리하는 [pgx](#) 드라이버 위에 인증 계층을 추가하여 기존 [pgx](#) 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 합니다.

Aurora DSQL 인증이란 무엇인가요?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 커넥터는 AWS 자격 증명을 사용하여 인증 토큰을 생성하며 이러한 토큰에는 구성 가능한 수명이 있습니다.

Go용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성하도록 설계되었습니다.

### Go용 Aurora DSQL 커넥터의 이점

Go용 Aurora DSQL 커넥터를 사용하면 기존 [pgx](#) 워크플로를 계속 사용하면서 IAM 인증을 활성화할 수 있습니다.

- 자동 토큰 생성: 커넥터가 각 연결에 대해 IAM 토큰을 자동으로 생성
- 연결 풀링: 연결마다 자동으로 토큰을 생성하여 [pgxpool](#) 기본 지원
- 유연한 구성: 리전 자동 감지를 통해 전체 엔드포인트 또는 클러스터 ID 지원

- AWS 자격 증명 지원: AWS 프로파일 및 사용자 지정 자격 증명 공급자 지원

## 주요 기능

### 자동 토큰 관리

커넥터가 사전 확인된 자격 증명을 사용하여 새 연결마다 IAM 토큰을 자동으로 생성합니다.

### 연결 풀링

연결마다 자동으로 토큰을 생성하여 pgxpool을 통해 연결을 풀링합니다.

### 유연한 호스트 구성

자동 리전 감지를 통해 전체 클러스터 엔드포인트와 클러스터 ID를 모두 지원합니다.

### SSL 보안

SSL은 항상 verify-full 모드와 직접 TLS 협상으로 활성화됩니다.

## 사전 조건

- Go 1.24 이상
- AWS 자격 증명이 구성됨
- Aurora DSQL 클러스터

커넥터는 다음 순서로 자격 증명을 확인하는 [AWS SDK for Go v2 기본 자격 증명 체인](#)을 사용합니다.

1. 환경 변수(AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY)
2. 공유 자격 증명 파일(~/.aws/credentials)
3. 공유 구성 파일(~/.aws/config)
4. Amazon EC2/ECS/Lambda에 대한 IAM 역할

## 설치

Go 모듈을 사용하여 커넥터 설치:

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

## 빠른 시작

다음 예제에서는 연결 풀을 만들고 쿼리를 실행하는 방법을 보여줍니다.

```
package main

import (
    "context"
    "log"

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

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

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

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

## 구성 옵션

커넥터는 다음과 같은 구성 옵션을 지원합니다.

| Field | Type | 기본값  | 설명                    |
|-------|------|------|-----------------------|
| Host  | 문자열  | (필수) | 클러스터 엔드포인트 또는 클러스터 ID |

| Field             | Type          | 기본값        | 설명                                  |
|-------------------|---------------|------------|-------------------------------------|
| 리전                | 문자열           | (자동 탐지됨)   | AWS 리전, 호스트가 클러스터 ID인 경우 필수         |
| User              | 문자열           | "admin"    | 데이터베이스 사용자                          |
| 데이터베이스            | 문자열           | "postgres" | 데이터베이스 이름                           |
| Port              | int           | 5432       | 데이터베이스 포트                           |
| 프로필               | 문자열           | ""         | 자격 증명의 AWS 프로파일 이름                  |
| TokenDurationSecs | int           | 900(15분)   | 초 단위의 토큰 유효 기간(최대 허용: 1주, 기본값: 15분) |
| MaxConns          | int32         | 0          | 최대 풀 연결(0 = pgxpool 기본값)            |
| MinConns          | int32         | 0          | 최소 풀 연결(0 = pgxpool 기본값)            |
| MaxConnLifetime   | time.Duration | 55분        | 최대 연결 수명                            |

## 연결 문자열 형식

커넥터는 다음의 PostgreSQL 및 DSQL 연결 문자열 형식을 지원합니다.

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

지원되는 쿼리 파라미터:

- region - AWS 리전
- profile - AWS 프로파일 이름
- tokenDurationSecs - 토큰 유효 기간(초 단위)

예시:

```
// Full endpoint (region auto-detected)
```

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

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

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

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

## 고급 사용

### 호스트 구성

커넥터는 다음의 두 가지 호스트 형식을 지원합니다.

전체 엔드포인트(리전 자동 감지됨):

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

클러스터 ID(리전 필요):

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

### 풀 구성 튜닝

워크로드에 대한 연결 풀을 구성합니다.

```
pool, err := dsql.NewPool(ctx, dsql.Config{
    Host:           "your-cluster.dsql.us-east-1.on.aws",
    MaxConns:       20,
    MinConns:       5,
    MaxConnLifetime: time.Hour,
    MaxConnIdleTime: 30 * time.Minute,
```

```
HealthCheckPeriod: time.Minute,
}))
```

## 단일 연결 사용

간단한 스크립트의 경우 또는 연결 풀링이 필요하지 않은 경우:

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

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

## AWS 프로파일 사용

자격 증명에 대한 AWS 프로파일 지정:

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

## OCC 재시도

Aurora DSQL은 낙관적 동시성 제어(OCC)를 사용합니다. 두 개의 트랜잭션이 동일한 데이터를 수정할 경우, 먼저 커밋된 트랜잭션이 우선권을 가지며 두 번째 트랜잭션은 OCC 오류를 수신합니다.

occretry 패키지는 지수 백오프 및 지터를 사용한 자동 재시도 도우미를 제공합니다. 다음을 사용하여 설치합니다.

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

트랜잭션 쓰기에는 WithRetry를 사용합니다.

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

```

    if err != nil {
        return err
    }
    _, err = tx.Exec(ctx, "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
100, toID)
    return err
})

```

DDL 또는 단일 문에는 ExecWithRetry를 사용합니다.

```

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

```

### Important

WithRetry는 BEGIN/COMMIT/ROLLBACK을 내부적으로 관리합니다. 콜백은 트랜잭션을 수신하며, 데이터베이스 작업만 포함하고 재시도해도 안전해야 합니다.

## 예제

보다 포괄적인 예제 및 사용 사례는 [Go용 Aurora DSQL 커넥터 예제](#)를 참조하세요.

| 예제                                | 설명                                 |
|-----------------------------------|------------------------------------|
| <a href="#">example_preferred</a> | 권장: 동시 쿼리가 있는 연결 풀                 |
| <a href="#">transaction</a>       | BEGIN/COMMIT/ROLLBACK을 사용한 트랜잭션 처리 |
| <a href="#">occ_retry</a>         | 지수 백오프와 OCC 충돌 처리                  |
| <a href="#">connection_string</a> | 구성에 연결 문자열 사용                      |

## Node.js용 Aurora DSQL 커넥터

node-postgres용 Aurora DSQL 커넥터와 Postgres.js용 Aurora DSQL 커넥터는 애플리케이션이 IAM 자격 증명을 사용하여 Aurora DSQL로 인증할 수 있도록 node-postgres 및 Postgres.js 클라이언트의 기능을 확장하는 인증 플러그인입니다.

## node-postgres용 Aurora DSQL 커넥터

[node-postgres용 Aurora DSQL 커넥터](#)는 JavaScript/TypeScript 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합하는 [node-postgres](#)를 기반으로 구축된 Node.js 커넥터입니다.

Aurora DSQL 커넥터는 node-postgres의 클라이언트 및 풀 기능을 확장하는 인증 플러그인으로 설계되어 애플리케이션이 IAM 자격 증명을 사용하여 Amazon Aurora DSQL과 인증할 수 있도록 합니다.

### 커넥터 정보

Amazon Aurora DSQL은 PostgreSQL과 호환되는 클라우드 네이티브 분산 데이터베이스입니다. IAM 인증 및 시간 제한 토큰이 필요하지만 기존 Node.js 데이터베이스 드라이버에는 이러한 기본 지원이 없습니다.

node-postgres용 Aurora DSQL 커넥터는 node-postgres와 원활하게 작동하는 인증 미들웨어를 구현하여 이 격차를 해소합니다. 이 접근 방식을 통해 개발자는 자동화된 토큰 관리를 통해 Aurora DSQL 클러스터에 대한 안전한 IAM 기반 액세스를 확보하면서 기존 노드 포스트 코드를 유지할 수 있습니다.

### Aurora DSQL 인증이란 무엇입니까?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 인증 토큰은 AWS 자격 증명을 사용하여 생성되며 구성 가능한 수명이 있습니다.

node-postgres용 Aurora DSQL JDBC 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성하도록 설계되었습니다.

### 특성

- 자동 IAM 인증 - DSQL 토큰 생성 및 새로 고침 처리
- node-postgres 기반 - Node.js에 널리 사용되는 PostgreSQL 클라이언트를 활용합니다.
- 원활한 통합: 기존 node-postgres 연결 패턴과 호환
- 리전 자동 검색 - DSQL 클러스터 호스트 이름에서 AWS 리전 추출
- 전체 TypeScript 지원 - 전체 유형 안전 제공
- AWS 자격 증명 지원 - 다양한 AWS 자격 증명 공급자(기본값, 프로필 기반, 사용자 지정) 지원
- 연결 풀링 호환성 - 기본 제공 연결 풀링과 원활하게 작동



## 애플리케이션 예제

[예제](#)에는 node-postgres용 Aurora DSQL 커넥터를 사용하는 방법을 보여주는 샘플 애플리케이션이 포함되어 있습니다. 포함된 예제를 실행하려면 예제 [README](#)를 참조하세요.

## 빠른 시작 설명서

### 요구 사항

- Node.js 20+
- [Aurora DSQL 클러스터에 대한 액세스](#)
- 애플리케이션이 Aurora DSQL에 연결할 수 있도록 적절한 IAM 권한 설정
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

### 설치

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

### 피어 종속성

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

## 사용법

### 클라이언트 연결

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

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

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

### 풀 연결

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

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

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

## 고급 사용

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

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

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

## 구성 옵션

| 옵션       | Type   | 필수  | 설명                                 |
|----------|--------|-----|------------------------------------|
| host     | string | 예   | DSQL 클러스터 호스트 이름                   |
| username | string | 예   | DSQL 사용자 이름                        |
| database | string | 아니요 | 데이터베이스 이름                          |
| region   | string | 아니요 | AWS 리전(제공되지 않은 경우 호스트 이름에서 자동 감지됨) |

| 옵션                        | Type                                                  | 필수  | 설명                                 |
|---------------------------|-------------------------------------------------------|-----|------------------------------------|
| port                      | number                                                | 아니요 | 기본값: 5432                          |
| customCredentialsProvider | AwsCredentialIdentity / AwsCredentialIdentityProvider | 아니요 | 사용자 지정 AWS 자격 증명 공급자               |
| profile                   | string                                                | 아니요 | IAM 프로파일 이름입니다. 기본값은 "default"입니다. |
| tokenDurationSecs         | number                                                | 아니요 | 초 단위의 토큰 만료 시간                     |

[클라이언트/풀](#)의 다른 모든 파라미터가 지원됩니다.

## Authentication

커넥터는 DSQL 클라이언트 토큰 생성기를 사용하여 토큰을 생성하여 DSQL 인증을 자동으로 처리합니다. AWS 리전이 제공되지 않으면 제공된 호스트 이름에서 자동으로 구문 분석됩니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [사용 설명서](#)를 참조하세요.

## 관리자 및 일반 사용자

- "admin"이라는 사용자는 관리자 인증 토큰을 자동으로 사용
- 다른 모든 사용자는 일반 인증 토큰을 사용합니다.
- 토큰은 각 연결에 대해 동적으로 생성됩니다.

## Postgres.js용 Aurora DSQL 커넥터

[Postgres.js용 Aurora DSQL 커넥터](#)는 JavaScript 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합하는 [Postgres.js](#)를 기반으로 구축된 Node.js 커넥터입니다.

Postgres.js용 Aurora DSQL 커넥터는 Postgres.js 클라이언트 및 풀 기능을 확장하는 인증 플러그인으로 설계되어 애플리케이션이 IAM 자격 증명을 사용하여 Amazon Aurora DSQL과 인증할 수 있도록 합니다. 커넥터는 데이터베이스에 직접 연결되지 않지만 기본 Postgres.js 드라이버를 기반으로 원활한 IAM 인증을 제공합니다.

## 커넥터 정보

Amazon Aurora DSQL은 PostgreSQL 호환 애플리케이션에 고가용성과 확장성을 제공하는 분산된 SQL 데이터베이스 서비스입니다. Aurora DSQL에는 기존 Node.js 드라이버가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 기반 인증이 필요합니다.

Postgres.js용 Aurora DSQL 커넥터의 기본 개념은 Postgres.js 클라이언트 위에 IAM 토큰 생성을 처리하는 인증 계층을 추가하여, 사용자가 기존 Postgres.js 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 하는 것입니다.

Postgres.js용 Aurora DSQL 커넥터는 대부분의 Postgres.js 버전에서 작동합니다. 사용자는 Postgres.js를 직접 설치하여 자체 버전을 제공합니다.

Aurora DSQL 인증이란 무엇입니까?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 인증 토큰은 AWS 자격 증명을 사용하여 생성되며 구성 가능한 수명이 있습니다.

Postgres.js용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성하도록 설계되었습니다.

## 특성

- 자동 IAM 인증 - DSQL 토큰 생성 및 새로 고침 처리
- Postgres.js 기반 - Node.js용 빠른 PostgreSQL 클라이언트 활용
- 원활한 통합 - 기존 Postgres.js 연결 패턴과 호환
- 리전 자동 검색 - DSQL 클러스터 호스트 이름에서 AWS 리전 추출
- 전체 TypeScript 지원 - 전체 유형 안전 제공
- AWS 자격 증명 지원 - 다양한 AWS 자격 증명 공급자(기본값, 프로파일 기반, 사용자 지정) 지원
- 연결 풀링 호환성 - Postgres.js의 기본 제공 연결 풀링과 원활하게 작동

## 빠른 시작 설명서

### 요구 사항

- Node.js 20+
- [Aurora DSQL 클러스터에 대한 액세스](#)
- 애플리케이션이 Aurora DSQL에 연결할 수 있도록 적절한 IAM 권한 설정
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

### 설치

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

### 기본 사용법

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

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

// Execute queries
const result = await sql`SELECT current_timestamp`;
console.log(result);

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

### 호스트 대신 클러스터 ID 사용

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

## 연결 문자열

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

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

## 고급 구성

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

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

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

## 구성 옵션

| 옵션       | Type    | 필수  | 설명                               |
|----------|---------|-----|----------------------------------|
| host     | string  | 예   | DSQL 클러스터 호스트 이름 또는 클러스터 ID      |
| database | string? | 아니요 | 데이터베이스 이름                        |
| username | string? | 아니요 | 데이터베이스 사용자 이름(제공되지 않은 경우 관리자 사용) |

| 옵션                        | Type                           | 필수  | 설명                                 |
|---------------------------|--------------------------------|-----|------------------------------------|
| region                    | string?                        | 아니요 | AWS 리전(제공되지 않은 경우 호스트 이름에서 자동 감지됨) |
| customCredentialsProvider | AwsCredentialIdentityProvider? | 아니요 | 사용자 지정 AWS 자격 증명 공급자               |
| tokenDurationSecs         | number?                        | 아니요 | 초 단위의 토큰 만료 시간                     |

모든 표준 [Postgres.js 옵션](#)도 지원됩니다.

## Authentication

커넥터는 DSQL 클라이언트 토큰 생성기를 사용하여 토큰을 생성하여 DSQL 인증을 자동으로 처리합니다. AWS 리전이 제공되지 않으면 제공된 호스트 이름에서 자동으로 구문 분석됩니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [사용 설명서](#)를 참조하세요.

## 관리자 및 일반 사용자

- "admin"이라는 사용자는 관리자 인증 토큰을 자동으로 사용
- 다른 모든 사용자는 일반 인증 토큰을 사용합니다.
- 토큰은 각 연결에 대해 동적으로 생성됩니다.

## 샘플 사용

Postgres.js용 Aurora DSQL 커넥터를 사용하는 JavaScript 예제는 GitHub에서 확인할 수 있습니다. 예제 실행 방법에 대한 자세한 내용은 [예제 디렉터리](#)를 참조하세요.

| 설명                                             | 예제                           |
|------------------------------------------------|------------------------------|
| 여러 작업자의 테이블 생성, 만들기 및 읽기를 포함한 동시 쿼리를 사용한 연결 풀링 | <a href="#">연결 풀 예제(선호됨)</a> |
| 연결 풀링이 없는 CRUD 작업(테이블 만들기, 삽입, 선택, 삭제)         | <a href="#">연결 풀이 없는 예제</a>  |

## Ruby 커넥터를 사용하여 Aurora DSQL 클러스터에 연결

[Ruby용 Aurora DSQL 커넥터](#)는 Ruby 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합하는 [pg](#)를 기반으로 구축된 .NET 커넥터입니다.

이 커넥터는 토큰 생성, SSL 구성 및 연결 풀링을 처리하므로 애플리케이션 로직에 집중할 수 있습니다.

### 커넥터 정보

Amazon Aurora DSQL에는 기존 Ruby PostgreSQL 드라이버가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 인증이 필요합니다. Ruby용 Aurora DSQL 커넥터는 IAM 토큰 생성을 처리하는 pg gem 위에 인증 계층을 추가하여 기존 pg 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 합니다.

Aurora DSQL 인증이란 무엇인가요?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 커넥터는 AWS 자격 증명을 사용하여 인증 토큰을 생성하며 이러한 토큰에는 구성 가능한 수명이 있습니다.

Ruby용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성합니다.

### 특성

- 자동 IAM 인증 - Aurora DSQL 토큰 생성 및 새로 고침 처리



- pg 기반 - 널리 사용되는 Ruby용 PostgreSQL gem 래핑
- 원활한 통합 - 기존 pg gem 워크플로와 함께 작동
- 연결 풀링 - 최대 수명 적용으로 connection\_pool gem을 통한 지원 기본 제공
- 리전 자동 감지 - DSQL 클러스터 호스트 이름에서 AWS 리전 추출
- AWS 자격 증명 지원: AWS 프로파일 및 사용자 지정 자격 증명 공급자 지원
- OCC 재시도 - 지수 백오프를 사용하여 옵트인 낙관적 동시성 제어 재시도

## 애플리케이션 예시

전체 예제는 GitHub의 [예제 애플리케이션](#)을 참조하세요.

## 빠른 시작 설명서

### 요구 사항

- Ruby 3.1 이상
- [Aurora DSQL 클러스터에 대한 액세스](#)
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

## 설치

Gemfile에 추가합니다.

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

또는 직접 설치합니다.

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

## 사용법

### 풀 연결

```
require "aurora_dsql_pg"
```

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

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

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

pool.shutdown
```

## 단일 연결

간단한 스크립트의 경우 또는 연결 풀링이 필요하지 않은 경우:

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

## 고급 사용

### 호스트 구성

커넥터는 전체 클러스터 엔드포인트(리전 자동 감지)와 클러스터 ID(리전 필수)를 모두 지원합니다.

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

# Cluster ID (region required)
pool = AuroraDsql::Pg.create_pool(
```

```

    host: "your-cluster-id",
    region: "us-east-1"
  )

```

## AWS 프로파일

자격 증명에 대한 AWS 프로파일 지정:

```

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

```

## 연결 문자열 형식

커넥터는 다음의 PostgreSQL 연결 문자열 형식을 지원합니다.

```

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

```

지원되는 쿼리 파라미터: `region`, `profile`, `tokenDurationSecs`.

```

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

```

## OCC 재시도

Aurora DSQL은 낙관적 동시성 제어(OCC)를 사용합니다. 두 개의 트랜잭션이 동일한 데이터를 수정할 경우, 먼저 커밋된 트랜잭션이 우선권을 가지며 두 번째 트랜잭션은 OCC 오류를 수신합니다.

OCC 재시도가 옵트인되었습니다. `pool.with`에서 지수 백오프 및 지터를 사용하여 자동 재시도를 활성화하려면 풀을 생성할 때 `occ_max_retries`를 설정합니다.

```

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

```

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

### Warning

`pool.with`는 블록을 트랜잭션에 자동으로 래핑하지 않습니다. 쓰기 작업을 위해 `conn.transaction`을 직접적으로 호출해야 합니다. OCC 충돌 시 커넥터는 전체 블록을 다시 실행하므로 데이터베이스 작업만 포함하고 재시도해도 안전해야 합니다.

개별 직접 호출에 대한 재시도를 건너뛰려면 `retry_occ: false`를 전달합니다.

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

## 구성 옵션

| Field   | Type    | 기본값        | 설명                          |
|---------|---------|------------|-----------------------------|
| host    | 문자열     | (필수)       | 클러스터 엔드포인트 또는 클러스터 ID       |
| region  | 문자열     | (자동 탐지됨)   | AWS 리전, 호스트가 클러스터 ID인 경우 필수 |
| user    | 문자열     | "admin"    | 데이터베이스 사용자                  |
| 데이터베이스  | 문자열     | "postgres" | 데이터베이스 이름                   |
| 포트      | Integer | 5432       | 데이터베이스 포트                   |
| profile | 문자열     | nil        | 자격 증명의 AWS 프로파일 이름          |

| Field                | Type             | 기본값        | 설명                                       |
|----------------------|------------------|------------|------------------------------------------|
| token_duration       | Integer          | 900(15분)   | 초 단위의 토큰 유효 기간(최대 허용: 1주, 기본값: 15분)      |
| credentials_provider | Aws::Credentials | nil        | 사용자 지정 자격 증명 공급자                         |
| max_lifetime         | Integer          | 3300(55분)  | 최대 연결 수명(초 단위)                           |
| application_name     | 문자열              | nil        | application_name의 ORM 접두사                |
| 로거                   | Logger           | nil        | OCC 재시도 경고에 대한 로거                        |
| occ_max_retries      | Integer          | nil(비활성화됨) | pool.with 에서 최대 OCC 재시도 횟수, 설정 시 재시도 활성화 |

create\_pool은 ConnectionPool.new에 직접 전달하는 옵션 해시가 있는 pool: 키워드도 허용합니다. pool:을 생략하는 경우 커넥터의 기본값은 {size: 5, timeout: 5}입니다. 제공하는 키는 특정 기본값만 재정의합니다.

```
pool = AuroraDsql::Pg.create_pool(
  host: "your-cluster.dsdl.us-east-1.on.aws",
  pool: { size: 10, timeout: 10 }
)
```

## Authentication

커넥터는 AWS 자격 증명을 사용해 토큰을 생성하여 Aurora DSQL 인증을 자동으로 처리합니다. AWS 리전을 제공하지 않으면 커넥터가 호스트 이름에서 리전을 구문 분석합니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [Aurora DSQL에 대한 인증 및 권한 부여](#) 섹션을 참조하세요.

### 관리자와 일반 사용자

- "admin"이라는 사용자는 관리자 인증 토큰을 자동으로 사용
- 다른 모든 사용자는 일반 인증 토큰을 사용합니다.

- 커넥터는 각 연결에 대해 동적으로 토큰을 생성합니다.

## .NET 커넥터를 사용하여 Aurora DSQL 클러스터에 연결

[.NET용 Amazon Aurora DSQL 커넥터](#)는 .NET 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합하는 [Npgsql](#)을 기반으로 구축된 .NET 커넥터입니다.

이 커넥터는 토큰 생성, SSL 구성 및 연결 풀링을 처리하므로 애플리케이션 로직에 집중할 수 있습니다.

### 커넥터 정보

Amazon Aurora DSQL에는 기존 .NET PostgreSQL 드라이버가 기본적으로 지원하지 않는 시간 제한 토큰을 사용한 IAM 인증이 필요합니다. .NET용 Aurora DSQL 커넥터는 IAM 토큰 생성을 처리하는 Npgsql 위에 인증 계층을 추가하여 기존 Npgsql 워크플로를 변경하지 않고도 Aurora DSQL에 연결할 수 있도록 합니다.

Aurora DSQL 인증이란 무엇인가요?

Aurora DSQL에서 인증에는 다음이 포함됩니다.

- IAM 인증: 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다.
- 토큰 생성: 커넥터는 AWS 자격 증명을 사용하여 인증 토큰을 생성하며 이러한 토큰에는 구성 가능한 수명이 있습니다.

.NET용 Aurora DSQL 커넥터는 이러한 요구 사항을 이해하고 연결을 설정할 때 IAM 인증 토큰을 자동으로 생성합니다.

### 특성

- 자동 IAM 인증 - Aurora DSQL 토큰 생성 및 새로 고침 처리
- Npgsql 기반 - .NET용 주요 PostgreSQL 드라이버 래핑
- 원활한 통합 - 기존 Npgsql 워크플로와 함께 작동
- 연결 풀링 - 최대 수명 적용으로 NpgsqlDataSource를 통한 지원 기본 제공
- 리전 자동 감지 - DSQL 클러스터 호스트 이름에서 AWS 리전 추출
- AWS 자격 증명 지원: AWS 프로파일 및 사용자 지정 자격 증명 공급자 지원
- OCC 재시도 - 지수 백오프를 사용하여 옵트인 낙관적 동시성 제어 재시도

- SSL 적용 - 항상 `verify-full` 모드 및 직접 TLS 협상과 함께 SSL 사용

## 애플리케이션 예시

전체 예제는 GitHub의 [예제 애플리케이션](#)을 참조하세요.

## 빠른 시작 설명서

### 요구 사항

- .NET 8.0 이상
- [Aurora DSQL 클러스터에 대한 액세스](#)
- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

## 설치

패키지를 프로젝트에 추가합니다.

```
dotnet add package Amazon.AuroraDsql.Npgsql
```

## 사용법

### 풀 연결

```
using Amazon.AuroraDsql.Npgsql;

// Create a connection pool
await using var ds = await AuroraDsql.CreateDataSourceAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws",
    OccMaxRetries = 3
});

// Read
await using (var conn = await ds.OpenConnectionAsync())
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT 'Hello, DSQL!'";
    var greeting = await cmd.ExecuteScalarAsync();
    Console.WriteLine(greeting);
}
```

```

}

// Transactional write with OCC retry
await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();
    cmd.CommandText = "INSERT INTO users (id, name) VALUES (gen_random_uuid(), @name)";
    cmd.Parameters.AddWithValue("name", "Alice");
    await cmd.ExecuteNonQueryAsync();
});

```

## 단일 연결

간단한 스크립트의 경우 또는 연결 풀링이 필요하지 않은 경우:

```

await using var conn = await AuroraDsql.ConnectAsync(new DsqlConfig
{
    Host = "your-cluster.dsql.us-east-1.on.aws"
});

await using var cmd = conn.CreateCommand("SELECT 1");
await cmd.ExecuteScalarAsync();

```

## OCC 재시도

Aurora DSQL은 낙관적 동시성 제어(OCC)를 사용합니다. 두 개의 트랜잭션이 동일한 데이터를 수정할 경우, 먼저 커밋된 트랜잭션이 우선권을 가지며 두 번째 트랜잭션은 OCC 오류를 수신합니다.

OCC 재시도가 옵트인되었습니다. 지수 백오프 및 지터를 사용하여 자동 재시도를 활성화하려면 구성에서 `OccMaxRetries`를 설정합니다. 트랜잭션 쓰기에는 `WithTransactionRetryAsync`를 사용합니다.

```

await ds.WithTransactionRetryAsync(async conn =>
{
    await using var cmd = conn.CreateCommand();

    cmd.CommandText = "UPDATE accounts SET balance = balance - 100 WHERE id = @from";
    cmd.Parameters.AddWithValue("from", fromId);
    await cmd.ExecuteNonQueryAsync();

    cmd.CommandText = "UPDATE accounts SET balance = balance + 100 WHERE id = @to";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("to", toId);

```



```
await cmd.ExecuteNonQueryAsync();
});
```

DDL 또는 단일 문에는 `ExecWithRetryAsync`를 사용합니다.

```
await ds.ExecWithRetryAsync("CREATE TABLE IF NOT EXISTS users (id UUID PRIMARY KEY,
name TEXT)");
```

### Important

`WithTransactionRetryAsync`는 `BEGIN/COMMIT/ROLLBACK`을 내부적으로 관리하고 각 시도에 대해 새 연결을 엽니다. 콜백은 데이터베이스 작업만 포함하고 재시도해도 안전해야 합니다.

## 구성 옵션

커넥터는 `region` 및 `profile` 쿼리 파라미터가 있는 `postgres://` 및 `postgresql://` 연결 문자열도 허용합니다.

| Field                     | Type            | 기본값        | 설명                           |
|---------------------------|-----------------|------------|------------------------------|
| Host                      | string          | (필수)       | 클러스터 엔드포인트 또는 26자 클러스터 ID    |
| Region                    | string?         | (자동 탐지됨)   | AWS 리전, Host가 클러스터 ID인 경우 필수 |
| User                      | string          | "admin"    | 데이터베이스 사용자                   |
| Database                  | string          | "postgres" | 데이터베이스 이름                    |
| Port                      | int             | 5432       | 데이터베이스 포트                    |
| Profile                   | string?         | null       | 자격 증명의 AWS 프로파일 이름           |
| CustomCredentialsProvider | AWSCredentials? | null       | 사용자 지정 AWS 자격 증명 공급자         |

| Field                     | Type                                   | 기본값                 | 설명                                                                                |
|---------------------------|----------------------------------------|---------------------|-----------------------------------------------------------------------------------|
| TokenDurationSecs         | int?                                   | null(SDK 기본값, 900초) | 토큰 유효 기간(초 단위)                                                                    |
| OccMaxRetries             | int?                                   | null(비활성)           | 데이터 소스의 재시도 메서드에 대한 기본 최대 OCC 재시도 횟수                                              |
| OrmPrefix                 | string?                                | null                | application_name 앞에 추가되는 ORM 접두사                                                  |
| LoggerFactory             | ILoggerFactory?                        | null                | 재시도 경고 및 진단을 위한 로거 팩토리                                                            |
| ConfigureConnectionString | Action<NpgsqlConnectionStringBuilder>? | null                | 풀 설정을 재정의하거나 추가 Npgsql 연결 문자열 속성을 설정하는 콜백입니다. SSL 및 Enlist는 보안 불변수이며 재정의할 수 없습니다. |

## Authentication

커넥터는 AWS 자격 증명을 사용해 토큰을 생성하여 Aurora DSQL 인증을 자동으로 처리합니다. AWS 리전을 제공하지 않으면 커넥터가 호스트 이름에서 리전을 구문 분석합니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [Aurora DSQL에 대한 인증 및 권한 부여](#) 섹션을 참조하세요.

### 관리자와 일반 사용자

- "admin"이라는 사용자는 관리자 인증 토큰을 자동으로 사용
- 다른 모든 사용자는 일반 인증 토큰을 사용합니다.
- 커넥터는 각 연결에 대해 동적으로 토큰을 생성합니다.

## PostgreSQL 호환 클라이언트를 사용하여 Aurora DSQL에 액세스

Aurora DSQL은 [PostgreSQL 유선 프로토콜](#)을 사용합니다. AWS CloudShell, psql, DBeaver 및 DataGrip과 같은 다양한 도구 및 클라이언트를 사용하여 PostgreSQL에 연결할 수 있습니다. 다음 테

이블에는 Aurora DSQL이 일반적인 PostgreSQL 연결 파라미터를 매핑하는 방법이 요약되어 있습니다.

| PostgreSQL                  | Aurora DSQL             | 참고                                                                                                                                                                     |
|-----------------------------|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 역할(사용자 또는 그룹이라고도 함)         | 데이터베이스 역할               | Aurora DSQL은 admin이라는 역할을 생성합니다. 사용자 지정 데이터베이스 역할을 생성할 경우 클러스터에 연결할 때 인증을 위해 admin 역할을 사용하여 IAM 역할과 연결해야 합니다. 자세한 내용은 <a href="#">데이터베이스 역할 및 IAM 인증 사용</a> 섹션을 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.                                                                                                                                                     |
| 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 <a href="#">Amazon Aurora DSQL에서 인증 토큰 생성</a> 를 참조하세요.                                                              |

연결할 때 Aurora DSQL에는 기존 암호 대신 서명된 IAM [인증 토큰](#)이 필요합니다. 이러한 임시 토큰은 AWS 서명 버전 4를 사용하여 생성되며 연결 설정 중에만 사용됩니다. 연결되면 세션이 종료되거나 클라이언트 연결이 끊어질 때까지 세션이 활성 상태로 유지됩니다.

만료된 토큰으로 새 세션을 열려고 하면 연결 요청이 실패하고 새 토큰을 생성해야 합니다. 자세한 내용은 [Amazon Aurora DSQL에서 인증 토큰 생성](#) 섹션을 참조하세요.

## SQL 클라이언트를 사용하여 Aurora DSQL에 액세스

Aurora DSQL은 클러스터에 연결하기 위해 여러 PostgreSQL 호환 클라이언트를 지원합니다. 다음 섹션에서는 AWS CloudShell 또는 로컬 명령줄과 함께 PostgreSQL을 사용하여 연결하는 방법과 DBeaver 및 JetBrains DataGrip과 같은 GUI 기반 도구를 설명합니다. 각 클라이언트에는 이전 섹션에 설명된 대로 유효한 인증 토큰이 필요합니다.

### 주제

- [DBeaver를 사용하여 Aurora DSQL에 액세스](#)
- [JetBrains DataGrip을 사용하여 Aurora DSQL에 액세스](#)
- [PostgreSQL 대화형 터미널\(psql\)을 사용하여 Aurora DSQL에 액세스](#)
- [SQLTools용 Aurora DSQL 드라이버 사용](#)
- [문제 해결](#)

## DBeaver를 사용하여 Aurora DSQL에 액세스

DBeaver는 JDBC 드라이버가 있는 데이터베이스를 관리하는 데 사용할 수 있는 범용 SQL 클라이언트입니다. 강력한 데이터 보기, 편집 및 관리 기능 덕분에 개발자와 데이터베이스 관리자 사이에서 널리 사용됩니다. DBeaver의 클라우드 연결 옵션을 사용하면 기본적으로 DBeaver를 Aurora DSQL에 연결할 수 있습니다.

### DBeaver Pro

DBeaver PRO 제품은 버전 25.3부터 Aurora DSQL과의 기본 통합을 제공합니다. [DBeaver 설명서](#)의 지침에 따라 Aurora DSQL 클러스터에 연결합니다.

### DBeaver Community Edition

DBeaver Community Edition은 무료 오픈 소스 버전입니다. 설치 지침은 [다운로드 페이지](#)를 참조하십시오. DBeaver Community Edition에서 DSQL에 연결하려면 [DBeaver용 Aurora DSQL 플러그인](#)을 설치해야 합니다.

[DBeaver용 Aurora DSQL 플러그인](#)은 [JDBC용 Aurora DSQL 커넥터](#)를 기반으로 구축되었으며 Aurora DSQL 클러스터에 대한 IAM 인증을 활성화합니다. DBeaver UI를 통해 편리하게 설치되므로 토큰 생성 코드를 작성하거나 유효한 IAM 토큰을 수동으로 제공할 필요가 없으므로 기존 사용자 생성 암호와 관련된 보안 위험을 제거하면서 인증을 간소화할 수 있습니다.

## 특성

- IAM 인증 지원: 안전하고 암호 없는 인증을 위해 AWS IAM 자격 증명을 사용하여 Aurora DSQL 클러스터에 연결
- 자동 드라이버 관리: JDBC용 Aurora DSQL 커넥터를 원활하게 설치하고 구성합니다.
- 유연한 연결 옵션: 호스트 기반 또는 JDBC URL 기반 연결 구성 중에서 선택

## DBeaver 설치를 위한 Aurora DSQL 플러그인

1. DBeaver를 연 상태에서 드롭다운 메뉴 도움말 → 새 소프트웨어 설치로 이동합니다.
2. 추가를 클릭하여 새 리포지토리를 추가합니다.
3. 입력:
  - 이름: Aurora DSQL Plugin
  - 위치: <https://awslabs.github.io/aurora-dsql-dbeaver-plugin/update-site/>
4. JDBC용 Aurora DSQL 커넥터 확인
5. 다음을 클릭하고 라이선스를 수락한 후 설치를 완료합니다.
6. 메시지가 표시되면 DBeaver 다시 시작

## Aurora DSQL 연결 생성

1. 새 데이터베이스 연결을 클릭합니다.
2. Aurora DSQL 선택
3. 다음 필드에 대해 사용자 인터페이스 텍스트 입력을 활성화하려면 서버에서 연결 기준 설정에 대해
  - 호스트:
    - 다음 중 하나를 선택합니다.
      - 엔드포인트: DSQL 클러스터 엔드포인트
      - 사용자 이름: DSQL 사용자 이름(예: admin)
      - AWS 프로파일: 기본값 예 - 특정 프로파일 지정되지 않은 경우 사용되는 표준 프로파일
      - AWS 리전(선택 사항): DSQL 클러스터가 있는 리전과 일치해야 합니다. 그렇지 않으면 인증이 실패합니다.
  - URL
    - 다음 형식의 JDBC URL:

```
jdbc:aws-dsql:postgresql://{cluster_endpoint}/{database}?
user=admin&profile=default&region=us-east-1
```

- 참고: 이 모드에서는 URL 입력만 활성화됩니다. JDBC 연결 문자열에 파라미터를 추가하려면 ?로 시작하는 URL 쿼리 파라미터 형식을 첫 번째 파라미터로 사용하고 후속 파라미터에 &를 추가합니다.
4. 연결 테스트를 클릭하여 Aurora DSQL 연결이 작동하는지 확인합니다.
  5. 완료를 클릭합니다.

## 문제 해결

### Windows Trust Store 문제

Windows 사용자는 Maven Central에서 JDBC 드라이버용 Aurora DSQL 커넥터를 다운로드하는 데 문제가 발생할 수 있습니다.

원인: Windows Trust Store에는 Maven Central 리포지토리에 액세스하는 데 필요한 인증서가 포함되지 않을 수 있습니다.

해결 방법:

1. DBeaver를 "관리자"로 실행
2. 이 설정 선택 취소 - Windows > 기본 설정 > 연결 > "Windows Trust Store 사용"

### 드라이버 누락 오류

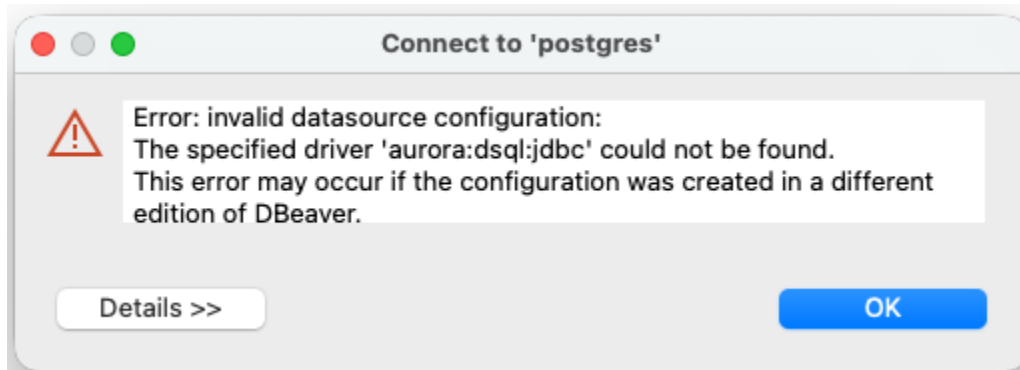
누락된 드라이버 아이콘 또는 연결 오류가 표시되면 현재 DBeaver 버전에 Aurora DSQL(Community 플러그인)이 설치되지 않을 수 있습니다. 아래 오류의 몇 가지 예와 이를 수정하는 방법을 참조하세요.

- 누락된 드라이버로 새 연결 생성:



aurora:dsql:jdbc

- 드라이버 없이 연결 시도:



원인: 여러 DBeaver 버전이 설치된 경우 연결 설정은 공유되지만 드라이버는 애플리케이션별로 설치됩니다.

해결 방법: 위의 설치 단계에 따라 Aurora DSQL(Community 플러그인)을 다시 설치합니다.

#### Important

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

## 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 웹 사이트의 [다운로드 페이지](#)로 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 5432를 사용합니다.

데이터베이스 - Aurora DSQL은 PostgreSQL 기본값인 postgres를 사용합니다.

인증 - User & Password 를 선택합니다.

사용자 이름 - admin을 입력합니다.

암호 - [토큰을 생성](#)하여 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 and PGSSLMODE=verify-full를 지원합니다. Aurora DSQL은 서버 측에서 SSL 통신을 적용하고 SSL이 아닌 연결을 거부합니다. verify-full 옵션의 경우 SSL 인증서를 로컬에 설치해야 합니다. 자세한 내용은 [SSL/TLS 인증서](#)를 참조하세요.

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 Aurora DSQL 데이터베이스에 적용되지 않습니다. 이러한 화면은 액세스 가능하지만 데이터베이스에 연결된 실제 세션에 대한 신뢰할 수 있는 정보를 제공하지 않습니다.

## PostgreSQL 대화형 터미널(psql)을 사용하여 Aurora DSQL에 액세스

### AWS CloudShell을 사용하여 PostgreSQL 대화형 터미널(psql)에서 Aurora DSQL에 액세스

다음 절차에 따라 PostgreSQL 대화형 터미널을 사용하여 AWS CloudShell에서 Aurora DSQL에 액세스합니다. 자세한 내용은 [AWS CloudShell란 무엇인가요?](#)를 참조하세요.

AWS CloudShell을 사용하여 연결

1. [Aurora DSQL 콘솔](#)에 로그인합니다.



2. CloudShell에서 열려있는 클러스터를 선택합니다. 클러스터를 아직 생성하지 않은 경우 [1단계: Aurora DSQL 단일 리전 클러스터 생성](#) 또는 [다중 리전 클러스터 생성](#)의 단계를 따릅니다.
3. 쿼리 편집기로 연결을 선택한 다음 CloudShell로 연결을 선택합니다.
4. admin으로 연결할지, 아니면 [사용자 지정 데이터베이스 역할](#)로 연결할지를 선택합니다.
5. CloudShell에서 시작을 선택하고 다음 CloudShell 대화 상자에서 실행을 선택합니다.

## 로컬 CLI를 사용하여 PostgreSQL 대화형 터미널(psql)을 사용하여 Aurora DSQL에 액세스

터미널 기반 프론트 엔드에서 PostgreSQL 유틸리티인 psql을 사용하여 쿼리에 대화식으로 입력하고, PostgreSQL에 발급하고, 쿼리 결과를 볼 수 있습니다.

### Note

쿼리 응답 시간을 개선하려면 PostgreSQL 버전 17 클라이언트를 사용합니다. 다른 환경에서 CLI를 사용하는 경우 Python 버전 3.8+ 및 psql 버전 14+를 수동으로 설정해야 합니다.

[PostgreSQL Downloads](#) 페이지에서 운영 체제의 설치 프로그램을 다운로드합니다. psql에 대한 자세한 내용은 PostgreSQL 웹 사이트의 [PostgreSQL 클라이언트 애플리케이션](#)을 참조하세요.

AWS CLI가 이미 설치되어 있는 경우 다음 예시를 사용하여 클러스터에 연결합니다.

```
# 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 dsq1 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
```

## SQLTools용 Aurora DSQL 드라이버 사용

SQLTools용 Aurora DSQL 드라이버는 SQLTools와 통합되는 Amazon Aurora DSQL용 Visual Studio Code 확장 프로그램입니다. 이를 통해 개발자는 VS Code에서 직접 Aurora DSQL 데이터베이스에 연결하고 쿼리할 수 있습니다. 드라이버는 [Visual Studio Marketplace](#) 및 [Open VSX Registry](#)에서 설치할 수 있습니다. Kiro, Cursor 및 기타 VSCode 기반 IDE는 [Open VSX 레지스트리](#)를 사용하여 이 페이지에 설명된 표준 설치 절차에 따라 드라이버를 설치할 수 있습니다.

### 특성

- 자동 IAM 인증
- 스키마, 테이블 검색 및 SQL 쿼리 실행과 같은 표준 데이터베이스 작업입니다.

### 설치

1. 확장 뷰를 엽니다.
2. "SQLTools용 Aurora DSQL 드라이버"를 검색합니다.
3. "설치"를 클릭합니다.

### 참고:

[SQLTools 확장](#)이 아직 없는 경우 자동으로 설치됩니다.

### Authentication

Aurora DSQL에서 모든 연결은 시간 제한 토큰이 있는 IAM 기반 인증을 사용합니다. 드라이버는 [Aurora DSQL Connector for node-postgres](#)를 사용하여 Aurora DSQL 인증을 자동으로 처리합니다.

Aurora DSQL의 인증에 대한 자세한 내용은 [사용 설명서](#)를 참조하세요.

### Aurora DSQL 연결 생성

#### 사전 조건

- AWS CLI, 환경 변수 또는 IAM 역할을 통해 AWS 자격 증명 구성

## 단계

1. 왼쪽 사이드바에서 SQLTools 아이콘을 클릭합니다.
2. SQLTools 창에서 연결 위에 마우스를 올려 놓고 새 연결 추가 아이콘을 클릭합니다.
3. SQLTools 설정 탭의 목록에서 Aurora DSQL 드라이버를 선택합니다.
4. 연결 파라미터를 입력합니다.
  - AWS 리전
    - 선택 사항 - 리전은 Aurora DSQL 클러스터 엔드포인트에서 구문 분석됩니다.
    - DSQL 클러스터 필드에 클러스터 ID만 지정된 경우 필요합니다.
  - AWS Profile(AWS 프로필)
    - 토큰 생성에 사용됩니다.
    - 지정되지 않은 경우 기본 프로필을 사용합니다.
5. "연결 테스트 버튼"을 클릭하여 연결을 테스트합니다.
6. 연결 저장을 클릭합니다.

## 문제 해결

### SQL 클라이언트의 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 연결의 암호 필드에 제공해야 합니다. 새 세션을 열려고 하면(예: 새 테이블 나열 또는 새 SQL 콘솔 열기) 새 인증 시도가 강제로 수행됩니다. 연결 설정에 구성된 인증 토큰이 더 이상 유효하지 않으면 새 세션이 실패하고 이전에 연 모든 세션이 무효화됩니다. 기본적으로 15분으로 설정할 수 있고 최대 값인 7일로 설정할 수 있는 `expires-in` 옵션을 사용하여 IAM 인증 토큰의 기간을 선택할 때는 이 점에 유의하세요.

또한 Aurora DSQL 설명서의 [문제 해결](#) 섹션을 참조하세요.

## Amazon Aurora DSQL 클러스터 연결 도구

Aurora DSQL은 여러 타사 데이터베이스 드라이버 및 ORM 라이브러리와 호환됩니다. AWS은 Aurora DSQL 작업을 간소화하는 두 가지 유형의 도구를 제공합니다.

- [커넥터](#) - IAM 토큰 생성을 자동으로 처리하도록 데이터베이스 드라이버를 확장하는 인증 플러그인입니다. 데이터베이스 드라이버로 직접 작업할 때는 커넥터를 사용합니다.
- 어댑터 및 언어 - IAM 인증 및 향상된 Aurora DSQL 호환성을 제공하는 특정 ORM 프레임워크에 대한 확장입니다. 지원되는 ORM 프레임워크로 작업할 때는 어댑터를 사용합니다.

## Aurora DSQL 어댑터 및 방언

다음 표에는 Aurora DSQL에 사용할 수 있는 어댑터 및 언어가 나와 있습니다.

| 프로그래밍 언어 | ORM/프레임워크    | 리포지토리 링크                                                                                                                                                            |
|----------|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Java     | Hibernate    | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/java/hibernate">https://github.com/aws-labs/aurora-dsql-orms/tree/main/java/hibernate</a>           |
| Python   | Django       | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/django">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/django</a>             |
| Python   | SQLAlchemy   | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/sqlalchemy">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/sqlalchemy</a>     |
| Python   | Tortoise ORM | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/tortoise-orm">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/tortoise-orm</a> |

## 데이터베이스 드라이버 샘플

다음 표에는 타사 데이터베이스 드라이버를 사용하여 Aurora DSQL에 연결하기 위한 샘플 코드가 나와 있습니다.

| 프로그래밍 언어  | 드라이버   | 샘플 리포지토리 링크                                                                                                                                                         |
|-----------|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C++       | libpq  | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/libpq">https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/libpq</a>         |
| C# (.NET) | Npgsql | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/npgsql">https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/npgsql</a> |

| 프로그래밍 언어   | 드라이버                       | 샘플 리포지토리 링크                                                                                                                                                                               |
|------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Go         | pgx                        | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/pgx">https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/pgx</a>                                     |
| Java       | HikariCP + pgJDBC          | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc">https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc</a>                           |
| JavaScript | node-postgres (AWS Lambda) | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/lambda">https://github.com/aws-samples/aurora-dsql-samples/tree/main/lambda</a>                                     |
| JavaScript | node-postgres              | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/node-postgres">https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/node-postgres</a> |
| JavaScript | Postgres.js                | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js">https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js</a>     |
| Python     | asyncpg                    | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/asyncpg">https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/asyncpg</a>                     |
| Python     | Psycopg                    | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg">https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg</a>                     |
| Python     | Psycopg2                   | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg2">https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg2</a>                   |
| Ruby       | pg                         | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/ruby-pg">https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/ruby-pg</a>                         |

| 프로그래밍 언어 | 드라이버 | 샘플 리포지토리 링크                                                                                                                                                 |
|----------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Rust     | SQLx | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/sqlx">https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/sqlx</a> |

## ORM 및 프레임워크 샘플

다음 표에는 Aurora DSQL에서 타사 ORM 라이브러리 및 프레임워크를 사용하기 위한 샘플 코드가 나와 있습니다.

| 프로그래밍 언어 | ORM/프레임워크   | 샘플 리포지토리 링크                                                                                                                                                                                                     |
|----------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Java     | Hibernate   | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/java/hibernate/examples/pet-clinic-app">https://github.com/aws-labs/aurora-dsql-orms/tree/main/java/hibernate/examples/pet-clinic-app</a>       |
| Java     | Liquibase   | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/liquibase">https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/liquibase</a>                                           |
| Java     | Spring Boot | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/spring_boot">https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/spring_boot</a>                                       |
| Python   | Django      | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/django/examples/pet-clinic-app">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/django/examples/pet-clinic-app</a>         |
| Python   | SQLAlchemy  | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/sqlalchemy/examples/pet-clinic-app">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/sqlalchemy/examples/pet-clinic-app</a> |

| 프로그래밍 언어   | ORM/프레임워크    | 샘플 리포지토리 링크                                                                                                                                                                         |
|------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Python     | Tortoise ORM | <a href="https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/tortoise-orm/example">https://github.com/aws-labs/aurora-dsql-orms/tree/main/python/tortoise-orm/example</a> |
| Ruby       | Rails        | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails">https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails</a>                       |
| TypeScript | Prisma       | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/prisma">https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/prisma</a>         |
| TypeScript | Sequelize    | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize">https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize</a>   |
| TypeScript | TypeORM      | <a href="https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm">https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm</a>     |

## Aurora DSQL로 데이터 로드

기존 데이터베이스에서 마이그레이션하든, Amazon Simple Storage Service에서 파일을 가져오든, 로컬 시스템에서 데이터를 로드하든, Aurora DSQL은 데이터를 가져올 수 있는 여러 접근 방식을 제공합니다. 이 섹션에서는 기가바이트에서 수백 테라바이트에 이르는 모든 규모의 데이터 로드와 권장되는 도구와 기술을 다룹니다.

### 로드 접근 방식 선택

Aurora DSQL은 표준 PostgreSQL 데이터 로드 명령을 지원하지만 대규모로 데이터를 효율적으로 로드하려면 병렬화, 연결 관리 및 오류 복구를 처리해야 합니다. 다음 표에는 사용 가능한 여러 가지 옵션이 요약되어 있습니다.

| 접근 방식                                                           | 최적의 용도                                   | 고려 사항                                                              |
|-----------------------------------------------------------------|------------------------------------------|--------------------------------------------------------------------|
| Aurora DSQL 로더 - Aurora DSQL을 사용할 때 삽입을 쉽게 병렬화할 수 있는 오픈 소스 유틸리티 | 대부분의 데이터 로드 시나리오, 특히 마이그레이션 및 대량 가져오기    | 병렬화, 연결 풀링, 충돌 해결 및 IAM 인증을 자동으로 처리합니다. 소스 코드 또는 바이너리로 사용할 수 있습니다. |
| PostgreSQL <code>\copy</code> - 클라이언트 측 <code>psql</code> 메타 명령 | <code>psql</code> 을 통해 이미 연결된 경우의 간단한 로드 | 클라이언트에서 파일을 읽고 연결을 통해 데이터를 스트리밍합니다. 사용자가 직접 병렬화를 관리합니다.            |
| <b>INSERT</b> 트랜잭션 - 표준 SQL DML                                 | 소규모 데이터세트 또는 애플리케이션 기반 삽입                | 가장 간단한 접근 방식이지만 대량 데이터의 경우 가장 느림                                   |

대부분의 데이터 로드 작업에서 Aurora DSQL 로더를 사용하세요. 이는 여러 연결에 걸친 병렬 실행과 실패한 작업의 자동 재시도를 포함하여 분산 데이터베이스에 데이터를 로드하는 데 따른 운영상의 복잡성을 처리합니다.

## Aurora DSQL 로더

[Aurora DSQL 로더](#)는 Aurora DSQL 클러스터에 데이터를 효율적으로 로드하도록 설계된 오픈 소스 명령줄 유틸리티입니다. 연결 풀링을 관리하고, 여러 워커 간의 데이터 전송을 병렬화하고, 충돌을 자동으로 처리하고 재시도합니다.

### 주요 기능

Aurora DSQL 로더는 다음 기능을 제공합니다.

#### 병렬 로드

구성 가능한 워커 스레드를 통해 여러 연결에서 동시에 데이터를 로드하여 성능을 향상시킬 수 있습니다.



## 연결 풀링

IAM 인증 및 연결 수명 주기를 자동으로 처리하며 Aurora DSQL 클러스터에 대한 연결 풀을 관리합니다.

## 다중 파일 형식 지원

CSV(쉼표로 구분된 값), TSV(탭으로 구분된 값) 및 Apache Parquet 열 기반 형식을 지원합니다. 이 로더는 소스 URI 확장명을 기반으로 파일 형식을 자동으로 감지합니다.

## 자동 스키마 추론

--if-not-exists 플래그와 함께 사용할 경우 로더가 데이터에 따라 적절한 열 유형으로 테이블을 자동으로 생성할 수 있습니다.

## 충돌 처리

대상 테이블에 고유한 제약 조건이 있는 경우 --on-conflict 옵션을 사용하여 로더가 충돌을 처리하는 방법(중복 건너뛰기, 레코드 업서트 또는 오류 반환)을 구성합니다.

## 내결함성

자동 재시도 및 작업 재개 기능을 사용하면 중단된 로드가 처음부터 다시 시작되지 않고 중단된 지점부터 계속되도록 할 수 있습니다.

## 로컬 및 S3 소스

로컬 파일 시스템 경로에서 데이터를 로드하거나 S3 URI를 사용하여 Amazon S3 버킷에서 직접 데이터를 로드할 수 있습니다.

## 사전 조건

Aurora DSQL 로더를 사용하기 전에 다음을 갖추었는지 확인하세요.

- 유효한 엔드포인트가 있는 활성 Aurora DSQL 클러스터.
- AWS CLI(aws configure), AWS Single Sign-On(aws sso login) 또는 IAM 역할을 통해 구성된 AWS 자격 증명.
- IAM 권한: Aurora DSQL 클러스터에 대한 `dsql:DbConnectAdmin` 또는 `dsql:DbConnect`.
- S3 소스의 경우 소스 버킷에서 읽을 수 있는 적절한 권한.

## 설치

[GitHub 릴리스 페이지](#)에서 최신 릴리스를 다운로드합니다. 일반적인 플랫폼용으로 미리 빌드된 바이너리가 제공됩니다. 소스에서 빌드하는 방법에 대한 지침은 [Aurora DSQL 로더 리포지토리](#)를 참조하세요.

## 사용 예제

다음 예시에서는 Aurora DSQL 로더의 일반적인 사용 사례를 보여줍니다.

### Example로컬 CSV 파일 로드

이 예시에서는 로컬 파일 시스템에서 기존 테이블로 CSV 파일을 로드합니다.

```
aurora-dsql-loader load \  
  --endpoint cluster-id.dsql.region.on.aws \  
  --source-uri data.csv \  
  --table my_table
```

### Example Amazon S3에서 데이터 로드

이 예시에서는 Amazon S3 버킷에서 Parquet 파일을 로드합니다.

```
aurora-dsql-loader load \  
  --endpoint cluster-id.dsql.region.on.aws \  
  --source-uri s3://my-bucket/data.parquet \  
  --table my_table
```

### Example자동 테이블 생성

이 예시에서는 데이터 스키마를 기반으로 새 테이블을 자동으로 생성합니다.

```
aurora-dsql-loader load \  
  --endpoint cluster-id.dsql.region.on.aws \  
  --source-uri data.csv \  
  --table my_table \  
  --if-not-exists
```

### Example로드 전 검증

이 예시에서는 데이터를 실제로 로드하지 않고 구성을 검증합니다.

```
aurora-dsql-loader load \  
  --dry-run
```

```
--endpoint cluster-id.dsql.region.on.aws \
--source-uri data.csv \
--table my_table \
--dry-run
```

## Example 중단된 로드 재개

로드 작업이 중단되면 이전 실행의 작업 ID를 사용하여 재개할 수 있습니다.

```
aurora-dsql-loader load \
--endpoint cluster-id.dsql.region.on.aws \
--source-uri data.csv \
--table my_table \
--resume-job-id job-id \
--manifest-dir ./loader-state
```

### Note

재개 시 로더는 이미 완료된 대부분의 작업을 건너뛰지만 일부 레코드를 다시 시도할 수 있습니다. 대상 테이블에 고유한 제약 조건이 있는 경우 `--on-conflict` 옵션을 사용하여 중복을 처리합니다(예: 중복을 건너뛰려면 `DO NOTHING`, upsert를 수행하려면 `DO UPDATE`).

## 명령줄 옵션

Aurora DSQL 로더는 다음 명령줄 옵션을 지원합니다.

### --endpoint

(필수) Aurora DSQL 클러스터 엔드포인트입니다. 예시: *cluster-id*.dsql.*region*.on.aws

### --source-uri

(필수) 데이터 파일의 경로입니다. 로컬 파일 경로 또는 S3 URI(예: *s3://bucket-name/file.parquet*)일 수 있습니다.

### --table

(필수) Aurora DSQL 데이터베이스의 대상 테이블 이름입니다.

### --if-not-exists

(선택 사항) 대상 테이블이 없는 경우 자동으로 생성됩니다. 로더는 데이터에서 스키마를 유추합니다.

**--dry-run**

(선택 사항) 실제로 데이터베이스에 로드하지 않고 구성 및 데이터를 검증합니다.

**--resume-job-id**

(선택 사항) 지정된 작업 ID를 사용하여 이전에 중단된 로드 작업을 재개합니다.

**--manifest-dir**

(선택 사항) 작업 재개에 사용되는 작업 상태 및 매니페스트를 저장하기 위한 디렉터리입니다.

**--on-conflict**

(선택 사항) 대상 테이블에서 고유한 제약 조건을 위반하는 행을 삽입할 때 충돌을 처리하는 방법을 지정합니다. 유효한 값은 `error`(오류 반환), `do-nothing`(중복된 행 건너뛰기) 또는 `do-update`(기존 행을 새 값으로 업데이트)입니다.

옵션 및 추가 구성 파라미터의 전체 목록을 보려면 다음을 실행합니다.

```
aurora-dsql-loader load --help
```

## 모범 사례

- 검증을 위해 모의 실행 사용 - 데이터를 프로덕션 테이블에 로드하기 전에 항상 `--dry-run`을 사용하여 로드 구성을 테스트합니다.
- 재개를 위한 고유 제약 조건 정의 - 중단된 로드를 재개해야 하는 경우 대상 테이블에 고유한 제약 조건을 정의하고 `--on-conflict` 옵션을 사용하여 이미 로드된 레코드를 처리합니다.
- 대규모 데이터세트에 Parquet 사용 - Parquet의 열 형식은 일반적으로 CSV 또는 TSV에 비해 대규모 데이터세트에서 더 나은 압축과 더 빠른 로드를 제공합니다.
- 매니페스트 디렉터리 보존 - 로드가 성공적으로 완료되었음을 확인할 때까지 로드 작업에 대한 매니페스트 디렉터리를 유지하여 필요한 경우 재개할 수 있도록 합니다.
- 가능한 경우 테이블 사전 생성 - 데이터를 로드하기 전에 열 데이터 유형 및 프라이머리 키를 명시적으로 지정하여 대상 테이블을 정의합니다. 사전 생성된 스키마를 사용하면 유형 정밀도와 인덱싱을 제어할 수 있기 때문에 일반적으로 자동 추론된 스키마에 비해 쿼리 성능이 향상됩니다.

## 문제 해결

### 인증 오류

AWS 자격 증명이 올바르게 구성되어 있고 IAM ID에 대상 클러스터에 대해 필요한 `dsql:DbConnect` 또는 `dsql:DbConnectAdmin` 권한이 있는지 확인합니다.

### S3 액세스 오류

IAM ID에 소스 버킷 및 객체에 대한 적절한 S3 읽기 권한이 있는지 확인합니다.

### 스키마 추론 오류

`--if-not-exists`를 사용할 때는 데이터 파일의 열 유형이 일관적인지 확인합니다. 열의 유형이 혼합되면 스키마 추론이 실패할 수 있습니다.

### 재개 시 중복 키 오류

로드를 재개할 때 중복 키 오류가 발생할 경우 로더가 `ON CONFLICT DO NOTHING`을 사용하여 이미 로드된 레코드를 건너뛴 수 있도록 대상 테이블에 고유한 제약 조건을 추가합니다.

추가 문제 해결 정보는 [Aurora DSQL 로더 GitHub 리포지토리](#)를 참조하세요.

## 마이그레이션 경로

다음 섹션에서는 일반적인 소스 시스템에서 Aurora DSQL로 데이터를 마이그레이션하는 방법을 설명합니다.

### PostgreSQL에서 마이그레이션

기존 PostgreSQL 데이터베이스에서 Aurora DSQL로 데이터를 마이그레이션하는 방법:

1. PostgreSQL에서 CSV 또는 Parquet 형식으로 데이터를 내보냅니다. PostgreSQL `COPY` 명령을 사용하여 각 테이블을 내보낼 수 있습니다.

```
COPY my_table TO '/path/to/my_table.csv' WITH (FORMAT csv, HEADER true);
```

2. Aurora DSQL에서 대상 테이블을 생성합니다. 스키마를 수동으로 생성하거나 로더의 `--if-not-exists` 플래그를 사용하여 데이터에서 스키마를 유추할 수 있습니다.
3. Aurora DSQL 로더를 사용하여 내보낸 데이터를 로드합니다.

```
aurora-dsql-loader load \
```

```
--endpoint cluster-id.dsql.region.on.aws \  
--source-uri /path/to/my_table.csv \  
--table my_table
```

### Tip

대규모 마이그레이션의 경우 더 나은 압축과 더 빠른 로드를 위해 Parquet 형식으로 내보내는 것이 좋습니다. DuckDB와 같은 도구는 CSV 파일을 Parquet으로 효율적으로 변환할 수 있습니다.

## MySQL에서 마이그레이션

MySQL에서 Aurora DSQL로 데이터를 마이그레이션하는 방법:

1. SELECT INTO OUTFILE을 사용하거나 mysqldump와 같은 도구를 --tab 옵션과 함께 사용하여 MySQL에서 CSV 형식으로 데이터를 내보냅니다.

```
SELECT * FROM my_table  
INTO OUTFILE '/path/to/my_table.csv'  
FIELDS TERMINATED BY ','  
ENCLOSED BY ''  
LINES TERMINATED BY '\n';
```

2. 적절한 PostgreSQL 호환 데이터 유형을 사용하여 Aurora DSQL에서 대상 테이블을 생성합니다.
3. Aurora DSQL 로더를 사용하여 내보낸 데이터를 로드합니다.

```
aurora-dsql-loader load \  
--endpoint cluster-id.dsql.region.on.aws \  
--source-uri /path/to/my_table.csv \  
--table my_table
```

### Note

MySQL과 PostgreSQL은 데이터 유형 시스템이 다릅니다. Aurora DSQL에서 테이블을 생성할 때 스키마를 검토하고 필요에 따라 데이터 유형을 조정합니다.

## Amazon S3에서 로드

데이터가 이미 Amazon S3에 있는 경우 로컬 시스템에 다운로드하지 않고도 직접 로드할 수 있습니다. Aurora DSQL 로더는 S3 URI를 기본적으로 지원합니다.

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/path/to/data.parquet \
  --table my_table
```

IAM ID에 소스 객체에 대한 s3:GetObject 권한이 있는지 확인합니다.

## PostgreSQL \copy 사용

IAM 인증을 처리하는 psql 세션을 통해 이미 Aurora DSQL에 연결되어 있는 경우 클라이언트 측 \copy 메타 명령을 사용하여 로컬 파일 시스템에서 데이터를 로드할 수 있습니다. 서버 측 COPY 문과 달리 \copy는 클라이언트 시스템에서 파일을 읽고 기존 연결을 통해 데이터를 스트리밍하므로 서버 측 파일 액세스가 필요하지 않습니다. 이 접근 방식은 간단한 단일 스레드 로드에는 적합합니다.

Example\copy를 사용하여 CSV 파일 로드

```
\copy my_table FROM '/path/to/data.csv' WITH (FORMAT csv, HEADER true);
```

\copy를 직접 사용할 경우 사용자에게 다음과 같은 책임이 있습니다.

- 여러 파일 또는 대용량 데이터셋을 로드하는 경우 병렬화 관리
- 연결 관리 및 인증 토큰 새로 고침 처리
- 실패한 작업에 대한 재시도 로직 구현

## INSERT 트랜잭션 모범 사례

INSERT 문을 사용하여 Aurora DSQL에 데이터를 로드할 때는 처리량과 신뢰성 향상을 위해 다음 모범 사례를 따르세요.

- 여러 행을 다중 행 INSERT로 묶어 처리 - 왕복 횟수를 줄이도록 여러 행을 단일 INSERT 문으로 그룹화합니다. 예를 들어 INSERT INTO my\_table VALUES (1, 'a'), (2, 'b'), (3, 'c')는 세 개의 개별 문보다 더 효율적입니다.

- 파라미터화된 쿼리 사용 - 문자열 연결 대신 파라미터 바인딩과 함께 준비된 문을 사용합니다. 이렇게 하면 SQL 인젝션 위험이 방지되고 데이터베이스가 쿼리 계획을 재사용할 수 있습니다.
- 트랜잭션을 작게 유지 - Aurora DSQL은 낙관적 동시성 제어를 사용하므로 많은 행을 처리하는 대규모 트랜잭션은 충돌이 발생할 가능성이 더 높습니다. 수천 행이 아닌 수백 행 정도의 트랜잭션을 목표로 하세요.
- 재시도 로직 구현 - 분산 시스템에서는 낙관적 동시성 제어(OCC) 충돌과 같은 일시적인 오류가 발생할 수 있습니다. 실패한 트랜잭션에 대한 재시도를 통해 지수 백오프를 구현합니다.
- 연결 간 병렬 처리 - 여러 연결을 열고 삽입을 분산 처리합니다. 각 연결은 서로 다른 데이터 하위 집합을 동시에 처리할 수 있습니다.

대부분의 사용 사례에서 Aurora DSQL 로더는 데이터 로드와 대한 더 간단하고 강력한 접근 방식을 제공합니다.

## 추가 리소스

- [GitHub의 Aurora DSQL 로더](#) - 소스 코드, 설명서 및 문제 추적
- [Amazon Aurora DSQL에서 인증 토큰 생성](#) - Aurora DSQL의 IAM 인증 토큰에 대한 자세한 정보
- [PostgreSQL 호환 클라이언트를 사용하여 Aurora DSQL에 액세스](#) - 다양한 클라이언트 및 도구를 사용하여 Aurora DSQL에 연결

## Aurora DSQL용 생성형 AI

이 섹션에서는 Aurora DSQL에서 생성형 AI 도구를 사용하는 방법에 대한 자세한 지침을 제공합니다.

### AWS Labs Aurora DSQL MCP 서버

Aurora DSQL용 AWS Labs Model Context Protocol(MCP) 서버

#### 특성

- 사람이 읽을 수 있는 질문과 명령을 구조화된 Postgres 호환 SQL 쿼리로 변환하고 구성된 Aurora DSQL 데이터베이스에 대해 실행합니다.
- 기본적으로 읽기 전용, `--allow-writes`에서 활성화된 트랜잭션
- 성능 향상을 위한 요청 간 연결 재사용
- Aurora DSQL 설명서, 검색 및 모범 사례 권장 사항에 대한 기본 제공 액세스



## 사용 가능한 도구

### 데이터베이스 작업

- `readonly_query` - DSQL 클러스터에 대해 읽기 전용 SQL 쿼리 실행
- 트랜잭션 - 트랜잭션에서 쓰기 작업 실행(필수 `--allow-writes`)
- `get_schema` - 테이블 스키마 정보 검색

### 설명서 및 권장 사항

- `dsql_search_documentation` - Aurora DSQL 설명서 검색
  - 파라미터: `search_phrase`(필수), `limit`(선택 사항)
- `dsql_read_documentation` - 특정 DSQL 설명서 페이지 읽기
  - 파라미터: `url`(필수), `start_index`(선택 사항), `max_length`(선택 사항)
- `dsql_recommend` - DSQL 모범 사례에 대한 권장 사항 가져오기
  - 파라미터: `url`(필수)

## 사전 조건

1. [Aurora DSQL 클러스터](#)를 사용하는 AWS 계정
2. 이 MCP 서버는 LLM 클라이언트와 동일한 호스트에서만 로컬로 실행할 수 있습니다.
3. AWS 서비스에 액세스할 수 있는 AWS 자격 증명 설정
  - 다음 권한이 포함된 역할이 있는 AWS 계정이 필요합니다.
    - `dsql:DbConnectAdmin` - 관리자 사용자로 DSQL 클러스터에 연결
    - `dsql:DbConnect` - 사용자 지정 데이터베이스 역할을 사용하여 DSQL 클러스터에 연결(관리자가 아닌 사용자를 사용하는 경우에만 필요)
  - `aws configure` 또는 환경 변수를 사용하여 AWS 자격 증명 구성

## 설치

대부분의 도구의 경우 [기본 설치](#) 지침에 따라 구성을 업데이트하는 것으로 충분합니다.

[Claude Code](#) 및 [Codex](#)에 대해서는 별도의 지침이 마련되어 있습니다.

기본 설치: 관련 MCP 구성 파일 업데이트

### uv 사용하기

1. [Astral](#) 또는 [GitHub README](#)에서 uv 설치

## 2. uv python install 3.10을 사용하여 Python 설치

MCP 클라이언트 구성에서 MCP 서버 구성([MCP 구성 파일 찾기](#))

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "command": "uvx",
      "args": [
        "awslabs.aurora-dsql-mcp-server@latest",
        "--cluster_endpoint",
        "[your dsql cluster endpoint, e.g. abcdefghijklmnopqrst234567.dsql.us-east-1.on.aws]",
        "--region",
        "[your dsql cluster region, e.g. us-east-1]",
        "--database_user",
        "[your dsql username, e.g. admin]",
        "--profile",
        "[your aws profile, e.g. default]"
      ],
      "env": {
        "FASTMCP_LOG_LEVEL": "ERROR"
      },
      "disabled": false,
      "autoApprove": []
    }
  }
}
```

## Windows 설치

Windows 사용자의 경우 MCP 서버 구성 형식이 약간 다릅니다.

```
{
  "mcpServers": {
    "awslabs.aurora-dsql-mcp-server": {
      "disabled": false,
      "timeout": 60,
      "type": "stdio",
      "command": "uv",
```

```

    "args": [
      "tool",
      "run",
      "--from",
      "awslabs.aurora-dsql-mcp-server@latest",
      "awslabs.aurora-dsql-mcp-server.exe"
    ],
    "env": {
      "FASTMCP_LOG_LEVEL": "ERROR",
      "AWS_PROFILE": "your-aws-profile",
      "AWS_REGION": "us-east-1"
    }
  }
}
}

```

## MCP 클라이언트 구성 파일 찾기

가장 일반적인 에이전틱 개발 도구의 경우 다음 파일 경로에서 MCP 클라이언트 구성을 찾을 수 있습니다.

- Kiro:
  - 사용자 구성: ~/.kiro/settings/mcp.json
  - Workspace 구성: /path/to/workspace/.kiro/settings/mcp.json
- Claude Code: 자세한 설정 도움말은 [Claude Code 설치](#)를 참조하세요.
  - 사용자 구성: "mcpServers"의 ~/.claude.json
  - 프로젝트 구성: /path/to/project/.mcp.json
  - 로컬 구성: "projects" -> "path/to/project" -> "mcpServers"의 ~/.claude.json
- Cursor:
  - 글로벌: ~/.cursor/mcp.json
  - 프로젝트: /path/to/project/.cursor/mcp.json
- Codex: ~/.codex/config.toml
  - 각 MCP 서버는 구성 파일의 [mcp\_servers.<server-name>] 테이블을 사용하여 구성됩니다. [사용자 지정 Codex 설치 지침](#)을 참조하세요.
- Warp:
  - 파일 편집: ~/.warp/mcp\_settings.json
  - 애플리케이션 편집기: Settings > AI > Manage MCP Servers 및 json 붙여넣기
- Amazon Q Developer CLI: ~/.aws/amazonq/mcp.json
- Cline: 일반적으로 중첩된 VS 코드 경로 - ~/.vscode-server/path/to/cline\_mcp\_settings.json

## Claude Code

### 사전 조건

중요: MCP 서버 관리는 VS Code 기본 패널 모드가 아닌 Claude Code CLI 터미널을 통해서만 가능합니다.

먼저 Claude의 [기본 설치 권장 프로세스](#)에 따라 Claude Code CLI를 설치합니다.

### 올바른 범위 선택

Claude Code는 로컬(기본값), 프로젝트, 사용자라는 세 가지 범위를 제공하며, 자격 증명의 민감도와 공유 필요성에 따라 어떤 범위를 선택해야 하는지 자세히 설명합니다. 자세한 내용은 Claude Code 설명서의 [MCP 설치 범위](#)를 참조하세요.

1. 로컬 범위 서버는 기본 구성 수준을 나타내며 프로젝트 경로 아래의 ~/.claude.json에 저장됩니다. 이 서버들은 사용자에게만 공개되며 현재 프로젝트 디렉터리 내에서만 액세스할 수 있습니다. 이는 MCP 서버를 생성할 때 기본 scope입니다.
2. 프로젝트 범위 서버는 프로젝트 디렉터리에서만 액세스할 수 있으면서도 팀 협업을 가능하게 합니다. 프로젝트 범위 서버는 프로젝트의 루트 디렉터리에 .mcp.json 파일을 추가합니다. 이 파일은 모든 팀 구성원이 동일한 MCP 도구 및 서비스에 액세스할 수 있도록 버전 관리 시스템에 체크인됩니다. 프로젝트 범위 서버를 추가하면 Claude Code가 자동으로 이 파일을 적절한 구성 구조로 생성하거나 업데이트합니다.
3. 사용자 범위 서버는 ~/.claude.json에 저장되고 교차 프로젝트 액세스가 가능하도록 합니다. 따라서 사용자 계정에 비공개로 유지되면서도 시스템의 모든 프로젝트에서 사용할 수 있습니다.

### Claude CLI 사용(권장)

대화형 claude CLI 세션을 사용하면 문제 해결 경험이 개선되므로 이 방법이 권장됩니다.

```
claude mcp add amazon-aurora-dsql \  
  --scope [one of local, project, or user] \  
  --env FASTMCP_LOG_LEVEL="ERROR" \  
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \  
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \  
  --region "[dsql cluster region, eg. us-east-1]" \  
  --database_user "[your-username]"
```

## 문제 해결: 다른 AWS 계정에서 Bedrock과 함께 Claude Code 사용

dsql 클러스터에 연결하는 데 필요한 프로필과 구별되는 Bedrock AWS 계정 또는 프로필로 Claude Code를 구성한 경우 추가 환경 인수를 제공해야 합니다.

```
--env AWS_PROFILE="[dsql profile, eg. default]" \
--env AWS_REGION="[dsql cluster region, eg. us-east-1]" \
```

### 구성 파일의 직접 수정

Claude Code는 영숫자 명명 규칙을 사용하므로 서버 이름을 `aurora-dsql-mcp-server`와 같이 지정하는 것이 좋습니다.

### 로컬 범위

프로젝트별 `mcpServers` 필드 내의 `~/.claude.json`을 다음과 같이 업데이트합니다.

```
{
  "projects": {
    "/path/to/project": {
      "mcpServers": {}
    }
  }
}
```

### 프로젝트 범위

`mcpServers` 필드의 `/path/to/project/root/.mcp.json`을 업데이트합니다.

```
{
  "mcpServers": {}
}
```

### 사용자 범위

프로젝트별 `mcpServers` 필드 내의 `~/.claude.json`을 다음과 같이 업데이트합니다.

```
{
  "mcpServers": {}
}
```

## Codex

### 옵션 1: Codex CLI

Codex CLI가 설치된 경우 `codex mcp` 명령을 사용하여 MCP 서버를 구성할 수 있습니다.

```
codex mcp add amazon-aurora-dsql \
  --env FASTMCP_LOG_LEVEL="ERROR" \
  -- uvx "awslabs.aurora-dsql-mcp-server@latest" \
  --cluster_endpoint "[dsql-cluster-id].dsql.[region].on.aws" \
  --region "[dsql cluster region, eg. us-east-1]" \
  --database_user "[your-username]"
```

### 옵션 2: config.toml

MCP 서버 옵션을 보다 세밀하게 제어하려면 `~/.codex/config.toml` 구성 파일을 수동으로 편집하면 됩니다. 각 MCP 서버는 구성 파일의 `[mcp_servers.<server-name>]` 테이블을 사용하여 구성됩니다.

```
[mcp_servers.amazon-aurora-dsql]
command = "uvx"
args = [
  "awslabs.aurora-dsql-mcp-server@latest",
  "--cluster_endpoint", "<DSQL_CLUSTER_ID>.dsql.<AWS_REGION>.on.aws",
  "--region", "<AWS_REGION>",
  "--database_user", "<DATABASE_USERNAME>"
]

[mcp_servers.amazon-aurora-dsql.env]
FASTMCP_LOG_LEVEL = "ERROR"
```

## 설치 확인

Amazon Q Developer CLI, Kiro CLI, Claude CLI/TUI 또는 Codex CLI/TUI의 경우 `/mcp`를 실행하여 MCP 서버의 상태를 확인합니다.

Kiro IDE의 경우 Kiro 패널의 MCP SERVERS 탭으로 이동하여 구성된 모든 MCP 서버와 연결 상태 표시기를 확인할 수도 있습니다.

## 서버 구성 옵션

### --allow-writes

기본적으로 dsql mcp 서버는 쓰기 작업('읽기 전용 모드')을 허용하지 않습니다. 이 모드에서는 트랜잭션 도구 호출이 실패합니다. 트랜잭션 도구를 사용하려면 --allow-writes 파라미터를 전달하여 쓰기를 허용합니다.

DSQL에 연결할 때 최소 권한 액세스를 사용하는 것이 좋습니다. 예를 들어 사용자는 가능하면 읽기 전용 역할을 사용해야 합니다. 읽기 전용 모드에는 변형을 거부하기 위한 최선의 클라이언트 측 적용이 있습니다.

### --cluster\_endpoint

연결할 클러스터를 지정하기 위한 필수 파라미터입니다. 이는 클러스터의 전체 엔드포인트여야 합니다. 예: 01abc2ldefg3hijklmnopqrstu.dsql.us-east-1.on.aws

### --database\_user

이는 연결할 사용자를 지정하는 필수 파라미터입니다. 예제: admin 또는 my\_user. 사용 중인 AWS 자격 증명에는 해당 사용자로 로그인할 수 있는 권한이 있어야 합니다. DSQL에서 데이터베이스 역할을 설정하고 사용하는 방법에 대한 자세한 내용은 [IAM 역할로 데이터베이스 역할 사용](#)을 참조하세요.

### --profile

자격 증명에 사용할 aws 프로파일을 지정할 수 있습니다. Docker 설치에는 지원되지 않습니다.

MCP 구성에서 AWS\_PROFILE 환경 변수를 사용하는 것도 지원됩니다.

```
"env": {
  "AWS_PROFILE": "your-aws-profile"
}
```

둘 다 제공되지 않으면 MCP 서버는 기본적으로 AWS 구성 파일에서 '기본' 프로파일을 사용합니다.

### --region

이는 DSQL 데이터베이스의 리전을 지정하기 위한 필수 파라미터입니다.

## --knowledge-server

DSQL 지식 도구(문서 검색, 읽기 및 권장 사항)에 대한 원격 MCP 서버 엔드포인트를 지정하는 선택적 파라미터입니다. 기본적으로 미리 구성되어 있습니다.

예제:

```
--knowledge-server https://custom-knowledge-server.example.com
```

참고: 보안을 위해 신뢰할 수 있는 지식 서버 엔드포인트만 사용합니다. 서버는 HTTPS 엔드포인트여야 합니다.

## --knowledge-timeout

지식 서버에 대한 요청의 제한 시간을 초 단위로 지정하는 선택적 파라미터입니다.

기본값: 30.0

예제:

```
--knowledge-timeout 60.0
```

느린 네트워크의 설명서에 액세스할 때 제한 시간이 발생하면이 값을 늘립니다.

## Aurora DSQL 운영: 스킬 및 파워

이 섹션에서는 스킬 및 파워를 사용하여 Aurora DSQL용 AI 운영을 구성하는 방법을 설명합니다. 이러한 마크다운 기반 구성 파일은 에이전트 개발의 품질을 개선하기 위해 코드를 생성할 때 AI 어시스턴트가 자동으로 적용하는 컨텍스트와 지침을 제공합니다.

### 개요

스킬 및 파워는 Aurora DSQL의 AI 어시스턴트 기능을 확장하는 모듈식 기능입니다. AI 어시스턴트가 Aurora DSQL 데이터베이스로 작업할 때 자동으로 사용하는 지침, 메타데이터 및 리소스를 패키징합니다.

스킬 및 파워를 사용하는 이유

스킬 및 파워는 Aurora DSQL 개발에 몇 가지 주요 이점을 제공합니다.



- AI 어시스턴트 전문화 - 모범 사례, Postgres 호환 SQL 패턴, 분산 데이터베이스 최적화를 포함하여 Aurora DSQL에 대한 도메인별 전문 지식을 제공합니다.
- 반복 감소 - 한 번 생성하고 자동으로 사용합니다. 여러 대화에서 동일한 지침을 반복적으로 제공할 필요가 없습니다.
- 컨텍스트 효율성 - 스킬은 컨텍스트를 미리 사용하는 대신 온디맨드 방식으로 로드합니다. AI는 필요에 따라 단계별로 정보를 로드합니다.
- 지속적인 학습 - Aurora DSQL 기능이 발전함에 따라 AI 어시스턴트는 기술이 업데이트될 때 업데이트된 패턴에 자동으로 액세스합니다.

## 권장 설정 경로

개발 환경과 일치하는 설정 경로를 선택합니다.

- [the section called “스킬 CLI”](#) (에이전트-무시)
- [the section called “Kiro 파워”](#)
- [the section called “Claude 스킬”](#)
- [the section called “Gemini 스킬”](#)
- [the section called “Codex 스킬”](#)

[DSQL 스킬](#)은 스킬 폴더를 도구의 rules 또는 skills 디렉터리에 복사하여 다른 AI 코딩 에이전트와 함께 사용할 수도 있습니다.

## 스킬 CLI

[DSQL 스킬](#)은 [스킬 CLI](#)를 사용하여 설치할 수 있습니다. 에이전트에 구매받지 않는 이 설정 방법은 대부분의 AI 코딩 어시스턴트에서 작동하며 한 번에 여러 에이전트에 스킬을 설치할 수 있습니다.

## 설정

다음 명령을 실행하여 Aurora DSQL 스킬을 설치합니다.

```
npx skills add awslabs/mcp --skill dsq1
```

CLI는 다음을 안내합니다.

- 에이전트 선택 - 설치할 에이전트를 선택합니다(Kiro, Claude Code, Cursor, Copilot, Gemini, Codex, Roo, Cline, OpenCode, Windsurf 등).
- 설치 범위 - 다음 중에서 선택합니다.
  - 프로젝트: 현재 디렉터리에 설치(프로젝트와 커밋됨)

- 글로벌: 홈 디렉터리에 설치(모든 프로젝트에서 사용 가능)
- 설치 방법 - 다음 중에서 선택합니다.
  - Symlink(권장): 단일 정보 소스, 간편한 업데이트
  - 모든 에이전트에 복사: 각 에이전트에 대해 독립적인 복사본

## 스킬 관리

다음을 사용하여 언제든지 스킬을 확인하고 업데이트합니다.

```
npx skills check
npx skills update
```

## Kiro 파워

Kiro 파워는 MCP 도구를 프레임워크 전문 지식 및 운영 지침과 함께 번들링하는 통합 패키지입니다. 각 파워에는 사용 가능한 MCP 도구 및 활성화 트리거, MCP 서버 구성, 온디맨드 로드된 추가 워크플로별 지침을 설명하는 진입점 문서가 포함되어 있습니다.

사용자 컨텍스트에 따라 파워가 동적으로 활성화됩니다. 파워는 모든 도구를 미리 로드하는 대신 관련 키워드가 활성화를 트리거할 때까지 거의 0에 가까운 기준 사용량을 유지합니다.

## 설정

Aurora DSQL에 대한 Kiro 파워를 설정하려면:

1. [Kiro 파워 레지스트리](#)에서 직접 설치
2. IDE에서 파워로 리디렉션되면 다음 중 하나를 수행합니다.
  - 파워 사용해 보기 버튼을 선택합니다. AI가 MCP 서버 설정 또는 Aurora DSQL을 사용한 대화형 온보딩 환경을 안내하여 새 클러스터를 생성하려는 사용자에게 제안됩니다.
  - 새 Kiro 채팅을 열고 Aurora DSQL과 관련된 모든 것을 질문합니다. 필요에 따라 기존 클러스터 세부 정보로 MCP 구성을 업데이트하여 파워로 즉시 사용할 수 있도록 MCP 서버 연결을 테스트합니다. Kiro 에이전트는 사용자의 작업을 완료하는 데 유용한 파워를 식별하면 자동으로 파워를 활성화합니다.

## Claude 스킬

Claude 스킬은 Claude의 기능을 확장하는 모듈식 기능입니다. 각 스킬은 관련된 경우 Claude가 자동으로 사용하는 지침, 메타데이터 및 선택적 리소스를 패키징합니다. 스킬은 파일 시스템 기반이며 온디맨드 방식으로 로드되어 컨텍스트 사용을 최소화합니다.

## 스킬 CLI를 사용한 간단한 설정

[the section called “스킬 CLI”](#)를 사용하여 스킬을 Claude Code에 설치할 수 있습니다. Claude Code만 설치할 에이전트로 지정하려면 다음을 사용합니다.

```
npx skills add awslabs/mcp --skill dsql --agent claude-code
```

## 대안: Git 복제본을 사용한 직접 설정

대체 설정은 dsql-skill 디렉터리의 희소 복제본을 가져와 이 복제본을 ~/.claude/skills/ 폴더에 symlink합니다. 이렇게 하면 스킬을 업데이트해야 할 때마다 스킬에 대한 변경 사항을 가져올 수 있습니다.

## 사전 조건

- Git 설치

## 설정 단계

### 1. 기본 리포지토리 디렉터리 생성

```
mkdir -p .dsql_skill_repos
```

### 2. MCP 리포지토리에서 스킬을 희소 복제

dsql-skill 폴더만 복제합니다(다른 파일 제외).

```
cd .dsql_skill_repos
git clone --filter=blob:none --no-checkout https://github.com/awslabs/mcp.git
cd mcp
git sparse-checkout init --cone
git sparse-checkout set src/aurora-dsql-mcp-server/skills/dsql-skill
git checkout
cd ../../..
```

### 3. 스킬을 스킬 디렉터리에 Symlink

스킬 디렉터리를 추가합니다(기본값: global/user-scoped).

```
mkdir -p ~/.claude/skills
```

**Note**

이를 프로젝트 범위 스킬로 만들려면 프로젝트 루트의 `.claude/skills/` 디렉터리를 대신 사용합니다.

symlink를 추가합니다.

```
ln -s "$(pwd)/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/dsql-skill"
~/.claude/skills/dsql-skill
```

**4. 설정 확인**

```
# Should show SKILL.md and other skill files
ls -la ~/.claude/skills/dsql-skill/
```

**5. 스킬 사용 확인**

스킬이 구성되면 새 스킬 명령 `/dsql`이 있어야 합니다. 감지할 스킬을 추가한 후 Claude 코드를 다시 시작해야 할 수 있습니다. Claude 코드 CLI 또는 패널에서 원하는 대로 이 명령을 사용할 수 있습니다.

**스킬 업데이트**

리포지토리에서 최신 변경 사항을 가져오려면:

```
cd .dsql_skill_repos/mcp
git pull
```

**디렉터리 구조**

글로벌 스킬을 설정한 후에는 다음 디렉터리가 표시됩니다.

```
.dsql_skill_repos/
### mcp/                                # Sparse git checkout
  ### src/
    ### aurora-dsql-mcp-server/
      ### skills/
        ### dsql-skill/
          ### SKILL.md
```

```
### ...
```

```
~/ .claude/  
### skills/  
    ### dsq1-skill -> /path/to/.dsq1_skill_repos/mcp/src/aurora-dsq1-mcp-server/skills/  
dsq1-skill
```

### Note

추적하지 않으려면 `.gitignore`에 `.dsq1_skill_repos/`를 추가합니다. 회소 체크아웃은 스킬 폴더만 유지하여 디스크 사용량을 최소화합니다.

## Gemini 스킬

Gemini에 Aurora DSQL 스킬을 직접 추가하려면 `workspace` (프로젝트에 포함됨) 또는 `user` (기본 값, 전역) 범위를 결정하고 스킬 설치 관리자를 사용합니다.

### 설정

```
gemini skills install https://github.com/aws-labs/mcp.git --path src/aurora-dsq1-mcp-  
server/skills/dsq1-skill --scope $SCOPE
```

`$SCOPE`를 `workspace` 또는 `user`로 바꿉니다.

그런 다음 Gemini와 함께 `/dsq1` 스킬 명령을 사용할 수 있으며, Gemini는 스킬을 사용해야 하는 시기를 자동으로 감지합니다.

## Codex 스킬

`$skill-installer` 스킬을 사용하여 Codex CLI 또는 TUI의 스킬 설치 관리자를 사용합니다.

### 설정

```
$skill-installer install dsq1 skill: https://github.com/aws-labs/mcp/tree/main/src/  
aurora-dsq1-mcp-server/skills/dsq1-skill
```

Codex를 다시 시작하여 스킬을 선택합니다. 그런 다음 `$dsq1`을 사용하여 스킬을 활성화할 수 있습니다.

# Aurora DSQL 쿼리 편집기 시작하기

Aurora DSQL 쿼리 편집기를 사용하면 외부 클라이언트를 설치하거나 구성하지 않고도 Aurora DSQL 클러스터에 안전하게 연결하고 AWS Management Console에서 직접 SQL 쿼리를 실행할 수 있습니다. 기본 제공 구문 강조 표시, 자동 완성 및 지능형 코드 지원이 포함된 직관적인 워크스페이스를 제공합니다. 단일 인터페이스 내에서 스키마 객체를 빠르게 탐색하고, SQL 쿼리를 개발 및 실행하고, 결과를 볼 수 있습니다.

이 주제에서는 클러스터에 연결하고, 쿼리를 실행하고, 결과를 보고, 실행 계획과 같은 고급 기능을 탐색하는 단계를 안내합니다.

## Note

쿼리 편집기는 Aurora DSQL이 지원되는 모든 리전에서 사용할 수 있습니다. 리전별 가용성에 대한 자세한 내용은 [리전별 AWS 서비스](#)를 참조하세요.

## 사전 조건

시작하기 전에 다음 요구 사항을 충족하는지 확인하세요.

- 사용 가능한 Aurora DSQL 클러스터가 하나 이상 있습니다. 클러스터 만들기에 대한 자세한 내용은 [1단계: Aurora DSQL 단일 리전 클러스터 생성](#) 섹션을 참조하세요.
- 클러스터 엔드포인트는 공개적으로 액세스할 수 있습니다. 쿼리 편집기는 현재 리소스 기반 정책에 의해 퍼블릭 액세스가 차단된 클러스터 또는 VPC 엔드포인트를 통해 관리되는 클러스터를 지원하지 않습니다. 액세스 제한에 대한 자세한 내용은 [Aurora DSQL에서 리소스 기반 정책으로 퍼블릭 액세스 차단](#) 및 [AWS PrivateLink를 사용하여 Amazon Aurora DSQL 클러스터 관리 및 연결](#) 섹션을 참조하세요.
- IAM 사용자 또는 역할에는 클러스터에 액세스하고 연결하는 데 필요한 권한이 있습니다. 권한에 대한 자세한 내용은 [데이터베이스 역할 및 IAM 인증 사용](#) 섹션을 참조하세요.

## 쿼리 편집기 사용하기

### 쿼리 편집기 열기

쿼리 편집기를 여는 방법

1. [Aurora DSQL 콘솔](#)을 엽니다.

## 2. 탐색 창에서 쿼리 편집기를 선택합니다.

또는 클러스터 페이지에서 쿼리할 클러스터를 선택하고 쿼리 편집기와 연결을 선택하여 편집기를 직접 시작합니다.

### Note

작업 및 연결 상태는 저장되지 않습니다. Aurora DSQL 콘솔에서 다른 곳으로 이동하거나 브라우저 탭을 닫거나 로그아웃하면 연결, 쿼리 텍스트 및 결과가 손실됩니다.

## 클러스터에 연결

### 클러스터에 연결

1. 클러스터 연결이 없는 경우 편집기에 연결된 클러스터 없음이 표시됩니다. 연결을 선택하거나 클러스터 탐색기 창에서 +(추가)를 선택하여 기존 클러스터에 연결합니다.
2. (선택 사항) 서로 다른 역할을 사용하여 여러 클러스터 또는 동일한 클러스터에 연결합니다.

## 클러스터 객체 탐색

클러스터 탐색기는 사용 가능한 모든 클러스터 연결을 표시하고 데이터베이스, 스키마, 테이블 및 뷰와 같은 객체를 검색할 수 있습니다. 또한 새로 고침, 테이블 생성 및 기타 컨텍스트별 옵션과 같은 일반적인 작업을 제공합니다.

## 쿼리 실행

### 쿼리를 실행하려면

1. 쿼리 편집기 탭 창에 SQL 문을 입력합니다. 예제:

```
SELECT * FROM public.orders LIMIT 10;
```

2. 쿼리 탭의 오른쪽 상단에 표시된 활성 클러스터 컨텍스트를 확인합니다. 이는 현재 쿼리 탭과 연결된 클러스터 연결을 나타냅니다.
3. (선택 사항) 연결 드롭다운을 사용하여 사용 가능한 모든 연결을 검토하거나 다른 클러스터로 전환합니다. 해당 탭의 쿼리가 실행되는 연결 업데이트 변경.
4. 그런 다음 Run(실행)을 선택하여 쿼리를 실행합니다.

**Note**

각 쿼리는 결과 창에서 최대 10,000개의 행을 반환할 수 있습니다. 더 큰 데이터세트의 경우 필터 또는 제한으로 쿼리를 구체화합니다.

## 결과 및 실행 계획 검토

쿼리가 실행된 후 편집기 하단의 결과 패널에서 출력을 검토합니다. 기본적으로 각 쿼리 실행에는 테이블 형식 쿼리 출력을 보여주는 결과(테이블) 탭이 표시됩니다.

쿼리 실행 계획을 가져오려면 `EXPLAIN ANALYZE` 또는 `EXPLAIN ANALYZE VERBOSE`를 실행하여 쿼리 성능에 대한 추가 인사이트를 얻습니다. 실행 계획에 대한 자세한 내용은 [Aurora DSQL EXPLAIN 계획 읽기](#) 섹션을 참조하세요.

**Tip**

`EXPLAIN ANALYZE VERBOSE` 명령은 컴퓨팅, 읽기, 쓰기 및 총 DPU 값을 포함한 DPU 사용량 추정치를 표시하여 개별 SQL 문에서 사용하는 리소스를 즉시 파악할 수 있습니다.

## 쿼리 편집기: Aurora DSQL과 함께 JupyterLab 사용

이 가이드에서는 Python과 함께 JupyterLab을 사용하여 Amazon Aurora DSQL을 연결하고 쿼리하는 방법에 대한 단계별 지침을 제공합니다. JupyterLab은 코드, 텍스트 및 시각화를 단일 문서로 결합하는 인기 있는 대화형 컴퓨팅 환경입니다. 데이터 과학 및 연구 애플리케이션에 널리 사용됩니다.

아래 지침은 JupyterLab의 로컬 설치에서 Aurora DSQL 사용의 기본 사항과 데이터 워크플로를 위한 UI를 통해 호스팅 환경을 제공하는 완전 관리형 기계 학습 서비스인 Amazon SageMaker AI를 사용하는 방법을 다룹니다.

## 시작하기

### 요구 사항

- Aurora DSQL 클러스터
- AWS 자격 증명 구성(로컬 설치만 해당)
- Python 버전 3.9 이상(로컬 설치만 해당)



## 로컬 JupyterLab 사용

JupyterLab을 시작하려면 사용자는 먼저 Python의 pip를 사용하여 애플리케이션을 설치해야 합니다.

```
pip install jupyterlab
```

그런 다음 **jupyter lab**을 실행하여 JupyterLab을 열 수 있습니다. 그러면 브라우저에서 액세스할 수 있는 localhost:8888에서 JupyterLab 애플리케이션이 열립니다. 계속하기 전에 로컬 환경에 AWS 자격 증명이 구성되어 있는지 확인합니다.

## Amazon SageMaker AI 사용

AWS Console에서 Amazon SageMaker AI 콘솔 페이지로 이동한 다음 애플리케이션 및 IDE 아래의 노트북 섹션으로 이동합니다. 여기에서 노트북 인스턴스 생성을 선택하여 SageMaker 환경 생성을 시작할 수 있습니다. 노트북 인스턴스 생성을 클릭하기 전에 인스턴스 유형과 플랫폼을 선택합니다.

설정 및 인스턴스 옵션에 대한 자세한 내용은 [Amazon SageMaker AI 설정 설명서](#)를 참조하세요.

### Note

경고: Amazon SageMaker AI를 사용하면 AWS 계정에 요금이 부과될 수 있습니다.

SageMaker 인스턴스가 활성화되면 JupyterLab 열기를 통해 노트북 인스턴스 섹션에서 인스턴스를 열 수 있습니다. 노트북에서 Aurora DSQL을 시작하기 전에 SageMaker 인스턴스의 IAM 역할에서 DSQL 클러스터에 대한 액세스를 제공해야 합니다. 가장 간단한 방법은 노트북 인스턴스 페이지의 IAM 역할에 대한 링크를 따르는 것입니다. 여기에서 SageMaker IAM 역할에 연결된 정책을 편집할 수 있습니다. Aurora DSQL에 대한 액세스를 허용하도록 IAM 정책을 구성하는 방법에 대한 자세한 내용은 [인증 및 권한](#) 부여를 참조하세요.

## JupyterLab을 사용하여 Aurora DSQL에 연결

JupyterLab 인스턴스를 설정한 후 Aurora DSQL에 연결하는 단계는 로컬 및 SageMaker AI에서 동일합니다. Python 코드를 사용하여 셀을 추가할 수 있는 빈 Python 3 노트북을 생성합니다.

Python 셀에서 공식 트러스트 스토어에서 Amazon 루트 인증서를 다운로드합니다.

```
import urllib.request
urllib.request.urlretrieve('https://www.amazontrust.com/repository/AmazonRootCA1.pem',
    'root.pem')
```

Aurora DSQL에 연결하려면 먼저 [Python용 Aurora DSQL 커넥터](#)와 Psycopg 드라이버를 Python 셸에 설치한 다음 가져옵니다.

```
pip install aurora_dsql_python_connector psycopg
```

```
import aurora_dsql_psycopg as dsql
```

커넥터를 가져온 상태에서 DSQL 구성을 생성하고 연결할 수 있습니다. Aurora DSQL Python 커넥터는 각 연결에서 인증 토큰 생성을 자동으로 처리합니다.

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

conn = dsql.connect(**config)
```

코드를 실행하면 이제 Aurora DSQL에 Psycopg로 연결할 수 있습니다. 그런 다음 Psycopg 커서를 사용하고 SQL 쿼리를 제공하여 쿼리를 실행할 수 있습니다. Postgres 호환 데이터베이스에서 Psycopg를 사용하는 방법에 대한 자세한 내용은 [Psycopg 설명서](#)를 참조하세요. 이 쿼리는 `results_list`에 튜플 목록을 생성합니다.

```
with conn:
    with conn.cursor() as cur:
        cur.execute("SELECT * FROM table")
        results_list = cur.fetchall()
```

그런 다음 [Pandas](#)와 같은 Python 프레임워크를 사용하여 쿼리 결과를 분석하거나 시각화할 수 있습니다. 예를 들면 다음과 같습니다.

```
pip install pandas

import pandas as pd

df = pd.DataFrame(tuples_list)
print(df)
```

```
print(f"Total records: {len(df)}")
```

## 예제 노트북

[Aurora DSQL을 사용하는 샘플 노트북은 Aurora DSQL 샘플 리포지토리에서 사용할 수 있습니다.](#)

## 참조 자료

[Amazon SageMaker AI 설정 설명서](#)

[Python용 Aurora DSQL 커넥터](#)

[Pandas 설명서](#)

# 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 availability](#)를 참조하세요.

## AWS Backup 시작

AWS Backup은 Aurora DSQL 클러스터의 전체 사본을 생성합니다. [Getting started with AWS Backup](#)의 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.

## 단일 리전 클러스터 복원

Aurora DSQL 단일 리전 클러스터를 복원하려면 <https://console.aws.amazon.com/backup> 콘솔 또는 CLI를 사용하여 복원할 복구 시점(백업)을 선택합니다. 백업에서 생성할 새 클러스터의 설정을 구성합니다. 자세한 지침은 [단일 리전 Aurora DSQL 클러스터 복원](#)을 참조하세요.

## 다중 리전 클러스터 복원

Aurora DSQL 다중 리전 클러스터 복원은 <https://console.aws.amazon.com/backup> 콘솔과 AWS CLI를 통해 지원됩니다. 자세한 지침은 [다중 리전 Aurora DSQL 클러스터 복원](#)을 참조하세요.

다중 리전 Aurora DSQL 클러스터로 복원하기 위해 단일 AWS 리전에서 가져온 백업을 사용할 수 있습니다. 그러나 복원 프로세스를 시작하기 전에 다중 리전 클러스터의 모든 AWS 리전에 동일한 백업 복사본이 있어야 합니다. 이러한 복사본이 아직 없는 경우 먼저 다중 리전 클러스터를 지원하는 다른 AWS 리전에 백업을 복사해야 합니다.

강력한 재해 복구 옵션을 활성화하고 규정 준수 요구 사항을 충족하려면 주요 AWS 리전에 백업 복사본을 생성하는 것이 좋습니다. Aurora DSQL을 사용할 수 있는 AWS 리전은 [the section called “AWS 리전 가용성”](#) 섹션을 참조하세요.

이러한 단계에 대한 자세한 지침은 [Amazon Aurora DSQL restore](#) 설명서를 참조하세요.

## 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 유지하는데 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.

- CloudWatch는 AWS에서 실행하는 AWS 리소스와 애플리케이션을 실시간으로 모니터링합니다. 지표를 수집 및 추적하고, 사용자 지정 대시보드를 생성할 수 있으며, 지정된 지표가 지정한 임계값에 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 설명서](#)를 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 설명서](#)를 참조하세요.

## 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 임계값을 감시하고 임계값에 도달하면 알림을 보내거나 조치를 취하도록 경보를 설정합니다. Aurora DSQL에 사용할 수 있는 다음 사용 및 관찰성 지표를 검토합니다.

자세한 내용은 [Amazon CloudWatch 사용 설명서](#)를 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 로드와 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 쓰기과 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                                                                                 |
|------------------|-------------|----------------|-----|------------------------------------------------------------------------------------|
| 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 CloudTrail](#)과 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 작업](#)을 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 trails](#)을 참조하세요. 이벤트 데이터 저장소 설정 및 관리에 대한 자세한 내용은 [CloudTrail Lake event data stores](#)를 참조하세요.

## CloudTrail의 Aurora DSQL 관리 이벤트

CloudTrail [관리 이벤트](#)는 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 reference](#)를 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 이벤트](#)는 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 Interface](#)를 참조하세요.

데이터 이벤트에는 추가 요금이 적용됩니다. CloudTrail 요금에 대한 자세한 내용은 [AWS CloudTrail 요금](#)을 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 | <ul style="list-style-type: none"> <li>• DbConnect</li> <li>• DbConnectAdmin</li> </ul> |

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 dsq1-data-events-trail을 구성하는 방법을 보여줍니다.

```
aws cloudtrail put-event-selectors \  
--region us-east-1 \  
--trail-name dsq1-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 모델](#)은 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 [AWS 규정 준수 프로그램](#)의 일환으로 보안 효과를 정기적으로 테스트하고 검증합니다. Amazon Aurora DSQL에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 [규정 준수 프로그램에 따른 AWS 서비스 범위](#)를 참조하세요.
- 클라우드 내 보안 - 귀하의 책임은 귀하가 사용하는 AWS 서비스에 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

- [Amazon Aurora DSQL의 AWS 관리형 정책](#)
- [Amazon Aurora DSQL의 데이터 보호](#)
- [Amazon Aurora DSQL에 대한 데이터 암호화](#)
- [Aurora DSQL의 ID 및 액세스 관리](#)
- [Aurora DSQL에 대한 리소스 기반 정책](#)
- [Aurora DSQL에서 서비스 연결 역할 사용](#)
- [Amazon Aurora DSQL에서 IAM 조건 키 사용](#)
- [Amazon Aurora DSQL의 사고 대응](#)
- [Amazon Aurora DSQL의 규정 준수 검증](#)
- [Amazon Aurora DSQL의 복원력](#)
- [Amazon Aurora DSQL의 인프라 보안](#)
- [Amazon Aurora DSQL의 구성 및 취약성 분석](#)

- [교차 서비스 혼동된 대리인 방지](#)
- [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 정책](#)을 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 보안 주체 ID(사용자, 그룹 및 역할)에도 업데이트가 적용됩니다. 새 AWS 서비스(를) 시작하거나 새 API 작업을 기존 서비스에 이용하는 경우, AWS가 AWS 관리형 정책을 업데이트할 가능성이 높습니다.

자세한 내용은 IAM 사용자 가이드의 [AWS 관리형 정책](#)을 참조하세요.

### AWS 관리형 정책: AmazonAuroraDSQLFullAccess

사용자, 그룹 및 역할에 AmazonAuroraDSQLFullAccess를 연결할 수 있습니다.

이 정책은 Aurora DSQL에 대한 전체 관리 액세스를 허용하는 권한을 부여합니다. 이러한 권한이 있는 보안 주체는 다음을 수행할 수 있습니다.

- Aurora DSQL 클러스터(다중 리전 클러스터 포함) 생성, 삭제 및 업데이트
- 클러스터 인라인 정책 관리(정책 생성, 보기, 업데이트 및 삭제)
- 클러스터에 태그 추가 및 제거
- 클러스터 목록 조회 및 개별 클러스터 정보 확인
- Aurora DSQL 클러스터에 연결된 태그 확인
- 관리자를 포함한 모든 사용자로 데이터베이스에 연결
- Aurora DSQL 클러스터에 대한 백업 및 복원 작업 수행(백업 및 복원 작업 시작, 중지, 모니터링 포함)

- 클러스터 암호화에 고객 관리형 AWS KMS 키 사용
- 해당 계정의 CloudWatch에서 모든 지표 보기
- AWS Fault Injection Service(AWS FIS)를 사용하여 내결함성 테스트를 위해 Aurora DSQL 클러스터에 장애 주입
- 클러스터 생성 시 필수인 `dsql.amazonaws.com` 서비스용 서비스 연결 역할 생성

## 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.
- `fis` - AWS Fault Injection Service(AWS FIS)를 사용하여 내결함성 테스트를 위해 Aurora DSQL 클러스터에 장애를 주입할 수 있는 권한 부여

`AmazonAuroraDSQLEFullAccess` 정책은 IAM 콘솔과 [AWS 관리형 정책 참조 안내서](#)에서 확인할 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 보고 클러스터 인라인 정책을 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 콘솔과 [AWS 관리형 정책 참조 안내서](#)에서 확인할 수 있습니다.

## AWS 관리형 정책: AmazonAuroraDSQLConsoleFullAccess

사용자, 그룹 및 역할에 AmazonAuroraDSQLConsoleFullAccess를 연결할 수 있습니다.

AWS Management Console을 통해 Amazon Aurora DSQL에 대한 전체 관리 액세스를 허용합니다. 이러한 권한이 있는 보안 주체는 다음을 수행할 수 있습니다.

- 콘솔을 사용하여 다중 리전 클러스터를 포함한 Aurora DSQL 클러스터 생성, 삭제 및 업데이트
- 콘솔을 통해 클러스터 인라인 정책 관리(정책 생성, 보기, 업데이트 및 삭제)
- 클러스터 목록 조회 및 개별 클러스터 정보 확인
- 계정의 어느 리소스에 대해서든 태그 확인
- 관리자를 포함한 모든 사용자로 데이터베이스에 연결
- Aurora DSQL 클러스터에 대한 백업 및 복원 작업 수행(백업 및 복원 작업 시작, 중지, 모니터링 포함)
- 클러스터 암호화에 고객 관리형 AWS KMS 키 사용
- AWS CloudShell에서 AWS Management Console 시작
- 계정의 CloudWatch에서 모든 지표 보기
- AWS Fault Injection Service(AWS FIS)를 사용하여 내결함성 테스트를 위해 Aurora DSQL 클러스터에 장애 주입
- 클러스터 생성 시 필수인 `dsql.amazonaws.com` 서비스용 서비스 연결 역할 생성

AmazonAuroraDSQLConsoleFullAccess 정책은 IAM 콘솔 및 AWS 관리형 정책 참조 안내서의 [AmazonAuroraDSQLConsoleFullAccess](#)에서 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.
- `fis` - AWS FIS를 사용하여 내결함성 테스트를 위해 Aurora DSQL 클러스터에 장애를 주입할 수 있는 권한을 부여합니다.
- `access-analyzer:ValidatePolicy`는 현재 정책의 오류, 경고 및 보안 문제에 대한 실시간 피드백을 제공하는 정책 편집기의 런터에 대한 권한을 부여합니다.
- `fis` - AWS Fault Injection Service(AWS FIS)를 사용하여 내결함성 테스트를 위해 Aurora DSQL 클러스터에 장애를 주입할 수 있는 권한 부여

`AmazonAuroraDSQLConsoleFullAccess` 정책은 IAM 콘솔과 [AWS 관리형 정책 참조 안내서](#)에서 확인할 수 있습니다.

## AWS 관리형 정책: AuroraDSQLServiceRolePolicy

`AuroraDSQLServiceRolePolicy`를 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 [AuroraDSQLServiceRolePolicy](#)에서 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                                                                                                                 | 설명                                                                                                                                                                                                                                                                                                                                                                     | 날짜            |
|--------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| AmazonAuroraDSQLFu<br>llAccess 및 AmazonAur<br>oraDSQLConsoleFullAccess<br>업데이트                                     | <p>AWS Fault Injection Service(AWS FIS)와 Aurora DSQL의 통합 지원이 추가되었습니다. 이를 통해 단일 리전 및 다중 리전 Aurora DSQL 클러스터에 장애를 주입하여 애플리케이션의 내결함성을 테스트할 수 있습니다. AWS FIS FIS 콘솔에서 실험 템플릿을 생성하여 장애 시나리오를 정의하고 테스트 대상 Aurora DSQL 클러스터를 지정할 수 있습니다.</p> <p>이러한 정책에 대한 자세한 내용은 <a href="#">AmazonAuroraDSQLFu llAccess</a> 및 <a href="#">AmazonAuroraDSQLConsoleFullAccess</a>를 참조하세요.</p> | 2025년 8월 19일  |
| AmazonAuroraDSQLFu<br>llAccess, AmazonAur<br>oraDSQLReadOnlyAccess<br>및 AmazonAuroraDSQLCo<br>nsoleFullAccess 업데이트 | <p>리소스 기반 정책(RBP) 지원을 PutClusterPolicy , GetClusterPolicy 및 DeleteClusterPolicy 와 같은 새 권한으로 추가했습니다. 이러한 권한을 통해 세분화된 액세스 제어를 위해 Aurora DSQL 클러스터에 연결된 인라인 정책을 관리할 수 있습니다.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLFullAccess</a>,</p>                                                                                                                                | 2025년 10월 15일 |

| 변경                                     | 설명                                                                                                                                                                                                                                                                                                | 날짜           |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
|                                        | <a href="#">AmazonAuroraDSQLReadOnlyAccess</a> 및 <a href="#">AmazonAuroraDSQLConsoleFullAccess</a> 를 참조하세요.                                                                                                                                                                                       |              |
| AmazonAuroraDSQLFullAccess 업데이트        | <p>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.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLFullAccess</a> 및 <a href="#">Using service-linked roles in Aurora DSQL</a>을 참조하세요.</p>                                       | 2025년 5월 21일 |
| AmazonAuroraDSQLConsoleFullAccess 업데이트 | <p>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 시작을 위해 고객 관리형 KMS 키를 사용하는 것을 지원합니다.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLConsoleFullAccess</a> 및 <a href="#">Aurora DSQL에서 서비스 연결 역할 사용</a>을 참조하세요.</p> | 2025년 5월 21일 |

| 변경                                  | 설명                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 날짜           |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| AmazonAuroraDSQLFu<br>llAccess 업데이트 | <p>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.</p> <p>또한 이 정책은 AWS PrivateLink를 통해 Aurora DSQL 클러스터에 연결하는 데 도움이 되는 GetVpcEndpointServiceName 권한을 추가합니다.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLFullAccess</a> 및 <a href="#">Using service-linked roles in Aurora DSQL</a>을 참조하세요.</p> | 2025년 5월 13일 |

| 변경                                     | 설명                                                                                                                                                                                                                                                                                                                   | 날짜           |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| AmazonAuroraDSQLReadOnlyAccess 업데이트    | <p>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.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLReadOnlyAccess</a> 및 <a href="#">Using service-linked roles in Aurora DSQL</a>을 참조하세요.</p> | 2025년 5월 13일 |
| AmazonAuroraDSQLConsoleFullAccess 업데이트 | <p>다중 리전 클러스터 관리 및 VPC 엔드포인트 연결을 지원하기 위해 Aurora DSQL에 새 권한을 추가합니다. 새 권한은 PutMultiRegionProperties , PutWitnessRegion , AddPeerCluster , RemovePeerCluster , GetVpcEndpointServiceName 입니다.</p> <p>자세한 내용은 <a href="#">AmazonAuroraDSQLConsoleFullAccess</a> 및 <a href="#">Aurora DSQL에서 서비스 연결 역할 사용</a>을 참조하세요.</p> | 2025년 5월 13일 |

| 변경                                      | 설명                                                                                                                                                                                                                                                                                               | 날짜           |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| AuroraDsqlServiceLinkedRole Policy 업데이트 | <p>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.</p> <p>자세한 내용은 <a href="#">AuroraDsqlServiceLinkedRolePolicy</a> 및 <a href="#">Using service-linked roles in Aurora DSQL</a>을 참조하세요.</p> | 2025년 5월 8일  |
| 페이지 생성됨                                 | Amazon Aurora DSQL과 관련된 AWS 관리형 정책 추적 시작                                                                                                                                                                                                                                                         | 2024년 12월 3일 |

## Amazon Aurora DSQL의 데이터 보호

[공동 책임 모델](#)은 의 데이터 보호에 적용됩니다. 이 모델에서 설명하는 것처럼은 모든 AWS 클라우드를 실행하는 글로벌 인프라를 보호할 책임이 있습니다. 사용자는 이 인프라에 호스팅되는 콘텐츠에 대한 통제 권한을 유지할 책임이 있습니다. 사용하는 의 보안 구성과 관리 태스크에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 관한 자세한 내용은 [데이터 프라이버시 FAQ](#)를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 [Security Blog](#)의 Shared Responsibility Model and 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 사용합니다.
- SSL/TLS를 사용하여 리소스와 통신하세요. TLS 1.2는 필수이며 TLS 1.3을 권장합니다.

- AWS CloudTrail으로 API 및 사용자 활동 로깅을 설정하세요. 활동 캡처에 추적을 사용하는 방법에 대한 자세한 내용은 사용 설명서의 [Working with trails](#)을 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 사용합니다.

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 프로세스](#)를 참조하세요.
- AWS CLI 또는 AWS SDK 중 하나를 사용하여 AWS에 요청합니다. 이들 도구는 도구를 구성할 때 지정한 액세스 키를 사용하여 사용자의 요청에 자동으로 서명합니다.



## FIPS 규정 준수

Aurora DSQL 데이터 영역 엔드포인트(데이터베이스 연결에 사용되는 클러스터 엔드포인트)는 기본적으로 FIPS 140-2 검증 암호화 모듈을 사용합니다. 클러스터 연결에는 별도의 FIPS 엔드포인트가 필요하지 않습니다.

컨트롤 플레인 작업의 경우 Aurora DSQL은 지원되는 리전에 전용 FIPS 엔드포인트를 제공합니다. 컨트롤 플레인 FIPS 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 [Aurora DSQL 엔드포인트 및 할당량](#)을 참조하세요.

저장 시 암호화는 [Aurora DSQL의 저장 시 암호화](#) 섹션을 참조하세요.

## 인터넷워크 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.
- Direct Connect 연결. 자세한 내용은 [Direct Connect란 무엇인가요?](#)를 참조하세요.

AWS 게시 API 작업을 사용하면 네트워크를 통해 Aurora DSQL에 액세스할 수 있습니다. 클라이언트는 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.

## 감시 리전의 데이터 보호

다중 리전 클러스터를 생성할 때, 감시 리전을 통해 암호화된 트랜잭션의 동기식 복제에 참여함으로써 자동화된 장애 복구를 지원합니다. 피어링된 클러스터를 사용할 수 없게 되는 경우에도, 감시 리전은 데이터베이스 쓰기 작업을 검증하고 처리할 수 있는 상태를 유지하여 가용성 손실이 발생하지 않도록 보장합니다.

감시 리전은 다음과 같은 설계 특성으로 데이터를 보호하고 안전성을 보장합니다.

- 감시 리전은 암호화된 트랜잭션 로그만 수신하고 저장합니다. 암호화 키를 호스팅하거나 저장하거나 전송하지 않습니다.
- 감시 리전은 오로지 쓰기 트랜잭션 로깅과 쿼리 기능에만 집중합니다. 데이터를 설계상 읽을 수 없습니다.
- 감시 리전은 클러스터 연결 엔드포인트나 쿼리 프로세서 없이 작동합니다. 이는 사용자 데이터베이스에 대한 액세스를 차단합니다.

AWS 리전에 대한 자세한 내용은 [다중 리전 클러스터 구성](#) 섹션을 참조하세요.

## 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 연결](#)을 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 설치](#)가 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 dsq1 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 [sslmode](#)를 참조하세요.

## macOS에서 연결

1. 인증 토큰을 생성하고 설정합니다.

```
export PGPASSWORD=$(aws dsq1 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. `psql`을 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 dsq 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 dsq1 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 DSQL의 저장 시 암호화](#)
- [Aurora DSQL에서 AWS KMS 및 데이터 키 사용](#)
- [Aurora DSQL에 AWS KMS key 사용 권한 부여](#)
- [Aurora DSQL 암호화 컨텍스트](#)
- [AWS KMS와 Aurora DSQL 상호 작용 모니터링](#)
- [암호화된 Aurora DSQL 클러스터 생성](#)
- [Aurora DSQL 클러스터의 키 제거 또는 업데이트](#)
- [Aurora DSQL에서 암호화 고려 사항](#)

## 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 concepts](#)를 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 키](#)를 사용하거나 자체 AWS KMS [고객 관리형 키](#)를 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 키](#)를 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 keys](#)를 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\_INACCESSIBLE로 변경됩니다. Aurora DSQL이 키에 액세스할 수 없는 경우 사용자는 클러스터에 연결할 수 없으며 클러스터의 암호화 상태가 KMS\_KEY\_INACCESSIBLE로 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\_INACCESSIBLE이 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\_INACCESSIBLE로 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.

예제 키 정책을 사용하기 전에 예제 보안 주체를 AWS 계정의 실제 보안 주체로 바꿉니다.

```
{
  "Sid": "Enable dsq1 IAM User Permissions",
  "Effect": "Allow",
  "Principal": {
    "Service": "dsq1.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey",
    "kms:Encrypt",
    "kms:ReEncryptFrom",
    "kms:ReEncryptTo"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:aws:dsq1:ClusterId": "w4abucpbwuxx",
      "aws:SourceArn": "arn:aws:dsq1:us-east-2:111122223333:cluster/w4abucpbwuxx"
    }
  }
},
{
  "Sid": "Enable dsq1 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 클러스터 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 dsq1 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 aiabtx6icfp6d53snkhseuiqq \  
--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 aiabtx6icfp6d53snkhseuiqq \  
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-ab1234a1b234
```

클러스터 설명의 EncryptionStatus는 UPDATING으로 전환되고 EncryptionType은 CUSTOMER\_MANAGED\_KMS\_KEY입니다. Aurora DSQL이 플랫폼을 전체에 새 키 전파를 완료하면 암호화 상태가 ENABLED로 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 encryptionStatus가 ENABLED이고 새 고객 관리형 키의 kmsKeyArn이 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)는 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](#)
- [ID를 통한 인증](#)
- [정책을 사용하여 액세스 관리](#)
- [Amazon Aurora DSQL에서 IAM을 사용하는 방법](#)
- [Amazon Aurora DSQL의 ID 기반 정책 예시](#)
- [Amazon Aurora DSQL ID 및 액세스 문제 해결](#)

## 고객

AWS Identity and Access Management(IAM)를 사용하는 방법은 역할에 따라 다릅니다.

- 서비스 사용자 - 기능에 액세스할 수 없는 경우 관리자에게 권한 요청([참조 Amazon Aurora DSQL ID 및 액세스 문제 해결](#))
- 서비스 관리자 - 사용자 액세스 결정 및 권한 요청 제출([Amazon Aurora DSQL에서 IAM을 사용하는 방법](#) 참조)
- IAM 관리자 - 액세스를 관리하기 위한 정책 작성([Amazon Aurora DSQL의 ID 기반 정책 예시](#) 참조)

## ID를 통한 인증

인증은 ID 자격 증명을 사용하여 AWS에 로그인하는 방식입니다. AWS 계정 루트 사용자인가 IAM 사용자로, 또는 IAM 역할을 수임하여 인증(에 로그인)받아야 합니다.

AWS IAM Identity Center(IAM Identity Center), Single Sign-On 인증 또는 Google/Facebook 자격 증명과 같은 자격 증명 소스의 자격 증명을 사용하여 페더레이션 ID로 로그인할 수 있습니다. 로그인하는 방법에 대한 자세한 내용은 AWS 로그인사용 설명서의 [AWS 계정에 로그인하는 방법](#) 섹션을 참조하세요.

프로그래밍 방식 액세스를 위해 AWS는 요청에 암호화 방식으로 서명할 수 있는 SDK 및 CLI를 제공합니다. 자세한 내용은 IAM 사용 설명서의 [API 요청용 AWS Signature Version 4](#) 섹션을 참조하세요.

### AWS 계정 루트 사용자

AWS 계정을 생성하는 경우에는 모든 AWS 서비스 서비스와 리소스에 대한 완전한 액세스 권한이 있는 AWS 계정 루트 사용자라는 단일 로그인 ID로 시작합니다. 일상적인 태스크에 루트 사용자를 사용하지 않을 것을 강력히 권장합니다. 루트 사용자 자격 증명이 필요한 작업은 IAM 사용 설명서의 [루트 사용자 자격 증명이 필요한 작업](#) 섹션을 참조하세요.

### 페더레이션 ID

가장 좋은 방법은 인간 사용자가 ID 공급자와의 페더레이션을 사용하여 임시 자격 증명으로 AWS 서비스에 액세스하도록 하는 것입니다.

페더레이션 ID는 엔터프라이즈 사용자 디렉터리, 웹 ID 제공업체 또는 Directory Service의 사용자로, ID의 자격 증명을 사용하여 AWS 서비스에 액세스합니다. 페더레이션 ID는 임시 자격 증명을 제공하는 역할을 수임합니다.

중앙 집중식 액세스 관리를 위해 AWS IAM Identity Center를 추천합니다. 자세한 정보는 AWS IAM Identity Center사용 설명서의 [What is IAM Identity Center?](#)를 참조하세요.

### IAM 사용자 및 그룹

[IAM 사용자](#)는 단일 개인 또는 애플리케이션에 대한 특정 권한을 가진 ID입니다. 장기 자격 증명이 있는 IAM 사용자 대신 임시 자격 증명을 사용하는 것이 좋습니다. 자세한 내용은 IAM 사용 설명서에서 [임시 자격 증명을 사용하여 AWS에 액세스하려면 인간 사용자가 ID 제공업체와의 페더레이션을 사용하도록 요구](#)를 참조하세요.

[IAM 그룹](#)은 IAM 사용자 모음을 지정하고 대규모 사용자 집합에 대한 관리 권한을 더 쉽게 만듭니다. 자세한 내용은 IAM 사용 설명서의 [IAM 사용자 사용 사례](#) 섹션을 참조하세요.

## IAM 역할

[IAM 역할](#)은 임시 자격 증명을 제공하는 특정 권한이 있는 자격 증명입니다. [사용자에서 IAM 역할\(콘솔\)로 전환](#)하거나 AWS CLI 또는 AWS API 작업을 직접적으로 호출하여 역할을 수입할 수 있습니다. 자세한 내용은 IAM 사용 설명서의 [역할 수입 방법](#)을 참조하세요.

IAM 역할은 페더레이션 사용자 액세스, 임시 IAM 사용자 권한, 교차 계정 액세스, 교차 서비스 액세스 및 Amazon EC2에서 실행되는 애플리케이션에 유용합니다. 자세한 내용은 IAM 사용 설명서의 [교차 계정 리소스 액세스](#)를 참조하세요.

## 정책을 사용하여 액세스 관리

정책을 생성하고 AWS ID 또는 리소스에 연결하여 AWS에서 내 액세스를 제어합니다. 정책은 자격 증명이나 리소스와 연결될 때 해당 권한을 정의합니다. AWS는 보안 주체가 요청을 보낼 때 이러한 정책을 평가합니다. 대부분의 정책은 AWS에 JSON 문서로 저장됩니다. JSON 정책 문서에 대한 자세한 내용은 IAM 사용 설명서의 [JSON 정책 개요](#) 섹션을 참조하세요.

정책을 사용하여 관리자는 어떤 보안 주체가 어떤 리소스에 대해 어떤 조건에서 작업을 수행할 수 있는지 정의하여 누가 무엇을 액세스할 수 있는지 지정합니다.

기본적으로 사용자 및 역할에는 어떠한 권한도 없습니다. IAM 관리자는 IAM 정책을 생성하고 사용자가 수입할 수 있는 역할에 추가합니다. IAM 정책은 작업을 수행하기 위해 사용하는 방법과 관계없이 작업에 대한 권한을 정의합니다.

## ID 기반 정책

ID 기반 정책은 ID(사용자, 사용자 그룹 또는 역할)에 연결하는 JSON 권한 정책 문서입니다. 이러한 정책은 자격 증명이 수행할 수 있는 작업, 대상 리소스 및 이에 관한 조건을 제어합니다. ID 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서에서 [고객 관리형 정책으로 사용자 지정 IAM 권한 정의](#)를 참조하세요.

ID 기반 정책은 인라인 정책(단일 ID에 직접 포함) 또는 관리형 정책(여러 ID에 연결된 독립 실행형 정책)일 수 있습니다. 관리형 정책 또는 인라인 정책을 선택하는 방법을 알아보려면 IAM 사용 설명서의 [관리형 정책 및 인라인 정책 중에서 선택](#) 섹션을 참조하세요.

## 리소스 기반 정책

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 예를 들어 IAM 역할 신뢰 정책 및 Amazon S3 버킷 정책이 있습니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 리소스 기반 정책에서 [보안 주체를 지정](#)해야 합니다.



리소스 기반 정책은 해당 서비스에 있는 인라인 정책입니다. 리소스 기반 정책에서는 IAM의 AWS관리형 정책을 사용할 수 없습니다.

## 기타 정책 유형

AWS는 이러한 정책 타입이 부여하는 최대 권한을 설정할 수 있는 추가 정책 타입을 지원합니다.

- 권한 경계 - ID 기반 정책에서 IAM 엔터티에 부여할 수 있는 최대 권한을 설정합니다. 자세한 정보는 IAM 사용 설명서의 [IAM 엔터티의 권한 범위](#)를 참조하세요.
- 서비스 제어 정책(SCP) - AWS Organizations내 조직 또는 조직 단위에 대한 최대 권한을 지정합니다. 자세한 내용은 AWS Organizations사용 설명서의 [서비스 제어 정책](#)을 참조하세요.
- 리소스 제어 정책(RCP) - 계정의 리소스에 사용할 수 있는 최대 권한을 설정합니다. 자세한 내용은 AWS Organizations사용 설명서의 [리소스 제어 정책\(RCP\)](#)을 참조하세요.
- 세션 정책 - 역할 또는 페더레이션 사용자에게 대해 임시 세션을 프로그래밍 방식으로 생성할 때 파라미터로 전달하는 고급 정책입니다. 자세한 내용은 IAM 사용 설명서의 [세션 정책](#)을 참조하세요.

## 여러 정책 유형

여러 정책 유형이 요청에 적용되는 경우, 결과 권한은 이해하기가 더 복잡합니다. 여러 정책 유형이 관련될 때 AWS가 요청을 허용할지 여부를 결정하는 방법을 알아보려면 IAM 사용 설명서의 [정책 평가 로직](#)을 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 특성                      | Aurora DSQL 지원 |
|-----------------------------|----------------|
| <a href="#">자격 증명 기반 정책</a> | 예              |
| <a href="#">리소스 기반 정책</a>   | 예              |
| <a href="#">정책 작업</a>       | 예              |
| <a href="#">정책 리소스</a>      | 예              |

| IAM 특성                       | Aurora DSQL 지원 |
|------------------------------|----------------|
| <a href="#">정책 조건 키</a>      | 예              |
| <a href="#">ACL</a>          | 아니요            |
| <a href="#">ABAC(정책의 태그)</a> | 예              |
| <a href="#">임시 보안 인증</a>     | 예              |
| <a href="#">엔터티 권한</a>       | 예              |
| <a href="#">서비스 역할</a>       | 예              |
| <a href="#">서비스 연결 역할</a>    | 예              |

Aurora DSQL 및 기타 AWS 서비스가 대부분의 IAM 기능과 작동하는 방법을 개괄적으로 알아보려면 IAM 사용 설명서의 [IAM으로 작업하는 AWS 서비스](#)를 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 정의](#)를 참조하세요.

IAM ID 기반 정책을 사용하면 허용되거나 거부되는 작업과 리소스뿐 아니라 작업이 허용되거나 거부되는 조건을 지정할 수 있습니다. JSON 정책에서 사용할 수 있는 모든 요소에 대해 알아보려면 IAM 사용 설명서의 [IAM JSON 정책 요소 참조](#)를 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: 예

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 보안 주체가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 정의합니다. 리소스 기반 정책에서 [보안 주체를 지정](#)해야 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.

Aurora DSQL 클러스터에 대한 리소스 기반 정책을 생성하고 관리하는 방법을 알아보려면 [Aurora DSQL에 대한 리소스 기반 정책](#)을 참조하세요.

## Aurora DSQL의 정책 작업

정책 작업 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.

JSON 정책의 Action요소는 정책에서 액세스를 허용하거나 거부하는 데 사용할 수 있는 작업을 설명합니다. 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 DSQL](#)을 참조하세요.

Aurora DSQL의 정책 작업은 작업 앞에 다음 접두사를 사용합니다.

```
dsql
```

단일 문에서 여러 작업을 지정하려면 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. 모범 사례에 따라 [Amazon 리소스 이름\(ARN\)](#)을 사용하여 리소스를 지정합니다. 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 DSQL](#)을 참조하세요. 각 리소스의 ARN을 지정할 수 있는 작업을 알아보려면 [Actions Defined by Amazon Aurora DSQL](#)을 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 요소는 정의된 기준에 따라 문이 실행되는 시기를 지정합니다. 같음(equals) 또는 미만(less than)과 같은 [조건 연산자](#)를 사용하여 정책의 조건을 요청의 값과 일치시키는 조건식을 생성할 수 있습니다. 모든 AWS 전역 조건 키를 보려면 IAM 사용자 설명서의 [AWS 전역 조건 컨텍스트 키](#)를 참조하세요.

Aurora DSQL 조건 키 목록을 보려면 서비스 승인 참조의 [Condition keys for Amazon Aurora DSQL](#)을 참조하세요. 조건 키를 사용할 수 있는 작업과 리소스를 알아보려면 [Actions defined by Amazon Aurora DSQL](#)을 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 보안 주체(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 태그라고 불리는 속성을 기반으로 권한을 정의하는 권한 부여 전략입니다. IAM 엔터티 및 AWS 리소스에 태그를 연결하면 보안 주체의 태그가 리소스 태그와 일치할 때 작업을 허용하도록 ABAC 정책을 설계할 수 있습니다.

태그에 근거하여 액세스를 제어하려면 `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` 또는 `aws:TagKeys` 조건 키를 사용하여 정책의 [조건 요소](#)에 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 정의](#)를 참조하세요. ABAC 설정 단계가 포함된 자습서를 보려면 IAM 사용 설명서의 [속성 기반 액세스 제어\(ABAC\) 사용](#)을 참조하세요.

## Aurora DSQL에서 임시 자격 증명 사용

임시 자격 증명 지원: 예

임시 자격 증명은 AWS 리소스에 대한 단기 액세스를 제공하며 페더레이션 또는 전환 역할을 사용할 때 자동으로 생성됩니다. AWS에서는 장기 액세스 키를 사용하는 대신 임시 자격 증명을 동적으로 생성할 것을 권장합니다. 자세한 내용은 IAM 사용 설명서의 [IAM의 임시 보안 자격 증명](#) 및 [IAM으로 작업하는 AWS 서비스](#) 섹션을 참조하세요.

## Aurora DSQL의 서비스 간 위탁자 권한

전달 액세스 세션(FAS) 지원: 예

전달 액세스 세션(FAS)은 AWS 서비스를 직접 호출하는 보안 주체의 권한과 요청하는 AWS 서비스를 함께 사용하여 다운스트림 서비스에 대한 요청을 수행합니다. FAS 요청 시 정책 세부 정보는 [전달 액세스 세션](#)을 참조하세요.

## Aurora DSQL의 서비스 역할

서비스 역할 지원: 예

서비스 역할은 서비스가 사용자를 대신하여 작업을 수행하는 것으로 가정하는 [IAM 역할](#)입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 내용은 IAM 사용 설명서의 [AWS 서비스 AWS에 권한을 위임할 역할 생성](#)을 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 연결 역할 지원: 예

서비스 연결 역할은 AWS 서비스에 연결된 서비스 역할의 한 유형입니다. 서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 연결 역할은 AWS 계정에 나타나고, 서비스가 소유합니다. IAM 관리자는 서비스 연결 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. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다.

이러한 예제 JSON 정책 문서를 사용하여 IAM ID 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 DSQL](#)을 참조하세요.

주제

- [정책 모범 사례](#)
- [Aurora DSQL 콘솔 사용](#)
- [사용자가 자신의 고유한 권한을 볼 수 있도록 허용](#)
- [클러스터 관리 및 데이터베이스 연결 허용](#)
- [태그 기반의 Aurora DSQL 리소스 액세스](#)

## 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 따르세요.

- 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 내용은 IAM 사용 설명서의 [AWS 관리형 정책](#) 또는 [AWS 직무에 대한 관리형 정책](#)을 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 권한](#)을 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. CloudFormation과 같이, 특정 AWS 서비스를 통해 사용되는 경우에만 서비스 작업에 대한 액세스 권한을 부여할 수도 있습니다. 자세한 내용은 IAM 사용자 설명서의 [IAM JSON 정책 요소: 조건](#)을 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 제공하여 안전하고 기능적인 정책을 작성하도록 돕습니다. 자세한 내용은 IAM 사용 설명서의 [IAM Access Analyzer에서 정책 검증](#)을 참조하세요.
- 다중 인증(MFA) 필요 - AWS 계정에 IAM 사용자 또는 루트 사용자가 필요한 시나리오가 있는 경우, 추가 보안을 위해 MFA를 설정합니다. API 작업을 직접적으로 호출할 때 MFA가 필요하다면 정책에 MFA 조건을 추가합니다. 자세한 내용은 IAM 사용 설명서의 [MFA를 통한 보안 API 액세스](#)를 참조하세요.

IAM의 모범 사례에 대한 자세한 내용은 IAM 사용 설명서의 [IAM의 보안 모범 사례](#)를 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 직접적으로 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 엔티티에 연결합니다. 자세한 내용은 IAM 사용 설명서의 [사용자에게 권한 추가](#)를 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 CLI나 AWS API를 사용하여 프로그래밍 방식으로 이 작업을 완료할 수 있는 권한이 포함됩니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsWithUser",
        "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": "*"
  }
]
}

```

## 클러스터 관리 및 데이터베이스 연결 허용

다음 정책은 IAM 사용자에게 특정 Aurora DSQL 클러스터를 관리하고 연결할 수 있는 권한을 부여합니다. 이 정책은 클러스터 관리 및 연결 작업의 범위를 단일 클러스터 Amazon 리소스 이름 (ARN)으로 제한하지만, 해당 작업이 리소스 수준 권한을 지원하지 않으므로 모든 리소스에 대해 `dsql:ListClusters`를 허용합니다.

이 예제에서는 `dsql:DbConnectAdmin`을 사용하여 `admin` 역할에 연결합니다. 대신 사용자 지정 데이터베이스 역할에 연결하려면 `dsql:DbConnectAdmin`을 `dsql:DbConnect`로 바꿉니다. 자세한 내용은 [Aurora DSQL에 대한 인증 및 권한 부여](#) 섹션을 참조하세요.

## JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowClusterManagement",
      "Effect": "Allow",
      "Action": [
        "dsql:GetCluster",
        "dsql:UpdateCluster",
        "dsql>DeleteCluster",
        "dsql:DbConnectAdmin",
        "dsql:TagResource",
        "dsql:ListTagsForResource",

```

```

        "dsql:UntagResource"
    ],
    "Resource": "arn:aws:dsql:*:123456789012:cluster/my-cluster-id"
},
{
    "Sid": "AllowListClusters",
    "Effect": "Allow",
    "Action": "dsql:ListClusters",
    "Resource": "*"
}
]
}

```

## 태그 기반의 Aurora DSQL 리소스 액세스

자격 증명 기반 정책의 조건을 사용하여 태그를 기반으로 Aurora DSQL 리소스에 대한 액세스를 제어할 수 있습니다. 다음 예제에서는 클러스터 보기를 허용하는 정책을 생성하는 방법을 보여 줍니다. 하지만 이 정책은 클러스터 태그 Owner가 해당 사용자의 사용자 이름 값을 가지고 있는 경우에만 권한을 부여합니다. 이 정책은 콘솔에서 이 태스크를 완료하는 데 필요한 권한도 부여합니다.

### JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ListClustersInConsole",
      "Effect": "Allow",
      "Action": "dsql:ListClusters",
      "Resource": "*"
    },
    {
      "Sid": "ViewClusterIfOwner",
      "Effect": "Allow",
      "Action": "dsql:GetCluster",
      "Resource": "arn:aws:dsql:*:cluster/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Owner": "${aws:username}"
        }
      }
    }
  ]
}

```

```
    }
  ]
}
```

이 정책을 계정의 IAM 사용자에게 연결할 수 있습니다. richard-roe라는 사용자가 Aurora DSQL 클러스터를 보려고 시도하는 경우 클러스터에 Owner=richard-roe 또는 owner=richard-roe 태그를 지정해야 합니다. 그러지 않으면 IAM은 액세스를 거부합니다. 조건 키 이름은 대소문자를 구분하지 않기 때문에 조건 태그 키 Owner는 Owner 및 owner 모두와 일치합니다. 자세한 내용은 IAM 사용자 설명서의 [IAM JSON 정책 요소: 조건](#)을 참조하세요.

다음 정책은 사용자가 Owner라는 사용자 이름으로 클러스터에 태그를 지정하는 경우에만 클러스터를 생성하도록 허용합니다. 또한 사용자가 이미 소유한 클러스터에서만 태그 지정을 허용합니다.

## JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreateTaggedCluster",
      "Effect": "Allow",
      "Action": "dsql:CreateCluster",
      "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/Owner": "${aws:username}"
        }
      }
    },
    {
      "Sid": "AllowTagOwnedClusters",
      "Effect": "Allow",
      "Action": "dsql:TagResource",
      "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Owner": "${aws:username}"
        }
      }
    }
  ]
}
```

}

## Amazon Aurora DSQL 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 권한이 없다는 오류가 표시되면 작업을 수행할 수 있도록 정책을 업데이트해야 합니다.

다음의 예시 오류는 mateojackson이 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.

자세한 내용은 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 사용자에게 대한 액세스 권한 제공](#)을 참조하세요.
- 리소스에 대한 액세스 권한을 서드 파티 AWS 계정에 제공하는 방법을 알아보려면 IAM 사용 설명서의 [서드 파티가 소유한 AWS 계정에 대한 액세스 제공](#)을 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 액세스](#)를 참조하세요.

## Aurora DSQL에 대한 리소스 기반 정책

Aurora DSQL에 대한 리소스 기반 정책을 사용하여 클러스터 리소스에 직접 연결하는 JSON 정책 문서를 통해 클러스터에 대한 액세스를 제한하거나 부여합니다. 이러한 정책은 클러스터에 액세스할 수 있는 사용자와 조건에 대한 세분화된 제어를 제공합니다.

Aurora DSQL 클러스터는 기본적으로 IAM 인증을 기본 보안 제어로 사용하여 퍼블릭 인터넷에서 액세스할 수 있습니다. 리소스 기반 정책을 사용하면 액세스 제한을 추가할 수 있으며, 특히 퍼블릭 인터넷에서 액세스를 차단할 수 있습니다.

리소스 기반 정책은 IAM 자격 증명 기반 정책과 함께 작동합니다. AWS는 두 가지 유형의 정책을 모두 평가하여 클러스터에 대한 액세스 요청에 대한 최종 권한을 결정합니다. 기본적으로 Aurora DSQL 클러스터는 계정 내에서 액세스할 수 있습니다. IAM 사용자 또는 역할에 Aurora DSQL 권한이 있는 경우 리소스 기반 정책이 연결되지 않은 클러스터에 액세스할 수 있습니다.

#### Note

리소스 기반 정책에 대한 변경 사항은 최종적으로 일관되며, 일반적으로 1분 이내에 적용됩니다.

ID 기반 정책과 리소스 기반 정책 간의 차이점에 대한 자세한 내용은 IAM 사용 설명서의 [ID 기반 정책 및 리소스 기반 정책](#)을 참조하세요.

## 리소스 기반 정책을 사용해야 하는 경우

리소스 기반 정책은 다음과 같은 시나리오에서 특히 유용합니다.

- 네트워크 기반 액세스 제어 - 요청이 시작되는 VPC 또는 IP 주소를 기반으로 액세스를 제한하거나 퍼블릭 인터넷 액세스를 완전히 차단합니다. `aws:SourceVpc` 및 `aws:SourceIp`와 같은 조건 키를 사용하여 네트워크 액세스를 제어합니다.
- 여러 팀 또는 애플리케이션 - 여러 팀 또는 애플리케이션에 대해 동일한 클러스터에 대한 액세스 권한을 부여합니다. 각 위탁자에 대한 개별 IAM 정책을 관리하는 대신 클러스터에서 액세스 규칙을 한번 정의합니다.
- 복잡한 조건부 액세스 - 네트워크 속성, 요청 컨텍스트 및 사용자 속성과 같은 여러 요소를 기반으로 액세스를 제어합니다. 단일 정책에서 여러 조건을 결합할 수 있습니다.
- 중앙 집중식 보안 거버넌스 - 클러스터 소유자가 기존 보안 관행과 통합되는 익숙한 AWS 정책 구문을 사용하여 액세스를 제어할 수 있습니다.

#### Note

Aurora DSQL 리소스 기반 정책에 대해서는 교차 계정 액세스가 아직 지원되지 않지만 향후 릴리스에서 사용할 수 있습니다.

누군가 Aurora DSQL 클러스터에 연결하려고 하면 AWS는 관련 IAM 정책과 함께 권한 부여 컨텍스트의 일부로 리소스 기반 정책을 평가하여 요청을 허용할지 거부할지를 결정합니다.

리소스 기반 정책은 클러스터와 동일한 AWS 계정 내의 위탁자에게 액세스 권한을 부여할 수 있습니다. 다중 리전 클러스터의 경우 각 리전 클러스터에는 자체 리소스 기반 정책이 있으므로 필요한 경우 리전별 액세스 제어를 허용합니다.

#### Note

조건 컨텍스트 키는 리전마다 다를 수 있습니다(예: VPC ID).

### 주제

- [리소스 기반 정책을 사용하여 클러스터 생성](#)
- [클러스터에 대한 리소스 기반 정책 추가 및 편집](#)
- [리소스 기반 정책 보기](#)
- [리소스 기반 정책 제거](#)
- [일반적인 리소스 기반 정책 예시](#)
- [Aurora DSQL에서 리소스 기반 정책으로 퍼블릭 액세스 차단](#)
- [Aurora DSQL API 작업 및 리소스 기반 정책](#)

## 리소스 기반 정책을 사용하여 클러스터 생성

새 클러스터를 생성할 때 리소스 기반 정책을 연결하여 처음부터 액세스 제어가 이루어지도록 할 수 있습니다. 각 클러스터에는 클러스터에 직접 연결된 단일 인라인 정책이 있을 수 있습니다.

### AWS Management Console

#### 클러스터 생성 중에 리소스 기반 정책 추가

1. AWS Management Console에 로그인하고 <https://console.aws.amazon.com/dsql/>에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 생성을 선택합니다.
3. 필요에 따라 클러스터 이름, 태그 및 다중 리전 설정을 구성합니다.
4. 클러스터 설정 섹션에서 리소스 기반 정책 옵션을 찾습니다.
5. 리소스 기반 정책 추가를 클릭합니다.

6. JSON 편집기에서 정책 문서를 입력합니다. 예를 들어 퍼블릭 인터넷 액세스를 차단하려면 다음을 수행합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

7. 문 편집 또는 새 문 추가를 사용하여 정책을 빌드할 수 있습니다.
8. 나머지 클러스터 구성을 완료하고 클러스터를 생성합니다.

## AWS CLI

클러스터를 생성할 때 `--policy` 파라미터를 사용하여 인라인 정책을 연결합니다.

```
aws dsql create-cluster --policy '{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
    }
  }]
}
```



```
    ]]
}'
```

## AWS SDK

### Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
        }
    }]
}

response = client.create_cluster(
    policy=json.dumps(policy)
)

print(f"Cluster created: {response['identifier']}")
```

### Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;

DsqlClient client = DsqlClient.create();

String policy = ""
{
    "Version": "2012-10-17",
    "Statement": [{
```

```

    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
        "StringNotEquals": { "aws:SourceVpc": "vpc-123456" }
    }
  }]
}
""";

CreateClusterRequest request = CreateClusterRequest.builder()
    .policy(policy)
    .build();

CreateClusterResponse response = client.createCluster(request);
System.out.println("Cluster created: " + response.identifier());

```

## 클러스터에 대한 리소스 기반 정책 추가 및 편집

### AWS Management Console

#### 기존 클러스터에 리소스 기반 정책 추가

1. AWS Management Console에 로그인하고 <https://console.aws.amazon.com/dsql/>에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 목록에서 클러스터를 선택하고 클러스터 세부 정보 페이지를 엽니다.
3. 권한 탭을 선택합니다.
4. 리소스 기반 정책 섹션에서 정책 추가를 선택합니다.
5. JSON 편집기에서 정책 문서를 입력합니다. 문 편집 또는 새 문 추가를 사용하여 정책을 빌드할 수 있습니다.
6. 정책 추가를 선택합니다.

#### 기존 리소스 기반 정책 편집

1. AWS Management Console에 로그인하고 <https://console.aws.amazon.com/dsql/>에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 목록에서 클러스터를 선택하고 클러스터 세부 정보 페이지를 엽니다.

3. 권한 탭을 선택합니다.
4. 리소스 기반 정책 섹션에서 편집을 선택합니다.
5. JSON 편집기에서 정책 문서를 수정합니다. 문 편집 또는 새 문 추가를 사용하여 정책을 업데이트할 수 있습니다.
6. 변경 사항 저장을 선택합니다.

## AWS CLI

`put-cluster-policy` 명령을 사용하여 새 정책을 연결하거나 클러스터의 기존 정책을 업데이트합니다.

```
aws dsq1 put-cluster-policy --identifier your_cluster_id --policy '{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsq1:DbConnect", "dsq1:DbConnectAdmin"],
    "Condition": {
      "Null": { "aws:SourceVpc": "true" }
    }
  }]
}'
```

## AWS SDK

### Python

```
import boto3
import json

client = boto3.client('dsq1')

policy = {
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
```

```

        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
    ]
}

response = client.put_cluster_policy(
    identifier='your_cluster_id',
    policy=json.dumps(policy)
)

```

## Java

```

import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.PutClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

String policy = ""
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
    }]
}
"";

PutClusterPolicyRequest request = PutClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .policy(policy)
    .build();

client.putClusterPolicy(request);

```

## 리소스 기반 정책 보기

클러스터에 연결된 리소스 기반 정책을 보고 현재 액세스 제어를 이해할 수 있습니다.

### AWS Management Console

#### 리소스 기반 정책 보기

1. AWS Management Console에 로그인하고 <https://console.aws.amazon.com/dsql/>에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 목록에서 클러스터를 선택하고 클러스터 세부 정보 페이지를 엽니다.
3. 권한 탭을 선택합니다.
4. 리소스 기반 정책 섹션에서 연결된 정책을 봅니다.

### AWS CLI

클러스터의 리소스 기반 정책을 보려면 다음 `get-cluster-policy` 명령을 사용합니다.

```
aws dsql get-cluster-policy --identifier your_cluster_id
```

### AWS SDK

#### Python

```
import boto3
import json

client = boto3.client('dsql')

response = client.get_cluster_policy(
    identifier='your_cluster_id'
)

# Parse and pretty-print the policy
policy = json.loads(response['policy'])
print(json.dumps(policy, indent=2))
```

## Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyResponse;

DsqlClient client = DsqlClient.create();

GetClusterPolicyRequest request = GetClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

GetClusterPolicyResponse response = client.getClusterPolicy(request);
System.out.println("Policy: " + response.policy());
```

## 리소스 기반 정책 제거

클러스터에서 리소스 기반 정책을 제거하여 액세스 제어를 변경할 수 있습니다.

### Important

클러스터에서 모든 리소스 기반 정책을 제거하면 IAM 자격 증명 기반 정책에 의해 액세스가 완전히 제어됩니다.

## AWS Management Console

### 리소스 기반 정책 제거

1. AWS Management Console에 로그인하고 <https://console.aws.amazon.com/dsql/>에서 Aurora DSQL 콘솔을 엽니다.
2. 클러스터 목록에서 클러스터를 선택하고 클러스터 세부 정보 페이지를 엽니다.
3. 권한 탭을 선택합니다.
4. 리소스 기반 정책 섹션에서 삭제를 선택합니다.
5. 확인 대화 상자에서 **confirm**을 입력하여 삭제를 확인합니다.
6. 삭제를 선택합니다.

## AWS CLI

`delete-cluster-policy` 명령을 사용하여 클러스터에서 정책을 제거합니다.

```
aws dsq1 delete-cluster-policy --identifier your_cluster_id
```

## AWS SDK

### Python

```
import boto3

client = boto3.client('dsq1')

response = client.delete_cluster_policy(
    identifier='your_cluster_id'
)

print("Policy deleted successfully")
```

### Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

DeleteClusterPolicyRequest request = DeleteClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

client.deleteClusterPolicy(request);
System.out.println("Policy deleted successfully");
```

## 일반적인 리소스 기반 정책 예시

이 예제에서는 Aurora DSQL 클러스터에 대한 액세스를 제어하는 일반적인 패턴을 보여줍니다. 특정 액세스 요구 사항에 맞게 이러한 패턴을 결합하고 수정할 수 있습니다.

## 퍼블릭 인터넷 액세스 차단

이 정책은 퍼블릭 인터넷(비 VPC)에서 Aurora DSQL 클러스터에 대한 연결을 차단합니다. 정책은 고객이 연결할 수 있는 VPC를 지정하지 않고 VPC에서 연결해야만 합니다. 특정 VPC에 대한 액세스를 제한하려면 StringEquals 조건 연산자와 함께 aws:SourceVpc를 사용합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

### Note

이 예제에서는 VPC 연결을 확인하는 데만 aws:SourceVpc를 사용합니다.

aws:VpcSourceIp 및 aws:SourceVpc 조건 키는 추가 세부 수준을 제공하지만 기본 VPC 전용 액세스 제어에는 필요하지 않습니다.

특정 역할에 대한 예외를 제공하려면 대신 이 정책을 사용합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {

```



```

    "Sid": "DenyAccessFromOutsideVPC",
    "Effect": "Deny",
    "Principal": {
      "AWS": "*"
    },
    "Resource": "*",
    "Action": [
      "dsql:DbConnect",
      "dsql:DbConnectAdmin"
    ],
    "Condition": {
      "Null": {
        "aws:SourceVpc": "true"
      },
      "StringNotEquals": {
        "aws:PrincipalArn": [
          "arn:aws:iam::123456789012:role/ExceptionRole",
          "arn:aws:iam::123456789012:role/AnotherExceptionRole"
        ]
      }
    }
  }
}
]
}

```

## AWS Organization에 대한 액세스 제한

이 정책은 AWS Organization 내 위탁자에 대한 액세스를 제한합니다.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mysqlclusterid0123456789a",
    }
  ]
}

```

```

    "Condition": {
      "StringNotEquals": {
        "aws:PrincipalOrgID": "o-exampleorgid"
      }
    }
  }
]
}

```

## 특정 조직 단위에 대한 액세스 제한

이 정책은 AWS Organization의 특정 조직 단위(OU) 내의 위탁자에 대한 액세스를 제한하여 조직 전체의 액세스보다 더 세분화된 제어를 제공합니다.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mysqlclusterid0123456789a",
      "Condition": {
        "StringNotLike": {
          "aws:PrincipalOrgPaths": "o-exampleorgid/r-examplerootid/ou-exampleouid/*"
        }
      }
    }
  ]
}

```

## 다중 리전 클러스터 정책

다중 리전 클러스터의 경우 각 리전 클러스터는 자체 리소스 정책을 유지 관리하여 리전별 제어를 허용합니다. 다음은 리전별로 정책이 다른 예입니다.

us-east-1 정책:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-east1-id"
        },
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

us-east-2 정책:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": "vpc-east2-id"
        }
      }
    }
  ]
}
```

```
}
]
}
```

### Note

조건 컨텍스트 키는 AWS 리전 간에 다를 수 있습니다(예: VPC ID).

## Aurora DSQL에서 리소스 기반 정책으로 퍼블릭 액세스 차단

퍼블릭 액세스 차단(BPA)은 AWS 계정의 Aurora DSQL 클러스터에 대한 퍼블릭 액세스를 허용하는 리소스 기반 정책의 연결을 식별하고 이를 방지하는 기능입니다. BPA를 사용하면 Aurora DSQL 리소스에 대한 퍼블릭 액세스를 차단할 수 있습니다. BPA는 리소스 기반 정책을 만들거나 수정하는 동안 검사를 수행하며 Aurora DSQL의 보안 태세를 개선하는 데 도움이 됩니다.

BPA는 [자동 추천](#)을 사용하여 리소스 기반 정책에 의해 부여된 액세스를 분석하고 리소스 기반 정책을 관리할 때 해당 권한이 발견되면 알려줍니다. 분석을 통해 정책에 사용되는 모든 리소스 기반 정책 문, 작업 및 조건 키 집합에 대한 액세스를 확인합니다.

### Important

BPA는 클러스터와 같은 Aurora DSQL 리소스에 직접 연결된 리소스 기반 정책을 통해 퍼블릭 액세스가 허용되지 않도록 함으로써 리소스를 보호합니다. BPA를 사용하는 것 외에도 다음 정책을 주의 깊게 검토하여 퍼블릭 액세스를 허용하지 않는지 확인하세요.

- 관련 AWS 보안 주체(예: IAM 역할)에 연결된 자격 증명 기반 정책
- 관련 AWS 리소스(예: AWS Key Management Service(KMS) 키)에 연결된 리소스 기반 정책

[보안 주체](#)에 \* 항목이 포함되지 않도록 하거나 지정된 조건 키 중 하나가 보안 주체의 리소스 액세스를 제한하는지 확인해야 합니다. AWS 전반에서 리소스 기반 정책이 클러스터에 대한 퍼블릭 액세스를 허용하는 경우 Aurora DSQL은 정책 내의 사양이 수정되어 퍼블릭이 아닌 것으로 간주될 때까지 정책을 생성하거나 수정하지 못하도록 차단합니다.

Principal 블록 내에 하나 이상의 보안 주체를 지정하여 정책을 퍼블릭이 아닌 상태로 만들 수 있습니다. 다음 리소스 기반 정책 예시는 두 개의 보안 주체를 지정하여 퍼블릭 액세스를 차단합니다.

```
{
```

```
"Effect": "Allow",
"Principal": {
  "AWS": [
    "123456789012",
    "111122223333"
  ]
},
"Action": "dsql:*",
"Resource": "arn:aws:dsq1:us-east-1:123456789012:cluster/cluster-id"
}
```

특정 조건 키를 지정하여 액세스를 제한하는 정책도 퍼블릭으로 간주되지 않습니다. 리소스 기반 정책에 지정된 보안 주체 평가와 함께 다음과 같은 [신뢰할 수 있는 조건 키](#)를 사용하여 퍼블릭이 아닌 액세스에 대한 리소스 기반 정책의 평가를 완료합니다.

- aws:PrincipalAccount
- aws:PrincipalArn
- aws:PrincipalOrgID
- aws:PrincipalOrgPaths
- aws:SourceAccount
- aws:SourceArn
- aws:SourceVpc
- aws:SourceVpce
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- aws:Ec2InstanceSourceVpc
- aws:SourceOrgID
- aws:SourceOrgPaths

또한 리소스 기반 정책을 퍼블릭이 아닌 상태로 설정하려면 Amazon 리소스 이름(ARN) 및 문자열 키의 값에 와일드카드나 변수가 포함되어서는 안 됩니다. 리소스 기반 정책에서 aws:PrincipalIsAWSService 키를 사용하는 경우 키 값을 true로 설정했는지 확인해야 합니다.

다음 정책은 지정된 계정의 Ben 사용자에게 액세스 권한을 제한합니다. 이 조건에 따라 Principal이 제한되며 퍼블릭으로 간주되지 않습니다.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsq1:us-east-1:123456789012:cluster/cluster-id",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/Ben"
    }
  }
}
```

다음은 StringEquals 연산자를 사용하는 퍼블릭이 아닌 리소스 기반 정책 제약 sourceVPC의 예입니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dsql:*",
      "Resource": "arn:aws:dsq1:us-east-1:123456789012:cluster/cluster-id",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
      }
    }
  ]
}
```

## Aurora DSQL API 작업 및 리소스 기반 정책

Aurora DSQL의 리소스 기반 정책은 특정 API 작업에 대한 액세스를 제어합니다. 다음 섹션에서는 범주별로 구성된 모든 Aurora DSQL API 작업을 나열하고 리소스 기반 정책을 지원하는 작업을 표시합니다.

지원 RBP 열은 정책이 클러스터에 연결될 때 API 작업이 리소스 기반 정책 평가를 받는지 여부를 나타냅니다.

### 태그 API

| API 작업                              | 설명                           | RBP 지원 |
|-------------------------------------|------------------------------|--------|
| <a href="#">ListTagsForResource</a> | Aurora DSQL 리소스에 대한 태그 나열    | 예      |
| <a href="#">TagResource</a>         | Aurora DSQL 리소스에 태그를 추가합니다.  | 예      |
| <a href="#">UntagResource</a>       | Aurora DSQL 리소스에서 태그를 제거합니다. | 예      |

### 클러스터 관리 API

| API 작업                                    | 설명                             | RBP 지원 |
|-------------------------------------------|--------------------------------|--------|
| <a href="#">CreateCluster</a> :           | 새 클러스터를 생성합니다.                 | 아니요    |
| <a href="#">DeleteCluster</a>             | 클러스터 삭제                        | 예      |
| <a href="#">GetCluster</a>                | 클러스터에 대한 정보 검색                 | 예      |
| <a href="#">GetVpcEndpointServiceName</a> | 클러스터의 VPC 엔드포인트 서비스 이름을 검색합니다. | 예      |
| <a href="#">ListClusters</a>              | 계정의 클러스터를 나열합니다.               | 아니요    |
| <a href="#">UpdateCluster</a>             | 클러스터의 구성 업데이트                  | 예      |

## 다중 리전 속성 API

| API 작업                                   | 설명                        | RBP 지원 |
|------------------------------------------|---------------------------|--------|
| <a href="#">AddPeerCluster</a>           | 다중 리전 구성에 피어 클러스터 추가      | 예      |
| <a href="#">PutMultiRegionProperties</a> | 클러스터에 대한 다중 리전 속성을 설정합니다. | 예      |
| <a href="#">PutWitnessRegion</a>         | 다중 리전 클러스터의 감시 리전을 설정합니다. | 예      |

## 리소스 기반 정책 API

| API 작업                              | 설명                            | RBP 지원 |
|-------------------------------------|-------------------------------|--------|
| <a href="#">DeleteClusterPolicy</a> | 클러스터에서 리소스 기반 정책 삭제           | 예      |
| <a href="#">GetClusterPolicy</a>    | 클러스터에 대한 리소스 기반 정책 검색         | 예      |
| <a href="#">PutClusterPolicy</a>    | 클러스터에 대한 리소스 기반 정책 생성 또는 업데이트 | 예      |

## AWS Fault Injection Service API

| API 작업                      | 설명                  | RBP 지원 |
|-----------------------------|---------------------|--------|
| <a href="#">InjectError</a> | 결함 주입 테스트를 위한 결함 주입 | 아니요    |

## 백업 및 복원 API

| API 작업                        | 설명          | RBP 지원 |
|-------------------------------|-------------|--------|
| <a href="#">GetBackupJob</a>  | 백업 작업 정보 검색 | 아니요    |
| <a href="#">GetRestoreJob</a> | 복원 작업 정보 검색 | 아니요    |



| API 작업                          | 설명                | RBP 지원 |
|---------------------------------|-------------------|--------|
| <a href="#">StartBackupJob</a>  | 클러스터에 대한 백업 작업 시작 | 예      |
| <a href="#">StartRestoreJob</a> | 백업에서 복원 작업 시작     | 아니요    |
| <a href="#">StopBackupJob</a>   | 실행 중인 백업 작업 중지    | 아니요    |
| <a href="#">StopRestoreJob</a>  | 실행 중인 복원 작업 중지    | 아니요    |

## Aurora DSQL에서 서비스 연결 역할 사용

Aurora DSQL은 AWS Identity and Access Management(IAM) [서비스 연결 역할](#)을 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 서비스](#)를 참조하고 서비스 연결 역할 옆에 예가 있는 서비스를 찾아보세요. 해당 서비스에 대한 서비스 연결 역할 설명서를 보려면 예(Yes) 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 연결 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.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CreateDsqlServiceLinkedRole",
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "iam:AWSServiceName": "dsql.amazonaws.com"
        }
      }
    }
  ]
}
```

자세한 내용은 [서비스 연결 역할 권한](#)을 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 특성 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 생성](#)을 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.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["dsql:CreateCluster"],
      "Resource": [
        "arn:aws:dsql:us-east-1:*:cluster/*",
        "arn:aws:dsql:us-east-2:*:cluster/*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

}

## 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.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dsql:CreateCluster",
        "dsql:PutMultiRegionProperties",
        "dsql:PutWitnessRegion",
        "dsql:AddPeerCluster"
      ],
      "Resource": [
        "arn:aws:dsql:us-east-1:123456789012:cluster/*",
        "arn:aws:dsql:us-east-2:123456789012:cluster/*"
      ]
    }
  ]
}
```

## 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.

## JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dsql:CreateCluster",
        "dsql:PutMultiRegionProperties",
        "dsql:AddPeerCluster"
      ],
      "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "dsql:PutWitnessRegion"
      ],
      "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
      "Condition": {
        "StringEquals": {
          "dsql:WitnessRegion": [
            "us-west-2"
          ]
        }
      }
    }
  ]
}
```

## Amazon Aurora 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 GuardDuty](#)는 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 서비스가 특정 규정 준수 프로그램의 범위에 포함되는지 알아보려면 [규정 준수 프로그램 제공 범위 내 AWS 서비스](#)를 참조하고 관심 있는 규정 준수 프로그램을 선택합니다. 일반적인 정보는 [AWS 규정 준수 프로그램](#)을 참조하세요.

AWS Artifact를 사용하여 타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 [AWS Artifact에서 보고서 다운로드](#)를 참조하세요.

AWS 서비스 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률 및 규정에 따라 결정됩니다. AWS 서비스 사용 시 규정 준수 책임에 대한 자세한 내용은 [AWS 보안 설명서](#) 참조하세요.

## 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 관한 자세한 정보는 [AWS 글로벌 인프라](#)를 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. 감시 리전이 손상될 경우 클러스터 가용성에 미치는 영향은 없습니다. 감시 리전이 복구될 때까지 쓰기 트랜잭션의 지연 시간이 약간 증가할 수 있습니다.

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 자

동화하므로 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.

## 결함 주입 테스트

Amazon Aurora DSQL은 애플리케이션의 복원력을 향상시키기 위해 제어된 결함 주입 실험을 실행하는 완전 관리형 서비스인 AWS Fault Injection Service(AWS FIS)와 통합됩니다. AWS FIS을 사용하여 다음 작업을 수행할 수 있습니다.

- 특정 장애 시나리오를 정의하는 실험 템플릿 생성
- 애플리케이션 오류 처리 및 복구 메커니즘을 검증하기 위해 장애(클러스터 연결 오류율 상승) 주입
- 단일 AWS 리전에서 높은 연결 오류율이 발생할 경우, 애플리케이션 트래픽이 다른 AWS 리전으로 전환되는지 검증하기 위해 다중 리전 애플리케이션 동작 테스트

예를 들어, 미국 동부(버지니아 북부)와 미국 동부(오하이오)에 걸쳐 있는 다중 리전 클러스터에서, 미국 동부(오하이오)에서 실험을 실행하여 해당 지역의 장애를 테스트하는 동시에 미국 동부(버지니아 북부)는 정상 운영을 계속할 수 있습니다. 이 제어된 테스트는 잠재적인 문제가 프로덕션 워크로드에 영향을 미치기 전에 이를 식별하고 해결하는 데 도움이 됩니다.

AWS FIS에서 지원하는 모든 작업의 전체 목록은 AWS FIS 사용 설명서의 [작업 대상](#)을 참조하세요.

AWS FIS에서 사용할 수 있는 Amazon Aurora DSQL 작업에 대한 자세한 내용은 AWS FIS 사용 설명서의 [Aurora DSQL 작업 참조](#)를 참조하세요.

결함 주입 실험 실행을 시작하려면 AWS FIS 사용 설명서의 [AWS FIS 실험 계획](#)을 참조하세요.



## Amazon Aurora DSQL의 인프라 보안

관리형 서비스인 Amazon Aurora DSQL은 [보안, 자격 증명 및 규정 준수 모범 사례](#)에 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 시크릿 액세스 키를 사용하여 서명해야 합니다. 또는 [AWS Security Token Service](#)(AWS STS)를 사용하여 임시 보안 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 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 [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 endpoint](#)를 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, list와 같은 관리 작업에 사용됩니다. [AWS PrivateLink를 사용하여 Aurora DSQL 클러스터 관리](#)을(를) 참조하세요.
2. 연결 엔드포인트 -이 엔드포인트는 PostgreSQL 클라이언트를 통해 Aurora DSQL 클러스터에 연결하는 데 사용됩니다. [AWS PrivateLink를 사용하여 Aurora DSQL 클러스터에 연결](#)을(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 [인터페이스 VPC 엔드포인트를 사용하여 AWS 서비스 액세스](#) 및 [AWS PrivateLink 할당량](#)을 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 endpoint](#)를 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.dsdl.us-east-1.vpce.amazonaws.com`)을 실제 정보로 바꿉니다.

```
aws dsdl --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsdl.us-east-1.vpce.amazonaws.com list-clusters
```

## API 작업

Aurora DSQL의 리소스 관리에 대한 설명서는 [Aurora DSQL API 참조](#)를 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-xxxxxxxxxxxxxxxxxx \
  --region region \
  --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": "*",
        "Action": "dsdl:*",
        "Resource": "*"
      }
    ]
  }'
```

예: 제한된 Aurora DSQL 액세스 정책

다음 정책은 이러한 Aurora DSQL 작업만 허용합니다.

- `CreateCluster`

- GetCluster
- ListClusters

다른 Aurora DSQL 작업은 모두 거부됩니다.

JSON

```
{
  "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

단계: 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-fnh4와 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);
```

## 단계: 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
```

응답 예제:

```
{
  "VpcEndpoint": {
    "VpcEndpointId": "vpce-0123456789abcdef0",
    "VpcEndpointType": "Interface",
    "VpcId": "vpc-0123456789abcdef0",
    "ServiceName": "com.amazonaws.us-east-1.dsqr-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.dsdl-fnh4"; // Use the service name
from previous step
String vpcId = "your-vpc-id";

Ec2Client ec2Client = Ec2Client.builder()
    .region(Region.of(region))
    .credentialsProvider(DefaultCredentialsProvider.create())
    .build();

CreateVpcEndpointRequest request = CreateVpcEndpointRequest.builder()
    .vpcId(vpcId)
    .serviceName(serviceName)
    .vpcEndpointType(VpcEndpointType.INTERFACE)
    .subnetIds("subnet-id-1", "subnet-id-2")
    .securityGroupIds("security-group-id")
    .build();

CreateVpcEndpointResponse response = ec2Client.createVpcEndpoint(request);
String vpcEndpointId = response.vpcEndpoint().vpcEndpointId();
System.out.println("VPC Endpoint created with ID: " + vpcEndpointId);
```

## Direct Connect 또는 Amazon VPC 피어링을 통한 연결 시 추가 설정

Amazon VPC 피어링 또는 Direct Connect를 통해 온프레미스 디바이스에서 AWS PrivateLink 연결 엔드포인트를 사용하여 Aurora DSQL 클러스터에 연결하려면 몇 가지 추가 설정이 필요할 수 있습니다. 애플리케이션이 AWS PrivateLink 엔드포인트와 동일한 Amazon VPC에서 실행되는 경우에는 이 설정이 필요하지 않습니다. 위에서 생성된 프라이빗 DNS 항목은 엔드포인트의 Amazon VPC 외부에서 올바르게 확인되지 않지만 AWS PrivateLink 연결 엔드포인트로 확인되는 자체 프라이빗 DNS 레코드를 생성할 수 있습니다.



AWS PrivateLink 엔드포인트의 정규화된 도메인 이름을 가리키는 프라이빗 CNAME DNS 레코드를 생성합니다. 생성된 DNS 레코드의 도메인 이름은 다음 구성 요소로 이루어져야 합니다.

1. 서비스 이름의 서비스 식별자입니다. 예: dsq1-fnh4
2. AWS 리전은

\* ***.service-identifier.region.on.aws*** 형식의 도메인 이름으로 CNAME DNS 레코드를 생성합니다.

도메인 이름의 형식은 다음 두 가지 이유로 중요합니다.

1. verify-full SSL 모드를 사용할 때 Aurora DSQL에 연결하는 데 사용되는 호스트 이름은 Aurora DSQL의 서버 인증서와 일치해야 합니다. 이렇게 할 때 최고 수준의 연결 보안이 유지됩니다.
2. Aurora DSQL은 Aurora DSQL에 연결하는 데 사용되는 호스트 이름의 클러스터 ID 부분을 사용하여 연결 클러스터를 식별합니다.

프라이빗 DNS 레코드를 생성할 수 없는 경우에도 Aurora DSQL에 연결할 수 있습니다. [프라이빗 DNS 없이 AWS PrivateLink 연결 엔드포인트를 사용하여 Aurora DSQL 클러스터에 연결](#)(를) 참조하세요.

AWS PrivateLink 연결 엔드포인트를 사용하여 Aurora DSQL 클러스터에 연결

AWS PrivateLink 엔드포인트가 설정되고 활성화되면(State가 available인지 확인) PostgreSQL 클라이언트를 사용하여 Aurora DSQL 클러스터에 연결할 수 있습니다. AWS SDK 사용에 대한 지침은 [Programming with Aurora DSQL](#)의 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: dsq1-fnh4
3. AWS 리전. 예: us-east-1

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 dsq1 --region $REGION generate-db-connect-admin-auth-token --
hostname $HOSTNAME)

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

프라이빗 DNS 없이 AWS PrivateLink 연결 엔드포인트를 사용하여 Aurora DSQL 클러스터에 연결 위의 연결 지침에서는 프라이빗 DNS 레코드를 사용합니다. 애플리케이션이 AWS PrivateLink 엔드포인트와 동일한 Amazon VPC에서 실행 중인 경우 DNS 레코드가 자동으로 생성됩니다. Amazon VPC 피어링 또는 Direct Connect를 통해 온프레미스 디바이스에서 연결하는 경우 자체 프라이빗 DNS 레코드를 생성할 수도 있습니다. 그러나 보안 팀에서 부과한 네트워크 제한으로 인해 DNS 레코드 설정이 항상 가능한 것은 아닙니다. 애플리케이션이 Direct Connect를 사용하거나 피어링된 Amazon VPC에서 연결해야 하고 DNS 레코드 설정이 불가능한 경우에도 Aurora DSQL에 연결할 수 있습니다.

Aurora DSQL은 호스트 이름의 클러스터 ID 부분을 사용하여 연결 클러스터를 식별하지만 DNS 레코드 설정이 불가능한 경우 Aurora DSQL은 `amzn-cluster-id` 연결 옵션을 사용한 대상 클러스터 지정을 지원합니다. 이 옵션을 사용하면 연결할 때 AWS PrivateLink 엔드포인트의 정규화된 도메인 이름을 호스트 이름으로 사용할 수 있습니다.

#### Important

AWS PrivateLink 엔드포인트의 정규화된 도메인 이름 또는 IP 주소와 연결할 때 `verify-full` SSL 모드는 지원되지 않습니다. 따라서 프라이빗 DNS를 설정하는 것이 좋습니다.

예: PostgreSQL을 사용하여 클러스터 ID 연결 옵션 지정

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
```

```
export HOSTNAME=vpce-04037adb76c111221-d849uc2p.ds1-fnh4.us-east-1.vpce.amazonaws.com
# This should match your endpoint's fully-qualified domain name

# Construct the hostname used to generate the authentication token
export AUTH_HOSTNAME="$CLUSTERID.ds1.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws ds1 --region $REGION generate-db-connect-admin-auth-token --
hostname $AUTH_HOSTNAME)

# Specify the amzn-cluster-id connection option
export PGOPTIONS="-c amzn-cluster-id=$CLUSTERID"

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

## AWS PrivateLink 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(직접 호출하는 서비스)가 다른 서비스(직접 호출되는 서비스)를 직접 호출할 때 발생할 수 있습니다. 직접 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 위탁자를 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](#)을(를) 사용합니다. 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:dsql:*: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:SourceArn의 값은 ResourceDescription이어야 합니다.

다음 예는 Aurora DSQL에서 aws:SourceArn 및 aws:SourceAccount 전역 조건 컨텍스트 키를 사용하여 혼동된 대리자 문제를 방지하는 방법을 보여줍니다.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "backup.amazonaws.com"
    },
    "Action": "dsql:GetCluster",
    "Resource": [
      "arn:aws:dsql:*:123456789012:cluster/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:backup:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

## Aurora 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 사용자의 환경에 적절하지 않거나 충분하지 않을 수 있으므로 규정이 아닌 참고용으로만 사용하세요.

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 [보안](#)을 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 지정한 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 Resources](#)를 참조하세요.

## Aurora DSQL의 예방 보안 모범 사례

Aurora DSQL을 안전하게 사용하는 다음 방법 외에도 클라우드 기술이 보안을 어떻게 개선하는지 알아보려면 AWS Well-Architected Tool의 [보안](#)을 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 자격 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 경계](#)를 사용할 수 있으며, 이를 통해 ID 기반 정책이 IAM 엔터티에 부여할 수 있는 최대 권한을 설정할 수 있습니다.

[AWS 계정에 대한 루트 사용자 모범 사례](#)와 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 DSQL](#)을 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 resources](#)를 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 제어](#)를 참조하세요.
- AWS에서 다양한 다른 활동에 태그를 사용할 수 있습니다. 자세한 내용은 [Common tagging strategies](#)를 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.

- DROP TABLE 명령을 실행한 후 스토리지 제한 계산에 여유 스토리지가 반영되는 데 시간이 걸릴 수 있습니다. 추가 스토리지 용량이 필요한 경우 [클러스터 할당량](#)를 참조하여 할당량 업데이트를 요청합니다.
- Aurora DSQL의 대형 테이블의 경우 시스템 카탈로그를 사용하여 COUNT(\*) 작업 대신 테이블 행 수를 검색합니다. 자세한 내용은 [Aurora DSQL에서 시스템 테이블 및 명령 사용](#)을 참조하세요.
- Aurora DSQL은 스키마 수준 권한 부여를 통해 권한을 관리합니다. 관리자 사용자는 CREATE SCHEMA를 사용하여 스키마를 만들고 GRANT USAGE ON SCHEMA를 사용하여 다른 역할에 대한 액세스 권한을 부여합니다. 관리자 사용자는 퍼블릭 스키마에서 객체를 관리하는 반면, 관리자가 아닌 사용자는 사용자가 만든 스키마에서 객체를 만듭니다. 관리자 역할은 다른 역할을 부여하여 사용자가 만든 객체에 대한 권한을 얻을 수 있습니다. 자세한 내용은 [데이터베이스 역할이 데이터베이스에서 SQL을 사용하도록 권한 부여](#) 섹션을 참조하세요.
- 드라이버가 PG\_PREPARED\_STATEMENTS를 직접적으로 호출하면 Aurora DSQL은 캐시된 준비된 문에 대한 클러스터 전체 보기를 제공합니다. 동일한 클러스터 및 IAM 역할에 대해 예상보다 연결당 더 많은 준비된 문이 표시될 수 있습니다. Aurora DSQL은 준비 중에 문 이름을 동적으로 관리합니다.
- IPv4 전용 인스턴스에서 연결할 때 클라이언트가 IPv4 연결에 맞게 구성되어 있는지 확인합니다. 일부 PostgreSQL 클라이언트는 듀얼 스택 모드에서 IPv4 및 IPv6 연결을 모두 시도합니다. IPv4 연결에 스로틀링이 발생하는 경우 클라이언트는 IPv6를 시도하고 IPv4 전용 호스트에서 NetworkUnreachable 오류를 반환할 수 있습니다. IPv4를 명시적으로 사용하도록 클라이언트를 구성하여 이러한 동작을 방지합니다.
- 관리자 사용자가 새 스키마를 만들고 GRANT 및 REVOKE 변경 사항이 연결 수명(최대 1시간) 내에 기존 연결로 전파된 후 즉시 적용하려면 권한 변경 후 새 연결을 설정합니다.
- 드문 다중 리전 연결 클러스터 복구 시나리오에서는 자동 클러스터 복구 작업이 고가용성을 유지하지만 일시적인 동시성 제어 또는 연결 오류가 발생할 수 있습니다. 대부분의 경우 워크로드의 일부만 영향을 받습니다. 이러한 전송 오류가 표시되면 트랜잭션을 다시 시도하거나 클라이언트와 다시 연결합니다.
- Datagrip과 같은 일부 SQL 클라이언트는 스키마 정보를 채우기 위해 광범위한 시스템 메타데이터를 요청합니다. Aurora DSQL은 SQL 쿼리 기능에 대한 핵심 메타데이터를 제공합니다. 이러한 클라이언트의 스키마 표시는 전체 기능 세트에 비해 제한된 정보를 표시할 수 있습니다.

- 쿼리가 새로 만들어진 스키마와 테이블을 인식하도록 하려면 데이터베이스 객체를 만들거나 삭제한 후 연결을 새로 고칩니다. 여기에는 스키마를 삭제한 후 또는 다른 연결에서 만들어진 객체를 쿼리할 때 Schema Already Exists 오류가 표시되는 시나리오가 포함됩니다. 연결 해제했다가 다시 연결하거나 SET search\_path를 다시 실행하여 카탈로그 캐시를 새로 고칩니다.
- 복잡한 쿼리의 경우 EXPLAIN ANALYZE VERBOSE를 사용하여 지연 시간이 긴 작업을 식별하고 쿼리 계획을 최적화합니다. 인덱스를 커버하면 전체 테이블 스캔 대신 인덱스 전용 스캔을 활성화하여 DPU 비용을 크게 줄일 수 있습니다. 자세한 내용은 [Aurora DSQL EXPLAIN 계획 작업](#) 섹션을 참조하세요.
- 연결 제한은 클러스터 수준에서 관리됩니다. 할당량 업데이트를 요청하려면 [클러스터 할당량](#) 섹션을 참조하세요.

# 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 20 개                         | 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 256TiB | 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 exceeds                  |
| 인덱스의 일부가 아닌 열의 최대 크기      | 1MiB  | 아니요    | 54000             | ERROR: maximum column size exceeds               |
| 프라이머리 키 또는 보조 인덱스의 최대 열 수 | 8     | 아니요    | 54011             | ERROR: more than 8 column keys 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 most 255 columns                                         |
| 테이블의 최대 열 수                    | 24                                                                                  | 아니요    | 54000             | ERROR: more than 24 indexes per table are allowed                                  |
| 한 쓰기 트랜잭션에서 수정된 모든 데이터의 최대 크기  | 10MiB                                                                               | 아니요    | 54000             | ERROR: transaction size limit exceeded<br>DETAIL: Current transaction size is 10mb |
| 트랜잭션 블록에서 변형할 수 있는 최대 테이블 행 수  | 트랜잭션당 행 3,000개.<br><a href="#">PostgreSQL 호환성을 위한 Aurora DSQL 고려 사항을</a> (를) 참조하세요. | 아니요    | 54000             | ERROR: transaction row limit exceeded                                              |
| 한 쿼리 작업에서 사용할 수 있는 최대 기본 메모리 양 | 트랜잭션당 128MiB                                                                        | 아니요    | 53200             | ERROR: query requires too much memory out of memory.                               |
| 데이터베이스에 정의된 최대 스키마 수           | 10                                                                                  | 아니요    | 54000             | ERROR: more than 10 schemas are allowed                                            |
| 데이터베이스의 최대 테이블 수               | 테이블 1,000 개                                                                         | 아니요    | 54000             | ERROR: creating more than 1000 tables is not 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 large          |
| 최대 시퀀스 수          | 5,000 | 아니요    | 54000             | ERROR: creating more than 500 not allowed |

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 [작업](#)을 참조하십시오.

데이터 형식에 대한 알파벳순 목록은 [데이터 형식](#)을 참조하십시오.

공통 쿼리 파라미터 목록은 [공통 파라미터](#)를 참조하십시오.

오류 코드에 대한 설명은 [공통 오류](#)를 참조하십시오.

AWS CLI에 대한 자세한 내용은 Aurora DSQL에 대한 AWS Command Line Interface 참조를 참조하십시오.



# 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](#)
- [인증 오류 문제 해결](#)
- [권한 부여 오류 문제 해결](#)
- [SQL 오류 문제 해결](#)
- [OCC 오류 문제 해결](#)
- [SSL/TLS 연결 문제 해결](#)

## 연결 오류 문제 해결

오류: 인식할 수 없는 SSL 오류 코드: 6 또는 연결을 수락할 수 없음, sni가 수신되지 않음

[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 --version을 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 DSQL](#)을 참조하세요.

GetObject 작업을 호출할 때 오류(InvalidAccessKeyId)가 발생했습니다. 제공한 AWS 액세스 키 ID가 레코드에 없습니다.

IAM이 요청을 거부했습니다. 자세한 내용은 [요청에 서명하는 이유](#)를 참조하세요.

<역할> IAM 역할이 존재하지 않음

Aurora DSQL에서 IAM 역할을 찾을 수 없습니다. 자세한 내용은 [IAM 역할](#)을 참조하세요.

IAM 역할은 IAM ARN과 같아야 합니다

자세한 내용은 [IAM 식별자 - IAM ARN](#)을 참조하세요.

## 잘못된 사용자-작업 매핑

이 오류는 인증 토큰 유형이 데이터베이스 역할과 일치하지 않을 때 발생합니다. Aurora DSQL은 admin 역할에 DbConnectAdmin, 사용자 지정 데이터베이스 역할에 DbConnect로 두 가지 토큰 유형을 사용합니다.

- Wrong user to action mapping. user: admin, action: DbConnect이 표시되는 경우 generate-db-connect-auth-token 대신 generate-db-connect-admin-auth-token을 사용합니다.
- Wrong user to action mapping. user: *myusername*, action: DbConnectAdmin이 표시되는 경우 generate-db-connect-admin-auth-token 대신 generate-db-connect-auth-token을 사용합니다.

## 권한 부여 오류 문제 해결

<역할> 역할이 지원되지 않음

Aurora DSQL은 GRANT 작업을 지원하지 않습니다. [Aurora DSQL에서 지원되는 SQL 명령 하위 집합](#)을 참조하세요.

<역할> 역할을 사용하여 신뢰를 구축할 수 없음

Aurora DSQL은 GRANT 작업을 지원하지 않습니다. [Aurora DSQL에서 지원되는 SQL 명령 하위 집합](#)을 참조하세요.

<역할> 역할이 존재하지 않음

Aurora DSQL에서 지정된 데이터베이스 사용자를 찾을 수 없습니다. [Authorize custom database roles to connect to a cluster](#)를 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 database](#)를 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 ROLE](#)을 참조하세요.

오류: 일부 객체가 해당 역할에 의존하기 때문에 <역할> 역할을 삭제할 수 없음

Aurora DSQL은 AWS IAM REVOKE를 사용하여 관계를 취소할 때까지 IAM 관계가 있는 데이터베이스 역할을 삭제하면 오류를 반환합니다. 자세한 내용은 [Revoking authorization](#)을 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 DSQL](#)을 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 DSQL](#)을 참조하세요.

## OCC 오류 문제 해결

OC000 '오류: 변형이 다른 트랜잭션과 충돌합니다. 필요에 따라 다시 시도하세요.'

이 트랜잭션은 다른 동시 트랜잭션과 동일한 튜플을 수정하려고 시도했습니다. 이는 수정된 튜플의 경합을 나타냅니다. 자세한 내용은 [Aurora DSQL의 동시성 제어](#)를 참조하세요.

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 DSQL에 대한 피드백 제공

마이그레이션에 중요하지만 현재 Aurora DSQL에서 지원되지 않는 기능이 있는 경우 피드백을 위한 여러 채널이 AWS에서 제공됩니다.

## 피드백 채널

### Aurora DSQL Discord 서버

[Aurora DSQL Discord 서버](#)에 참여하여 AWS 팀 및 커뮤니티와 소통하세요. 기능 요청을 공유하고, 마이그레이션 관련 어려움을 논의하고, 실시간 피드백을 받을 수 있습니다.

### AWS Support

AWS Support 플랜이 있는 경우 지원 사례를 생성하여 특정 요구 사항 및 필요한 일정을 논의하세요.

### AWS re:Post

[AWS re:Post](#)를 사용하여 커뮤니티 및 AWS 전문가에게 질문하고 피드백을 공유하세요.

## 효과적인 기능 요청

기능을 요청할 때는 다음 정보를 제공해 주세요.

- 사용 사례 설명: 달성하고자 하는 것과 그 이유를 설명합니다.
- 현재 해결 방법: 시도해 본 방법을 설명합니다.
- 비즈니스 영향: 누락된 기능이 마이그레이션 일정 또는 애플리케이션 기능에 어떤 영향을 미치는지 설명합니다.
- 우선순위: 해당 기능이 마이그레이션을 방해하는지, 아니면 있으면 좋은 개선 사항인지 나타내 주세요.

# Amazon Aurora DSQL 사용 설명서의 문서 기록

다음 표에서는 Aurora DSQL에 대한 문서 릴리스를 소개합니다.

| 변경 사항                                               | 설명                                                                                                                                                                                                    | 날짜           |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| <a href="#">새 콘텐츠: .NET Npgsql용 Aurora DSQL 커넥터</a> | 자동 IAM 인증으로 Npgsql을 래핑하는 .NET Npgsql용 Aurora DSQL 커넥터에 대한 설명서가 추가되었습니다. 이 커넥터는 .NET 애플리케이션의 토큰 생성, SSL 구성 및 연결 관리를 처리합니다. 자세한 내용은 <a href="#">.NET Npgsql 커넥터를 사용하여 Aurora DSQL 클러스터에 연결</a> 을 참조하세요. | 2026년 3월 20일 |
| <a href="#">업데이트된 콘텐츠: SQL 명령 참조 및 시스템 쿼리</a>       | END 및 ABORT를 별칭으로 사용한 START TRANSACTION 및 ROLLBACK이 트랜잭션 시작 명령 참조에 추가되었습니다. Aurora DSQL 및 PostgreSQL 버전 정보를 검색하는 데 유용한 시스템 쿼리가 추가되었습니다. 자세한 내용은 <a href="#">PostgreSQL 호환성 참조</a> 를 참조하세요.            | 2026년 3월 13일 |
| <a href="#">업데이트된 콘텐츠: Aurora DSQL로 데이터 로드</a>      | 클라이언트 측 \copy 사용법, INSERT 모범 사례 및 로드 전 테이블 사전 생성 지침을 포함하여 데이터 로드 가이드가 업데이트되었습니다. 자세한 내용은 <a href="#">Aurora DSQL로 데이터 로드</a> 를 참조하세요.                                                                 | 2026년 3월 13일 |

### [새 콘텐츠: Ruby pg용 Aurora DSQL 커넥터](#)

자동 IAM 인증으로 pg gem을 래핑하는 Ruby pg용 Aurora DSQL 커넥터에 대한 설명서가 추가되었습니다. 커넥터는 Ruby 애플리케이션의 토큰 생성, SSL 구성 및 연결 관리를 처리합니다. 자세한 내용은 [Ruby pg 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#)을 참조하세요.

2026년 3월 12일

### [업데이트된 콘텐츠: 비동기 DDL 작업](#)

비동기 DDL 작업 모니터링 및 관리에 대한 자세한 정보를 포함하여 sys.jobs 설명서가 업데이트되었습니다. 자세한 내용은 [PostgreSQL 호환성 참조](#)를 참조하세요.

2026년 3월 6일

### [업데이트된 콘텐츠: PHP 인증 토큰 생성](#)

인증 토큰 생성 페이지에 PHP SDK 탭이 추가되었습니다. 자세한 내용은 [인증 토큰 생성](#)을 참조하세요.

2026년 3월 5일

### [새 콘텐츠: Aurora DSQL로 데이터 로드](#)

Aurora DSQL 로더 유틸리티 사용법을 포함하여 Aurora DSQL 클러스터로 데이터를 로드하기 위한 가이드가 추가되었습니다. 자세한 내용은 [Aurora DSQL로 데이터 로드](#)를 참조하세요.

2026년 3월 5일



### [새 콘텐츠: 시퀀스 및 ID 열](#)

시퀀스 및 ID 열에 대한 지원이 추가되었습니다. CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE 및 시퀀스 조작 함수에 대한 새로운 SQL 명령 참조 페이지입니다. ID 열 구문을 포함하도록 CREATE TABLE 및 ALTER TABLE을 업데이트했습니다. 식별자 유형 및 캐시 크기 선택에 대한 새 가이드가 추가되었습니다. 자세한 내용은 [시퀀스 및 ID 열](#)을 참조하세요.

2026년 2월 11일

### [새 콘텐츠: Go용 Aurora DSQL 커넥터](#)

자동 IAM 인증으로 pgx를 래핑하는 Go용 Aurora DSQL 커넥터에 대한 설명서가 추가되었습니다. 커넥터는 Go 애플리케이션의 토큰 생성, SSL 구성 및 연결 관리를 처리합니다. 자세한 내용은 [Go 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#)을 참조하세요.

2026년 2월 5일

### [업데이트된 콘텐츠: Amazon Aurora DSQL 클러스터 연결 도구](#)

AWS에서 제공하는 커넥터, 어댑터 및 타사 도구 간의 구분을 명확히 하기 위해 클러스터 연결 도구 설명서를 재구성했습니다. 코드 샘플에 대한 누락된 링크가 추가되었습니다. 자세한 내용은 [Amazon Aurora DSQL 클러스터 연결 도구](#)를 참조하세요.

2026년 1월 26일

[새 콘텐츠: DBeaver](#)[Community Edition용 Aurora DSQL 플러그인](#)

Aurora DSQL 클러스터에 대한 IAM 인증 및 간소화된 연결 설정을 지원하는 DBeaver Community Edition용 Aurora DSQL 플러그인에 대한 설명서가 추가되었습니다. 설치 지침, 연결 구성 및 문제 해결 지침이 포함되어 있습니다. 자세한 내용은 [DBeaver를 사용하여 Aurora DSQL 액세스](#)를 참조하세요.

2026년 1월 26일

[새 콘텐츠: SQLTools용 Aurora DSQL 드라이버](#)

개발자가 자동 IAM 인증을 사용하여 VS Code에서 직접 Aurora DSQL 데이터베이스에 연결하고 쿼리할 수 있는 Visual Studio Code 확장 프로그램인 SQLTools용 Aurora DSQL 드라이버에 대한 설명서가 추가되었습니다. 자세한 내용은 [SQLTools용 Aurora DSQL 드라이버 사용](#)을 참조하세요.

2026년 1월 26일

[업데이트된 콘텐츠: Aurora DSQL 운영: 스킬 및 파워](#)

에이전트에 구매받지 않는 지원을 위해 스킬 CLI를 사용하여 설치를 지원하는 설명서가 추가되었습니다. 스킬 CLI는 Claude 코드, Cursor, Copilot, Gemini 등을 포함한 여러 AI 코딩 어시스턴트에서 작동하는 간소화된 설정 방법을 제공합니다. 자세한 내용은 [Aurora DSQL 운영: 스킬 및 파워](#)를 참조하세요.

2026년 1월 23일

### [새 콘텐츠: Tortoise ORM용 Aurora DSQL 어댑터](#)

Python 비동기 ORM 프레임워크인 Tortoise ORM에 대한 지원이 추가되었습니다. Tortoise ORM용 Aurora DSQL 어댑터를 사용하면 개발자가 Aurora DSQL 클러스터와 함께 Tortoise ORM을 사용할 수 있습니다. 자세한 내용은 [Aurora DSQL 어댑터 및 언어](#)를 참조하세요.

2026년 1월 23일

### [새 콘텐츠: Aurora DSQL 운영: 스کیل 및 파워](#)

스کیل 및 파워를 사용하여 Aurora DSQL의 AI 운영을 구성하기 위한 새로운 설명서가 추가되었습니다. Kiro 파워, Claude 스کیل, Gemini 스کیل 및 Codex 스کیل에 대한 설정 지침이 포함되어 있습니다. 자세한 내용은 [Aurora DSQL 운영: 스کیل 및 파워](#)를 참조하세요.

2026년 1월 16일

### [숫자 데이터 유형 인덱스 지원](#)

Aurora DSQL의 numeric 데이터 유형에 대한 인덱스 지원이 추가되었습니다. 이제 numeric 열을 기본 키로 사용하고 보조 인덱스에서 사용할 수 있습니다. 자세한 내용은 [Aurora DSQL에서 지원되는 데이터 유형](#)을 참조하세요.

2026년 1월 13일

[업데이트된 콘텐츠: 지원되는 SQL 명령 하위 집합](#)

탐색 및 명확성을 개선하기 위해 SQL 명령 설명서를 별도의 페이지로 재구성했습니다. 이제 각 명령(CREATE TABLE, ALTER TABLE, CREATE VIEW, ALTER VIEW, DROP VIEW)에 자체 전용 페이지가 있습니다. 자세한 내용은 [지원되는 SQL 명령 하위 집합](#)을 참조하세요.

2026년 1월 6일

[업데이트된 콘텐츠: AWS Labs Aurora DSQL MCP 서버](#)

CLI 기반 설정 및 구성 파일 예제를 포함하여 Claude Code 및 Codex에 대한 자세한 설치 접근 방식으로 MCP 서버 설명서를 업데이트했습니다. 다양한 개발 도구에서 MCP 클라이언트 구성 파일을 찾기 위한 포괄적인 지침이 추가되었습니다. 자세한 내용은 [AWS Labs Aurora DSQL MCP 서버](#)를 참조하세요.

2025년 12월 19일

[업데이트된 콘텐츠: PostgreSQL에서 Aurora DSQL로 마이그레이션](#)

포괄적인 마이그레이션 가이드로 PostgreSQL 호환성 섹션을 수정했습니다. 프레임워크 호환성 정보, 일반적인 마이그레이션 패턴, 아키텍처 차이 및 AI 지원 마이그레이션 지침을 포함합니다. Aurora DSQL에 대한 피드백을 제공하는 새로운 장을 추가했습니다. 자세한 내용은 [PostgreSQL에서 Aurora DSQL로 마이그레이션](#)을 참조하세요.

2025년 12월 16일

[업데이트된 콘텐츠: AWS PrivateLink를 사용하여 Aurora DSQL에 연결](#)

Direct Connect 또는 Amazon VPC 피어링과 함께 AWS PrivateLink를 사용하는 고객을 지원하기 위해 프라이빗 DNS 설정 및 클러스터 ID 연결 옵션에 대한 설명서가 추가되었습니다. `amzn-cluster-id` 연결 옵션을 사용하여 프라이빗 DNS 없이 연결하기 위한 지침이 포함되어 있습니다. 자세한 내용은 [AWS PrivateLink를 사용하여 Amazon Aurora DSQL 클러스터 관리 및 연결](#)을 참조하세요.

2025년 12월 11일

[업데이트된 콘텐츠: Aurora DSQL 클러스터 수명 주기](#)

Aurora DSQL 클러스터 수명 주기 관리에 대한 설명서가 업데이트되었습니다. 유희 및 비활성 상태에서 사용할 수 있는 클러스터 상태 정의, 상태 전환 및 작업에 대해 설명합니다. 자세한 내용은 [Aurora DSQL 클러스터 수명 주기](#)를 참조하세요.

2025년 12월 4일

[새 콘텐츠: Python 및 Node.js용 Aurora DSQL 커넥터](#)

Python(`psycopg`, `psycopg2`, `asyncpg`) 및 Node.js(`node-postgres`, `Postgres.js`)용 Aurora DSQL 커넥터에 대한 설명서가 추가되었습니다. 이 커넥터는 애플리케이션을 Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합합니다. 자세한 내용은 [Amazon DSQL 커넥터](#)를 참조하세요.

2025년 11월 21일

[새 콘텐츠: Aurora DSQL에서 JupyterLab 사용](#)

Python과 함께 JupyterLab을 사용하여 Aurora DSQL을 연결하고 쿼리하기 위한 단계별 가이드가 추가되었습니다. 로컬 JupyterLab 설치 및 Amazon SageMaker AI 환경에 대한 지침을 모두 포함합니다. 자세한 내용은 [Aurora DSQL에서 JupyterLab 사용](#)을 참조하세요.

2025년 11월 20일

[업데이트된 콘텐츠: Aurora DSQL 할당량](#)

최대 클러스터 스토리지 할당량을 128TiB에서 256TiB로 업데이트했습니다. 자세한 내용은 [Aurora DSQL 할당량](#)을 참조하세요.

2025년 11월 19일

[새 콘텐츠: Aurora DSQL 쿼리 편집기 시작하기](#)

AWS 관리 콘솔에서 Aurora DSQL 쿼리 편집기 사용에 대한 설명서가 추가되었습니다. 사전 조건, 연결 설정 및 쿼리 실행 지침이 포함되어 있습니다. 자세한 내용은 [Aurora DSQL 쿼리 편집기 시작하기](#)를 참조하세요.

2025년 11월 18일

## [Amazon Aurora DSQL에 대한 리소스 기반 정책 지원](#)

리소스 기반 정책(RBP) 지원을 PutClusterPolicy , GetClusterPolicy 및 DeleteClusterPolicy 와 같은 새 권한으로 추가했습니다. 이러한 권한을 통해 세분화된 액세스 제어를 위해 Aurora DSQL 클러스터에 연결된 인라인 정책을 관리할 수 있습니다. RBP 기능을 포함하도록 관리형 정책 AmazonAuroraDSQFullAccess, AmazonAuroraDSQLReadOnlyAccess 및 AmazonAuroraDSQLConsoleFullAccess를 업데이트했습니다. 자세한 내용은 [Amazon Aurora DSQL에 대한 AWS 관리형 정책](#)을 참조하세요.

2025년 10월 15일

## [Aurora DSQL JDBC 커넥터](#)

Java 애플리케이션을 Amazon Aurora DSQL 클러스터에 연결하기 위한 IAM 인증을 통합하는 PgJDBC 커넥터인 Aurora DSQL JDBC 커넥터에 대한 설명서가 추가되었습니다. 자세한 내용은 [JDBC 커넥터를 사용하여 Aurora DSQL 클러스터에 연결](#)을 참조하세요.

2025년 9월 2일

### [AWS FIS 통합을 위한 AWS 관리형 정책 업데이트](#)

Aurora DSQL과의 AWS Fault Injection Service 통합을 지원하도록 AmazonAuroraDSQLEFullAccess 및 AmazonAuroraDSQLEConsoleFullAccess 정책이 업데이트되었습니다. 이를 통해 단일 리전 및 다중 리전 Aurora DSQL 클러스터에 장애를 주입하여 애플리케이션의 내결함성을 테스트할 수 있습니다. 이러한 정책에 대한 자세한 내용은 [AWS 관리형 정책 업데이트](#)를 참조하세요.

2025년 8월 19일

### [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 DSQL](#)을 참조하세요.

2025년 5월 27일



### [AmazonAuroraDSQLFu llAccess 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 DSQL](#)을 참조하세요.

2025년 5월 21일

### [AmazonAuroraDSQLCo nsoleFullAccess 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 시작을 위해 고객 관리형 KMS 키를 사용하는 것을 지원합니다. 자세한 내용은 [AmazonAuroraDSQLConsoleFullAccess](#) 및 [Aurora DSQL에서 서비스 연결 역할 사용](#)을 참조하세요.

2025년 5월 21일

## [AmazonAuroraDSQLReadOnlyAccess 업데이트](#)

2025년 5월 13일

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 DSQL](#)을 참조하세요.

## [AmazonAuroraDSQLFu](#) [llAccess 업데이트](#)

2025년 5월 13일

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 [Aurora DSQL에서 서비스 연결 역할 사용](#)을 참조하세요.

### [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 [Aurora DSQL에서 서비스 연결 역할 사용](#)을 참조하세요.

2025년 5월 13일

### [AuroraDsqlServiceLinkedRole Policy 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 [AuroraDsqlServiceLinkedRole Policy](#) 및 [Using service-linked roles in Aurora DSQL](#)을 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 PrivateLink](#)를 참조하세요.

2025년 5월 8일

## [최초 릴리스입니다.](#)

Amazon Aurora DSQL 사용 설명서의 최초 릴리스입니다.

2024년 12월 3일