使用者指南

Amazon Aurora DSQL

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

Amazon Aurora DSQL: 使用者指南

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

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

什麼是 Amazon Aurora DSQL？ 1
使用情況 1
主要功能 1
AWS 區域 可用性 2
多區域叢集 4
定價 5
後續步驟？ 5
開始使用 6
先決條件 6
建立單一區域叢集 6
連接至叢集 7
執行 SQL 指令。 7
建立多區域叢集 8
疑難排解 10
身分驗證和授權 11
管理叢集 11
連線至叢集 11
PostgreSQL 和 IAM 角色 12
以 Aurora DSQL 使用 IAM 政策動作 13
使用 IAM 政策動作連線至叢集 13
使用 IAM 政策動作管理叢集 13
使用 IAM 和 PostgreSQL 撤銷授權 14
產生身分驗證記號 15
主控台 15
AWS CloudShell 16
AWS CLI 17
Aurora DSQL SDK 18
資料庫角色和 IAM 身分驗證 26
IAM 角色 26
IAM 使用者 27
連接 27
Query 27
檢視映射 28
撤銷 28

Aurora DSQL 和 PostgreSQL ... 29
相容性重點 ... 29
分散式架構優點 ... 30
SQL 相容性 ... 30
支援的資料類型 ... 30
支援的 SQL 功能 ... 35
支援的 SQL 命令子集 ... 39
遷移指南 ... 57
並行控制 ... 63
交易衝突 ... 64
最佳化交易效能的指導方針 ... 64
DDL 和分散式交易 ... 64
主索引鍵 ... 65
資料結構和儲存 ... 66
選擇主索引鍵的準則 ... 66
序列和身分資料欄 ... 66
序列處理函式 ... 67
身分資料欄 ... 69
使用序列和身分資料欄 ... 70
非同步索引 ... 71
語法 ... 72
Parameters ... 72
使用須知 ... 73
建立 索引 ... 74
查詢索引 ... 74
唯一索引建置失敗 ... 76
違反唯一性 ... 76
系統資料表和命令 ... 78
系統表 ... 78
有用的系統查詢 ... 88
ANALYZE 命令。 ... 89
EXPLAIN 計劃 ... 90
PostgreSQL EXPLAIN 計劃 ... 90
金鑰元素 ... 91
篩選 ... 92
讀取 EXPLAIN 計劃 ... 92

EXPLAIN ANALYZE 中的 DPUs 95
管理 Aurora 資料庫叢集 99
單一區域叢集 99
使用 AWS SDK 99
使用 AWS CLI 137
多區域叢集 141
使用 AWS SDK 141
使用 AWS CLI 194
CloudFormation 200
初始組態 200
尋找叢集 201
更新組態 201
Aurora DSQL 叢集生命週期 202
叢集狀態 202
檢視叢集狀態 204
使用 Aurora DSQL 進行程式設計 206
連接器 206
JDBC 連接器 207
Python 連接器 211
Go 連接器 222
Node.js 連接器 229
Ruby 連接器 236
.NET 連接器 242
存取 Aurora DSQL 247
SQL 用戶端 248
DBeaver 248
JetBrains DataGrip 251
Psql 252
VSCode 253
疑難排解 255
資料庫連線工具 255
Aurora DSQL 轉接器 255
資料庫驅動程式範例 256
ORM 和架構範例 257
載入資料 259
選擇載入方法 259

Aurora DSQL Loader ... 260
遷移途徑 ... 264
使用 PostgreSQL \copy ... 266
其他資源 ... 267
生成式 AI ... 267
AWS Labs Aurora DSQL MCP 伺服器 ... 267
Aurora DSQL 轉向：技能和能力 ... 275
查詢編輯器 ... 280
先決條件 ... 280
使用查詢編輯器 ... 281
查詢編輯器：搭配 Aurora DSQL 使用 JupyterLab ... 282
開始使用 ... 283
筆記本範例 ... 285
深入閱讀 ... 285
備份和還原 ... 286
AWS Backup 入門 ... 286
還原您的備份 ... 286
還原單一區域叢集 ... 287
還原多區域叢集 ... 287
監控與法規遵循 ... 287
其他資源 ... 287
監控和記錄 ... 289
使用 CloudWatch 進行監控 ... 289
可觀測性 ... 289
用量 ... 290
使用 CloudTrail 進行記錄 ... 292
管理事件 ... 292
資料事件 ... 294
安全 ... 295
AWS 受管政策 ... 296
AmazonAuroraDSQLFullAccess ... 296
AmazonAuroraDSQLReadOnlyAccess ... 297
AmazonAuroraDSQLConsoleFullAccess ... 297
AuroraDSQLServiceRolePolicy ... 299
政策更新 ... 299
資料保護 ... 303

資料加密 304
見證區域中的資料保護 305
SSL/TLS 憑證 305
資料加密 304
KMS 金鑰類型 311
靜態加密 312
使用 KMS 金鑰和資料金鑰 313
授權 KMS 金鑰 315
加密內容 316
監控 AWS KMS 317
建立加密的叢集 319
移除或更新金鑰 321
考量事項 323
身分與存取管理 324
目標對象 324
使用身分驗證 324
使用政策管理存取權 326
Aurora DSQL 如何搭配使用 IAM 327
身分型政策範例 331
疑難排解 336
資源型政策 338
何時使用 338
使用 政策建立 339
新增和編輯政策 342
檢視政策 345
移除政策 346
政策範例 348
封鎖公有存取權 352
API 操作 354
使用服務連結角色 356
Aurora DSQL 的服務連結角色許可權限 357
建立服務連結角色 358
編輯服務連結角色 358
刪除服務連結角色 358
Aurora DSQL 服務連結角色的支援區域 358
使用 IAM 條件索引鍵 358

- 在特定區域中建立叢集 359
- 在特定區域中建立多區域叢集 359
- 建立具有特定見證區域的多區域叢集 360
- 事件回應 361
- 法規遵循驗證 361
- 恢復能力 362
- 備份與還原 362
- 複寫 362
- 高可用性 363
- 故障注入測試 363
- 基礎設施安全性 364
- 使用 管理叢集 AWS PrivateLink 364
- 組態與漏洞分析 374
- 預防跨服務混淆代理人 375
- 安全最佳實務 376
- 偵測性安全最佳實務 376
- 預防性安全最佳實務 377
- 標記資源 379
- Name tag (名稱標籤) 379
- 標記需求 379
- 標籤使用注意事項 379
- 考量事項 381
- 配額和限制 383
- 叢集配額 383
- 資料庫限制 384
- API 參考 387
- 疑難排解 250
- 連線錯誤 388
- 身分驗證錯誤 388
- 授權錯誤 389
- SQL 錯誤 390
- OCC 錯誤 390
- SSL/TLS 連線 391
- 提供回饋 392
- 意見回饋管道 392
- 有效的功能請求 392

文件歷史紀錄 ... 393
... cdiv

什麼是 Amazon Aurora DSQL？

Amazon Aurora DSQL 是一項針對交易型工作負載最佳化的無伺服器分散式關聯式資料庫服務。Aurora DSQL 提供幾乎無限制的擴展能力，且無需您管理任何基礎設施。主動-主動高可用性架構可提供 99.99% 的單一區域可用性，以及 99.999% 的多區域可用性。

何時使用 Aurora DSQL

Aurora DSQL 已針對可受益於 ACID 交易特性與關聯式資料模型的交易型工作負載進行最佳化。由於採用無伺服器架構，Aurora DSQL 特別適合微型服務、無伺服器及事件驅動架構的應用程式設計模式。Aurora DSQL 與 PostgreSQL 相容，因此您可以使用熟悉的驅動程式、物件關聯對應 (ORM)、開發框架及 SQL 功能。

Aurora DSQL 會自動管理系統基礎設施，並依工作負載自動擴展運算、I/O 與儲存體資源。由於您無需佈建或管理伺服器，因此不必擔心與佈建、修補或基礎設施升級相關的維護停機時間。

Aurora DSQL 可協助您建置並維護在任何規模下皆可隨時使用的企業應用程式。主動-主動的無伺服器設計可自動處理故障復原，因此您不需擔心傳統資料庫的容錯移轉。您的應用程式可受益於多可用區域與多區域的高可用性，且無需擔心最終一致性或容錯移轉導致的資料遺失。

Aurora DSQL 的主要功能

下列主要功能可協助您建立無伺服器分散式資料庫，以支援高可用性的應用程式：

分散式架構

Aurora DSQL 由下列多租戶元件組成：

- 中繼與連線
- 運算與資料庫
- 交易日誌、並行控制與隔離性
- 儲存

控制平面負責協調前述各元件。各元件在三個可用區域 (AZ) 之間提供備援，並在元件發生故障時自動進行叢集擴展與自我修復。若要深入了解此架構如何支援高可用性，請參閱 Amazon Aurora DSQL 的恢復能力。

單一區域叢集與多區域叢集

Aurora DSQL 叢集提供以下優點：

- 同步資料複寫
- 一致性讀取作業
- 自動故障復原
- 跨多個可用區域 (AZ) 或區域的資料一致性

當基礎設施元件發生故障時，Aurora DSQL 會自動將請求導向運作正常的基礎設施，無需人工介入。Aurora DSQL 提供具備原子性、一致性、隔離性與耐久性 (ACID) 的交易，並實現強一致性、快照隔離、原子性以及跨可用區域與跨區域的耐久性。

多區域對等叢集提供與單一區域叢集相同的彈性及連線能力。但它們透過提供兩個區域端點 (每個對等叢集區域各一個) 提升可用性。對等叢集的兩個端點共同呈現單一邏輯資料庫。它們可同時進行讀取與寫入操作，並提供強大的資料一致性。您可以建置同時在多個區域運行的應用程式，以提升效能與彈性，且讀者始終能看到相同資料。

與 PostgreSQL 的相容性

Aurora DSQL 中的分散式資料庫層 (運算) 基於 PostgreSQL 的最新主要版本。您可以使用熟悉的 PostgreSQL 驅動程式與工具連接 Aurora DSQL，例如 `psql`。Aurora DSQL 目前與 PostgreSQL 第 16 版相容，並支援各種 PostgreSQL 功能、表達式和資料類型。如需瞭解支援的 SQL 功能詳情，請參閱 Aurora DSQL 中的 SQL 功能相容性。

Aurora DSQL 的區域可用性

使用 Amazon Aurora DSQL，您可以在多個 之間部署資料庫執行個體 AWS 區域 ，以支援全域應用程式並滿足資料駐留需求。區域可用性決定您可在何處建立並管理 Aurora DSQL 資料庫叢集。需設計高可用性、全球分散式資料庫系統的資料庫管理員與應用程式架構師，通常必須了解區域對其工作負載的支援情況。常見使用案例包括設定跨區域災難復原、讓使用者透過地理位置更接近的資料庫執行個體以降低延遲，並在特定位置維護資料副本以符合法規要求。

下表顯示目前可使用 Aurora DSQL AWS 區域 的 ，以及每個 DSQL 的端點 AWS 區域。

區域名稱	區域	端點	通訊協定
美國東部 (俄亥俄)	us-east-2	dsql.us-east-2.api.aws	HTTPS
		dsql-fips.us-east-2.api.aws	HTTPS

區域名稱	區域	端點	通訊協定
美國東部 (維吉尼亞北部)	us-east-1	dsql.us-east-1.api.aws dsql-fips.us-east-1.api.aws	HTTPS HTTPS
美國西部 (奧勒岡)	us-west-2	dsql.us-west-2.api.aws dsql-fips.us-west-2.api.aws	HTTPS HTTPS
亞太區域 (墨爾本)	ap-southeast-4	dsql.ap-southeast-4.api.aws	HTTPS
亞太區域 (大阪)	ap-northeast-3	dsql.ap-northeast-3.api.aws	HTTPS
亞太區域 (首爾)	ap-northeast-2	dsql.ap-northeast-2.api.aws	HTTPS
亞太地區 (悉尼)	ap-southeast-2	dsql.ap-southeast-2.api.aws	HTTPS
亞太區域 (東京)	ap-northeast-1	dsql.ap-northeast-1.api.aws	HTTPS
加拿大 (中部)	ca-central-1	dsql.ca-central-1.api.aws dsql-fips.ca-central-1.api.aws	HTTPS HTTPS
加拿大西部 (卡加利)	ca-west-1	dsql.ca-west-1.api.aws dsql-fips.ca-west-1.api.aws	HTTPS HTTPS
歐洲 (法蘭克福)	eu-central-1	dsql.eu-central-1.api.aws	HTTPS
歐洲 (愛爾蘭)	eu-west-1	dsql.eu-west-1.api.aws	HTTPS

區域名稱	區域	端點	通訊協定
歐洲 (倫敦)	eu-west-2	dsql.eu-west-2.api.aws	HTTPS
歐洲 (巴黎)	eu-west-3	dsql.eu-west-3.api.aws	HTTPS

Aurora DSQL 的多區域叢集可用性

您可以在特定 AWS 區域集中建立 Aurora DSQL 多區域叢集。每個區域集會將地理相關、可於多區域叢集中共同運作的區域分組。

美國區域

- 美國東部 (維吉尼亞北部)
- 美國東部 (俄亥俄)
- 美國西部 (奧勒岡)

亞太區域

- 亞太地區 (大阪)
- 亞太地區 (首爾)
- 亞太地區 (東京)

歐洲區域

- 歐洲 (法蘭克福)
- 歐洲 (愛爾蘭)
- 歐洲 (倫敦)
- Europe (Paris)

重要限制

多區域叢集必須在單一區域集內建立。例如，您無法建立同時包含美國東部 (維吉尼亞北部) 與歐洲 (愛爾蘭) 區域的叢集。

Important

Aurora DSQL 目前不支援跨洲多區域叢集。

Aurora DSQL 定價

如需成本相關資訊，請參閱 Aurora DSQL 定價。

後續步驟？

如需了解 Aurora DSQL 的核心元件與服務入門資訊，請參閱以下內容：

- Aurora DSQL 入門指南
- Aurora DSQL 中的 SQL 功能相容性
- 使用 PostgreSQL 相容用戶端存取 Aurora DSQL
- Aurora DSQL 和 PostgreSQL

Aurora DSQL 入門指南

Amazon Aurora DSQL 是針對交易工作負載最佳化的無伺服器、全受管、分散式關聯式資料庫。在下列各節中，您將了解如何建立單一區域和多區域 Aurora DSQL 叢集、與其連線，以及建立和載入範例結構描述。您將使用 AWS 主控台存取叢集，並選擇性地使用其他 PostgreSQL 用戶端與資料庫互動。最後，您將擁有可運作的 Aurora DSQL 叢集設定，可用於測試或生產工作負載。

主題

- 先決條件
- 步驟 1：建立 Aurora DSQL 單一區域叢集
- 步驟 2：連線至 Aurora DSQL 叢集
- 步驟 3：在 Aurora DSQL 中執行範例 SQL 指令
- 步驟 4（選用）：建立多區域叢集
- 疑難排解

先決條件

在開始使用 Aurora DSQL 前，請確認您已符合以下先決條件：

- 您的 IAM 身分必須具有登入 主控台的許可。
- 您的 IAM 身分必須符合下列條件：
  - 存取 以對 中的任何資源執行任何動作 AWS 帳戶
  - `AmazonAuroraDSQLConsoleFullAccess` AWS 已連接 受管政策。

步驟 1：建立 Aurora DSQL 單一區域叢集

Aurora DSQL 的基本單位為叢集，即資料儲存的位置。在此任務中，您將在單一 AWS 區域中建立叢集。

在 Aurora DSQL 中建立單一區域叢集

1. 登入 AWS 管理主控台 並開啟位於 的 Aurora DSQL 主控台https://console.aws.amazon.com/dsql。
2. 選擇建立叢集，然後選擇單一區域。

3. （選用） 變更預設名稱標籤的值。
4. （選用） 為此叢集新增其他標籤。
5. (選用) 在叢集設定中，選取以下任一選項：
   - 選取自訂加密設定 (進階) 以選擇或建立 AWS KMS key。如果您使用客戶受管金鑰，請確定金鑰政策授予 Aurora DSQL 必要的許可。如需詳細資訊，請參閱客戶自管金鑰的金鑰政策。
   - 選取啟用刪除保護，可防止刪除操作誤刪叢集。預設會啟用刪除保護。
   - 選取資源型政策 （ 進階 ），指定此叢集的存取控制政策。
6. 選擇 Create Cluster (建立叢集)。
7. 主控台會讓您返回叢集頁面。隨即出現通知橫幅，指出正在建立叢集。選取叢集 ID 以開啟叢集詳細資訊檢視。

步驟 2：連線至 Aurora DSQL 叢集

Aurora DSQL 支援多種連接到叢集的方式，包括 DSQL 查詢編輯器、 AWS CloudShell、本機 psql 用戶端和其他 PostgreSQL 相容工具。在此步驟中，您會使用 Aurora DSQL 查詢編輯器進行連線，這可讓您快速開始與新叢集互動。

使用查詢編輯器連線

1. 在 Aurora DSQL 主控台 (https://console.aws.amazon.com/dsql) 中，開啟叢集頁面，確認您的叢集建立已完成，且其狀態為作用中。
2. 從清單中選擇叢集，或選擇叢集 ID 以開啟叢集詳細資訊頁面。
3. 選擇使用查詢編輯器連線。
4. 為剛建立的叢集選擇 Connect as admin。
   - 您可以選擇性地與自訂角色連線，請參閱 使用資料庫角色和 IAM 身分驗證。

步驟 3：在 Aurora DSQL 中執行範例 SQL 指令

執行 SQL 陳述式以測試 Aurora DSQL 叢集。在查詢編輯器中開啟叢集後，依序選取並執行每個範例查詢。

在 Aurora DSQL 中執行範例 SQL 命令

1. 建立名為 `test` 的結構描述。

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

2. 建立 hello_world 資料表，使用自動產生的 UUID 做為主索引鍵。

```
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. 插入範例資料列。

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

4. 讀取插入的值。

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

5. 選擇性清除

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

步驟 4（選用）：建立多區域叢集

建立多區域叢集時，請指定下列區域：

遠端區域

此區域為建立第二個叢集的位置。在此區域建立第二個叢集，並將其與初始叢集對等互連。Aurora DSQL 會將初始叢集的所有寫入動作複寫至遠端叢集。您可在任一叢集上進行讀寫操作。

見證區域

此區域會接收寫入多區域叢集的所有資料。然而，見證區域不會託管用戶端端點，也不提供使用者資料存取功能。見證區域中會保留一段加密交易日誌。此日誌有助於資料復原，並在區域無法使用時支援交易仲裁機制。

使用下列程序建立初始叢集、在不同區域中建立第二個叢集，然後對等兩個叢集以建立多區域叢集。同時示範跨區域寫入複寫與兩個區域端點間的一致讀取。

建立多區域叢集

1. 登入 Aurora DSQL 主控台。
2. 在導覽窗格中，選擇叢集。
3. 選擇建立叢集，再選擇多區域。
4. （選用） 變更預設名稱標籤的值。
5. （選用） 為此叢集新增其他標籤。
6. 在多區域設定中，為初始叢集選擇以下設定項目：
    - 在見證區域中選擇區域。目前多區域叢集的見證區域僅支援美國地區。
    - (選用) 在遠端區域叢集 ARN 欄位中，輸入另一區域現有叢集的 ARN。若目前無可作為第二叢集的現有叢集，請於建立初始叢集後再完成設定。
7. (選用) 在叢集設定中，為初始叢集選擇以下任一選項：
    - 選取自訂加密設定 (進階) 以選擇或建立 AWS KMS key。如果您使用客戶受管金鑰，請確定金鑰政策授予 Aurora DSQL 必要的許可。如需詳細資訊，請參閱客戶自管金鑰的金鑰政策。
    - 選取啟用刪除保護，可防止刪除操作誤刪叢集。預設會啟用刪除保護。
    - 選取資源型政策 （ 進階 ），指定此叢集的存取控制政策。
8. 選擇建立叢集以建立初始叢集。如果您在前一個步驟中未輸入 ARN，主控台會顯示叢集設定擱置中通知。
9. 在叢集設定擱置中通知中，選擇完成多區域叢集設定。此操作會開始在另一個區域建立第二個叢集。
10. 針對第二個叢集，請選擇下列其中一個選項：
    - 新增遠端區域叢集 ARN – 若已存在叢集，且您希望將其設為多區域叢集中的第二個叢集，請選擇此選項。
    - 在另一個區域建立叢集 – 選擇此選項以建立第二個叢集。在遠端區域中，選擇第二個叢集所屬的區域。
11. 選擇在 ***your-second-region*** 建立叢集，其中 *your-second-region* 為第二個叢集的位置。主控台會在您的第二個區域開啟。
12. (選用) 選擇第二個叢集的設定。例如，您可以選擇 AWS KMS key。如果您使用客戶受管金鑰，請確定金鑰政策授予 Aurora DSQL 必要的許可。如需詳細資訊，請參閱客戶自管金鑰的金鑰政策。

13. 選擇建立叢集以建立第二個叢集。
14. 選擇在 ***initial-cluster-region*** 建立對等連線，其中 *initial-cluster-region* 為託管您第一個叢集的區域。
15. 出現提示時，請選擇確認。此步驟將完成多區域叢集的建立。

連線至第二個叢集

1. 開啟 Aurora DSQL 主控台，並選擇第二個叢集所在的區域。
2. 選擇 Clusters (叢集)。
3. 選取多區域叢集內第二個叢集的資料列。
4. 選擇使用查詢編輯器連線。
5. 選擇以管理員身分連線。
6. 建立範例結構描述和資料表，並依照中的步驟插入資料步驟 3：在 Aurora DSQL 中執行範例 SQL 指令。

從託管初始叢集的區域查詢第二個叢集的資料

1. 在 Aurora DSQL 主控台中，選擇初始叢集所在的區域。
2. 選擇 Clusters (叢集)。
3. 選取多區域叢集內第二個叢集的資料列。
4. 選擇使用查詢編輯器連線。
5. 選擇以管理員身分連線。
6. 查詢您匯入至第二個叢集的資料。

   Example

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

疑難排解

請參閱 Aurora DSQL 文件的故障診斷一節。

Aurora DSQL 的身分驗證和授權

Aurora DSQL 使用 IAM 角色和政策進行叢集授權。您可將 IAM 角色與 PostgreSQL 資料庫角色建立關聯以進行資料庫授權。這種方法結合了 IAM 的優點與 PostgreSQL 權限。Aurora DSQL 使用這些功能為您的叢集、資料庫和資料提供完整的授權和存取政策。

使用 IAM 管理叢集

管理叢集時請使用 IAM 進行身分驗證和授權：

IAM 身分驗證

若要在管理 Aurora DSQL 叢集時驗證 IAM 身分，就一定要使用 IAM。您可以使用 AWS 管理主控台、AWS CLI 或 AWS SDK 提供身分驗證。

IAM 授權

若要管理 Aurora DSQL 叢集，請使用 Aurora DSQL 的 IAM 動作授予授權。例如若要描述叢集，請確保您的 IAM 身分具有許可權限可以採取 IAM 動作 `dsql:GetCluster`，如下列政策動作範例所示。

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

如需詳細資訊，請參閱使用 IAM 政策動作管理叢集。

使用 IAM 連線至叢集

若要連線至叢集，請使用 IAM 進行身分驗證和授權：

IAM 身分驗證

使用具有授權的 IAM 身分產生臨時身分驗證記號以連線至叢集。如需詳細資訊，請參閱 在 Amazon Aurora DSQL 產生身分驗證記號。

IAM 授權

將下列 IAM 政策動作授予至您用來建立連線至叢集端點的 IAM 身分：

- 如果您使用的是 admin 角色，請使用 dsql:DbConnectAdmin。Aurora DSQL 會為您建立和管理此角色。下列 IAM 政策動作範例允許 admin 連線至 *my-cluster*。

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

- 如果您使用的是自訂資料庫角色，請使用 dsql:DbConnect。您可在資料庫使用 SQL 命令建立和管理此角色。下列 IAM 政策動作範例允許自訂資料庫角色連線至 *my-cluster* 長達一小時。

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

您建立連線後，角色就獲得授權可連線最長一小時。

使用 PostgreSQL 資料庫角色和 IAM 角色與資料庫互動

PostgreSQL 使用角色概念管理資料庫存取許可權限。視角色的設定方式而定，可將角色視為資料庫使用者或資料庫使用者群組。您可使用 SQL 命令建立 PostgreSQL 角色。若要管理資料庫層級授權，請將 PostgreSQL 許可權限授予 PostgreSQL 資料庫角色。

Aurora DSQL 支援兩種類型的資料庫角色：admin 角色和自訂角色。Aurora DSQL 會自動在 Aurora DSQL 叢集中為您建立預先定義的 admin 角色。您無法修改 admin 角色。您以 admin 身分連線至資料庫時，可以發出 SQL 建立新的資料庫層級角色，以與您的 IAM 角色建立關聯。若要讓 IAM 角色連線至您的資料庫，請將您的自訂資料庫角色與 IAM 角色建立關聯。

身分驗證

使用 admin 角色連線至您的叢集。連線資料庫後，請使用 AWS IAM GRANT 命令將自訂資料庫角色與有權連線叢集的 IAM 身分建立關聯，如下列範例所示。

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

如需詳細資訊，請參閱 授權資料庫角色連線至您的叢集。

授權

使用 admin 角色連線至您的叢集。執行 SQL 命令以設定自訂資料庫角色並授予許可權限。如需詳細資訊，請參閱 PostgreSQL 文件中的 PostgreSQL 資料庫角色和 PostgreSQL 權限。

以 Aurora DSQL 使用 IAM 政策動作

您所能使用的 IAM 政策動作，取決於您用來連線至叢集的角色：可能是 admin 或自訂資料庫角色。政策也取決於此角色所需的 IAM 動作。

使用 IAM 政策動作連線至叢集

您使用 admin 預設資料庫角色連線至叢集時，請使用有權執行下列 IAM 政策動作的 IAM 身分。

```
"dsql:DbConnectAdmin"
```

您使用自訂資料庫角色連線至叢集時，請先將 IAM 角色與資料庫角色建立關聯。您用來連線至叢集的 IAM 身分，必須有權執行下列 IAM 政策動作。

```
"dsql:DbConnect"
```

若要進一步瞭解自訂資料庫角色，請參閱 使用資料庫角色和 IAM 身分驗證。

使用 IAM 政策動作管理叢集

管理 Aurora DSQL 叢集時，請僅為角色需要執行的動作指定政策動作。例如若您的角色只需取得叢集資訊，您可以限制角色許可權限為僅限 GetCluster 和 ListClusters 許可權限，如下列範例政策所示

JSON

```
{
"Version":"2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
```

```
          "dsql:GetCluster",
          "dsql:ListClusters"
        ],
        "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/my-cluster"
      }
    ]
}
```

下列範例政策顯示所有可用於管理叢集的 IAM 政策動作。

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" : "*"
    }
  ]
}
```

使用 IAM 和 PostgreSQL 撤銷授權

您可以撤銷 IAM 角色存取資料庫層級角色的許可權限：

撤銷連線至叢集的管理員授權

若要撤銷以 `admin` 角色連線至叢集的授權，請撤銷 IAM 身分對 `dsql:DbConnectAdmin` 的存取權限。您可編輯 IAM 政策，或是讓身分與政策分離。

從 IAM 身分撤銷連線授權後，Aurora DSQL 會拒絕該 IAM 身分嘗試的所有新連線。任何使用該 IAM 身分的作用中連線，在連線期間可能會保持授權狀態。如需有關連線期間的詳細資訊，請參閱配額與限制。

撤銷自訂角色連線叢集的授權

若要撤銷 admin 以外存取資料庫角色的權限，請撤銷 IAM 身分對 dsql:DbConnect 的存取權限。您可編輯 IAM 政策，或是讓身分與政策分離。

您也可以使用資料庫的命令 AWS IAM REVOKE，以移除資料庫角色與 IAM 之間的關聯。若要進一步瞭解如何從資料庫角色撤銷存取權限，請參閱 從 IAM 角色撤銷資料庫授權。

您無法管理預先定義 admin 資料庫角色的許可權限。若要瞭解如何管理自訂資料庫角色的許可權限，請參閱 PostgreSQL 權限。Aurora DSQL 成功認可修改交易後，權限修改會在下一個交易生效。

在 Amazon Aurora DSQL 產生身分驗證記號

若要使用 SQL 用戶端連線至 Amazon Aurora DSQL，請產生身分驗證記號作為密碼。此記號僅用於驗證連線。連線建立後，即使身分驗證記號過期，連線仍然有效。

如果您使用 AWS 主控台 AWS CLI、 或 SDKs 建立身分驗證字符，字符預設會在 15 分鐘內自動過期。期限上限為 604,800 秒，也就是一週。若要從用戶端再次連線至 Aurora DSQL，您可以使用尚未過期的相同身分驗證記號，也可以產生新的記號。

若要開始產生記號，請在 Aurora DSQL 中建立 IAM 政策和叢集。然後使用 AWS 主控台 AWS CLI、 或 AWS SDKs來產生字符。

視您用來連線的資料庫角色而定，您至少必須擁有 使用 IAM 連線至叢集 列出的 IAM 許可權限。

主題

- 使用 AWS 主控台在 Aurora DSQL 中產生身分驗證字符
- 使用 在 Aurora DSQL 中 AWS CloudShell 產生身分驗證字符
- 使用 AWS CLI 在 Aurora DSQL 中產生身分驗證字符
- 使用 SDK 在 Aurora DSQL 中產生記號

使用 AWS 主控台在 Aurora DSQL 中產生身分驗證字符

Aurora DSQL 會使用記號而非密碼驗證使用者身分。您可以從主控台產生記號。

產生身分驗證記號

1. 登入 AWS 管理主控台 並開啟位於 的 Aurora DSQL 主控台https://console.aws.amazon.com/dsql。
2. 為您要產生身分驗證記號的叢集選擇叢集 ID。如果您尚未建立叢集，請遵循 步驟 1：建立 Aurora DSQL 單一區域叢集 或 步驟 4（選用）：建立多區域叢集 中的步驟。
3. 選擇連線，然後選擇取得記號。
4. 選擇您要以 admin 或自訂資料庫角色進行連線。
5. 複製產生的身分驗證記號並將其用於 使用 SQL 用戶端存取 Aurora DSQL。

若要進一步瞭解 Aurora DSQL 的自訂資料庫角色和 IAM，請參閱 身分驗證和授權。

使用 在 Aurora DSQL 中 AWS CloudShell 產生身分驗證字符

在使用 產生身分驗證字符之前 AWS CloudShell，請確定您建立 Aurora DSQL 叢集。

使用 產生身分驗證字符 AWS CloudShell

1. 登入 AWS 管理主控台 並開啟位於 的 Aurora DSQL 主控台https://console.aws.amazon.com/dsql。
2. 在 AWS 主控台的左下角，選擇 AWS CloudShell。
3. 執行下列命令以產生 admin 角色的身分驗證記號。以您的區域取代 *us-east-1*，並以您的叢集端點取代 *your_cluster_endpoint*。

   > **Note**
   >
   > 如果您不是以 admin 身分連線，請改用 generate-db-connect-auth-token。

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

   如果您遇到問題，請參閱對 IAM 進行疑難排解和如何對 IAM 政策的存取遭拒或未經授權的操作錯誤進行疑難排解？。
4. 使用下列命令以 psql 開始連線至叢集。

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

5. 您應該會看到提示要求提供密碼。複製您產生的記號，並確保其中不包含任何額外的空格或字元。從 psql 將其貼上至下列提示。

```
Password for user admin:
```

6. 按 Enter。系統應顯示 PostgreSQL 提示字元。

```
postgres=>
```

如果您收到存取遭拒錯誤，請確保您的 IAM 身分具有 dsql:DbConnectAdmin 許可權限。如果您具有許可權限但仍繼續發生存取遭拒錯誤，請參閱對 IAM 進行疑難排解和如何對 IAM 政策的存取遭拒或未經授權的操作錯誤進行疑難排解？。

若要進一步瞭解 Aurora DSQL 的自訂資料庫角色和 IAM，請參閱 身分驗證和授權。

使用 AWS CLI 在 Aurora DSQL 中產生身分驗證字符

若您的叢集為 ACTIVE，可以使用 aws dsql 命令在 CLI 上產生身分驗證記號。請使用下列任一技巧：

> **Note**
>
> 權杖產生是使用您目前的 IAM 登入資料簽署請求的本機操作。它不會聯絡 AWS 來驗證登入資料。如果您的登入資料已過期或無效，權杖產生仍然成功，但連線嘗試失敗。在產生字符之前，請確定您的 IAM 登入資料有效。

- 如果您要以 admin 角色連線，請使用 generate-db-connect-admin-auth-token 選項。
- 如果您要以自訂資料庫角色連線，請使用 generate-db-connect-auth-token 選項。

下列範例使用下列屬性產生 admin 角色的身分驗證記號。

- *your_cluster_endpoint*：叢集端點。其中遵循格式 *your_cluster_identifier*.dsql.*region*.on.aws，如範例 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws 所示。
- *region* – AWS 區域，例如 us-east-2或 us-east-1。

下列範例設定的到期時間為記號在 3,600 秒 (1 小時) 後到期。

Linux and macOS

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

Windows

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

使用 SDK 在 Aurora DSQL 中產生記號

您可以在叢集處於 ACTIVE 狀態時為叢集產生身分驗證記號。SDK 範例使用以下屬性產生 admin 角色的身分驗證記號：

- *your_cluster_endpoint* (或 *yourClusterEndpoint*)：Aurora DSQL 叢集端點。命名格式為 *your_cluster_identifier*.dsql.*region*.on.aws，如範例 01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws 所示。
- *region* （或 *RegionEndpoint*）– AWS 區域 叢集所在的 ，例如 us-east-2或 us-east-1。

Python SDK

Tip

AWS 建議使用 適用於 Python 的 Aurora DSQL 連接器，其會自動處理字符產生。

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 generate_db_connect_admin_auth_token。
- 如果您要與自訂資料庫角色連線，請使用 generate_connect_auth_token。

```
import boto3

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

C++ SDK

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 GenerateDBConnectAdminAuthToken。
- 如果您要與自訂資料庫角色連線，請使用 GenerateDBConnectAuthToken。

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

using namespace Aws;
using namespace Aws::DSQL;

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

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

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

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

JavaScript SDK

Tip

AWS 建議使用 適用於 Node.js 的 Aurora DSQL 連接器，其會自動處理字符產生。

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 getDbConnectAdminAuthToken。
- 如果您要與自訂資料庫角色連線，請使用 getDbConnectAuthToken。

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

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

Java SDK

> **Tip**
>
> AWS 建議使用 使用 JDBC 連接器連線至 Aurora DSQL 叢集，其會自動處理字符產生。

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 generateDbConnectAdminAuthToken。
- 如果您要與自訂資料庫角色連線，請使用 generateDbConnectAuthToken。

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

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

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

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

Rust SDK

您可以透過下列方式產生記號：

- 如果您要與 `admin`角色連線，請使用 `db_connect_admin_auth_token`。
- 如果您要與自訂資料庫角色連線，請使用 `db_connect_auth_token`。

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

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

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

Ruby SDK

> **Tip**
>
> AWS 建議使用 使用 Ruby 連接器連接到 Aurora DSQL 叢集，其會自動處理字符產生。

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 generate_db_connect_admin_auth_token。
- 如果您要與自訂資料庫角色連線，請使用 generate_db_connect_auth_token。

```
require 'aws-sdk-dsql'

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

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

  # if you're not using admin role, use generate_db_connect_auth_token instead
```

```
  token = token_generator.generate_db_connect_admin_auth_token({
    :endpoint => your_cluster_endpoint,
    :region => region
  })
end
```

PHP SDK

您可以透過下列方式產生記號：

- 如果您要與 admin角色連線，請使用 generateDbConnectAdminAuthToken。
- 如果您要與自訂資料庫角色連線，請使用 generateDbConnectAuthToken。

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

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

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

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

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

.NET

Tip

AWS 建議使用 使用 .NET 連接器連接到 Aurora DSQL 叢集，其會自動處理字符產生。

適用於 .NET 的官方 SDK 並未包含內建 API 呼叫用於產生 Aurora DSQL 的身分驗證記號。您必須改用 `DSQLAuthTokenGenerator` 這項公用程式類別。下列程式碼範例示範如何產生 .NET 的身分驗證記號。

您可以透過下列方式產生記號：

- 如果您要與 `admin`角色連線，請使用 `DbConnectAdmin`。
- 如果您要與自訂資料庫角色連線，請使用 `DbConnect`。

下列範例使用 `DSQLAuthTokenGenerator` 公用程式類別，為具有 `admin` 角色的使用者產生身分驗證記號。以您的叢集端點取代 *`insert-dsql-cluster-endpoint`*。

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

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

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

Console.WriteLine(token);
```

Go

AWS 建議使用 使用 Go 連接器連線至 Aurora DSQL 叢集，其會自動處理字符產生。

SDK AWS for Go v2 提供在`github.com/aws/aws-sdk-go-v2/feature/dsql/auth`套件中產生身分驗證字符的內建方法。

- 如果您要與 `admin`角色連線，請使用 `auth.GenerateDBConnectAdminAuthToken`。

- 如果您要與自訂資料庫角色連線，請使用 `auth.GenerateDbConnectAuthToken`。

```
package main

import (
 "context"
 "fmt"

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

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

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

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

 fmt.Println(token)
}
```

使用資料庫角色和 IAM 身分驗證

Aurora DSQL 支援使用 IAM 角色和 IAM 使用者進行身分驗證。您可使用任一種方法驗證和存取 Aurora DSQL 資料庫。

IAM 角色

IAM 角色是 中的身分 AWS 帳戶 ，具有特定許可，但未與特定人員建立關聯。使用 IAM 角色提供臨時安全憑證。您可透過多種方式暫時擔任 IAM 角色：

- 透過在 中切換角色 AWS 管理主控台
- 透過呼叫 AWS CLI 或 AWS API 操作
- 使用自訂 URL

您擔任角色之後，就可使用角色的臨時憑證存取 Aurora DSQL。如需使用角色方法的詳細資訊，請參閱 IAM 使用者指南中的 IAM 身分。

IAM 使用者

IAM 使用者是 中的身分 AWS 帳戶 ，具有特定許可，並與單一人員或應用程式相關聯。IAM 使用者具有長期憑證，例如可用於存取 Aurora DSQL 的密碼和存取金鑰。

> **Note**
> 若要使用 IAM 身分驗證執行 SQL 命令，您可在以下範例中使用 IAM 角色 ARN 或 IAM 使用者 ARN。

授權資料庫角色連線至您的叢集

建立 IAM 角色並使用 IAM 政策動作授予連線授權：`dsql:DbConnect`。

IAM 政策也必須授予許可權限以存取叢集資源。使用萬用字元 (*) 或遵循使用 IAM 條件索引鍵搭配 Amazon Aurora DSQL 中的指示。

授權資料庫角色在您的資料庫使用 SQL

您必須使用具有授權的 IAM 角色連線至叢集。

1. 使用 SQL 公用程式連線至 Aurora DSQL 叢集。

   使用 `admin` 資料庫角色搭配 IAM 身分，且該身分已獲授權讓 IAM 動作 `dsql:DbConnectAdmin` 連線至您的叢集。

2. 建立新的資料庫角色，請務必指定 `WITH LOGIN` 選項。

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

3. 將資料庫角色與 IAM 角色 ARN 建立關聯。

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

4. 將資料庫層級許可權限授予資料庫角色

   下列範例使用 GRANT 命令在資料庫中提供授權。

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

如需更多詳細資訊，請參閱 PostgreSQL 文件中的 PostgreSQLGRANT 和 PostgreSQL 權限。

檢視 IAM 至資料庫角色映射

若要檢視 IAM 角色和資料庫角色之間的映射，請查詢 `sys.iam_pg_role_mappings` 系統資料表。

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

輸出範例：

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

此資料表顯示 IAM 角色 (依其 ARN 識別) 和 PostgreSQL 資料庫角色之間的所有映射。

從 IAM 角色撤銷資料庫授權

若要撤銷資料庫授權，請使用 `AWS IAM REVOKE` 操作。

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

若要進一步瞭解撤銷授權，請參閱 使用 IAM 和 PostgreSQL 撤銷授權。

Aurora DSQL 和 PostgreSQL

Aurora DSQL 是與 PostgreSQL 相容的分散式關聯式資料庫，專為交易工作負載而設計。Aurora DSQL 使用核心 PostgreSQL 元件，例如解析器、規劃器、最佳化工具和類型系統。

Aurora DSQL 設計確保所有支援的 PostgreSQL 語法提供相容的行為，並產生相同的查詢結果。例如，Aurora DSQL 提供類型轉換、算術運算，以及與 PostgreSQL 相同的數值精確度和小數位數。系統會記錄任何偏差。

Aurora DSQL 也導入進階功能，例如開放式並行控制和分散式結構描述管理。透過這些功能，您可以使用熟悉的 PostgreSQL 工具，同時受益於現代、雲端原生、分散式應用程式的效能和可擴展性。

PostgreSQL 相容性重點

Aurora DSQL 目前是以 PostgreSQL 第 16 版為基礎。重點包括下列項目：

有線通訊協定

Aurora DSQL 使用標準 PostgreSQL v3 線路通訊協定。因此可以整合標準 PostgreSQL 用戶端、驅動程式和工具。例如，Aurora DSQL 與 `psql`、`pgjdbc` 和 `psycopg` 相容。

SQL 語法

Aurora DSQL 支援交易工作負載中常用的各種標準 PostgreSQL 運算式和函數。支援的 SQL 運算式會產生與 PostgreSQL 相同的結果，包括下列項目：

- 處理 null
- 排序訂單行為
- 數值運算的小數位數和精確度
- 字串運算的等效性

如需詳細資訊，請參閱Aurora DSQL 中的 SQL 功能相容性。

交易管理

Aurora DSQL 會保留 PostgreSQL 的主要特性，例如 ACID 交易和相當於 PostgreSQL 可重複讀取的隔離層級。如需詳細資訊，請參閱Aurora DSQL 的並行控制。

分散式架構優點

Aurora DSQL 的分散式、共用物件設計可提供超越傳統單一節點資料庫的效能和可擴展性優勢。主要功能包括下列項目：

開放式並行存取控制 (OCC)

Aurora DSQL 使用開放式並行存取控制模型。這種無鎖定方法可防止交易彼此封鎖、消除死結，並啟用高輸送量平行執行。這些功能使 Aurora DSQL 特別適用於需要大規模一致效能的應用程式。如需更多範例，請參閱 Aurora DSQL 的並行控制。

非同步 DDL 作業

Aurora DSQL 會以非同步方式執行 DDL 作業，確保在結構描述變更期間不間斷的讀取和寫入。其分散式架構可讓 Aurora DSQL 執行下列動作：

- 以背景任務執行 DDL 作業，將中斷降至最低。
- 座標目錄會隨著高度一致的分散式交易而變更。這可確保所有節點的原子可見度，即使在故障或並行作業期間也是如此。
- 使用分開的運算和儲存層，在多個可用區域中以完全分散式、無領導的方式操作。

如需在 PostgreSQL 中使用 EXPLAIN 命令的詳細資訊，請參閱 Aurora DSQL 的 DDL 和分散式交易。

Aurora DSQL 中的 SQL 功能相容性

下列各節說明 Aurora DSQL 支援的 PostgreSQL 資料類型和 SQL 命令。

主題

- Aurora DSQL 支援的資料類型
- 支援的 SQL for Aurora DSQL
- Aurora DSQL 中支援的 SQL 命令子集
- 從 PostgreSQL 遷移至 Aurora DSQL

Aurora DSQL 支援的資料類型

Aurora DSQL 支援一部分常見的 PostgreSQL 類型。

主題

- 數值資料類型
- 字元資料類型
- 日期和時間資料類型
- 其他資料類型
- 查詢執行時期資料類型

數值資料類型

Aurora DSQL 支援下列 PostgreSQL 數值資料類型。

名稱	別名	範圍和精確度	儲存大小	索引支援
`smallint`	`int2`	-32768 到 +32767	2 位元組	是
`integer`	`int`, `int4`	-2147483648 到 +2147483647	4 位元組	是
`bigint`	`int8`	-9223372036854775808 至 +9223372036854775807	8 位元組	是
`real`	`float4`	6 個小數位數精確度	4 位元組	是
`double precision`	`float8`	15 個小數位數精確度	8 位元組	是
`numeric [ (p, s) ]`	`decimal [ (p, s) ]` `dec[ (p,s)]`	可選擇精確度 (有效位數) 的精確數值。最高精確度為 38、最大的小數位數為 37。[1] 預設值為 `numeric (18,6)`。	8 位元組 + 每個精確度位數 2 位元組。大小上限為 27 個位元組。	是

[1] – 如果您在執行 `CREATE TABLE` 或 `ALTER TABLE ADD COLUMN` 時未明確指定大小，Aurora DSQL 會強制執行預設值。當您執行 `INSERT` 或 `UPDATE` 陳述式時，Aurora DSQL 會套用限制。

字元資料類型

Aurora DSQL 支援下列 PostgreSQL 字元資料類型。

名稱	別名	Description	Aurora DSQL 限制	儲存大小	索引支援
character [(*n*)]	char [(*n*)]	固定長度的字元字串	4096 位元組[1]	可變上限為 4100 位元組	是
character varying [(*n*)]	varchar [(*n*)]	可變長度字元字串	65535 位元組[1]	可變上限為 65539 位元組	是
bpchar [(*n*)]		如果長度固定，這是 char 的別名。如果長度可變，這是 varchar 的別名，其中結尾空格在語義上是無關緊要的。	4096 位元組[1]	可變上限為 4100 位元組	是
text		可變長度字元字串	1 MiB[1]	可變上限為 1 MiB	是

1 – 如果您在執行 CREATE TABLE 或 ALTER TABLE ADD COLUMN 時未明確指定大小，Aurora DSQL 會強制執行預設值。當您執行 INSERT 或 UPDATE 陳述式時，Aurora DSQL 會套用限制。

日期和時間資料類型

Aurora DSQL 支援下列 PostgreSQL 日期和時間資料類型。

名稱	別名	Description	範圍	Resolution	儲存大小	索引支援
date		日曆日期 (年、月、日)	4713 BC – 5874897 AD	1 天	4 位元組	是

名稱	別名	Description	範圍	Resolution	儲存大小	索引支援
time [(p)] [without time zone]	times	當日時間 (不含時區)	0 – 1	1 毫秒	8 位元組	是
time [(p)] with time zone	timet	當日時間 (含時區)	00:00:00+1559 – 24:00:00 –1559	1 毫秒	12 位元組	否
timestamp [(p)] [without time zone]		日期和時間 (不含時區)	4713 BC – 294276 AD	1 毫秒	8 位元組	是
timestamp [(p)] with time zone	times tz	日期和時間 (含時區)	4713 BC – 294276 AD	1 毫秒	8 位元組	是
interval [fields] [(p)]		時間範圍	-178000000 年 – 178000000 年	1 毫秒	16 位元組	否

其他資料類型

Aurora DSQL 支援下列其他 PostgreSQL 資料類型。

名稱	別名	Description	Aurora DSQL 限制	儲存大小	索引支援
boolean	bool	邏輯布林值 (true/false)		1 位元組	是
bytea		二進位資料 (「位元組陣列」)	1 MiB[1]	可變上限限制為 1 MiB	否

名稱	別名	Description	Aurora DSQL 限制	儲存大小	索引支援
UUID		通用唯一識別碼		16 位元組	是

1 – 如果您在執行 `CREATE TABLE` 或 `ALTER TABLE ADD COLUMN` 時未明確指定大小，Aurora DSQL 會強制執行預設值。當您執行 `INSERT` 或 `UPDATE` 陳述式時，Aurora DSQL 會套用限制。

查詢執行時期資料類型

查詢執行時期資料類型是在查詢執行時間使用的內部資料類型。這些類型與您在結構描述中定義的 PostgreSQL 相容類型不同，例如 `varchar` 和 `integer`。相反地，這些類型是 Aurora DSQL 在處理查詢時使用的執行時期表示法。

下列資料類型唯有在查詢執行時期受支援：

陣列類型

Aurora DSQL 支援受支援的資料類型陣列。例如，您可以使用整數陣列。函數 `string_to_array` 會將字串分割為 PostgreSQL 樣式陣列，並以逗號分隔符號 (`,`) 表示，如下列範例所示。查詢執行期間，您可以在運算式、函數輸出或暫時運算中使用陣列。

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

函數會傳回類似以下內容的回應：

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

inet 類型

資料類型代表 IPv4、IPv6 主機位址及其子網路。此類型有助於剖析日誌、篩選 IP 子網路或在查詢中進行網路計算。如需詳細資訊，請參閱 PostgreSQL 文件中的 inet。

JSON 執行時期函數

Aurora DSQL 支援 JSON 和 JSONB 作為查詢處理的執行時間資料類型。將 JSON 資料儲存為`text`資料欄，並在查詢執行期間轉換為 JSON，以使用 PostgreSQL JSON 函數和運算子。

Aurora DSQL 支援 9.1.6 節 JSON 函數和運算子中的大多數 PostgreSQL JSON 函數，其行為相同。

傳回 JSON 或 JSONB 類型的函數可能需要額外轉換為 `text`，才能正確顯示。

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

函數會傳回類似以下內容的回應：

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

支援的 SQL for Aurora DSQL

Aurora DSQL 支援各種核心 PostgreSQL SQL 功能。下列各節說明一般 PostgreSQL 運算式支援。此清單並不詳盡。

SELECT 命令

Aurora DSQL 支援 `SELECT` 命令的下列子句。

主要子句	支援的子句
`FROM`	
`GROUP BY`	`ALL`, `DISTINCT`
`ORDER BY`	`ASC`, `DESC`, `NULLS`
`LIMIT`	
`DISTINCT`	
`HAVING`	
`USING`	
`WITH` (一般資料表表達式)	

主要子句	支援的子句
INNER JOIN	ON
OUTER JOIN	LEFT, RIGHT, FULL, ON
CROSS JOIN	ON
UNION	ALL
INTERSECT	ALL
EXCEPT	ALL
OVER	RANK (), PARTITION BY
FOR UPDATE	

資料定義語言 (DDL)

Aurora DSQL 支援下列 PostgreSQL DDL 命令。

命令	主要子句	支援的子句
CREATE	TABLE	如需 CREATE TABLE 命令支援語法的相關資訊，請參閱 CREATE TABLE。
ALTER	TABLE	如需 ALTER TABLE 命令支援語法的相關資訊，請參閱 ALTER TABLE。
DROP	TABLE	
CREATE	[UNIQUE] INDEX ASYNC	此命令可搭配使用以下參數：ON、NULLS FIRST、NULLS LAST。 如需 CREATE INDEX ASYNC 命令支援語法的相關資訊，請參閱 Aurora DSQL 的非同步索引。

命令	主要子句	支援的子句
DROP	INDEX	
CREATE	VIEW	如需 CREATE VIEW 命令支援語法的詳細資訊，請參閱 CREATE VIEW。
ALTER	VIEW	如需 ALTER VIEW 命令支援語法的相關資訊，請參閱 ALTER VIEW。
DROP	VIEW	如需 DROP VIEW 命令支援語法的相關資訊，請參閱 DROP VIEW。
CREATE	SEQUENCE	如需 CREATE SEQUENCE 命令支援語法的相關資訊，請參閱 CREATE SEQUENCE。
ALTER	SEQUENCE	如需 ALTER SEQUENCE 命令支援語法的相關資訊，請參閱 ALTER SEQUENCE。
DROP	SEQUENCE	如需 DROP SEQUENCE 命令支援語法的相關資訊，請參閱 DROP SEQUENCE。
CREATE	ROLE, WITH	
CREATE	FUNCTION	LANGUAGE SQL
CREATE	DOMAIN	

資料處理語言 (DML)

Aurora DSQL 支援下列 PostgreSQL DML 命令。

命令	主要子句	支援的子句
INSERT	INTO	VALUES SELECT
UPDATE	SET	WHERE (SELECT)

命令	主要子句	支援的子句
		FROM, WITH
DELETE	FROM	USING, WHERE

資料控制語言 (DCL)

Aurora DSQL 支援下列 PostgreSQL DCL 命令。

命令	支援的子句
GRANT	ON, TO
REVOKE	ON, FROM, CASCADE, RESTRICT

交易控制語言 (TCL)

Aurora DSQL 支援下列 PostgreSQL TCL 命令。

命令	支援的子句	Alias (別名)
COMMIT	[WORK | TRANSACTION] [AND NO CHAIN]	END
BEGIN	[WORK | TRANSACTION] [ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
START TRANSACTION	[ISOLATION LEVEL REPEATABLE READ] [READ WRITE | READ ONLY]	
ROLLBACK	[WORK | TRANSACTION]	ABORT

命令	支援的子句	Alias (別名)
	[AND NO CHAIN]	

公用程式命令

Aurora DSQL 支援下列 PostgreSQL 公用程式命令：

- EXPLAIN
- ANALYZE (僅限關聯名稱)

Aurora DSQL 中支援的 SQL 命令子集

本節提供有關支援的 SQL 命令的詳細資訊，著重於具有大量參數集和子命令的命令。例如，PostgreSQL 中的 CREATE TABLE 提供許多子句和參數，其中一部分由 Aurora DSQL 支援。本節使用 Aurora DSQL 支援的熟悉 PostgreSQL 語法元素，說明常見 SQL 命令的支援子集。

主題

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

CREATE TABLE

CREATE TABLE 可定義新的資料表。

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

身分資料欄

> **Note**
>
> 使用身分資料欄時，應仔細考慮快取值。如需詳細資訊，請參閱 CREATE SEQUENCE 頁面上的重要標註。
> 如需如何根據工作負載模式最佳使用身分資料欄的指引，請參閱 使用序列和身分資料欄。

`GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY ( sequence_options )` 子句會將資料欄建立為身分資料欄。它將有一個隱含序列連接到它，在新插入的資料列中，資料欄將自動具有來自指派給它的序列的值。這類資料欄隱含為 `NOT NULL`。

子句ALWAYS和 會BY DEFAULT決定 INSERT和 UPDATE命令中使用者指定值的明確處理方式。

在 INSERT命令中，如果選取 ALWAYS ，則只有在INSERT陳述式指定 時，才會接受使用者指定的值OVERRIDING SYSTEM VALUE。如果選取 BY DEFAULT ，則使用者指定的值優先。

在 UPDATE命令中，如果選取 ALWAYS ，則會DEFAULT拒絕將資料欄更新為 以外的任何值。BY DEFAULT 如果選取 ，則欄可以正常更新。(UPDATE命令沒有 OVERRIDING子句。)

sequence_options 子句可用來覆寫序列的參數。可用的選項包括針對 CREATE SEQUENCE和 顯示的選項SEQUENCE NAME *name*。如果沒有 SEQUENCE NAME，系統會為序列選擇未使用的名稱。

ALTER TABLE

ALTER TABLE 可變更資料表的定義。

```
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 }
```

身分欄動作

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

這些表單會變更資料欄是身分資料欄，或變更現有身分資料欄的產生屬性。如需詳細資訊，請參閱 CREATE TABLE。如同 SET DEFAULT，這些表單只會影響後續 INSERT和 UPDATE命令的行為；它們不會導致資料表中已有的資料列變更。

sequence_option 是 支援的選項，[ALTER SEQUENCE]例如 INCREMENT BY。這些表單會變更以現有身分資料欄為依據的序列。

DROP IDENTITY [IF EXISTS]

此表單會從資料欄移除身分屬性。如果已指定 DROP IDENTITY IF EXISTS，且資料欄不是身分資料欄，則不會擲出錯誤。在此情況下，會改為發出通知。

CREATE SEQUENCE

CREATE SEQUENCE — 定義新的序列產生器。

> **Important**
>
> 在 PostgreSQL 中，指定 CACHE 是選用的，預設為 1。在 Amazon Aurora DSQL 等分散式系統中，序列操作涉及協調，快取大小為 1 可以增加高並行下的協調負荷。雖然較大的快取值允許從本機預先配置的範圍提供序號，提高輸送量，但未使用的預留值可能會遺失，使得差距和排序效果更加明顯。由於應用程式對配置排序的敏感度與輸送量不同，因此 Amazon Aurora DSQL CACHE 需要明確指定，且目前支援 CACHE = 1或 CACHE >= 65536，提供配置行為之間的明確區別，更接近針對高度並行工作負載進行最佳化的嚴格循序產生和配置。
> 當 時CACHE >= 65536，序列值仍保證是唯一的，但可能不會在工作階段間以嚴格增加的順序產生，並且可能發生差距，特別是快取值未完全耗用時。這些特性與同時使用下快取序列的 PostgreSQL 語意一致，其中兩個系統都保證不同的值，但不保證跨工作階段嚴格循序排序。
> 在單一用戶端工作階段中，序列值可能不會總是出現嚴格增加，尤其是在明確交易之外。此行為類似於使用連線集區的 PostgreSQL 部署。接近單一工作階段 PostgreSQL 環境的配置行為，可以透過在明確交易中使用 CACHE = 1或 取得序列值來達成。
> 使用 CACHE = 1，序列配置遵循 PostgreSQL 的非快取序列行為。
> 如需如何根據工作負載模式以最佳方式使用序列的指引，請參閱 [使用序列和身分資料欄]。

支援的語法

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

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

Description

CREATE SEQUENCE 會建立新的序號產生器。這包括使用*##*建立和初始化新的特殊單列資料表。產生器將由發出 命令的使用者擁有。

如果指定結構描述名稱，則會在指定的結構描述中建立序列。否則會在目前的結構描述中建立。序列名稱必須與相同結構描述中任何其他關係（資料表、序列、索引、檢視、具體化檢視或外部資料表）的名稱不同。

建立序列之後，您可以使用函數 nextval、 currval和 setval 來操作序列。這些函數會記錄在中[序列處理函式](#)。

雖然您無法直接更新序列，但您可以使用下列查詢：

```
SELECT * FROM name;
```

檢查序列的某些參數和目前狀態。特別是，序列last_value的欄位會顯示任何工作階段配置的最後一個值。（當然，如果其他工作階段正在執行nextval呼叫，此值在列印時可能會淘汰。）您可以在pg_sequences檢視中觀察其他參數，例如*##*和*###*。

Parameters

IF NOT EXISTS

如果具有相同名稱的關係已存在，請勿擲回錯誤。在此情況下會發出通知。請注意，無法保證現有關係與已建立的序列類似，甚至可能不是序列。

name

要建立之序列的名稱（選擇性符合結構描述資格）。

data_type

選用的 子句AS *data_type*指定序列的資料類型。有效類型為 bigint。 bigint 是預設值。資料類型會決定序列的預設最小值和最大值。

##

選用的 子句會INCREMENT BY *increment*指定要新增至目前序列值的值，以建立新的值。正值將使遞增序列，負值將遞減序列。預設值為 1.

***###*/ NO MINVALUE**

選用子句MINVALUE *minvalue*決定序列可以產生的最小值。如果未提供或NO MINVALUE指定此子句，則會使用預設值。遞增序列的預設值為 1。遞減序列的預設值為 資料類型的最小值。

***maxvalue* / NO MAXVALUE**

選用子句MAXVALUE *maxvalue*決定序列的最大值。如果未提供或NO MAXVALUE指定此子句，則會使用預設值。遞增序列的預設值為 資料類型的最大值。遞減序列的預設值為 -1。

CYCLE / NO CYCLE

當遞增或遞減序列分別達到*###*或*###*時， CYCLE選項允許序列包裝。如果達到限制，產生的下一個數字將分別是*###*或*###*。

如果指定 NO CYCLE ，則在序列達到其最大值nextval之後對 的任何呼叫都會傳回錯誤。如果未指定 CYCLE或 NO CYCLE ，NO CYCLE則預設為 。

start

選用的 子句START WITH *start*允許序列從任何位置開始。預設的開始值是遞增序列的*###*，以及遞減序列的*###*。

##

子句CACHE *cache*指定要預先配置和儲存在記憶體中的序號數量，以加快存取速度。Aurora DSQL CACHE中 可接受的值為 1 或任何數字 >= 65536。最小值為 1（一次只能產生一個值，表示沒有快取）。

OWNED BY *table_name.column_name* / OWNED BY NONE

OWNED BY 選項會導致序列與特定資料表資料欄相關聯，因此如果捨棄該資料欄（或其整個資料表），序列也會自動捨棄。指定的資料表必須具有相同的擁有者，且與序列位於相同的結構描述中。預設 OWNED BY NONE會指定沒有此類關聯。

備註

使用 DROP SEQUENCE移除序列。

序列是以算術為基礎，因此範圍不能超過八位元組整數 (-9223372036854775808 bigint 到 9223372036854775807) 的範圍。

由於 nextval和 setval呼叫永遠不會轉返，如果需要序號的「無間隙」指派，則無法使用序列物件。

每個工作階段都會在一次存取序列物件期間配置和快取連續的序列值，並last_value相應地增加序列物件的 。然後，該工作階段nextval中的下一個 *cache*-1 使用 只會傳回預先配置的值，而不會接觸序列物件。因此，當工作階段結束時，任何配置但未在工作階段中使用的數字都會遺失，導致序列中出現「洞」。

此外，雖然保證多個工作階段會配置不同的序列值，但當考慮所有工作階段時，可能會不按順序產生這些值。例如，如果*##*設定為 10，工作階段 A 可能會保留值 1..10 並傳回 nextval=1，則工作階段 B 可能會保留值 11..20，並在工作階段 A 產生 nextval=2 之前傳回 nextval=11。因此，在*##*設定為一個時，您可以安全地假設nextval值是循序產生的；*##*設定大於一個時，您應該只假設這些nextval值都是不同的，而不是完全循序產生。此外， last_value也會反映任何工作階段保留的最新值，無論 是否已傳回。 nextval

另一個考量是，在此類序列setval上執行的 不會被其他工作階段注意到，直到它們用完他們快取的任何預先配置值為止。

範例

建立名為 的遞增序列serial，從 101 開始：

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

從此序列中選取下一個數字：

```
SELECT nextval('serial');

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

從此序列中選取下一個數字：

```
SELECT nextval('serial');

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

在 INSERT命令中使用此序列：

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

使用 將序列重設為特定值setval：

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

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

相容性

CREATE SEQUENCE 符合 SQL 標準，但有下列例外：

- 取得下一個值是使用 nextval()函數而非標準NEXT VALUE FOR表達式來完成。
- OWNED BY 子句是 PostgreSQL 延伸模組。

ALTER SEQUENCE

ALTER SEQUENCE — 變更序列產生器的定義。

> **Important**
>
> 使用序列時，應仔細考慮快取值。如需詳細資訊，請參閱 CREATE SEQUENCE 頁面上的重要標註。
> 如需如何根據工作負載模式以最佳方式使用序列的指引，請參閱 使用序列和身分資料欄。

支援的語法

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

```
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema

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

Description

ALTER SEQUENCE 會變更現有序列產生器的參數。ALTER SEQUENCE 命令中未特別設定的任何參數都會保留其先前的設定。

您必須擁有序列才能使用 ALTER SEQUENCE。若要變更序列的結構描述，您還必須具有新結構描述CREATE的權限。若要更改擁有者，您必須能夠SET ROLE使用新的擁有角色，且該角色必須具有序列結構描述CREATE的權限。（這些限制會強制變更擁有者不會執行您捨棄並重新建立序列而無法執行的任何動作。 不過，超級使用者可以改變任何序列的擁有權。）

Parameters

name

要變更之序列的名稱（選擇性符合結構描述資格）。

IF EXISTS

如果序列不存在，請勿擲回錯誤。在此情況下會發出通知。

##

子句INCREMENT BY *increment*是選用的。正值將使遞增序列，負值將遞減序列。如果未指定，則會維持舊的增量值。

minvalue / **NO MINVALUE**

選用子句MINVALUE *minvalue*決定序列可以產生的最小值。如果NO MINVALUE指定 ，則會分別使用 1 的預設值和遞增和遞減序列的資料類型最小值。如果未指定任何選項，則會維持目前的最小值。

maxvalue / **NO MAXVALUE**

選用子句MAXVALUE *maxvalue*決定序列的最大值。如果NO MAXVALUE指定 ，則會分別使用資料類型的最大值預設值和遞增和遞減序列的 -1。如果未指定任何選項，則會維持目前的最大值。

CYCLE

當遞增或遞減序列分別達到*###*或*###*時，選用CYCLE的關鍵字可用來讓序列包裝。如果達到限制，產生的下一個數字將分別是*###*或*###*。

NO CYCLE

如果指定選用NO CYCLE關鍵字，則在序列達到其最大值nextval之後對 的任何呼叫都會傳回錯誤。如果未指定 CYCLE或 NO CYCLE ，則會維持舊的週期行為。

start

選用子句會START WITH *start*變更序列的記錄開始值。這不會影響目前的序列值；它只會設定未來ALTER SEQUENCE RESTART命令將使用的值。

####

選用子句會RESTART [WITH *restart*]變更序列的目前值。這類似於使用 is_called = 呼叫setval函數false：下次呼叫 會傳回指定的值nextval。沒有*####*值RESTART的寫入相當於提供由 記錄的開始值CREATE SEQUENCE，或由 上次設定ALTER SEQUENCE START WITH。

與setval呼叫相反，序列上的RESTART操作是交易的，並封鎖並行交易從相同序列取得數字。如果這不是所需的操作模式，setval則應使用 。

##

子句CACHE *cache*可讓序號預先配置並存放在記憶體中，以加快存取速度。值必須為 1 或某個值 >= 65536。如果未指定，則會保留舊快取值。如需快取行為的詳細資訊，請參閱 下的指引CREATE SEQUENCE。

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

OWNED BY 選項會導致序列與特定資料表資料欄相關聯，因此如果捨棄該資料欄（或其整個資料表），序列也會自動捨棄。如果指定，此關聯會取代先前為序列指定的任何關聯。指定的資料表必須具有相同的擁有者，且與序列位於相同的結構描述中。指定 會OWNED BY NONE移除任何現有的關聯，使序列「獨立」。

new_owner

序列新擁有者的使用者名稱。

new_name

序列的新名稱。

new_schema

序列的新結構描述。

備註

ALTER SEQUENCE 除了目前已預先配置（快取）序列值的後端之外，不會立即影響後端nextval的結果。在注意到已變更的序列產生參數之前，他們會先用完所有快取的值。目前的後端會立即受到影響。

ALTER SEQUENCE 不會影響序列currval的狀態。

ALTER SEQUENCE 可能會導致其他交易進入 OCC。

基於歷史原因，也可以與序列ALTER TABLE搭配使用；但唯一允許與序列搭配使用ALTER TABLE的變體等同於上述表單。

範例

在 105 serial重新啟動名為 的序列：

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

相容性

ALTER SEQUENCE 符合 SQL 標準，但 AS、、RENAME TO、、 START WITH OWNED BY OWNER TO和 SET SCHEMA子句除外，其為 PostgreSQL 擴充功能。

DROP SEQUENCE

DROP SEQUENCE — 移除序列。

支援的語法

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

Description

DROP SEQUENCE 會移除序號產生器。序列只能由其擁有者或超級使用者捨棄。

Parameters

IF EXISTS

如果序列不存在，請勿擲回錯誤。在此情況下會發出通知。

name

序列的名稱（選擇性符合結構描述資格）。

CASCADE

自動捨棄相依於序列的物件，然後捨棄相依於這些物件的所有物件。

RESTRICT

如果有任何物件相依於序列，請拒絕捨棄序列。這是預設值。

範例

若要移除序列 seq：

```
DROP SEQUENCE seq;
```

相容性

DROP SEQUENCE 符合 SQL 標準，但標準只允許每個命令捨棄一個序列，並且與 PostgreSQL 擴充IF EXISTS功能的選項不同。

CREATE VIEW

CREATE VIEW 可定義新的持久性檢視。Aurora DSQL 不支援暫時檢視；僅支援持久性檢視。

支援的語法

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

Description

CREATE VIEW 可定義查詢的檢視。實際上，檢視並沒有具體化。相反地，每次在查詢中參考檢視時，即會執行查詢。

CREATE or REPLACE VIEW 類似，但如果有同名的檢視存在，則會取代該檢視。新查詢產生的資料欄，必須與現有檢視查詢產生的資料欄相同 (亦即相同資料欄名稱、相同順序且資料類型相同)，但其可能會將其他資料欄新增至清單結尾。計算引發的輸出資料欄可能不同。

如果有提供結構描述名稱 (例如 CREATE VIEW myschema.myview ...)，則會使用指定的結構描述建立檢視。否則，會使用目前的結構描述建立。

檢視的名稱必須不同於相同結構描述中任何其他關係 (資料表、索引、檢視) 的名稱。

Parameters

CREATE VIEW 支援各種參數，以控制可自動更新檢視的行為。

RECURSIVE

建立遞迴檢視。語法：CREATE RECURSIVE VIEW [schema .] view_name (column_names) AS SELECT ...; 相當於 CREATE VIEW [schema .] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...) SELECT column_names FROM view_name;。

您必須為遞迴檢視指定檢視資料欄名稱清單。

name

要建立的檢視名稱；其可以選擇性符合結構描述資格。您必須為遞迴檢視指定資料欄名稱清單。

column_name

要用於檢視資料欄的選用名稱清單。若未提供，則會從查詢衍生資料欄名稱。

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

此子句指定檢視的選用參數；下列為支援的參數。

- check_option (enum) — 此參數可以是 local 或 cascaded，且等同於指定 WITH [CASCADED | LOCAL] CHECK OPTION。
- security_barrier (boolean) — 如果檢視是為了提供資料列層級安全性，則應使用此項目。Aurora DSQL 目前不支援資料列層級安全性，但此選項仍會強制先評估檢視WHERE的條件（以及任何使用標記為 之運算子的條件LEAKPROOF)。
- security_invoker (boolean) — 此選項會根據檢視使用者的權限檢查基礎關係，而不是檢視擁有者。如需完整詳細資訊，請參閱以下備註。

您可以使用 ALTER VIEW，變更現有檢視上的上述所有選項。

query

提供檢視資料欄和資料列的 SELECT或 VALUES命令。

WITH [CASCADED | LOCAL] CHECK OPTION

此選項可控制自動更新檢視的行為。指定此選項時，系統會檢查檢視上的 INSERT 和 UPDATE 命令，以確保新資料列符合檢視定義條件 (亦即會檢查新的資料列，確保它們可透過檢視顯示)。若不符合，則會拒絕更新。如果未指定 CHECK OPTION，則系統會允許檢視上的 INSERT 和 UPDATE 命令建立無法透過檢視顯示的資料列。

LOCAL — 僅依據檢視本身直接定義的條件，檢查新資料列。不依據在基礎檢視上定義的任何條件進行檢查 (除非它們也指定 CHECK OPTION)。

CASCADED — 依據檢視和所有基礎檢視的條件，檢查新資料列。如果指定 CHECK OPTION，而未指定 LOCAL 或 CASCADED，則會假設 CASCADED。

> **Note**
>
> CHECK OPTION 可能無法與 RECURSIVE 檢視搭配使用。CHECK OPTION 僅支援可自動更新的檢視。

備註

使用 DROP VIEW 陳述式捨棄檢視。

您應謹慎考慮檢視的資料欄名稱和資料類型。例如，CREATE VIEW vista AS SELECT 'Hello World'; 不建議使用，因為資料欄名稱預設為 ?column?;。此外，您可能不希望資料欄資料類型預設為 text。

更好的方法是明確指定資料欄名稱和資料類型，例如：CREATE VIEW vista AS SELECT text 'Hello World' AS hello;。

根據預設，檢視中參考的基礎關係存取權，取決於檢視擁有者的許可而定。在某些情況下，這可用來提供對基礎資料表的安全但受限制的存取。不過，並非所有檢視都安全，不會遭到竄改。

- 如果檢視的 security_invoker 屬性設為 true，則基礎關係的存取權取決於執行查詢的使用者許可，而不是檢視擁有者。因此，安全調用端檢視的使用者，必須具有檢視及其基礎關係的相關許可。
- 如果任何基礎關係是安全調用端檢視，則會視為直接透過原始查詢存取。因此，安全調用端檢視一律會使用目前使用者的許可檢查其基礎關係，即使是透過不含 security_invoker 屬性的檢視存取亦同。

- 檢視中呼叫的函數與使用檢視直接透過查詢呼叫的函數，兩者視為相同。因此，檢視使用者必須有權呼叫檢視所用的全部函數。檢視中的函數會以執行查詢的使用者或函數擁有者的權限執行，取決於函數定義為 `SECURITY INVOKER` 或 `SECURITY DEFINER`。
- 建立或取代檢視的使用者，必須擁有檢視查詢中所參考任何結構描述的 `USAGE` 權限，才能在這些結構描述中查詢參考的物件。
- 在現有檢視上使用 `CREATE OR REPLACE VIEW` 時，只會變更檢視的定義 `SELECT` 規則，以及任何 `WITH ( ... )` 參數和其 `CHECK OPTION`。其他檢視屬性，包括擁有權、許可和非 SELECT 規則，保持不變。您必須擁有檢視才能將其取代 (包括成為擁有角色的成員)。

可更新的檢視

簡單檢視可自動更新：系統允許檢視使用 `INSERT`、`UPDATE` 和 `DELETE` 陳述式；方法與一般資料表相同。如果檢視符合下列所有條件，則可自動更新檢視：

- 檢視的 `FROM` 清單中必須只有一個項目，且該項目必須是資料表或其他可更新的檢視。
- 檢視定義最上層不得包含 `WITH`、`DISTINCT`、`GROUP BY`、`HAVING`、`LIMIT` 或 `OFFSET` 子句。
- 檢視定義最上層不得包含集合操作 (`UNION`、`INTERSECT` 或 `EXCEPT`)。
- 檢視的選取清單不得包含任何彙總、視窗函數或集合傳回函數。

可自動更新的檢視可能混合可更新和不可更新資料欄。如果資料欄是基礎關係可更新資料欄的簡單參考，則可更新資料欄。否則，資料欄為唯讀，一旦 `INSERT` 或 `UPDATE` 陳述式嘗試為其指派值，則會發生錯誤。

根據預設，不符合上述所有條件的更複雜檢視為唯讀：系統不允許在這類檢視上插入、更新或刪除。

> **Note**
>
> 在檢視上執行插入、更新或刪除的使用者，必須具有該檢視的插入、更新或刪除相應權限。根據預設，檢視的擁有者必須具有基礎關係的相關權限，而執行更新的使用者不需要基礎關係的任何許可。不過，如果檢視的 security_invoker 設為 true，則執行更新的使用者 (而不是檢視擁有者) 必須具有基礎關係的相關權限。

範例

建立包含所有喜劇影片的檢視。

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

使用 LOCAL CHECK OPTION 建立檢視。

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

建立遞迴檢視。

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

相容性

CREATE OR REPLACE VIEW 是 PostgreSQL 語言擴充功能。WITH (...) 子句、安全屏障檢視和安全調用端檢視也是擴充功能。Aurora DSQL 支援這些語言擴充功能。

ALTER VIEW

ALTER VIEW 陳述式允許變更現有檢視的各種屬性，而 Aurora DSQL 支援此命令的所有 PostgreSQL 語法。

支援的語法

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_column_name
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
```

```
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

Description

ALTER VIEW 會變更檢視的各種輔助屬性。(如果您想要修改檢視的定義查詢，請使用 CREATE OR REPLACE VIEW。) 您必須擁有檢視才能使用 ALTER VIEW。若要變更檢視的結構描述，您還必須具備新結構描述的 CREATE 權限。若要變更擁有者，您必須能夠對新的擁有角色使用 SET ROLE，且該角色必須具有檢視結構描述的 CREATE 權限。

Parameters

name

現有檢視的名稱 (選擇性符合結構描述資格)。

column_name

現有資料欄的名稱，或現有資料欄的新名稱。

IF EXISTS

如果檢視不存在，不要擲回錯誤。在此情況下會發出通知。

SET/DROP DEFAULT

這些表單會設定或移除資料欄的預設值。系統會以目標為檢視的任何 INSERT 或 UPDATE 命令，替代檢視資料欄的預設值。

new_owner

檢視新擁有者的使用者名稱。

new_name

檢視的新名稱。

new_schema

檢視的新結構描述。

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

設定檢視選項。以下是支援的選項：

- check_option (enum) - 變更檢視的檢查選項。值必須為 local 或 cascaded。
- security_barrier (boolean) - 變更檢視的安全屏障屬性。
- security_invoker (boolean) - 變更檢視的安全呼叫器屬性。

RESET (view_option_name [, ...])

將檢視選項重設為其預設值。

範例

將檢視重新命名foo為 bar：

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

將預設資料欄值附加至可更新的檢視：

```
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
```

相容性

`ALTER VIEW` 是 Aurora DSQL 支援的 SQL 標準 PostgreSQL 擴充功能。

DROP VIEW

`DROP VIEW` 陳述式會移除現有的檢視。Aurora DSQL 支援此命令的完整 PostgreSQL 語法。

支援的語法

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

Description

`DROP VIEW` 會捨棄現有的檢視。您必須是檢視的擁有者才能執行此命令。

Parameters

IF EXISTS

如果檢視不存在，不要擲回錯誤。在此情況下會發出通知。

name

要移除的檢視名稱 (選擇性符合結構描述資格)。

CASCADE

自動捨棄相依於檢視的物件 (例如其他檢視)，進而捨棄所有相依於這些物件的物件。

RESTRICT

如果有任何物件相依於檢視，即拒絕捨棄該檢視。這是預設值。

範例

```
DROP VIEW kinds;
```

相容性

此命令符合 SQL 標準，但標準只允許每個命令捨棄一個檢視；`IF EXISTS` 選項除外，這是 Aurora DSQL 支援的 PostgreSQL 擴充功能。

從 PostgreSQL 遷移至 Aurora DSQL

Aurora DSQL 的設計與 PostgreSQL 相容，支援 ACID 交易、次要索引、聯結和標準 DML 操作等核心關聯式功能。大多數現有的 PostgreSQL 應用程式都可以遷移至 Aurora DSQL，且變更最少。

本節提供將應用程式遷移至 Aurora DSQL 的實際指導，包括架構相容性、遷移模式和架構考量。

架構和 ORM 相容性

Aurora DSQL 使用標準 PostgreSQL 線路通訊協定，以確保與 PostgreSQL 驅動程式和架構的相容性。最熱門ORMs 可搭配 Aurora DSQL 使用，但幾乎沒有變更。如需參考實作和可用的 ORM 整合the section called “Aurora DSQL 轉接器”，請參閱 。

常見的遷移模式

從 PostgreSQL 遷移到 Aurora DSQL 時，某些功能的運作方式不同，或有其他語法。本節提供常見遷移案例的指引。

DDL 操作替代方案

Aurora DSQL 提供傳統 PostgreSQL DDL 操作的現代替代方案：

建立索引

使用 `CREATE INDEX ASYNC`而非 `CREATE INDEX` 建立非封鎖索引。

優點：在大型資料表上建立零停機時間索引。

資料移除

使用 `DELETE FROM table_name`而非 `TRUNCATE`。

替代方案：若要完成資料表重新建立，請使用 `DROP TABLE`，後面接著 `CREATE TABLE`。

系統組態

Aurora DSQL 是全受管的，因此會根據工作負載模式自動處理組態。使用 AWS 管理主控台或 API 來管理叢集設定。

優點：不需要進行資料庫調校或參數管理。

結構描述設計模式

為 Aurora DSQL 相容性調整這些常見的 PostgreSQL 模式：

參考完整性模式

Aurora DSQL 支援資料表關係和`JOIN`操作。如需參考完整性，請在應用程式層中實作驗證。此設計符合現代分散式資料庫模式，其中應用程式層驗證提供更多彈性，並避免層疊操作造成效能瓶頸。

模式：使用一致的命名慣例、驗證邏輯和交易界限，在您的應用程式層中實作參考完整性檢查。許多大規模應用程式偏好這種方法來更好地控制錯誤處理和效能。

暫時資料處理

使用具有清除邏輯CTEs、子查詢或一般資料表，而非暫存資料表。

替代方案：建立具有工作階段特定名稱的資料表，並在應用程式中清除它們。

了解架構差異

Aurora DSQL 的分散式無伺服器架構刻意在數個區域與傳統 PostgreSQL 不同。這些差異可實現 Aurora DSQL 簡化和擴展的主要優勢。

簡化的資料庫模型

每個叢集的單一資料庫

Aurora DSQL `postgres`為每個叢集提供一個名為 的內建資料庫。

遷移秘訣：如果您的應用程式使用多個資料庫，請建立個別的 Aurora DSQL 叢集進行邏輯分離，或在單一叢集中使用結構描述。

沒有暫存資料表

對於臨時資料處理，您應該使用通用資料表表達式 (CTEs) 和子查詢，為複雜的查詢提供靈活的替代方案。

替代方案：將 CTEs與暫時結果集的 `WITH`子句搭配使用，或使用工作階段特定資料的唯一命名規則資料表。

自動儲存管理

Aurora DSQL 消除了資料表空間和手動儲存管理。儲存會根據您的資料模式自動擴展和最佳化。

優點：不需要監控磁碟空間、規劃儲存配置或管理資料表空間組態。

現代應用程式模式

Aurora DSQL 鼓勵現代應用程式開發模式，以改善可維護性和效能：

應用程式層級邏輯而非資料庫觸發

如需類似觸發的功能，請在應用程式層中實作事件驅動的邏輯。

遷移策略：將觸發邏輯移至應用程式程式碼、搭配 EventBridge 等 AWS 服務使用事件驅動型架構，或使用應用程式記錄實作稽核線索。

資料處理的 SQL 函數

Aurora DSQL 支援以 SQL 為基礎的函數，但不支援程序語言，例如 PL/pgSQL。

替代方案：使用 SQL 函數進行資料轉換，或將複雜的邏輯移至您的應用程式層或 AWS Lambda 函數。

樂觀並行控制而非幂等鎖定

Aurora DSQL 使用樂觀並行控制 (OCC)，這是一種與傳統資料庫鎖定機制不同的無鎖定方法。Aurora DSQL 不會取得封鎖其他交易的鎖定，而是允許交易繼續進行，而不會封鎖和偵測遞交時的衝突。這可消除死結，並防止慢速交易封鎖其他操作。

關鍵差異：發生衝突時，Aurora DSQL 會傳回序列化錯誤，而不是讓交易等待鎖定。這需要應用程式實作重試邏輯，類似於處理傳統資料庫中的鎖定逾時，但衝突會立即解決，而不是導致封鎖等待。

設計模式：使用重試機制實作等冪交易邏輯。設計結構描述，透過使用隨機主索引鍵並將更新分散到您的索引鍵範圍，將爭用降至最低。如需詳細資訊，請參閱Aurora DSQL 的並行控制。

關係和參考完整性

Aurora DSQL 支援資料表之間的外部金鑰關係，包括 `JOIN` 操作。如需參考完整性，請在應用程式層中實作驗證。雖然強制執行參考完整性可能很有價值，但串聯操作 (例如串聯刪除) 可能會產生非預期的效能問題，例如，刪除具有 1，000 個明細項目的訂單會變成 1，001 列交易。因此，許多客戶會避免外部金鑰限制。

設計模式：在您的應用程式層中實作參考完整性檢查、使用最終一致性模式，或利用 AWS 服務進行資料驗證。

操作簡化

Aurora DSQL 可消除許多傳統資料庫維護任務，減少營運開銷：

不需要手動維護

Aurora DSQL 會自動管理儲存最佳化、統計資料收集和效能調校。這類傳統維護命令由系統`VACUUM`處理。

優點：不需要資料庫維護時段、清空排程和系統參數調校。

自動分割和擴展

Aurora DSQL 會根據存取模式自動分割和分配您的資料。使用 UUIDs或應用程式產生的 IDs以獲得最佳分佈。

遷移秘訣：移除手動分割邏輯，並讓 Aurora DSQL 處理資料分佈。使用 UUIDs或應用程式產生的 IDs以獲得最佳分佈。如果您的應用程式需要循序識別符，請參閱 序列和身分資料欄。

使用 AI 工具進行代理遷移

AI 編碼代理器可以透過分析結構描述、轉換程式碼，以及使用內建安全檢查執行 DDL 遷移，加速遷移至 Aurora DSQL。

使用 Kiro 進行遷移

Kiro 等編碼代理程式可協助您分析並將 PostgreSQL 程式碼遷移至 Aurora DSQL：

- 結構描述分析：上傳現有的結構描述檔案，並要求 Kiro 識別潛在的相容性問題，並建議替代方案

- 程式碼轉換：提供您的應用程式程式碼，並要求 Kiro 協助重構觸發邏輯、以 UUIDs取代序列，或修改交易模式
- 遷移規劃：請 Kiro 根據您的特定應用程式架構建立step-by-step遷移計畫
- DDL 遷移：使用具有內建安全檢查和使用者驗證的資料表重建模式來執行結構描述修改

範例提示：

```
"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"
```

使用資料表重新建立進行 DDL 遷移

搭配 Aurora DSQL MCP 伺服器使用 AI 代理器時，某些 ALTER TABLE 操作會使用可安全遷移資料的資料表重建模式。代理程式會處理複雜性，同時在每個步驟通知您。

下列操作使用資料表重新建立模式：

作業	方法
DROP COLUMN	從新資料表中排除資料欄
ALTER COLUMN TYPE	在遷移期間投射資料類型
ALTER COLUMN SET/DROP NOT NULL	變更新資料表定義的限制條件
ALTER COLUMN SET/DROP DEFAULT	在新資料表定義中定義預設值
ADD/DROP CONSTRAINT	在新資料表中包含或移除限制條件

作業	方法
MODIFY PRIMARY KEY	使用唯一性驗證定義新的 PK
分割/合併資料欄	使用 SPLIT_PART、SUBSTRING 或 CONCAT

無需重新建立資料表，即可直接支援下列 ALTER TABLE 操作：

- `ALTER TABLE ... RENAME COLUMN` – 重新命名資料欄
- `ALTER TABLE ... RENAME TO` – 重新命名資料表
- `ALTER TABLE ... ADD COLUMN` – 新增資料欄

安全功能：執行 DDL 遷移時，AI 代理器會呈現遷移計畫、驗證資料相容性、確認資料列計數，並在 DROP TABLE 等任何破壞性操作之前請求明確核准。

批次遷移：對於超過 3，000 個資料列的資料表，代理程式會自動以 500-1，000 個資料列的增量批次遷移，以保持在交易限制內。

Aurora DSQL MCP 伺服器

Aurora DSQL 模型內容通訊協定 (MCP) 伺服器可讓 AI 助理直接連線至您的 Aurora DSQL 叢集，並搜尋 Aurora DSQL 文件。這可讓 AI：

- 分析現有的結構描述並建議遷移變更
- 使用資料表重新建立模式執行 DDL 遷移
- 在遷移期間測試查詢並驗證相容性
- 根據最新的 Aurora DSQL 文件提供準確up-to-date指引

若要搭配 AI 助理使用 Aurora DSQL MCP 伺服器，請參閱 Aurora DSQL MCP 伺服器的設定說明。

Aurora DSQL 的 PostgreSQL 相容性考量

Aurora DSQL 具有與自我管理 PostgreSQL 不同的功能，可啟用其分散式架構、無伺服器操作和自動擴展。大多數應用程式會在這些差異中運作，無需修改。

如需一般考量，請參閱 使用 Amazon Aurora DSQL 的考量事項。如需了解配額和限制，請參閱 Amazon Aurora DSQL 中的叢集配額與資料庫限制。

- Aurora DSQL `postgres`使用每個叢集名為 的單一內建資料庫。對於邏輯分離，請建立個別的 Aurora DSQL 叢集，或在單一叢集中使用結構描述。
- `postgres` 資料庫使用 UTF-8 字元編碼，可提供廣泛的國際字元支援。
- 資料庫只會使用 `C` 定序。
- Aurora DSQL 以 UTC 作為系統時區。Postgres 會將所有時區感知日期和時間儲存在 UTC 內部。您可以設定`TimeZone`組態參數來轉換用戶端的顯示方式，並做為伺服器將用來在內部轉換為 UTC 的用戶端輸入預設值。
- 交易隔離層級固定為 PostgreSQL `Repeatable Read`。
- 交易有下列限制：
  - DDL 和 DML 操作需要個別的交易
  - 交易只能包含 1 個 DDL 陳述式
  - 無論次要索引的數量為何，交易最多可以修改 3,000 個資料列
  - 3,000 個資料列的限制適用於所有 DML 陳述式 (`INSERT`、`UPDATE`、`DELETE`)
- 資料庫連線會在 1 小時後逾時。
- Aurora DSQL 透過結構描述層級授予來管理許可。管理員使用者使用 建立結構描述`CREATE SCHEMA`，並使用 授予存取權`GRANT USAGE ON SCHEMA`。管理員使用者管理公有結構描述中的物件，而非管理員使用者則在使用者建立的結構描述中建立物件，以實現明確的擁有權界限。如需詳細資訊，請參閱授權資料庫角色在您的資料庫使用 SQL。

需要遷移協助嗎？

如果您遇到對遷移至關重要但 Aurora DSQL 目前不支援的功能，請參閱 以了解如何與 AWS 共用意見回饋提供有關 Amazon Aurora DSQL 的意見回饋的相關資訊。

Aurora DSQL 的並行控制

並行功能可讓多個工作階段同時存取和修改資料，而不會影響資料完整性和一致性。Aurora DSQL 提供 PostgreSQL 相容性，同時實作現代、無需鎖定的並行控制機制。其可透過快照隔離維持完整的 ACID 合規，以確保資料一致性和可靠性。

Aurora DSQL 的主要優點是無需鎖定架構，可消除常見的資料庫效能瓶頸。Aurora DSQL 可防止緩慢交易封鎖其他操作，並消除死結的風險。這種方法讓 Aurora DSQL 特別適合效能和可擴展性至關重要的高輸送量應用程式。

交易衝突

Aurora DSQL 使用開放式並行控制 (OCC)，其運作方式與傳統的鎖定型系統不同。OCC 會評估遞交時的衝突，而不是使用鎖定。如果更新相同資料列時發生多個交易衝突，Aurora DSQL 會管理交易，如下所示：

- Aurora DSQL 會處理遞交時間最早的交易。
- 衝突的交易會收到 PostgreSQL 序列化錯誤，指出需要重試。

您可以設計應用程式，藉由實作重試邏輯處理衝突。理想設計模式為等冪，並盡可能讓交易重試成為優先求助方法。建議邏輯類似於，標準 PostgreSQL 鎖定逾時或死結情況下的中止和重試邏輯。不過，使用 OCC 時，您的應用程式必須更頻繁地執行此邏輯。

最佳化交易效能的指導方針

為了最佳化效能，請將單一索引鍵或小型索引鍵範圍上的激烈爭用性降至最低。若要達成此目標，請使用下列指導方針設計結構描述，將更新分散至叢集索引鍵範圍：

- 為您的資料表選擇隨機主索引鍵。
- 避免使用會提高單一索引鍵爭用性的模式。即使交易量增加，此方法也能確保最佳效能。

Aurora DSQL 的 DDL 和分散式交易

Aurora DSQL 的資料定義語言 (DDL) 行為與 PostgreSQL 不同。Aurora DSQL 具有多可用區域的分散式、不共用物件資料庫層，其建置在多租用戶運算和儲存機群之上。由於不存在單一主要資料庫節點或領導節點，因此資料庫目錄為分散式。亦即，Aurora DSQL 會以分散式交易方式管理 DDL 結構描述變更。

具體來說，Aurora DSQL 的 DDL 行為有下列不同：

並行控制錯誤

如果執行交易時其他交易正在更新資源，Aurora DSQL 會傳回並行控制違規錯誤。例如，假設發生下列一系列動作：

1. 在工作階段 1 中，使用者將資料欄新增至資料表 `mytable`。
2. 在工作階段 2 中，使用者嘗試將資料列插入 `mytable`。

Aurora DSQL 傳回錯誤 `SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (OC001).`

相同交易中的 DDL 和 DML

Aurora DSQL 的交易只能包含一個 DDL 陳述式，且不能同時包含 DDL 和 DML 陳述式。此限制表示您無法建立資料表，並將資料插入相同交易的相同資料表中。例如，Aurora DSQL 支援以下序列交易。

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

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

Aurora DSQL 不支援下列交易，因為其中包含 `CREATE` 和 `INSERT` 兩種陳述式。

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

非同步 DDL

在標準 PostgreSQL 中，DDL 操作 (例如 `CREATE INDEX` 鎖定受影響的資料表) 會導致其他工作階段無法讀取和寫入。在 Aurora DSQL 中，這些 DDL 陳述式會使用背景管理員以非同步方式執行。因此不會封鎖存取受影響的資料表。也就是說，大型資料表上的 DDL 可在不造成停機或影響效能的情況下執行。如需 Aurora DSQL 非同步作業管理員的詳細資訊，請參閱 Aurora DSQL 的非同步索引。

Aurora DSQL 的主索引鍵

在 Aurora DSQL 中，主索引鍵是實際整理資料表資料的功能。其類似 PostgreSQL 的 `CLUSTER` 操作，或其他資料庫的叢集索引。當您定義主索引鍵時，Aurora DSQL 會建立包含資料表中所有資料欄的索引。Aurora DSQL 中的主索引鍵結構可確保有效的資料存取和管理。

資料結構和儲存

當您定義主索引鍵時，Aurora DSQL 會按照主索引鍵順序存放資料表資料。這種索引組織結構可讓主索引鍵查詢直接擷取所有資料欄值，而不用像傳統 B 型樹狀結構索引一樣遵循資料指標。PostgreSQL 的 CLUSTER 操作只會重新整理一次資料；不同的是，Aurora DSQL 會自動且持續地維護此順序。此方法可改善依賴主索引鍵存取的查詢效能。

Aurora DSQL 也會使用主索引鍵，為資料表和索引中的每個資料列產生整個叢集的唯一索引鍵。此唯一索引鍵也支援分散式資料管理。其可自動分割多個節點的資料，支援可擴展的儲存和高度並行。因此，主索引鍵結構可協助 Aurora DSQL 自動擴展並有效管理並行工作負載。

選擇主索引鍵的準則

在 Aurora DSQL 中選擇和使用主索引鍵時，請考慮下列準則：

- 在建立資料表時定義主索引鍵。您稍後無法變更此索引鍵或新增新的主索引鍵。主索引鍵會成為整個叢集的索引鍵的一部分，用於資料分割和自動擴展寫入輸送量。如果您未指定主索引鍵，Aurora DSQL 會指派合成的隱藏 ID。
- 若是具大量寫入的資料表，請避免使用單調遞增的整數作為主索引鍵。這可能會將所有新的插入導向單一分割區，而造成效能問題。反之，請使用隨機分佈的主索引鍵，以確保寫入平均分佈至各儲存分割區。
- 若是不常變更或唯讀的資料表，您可以使用遞增索引鍵。時間戳記或序號就是遞增索引鍵的範例。密集索引鍵有許多緊密間隔或重複的值。即使是密集索引鍵，您也可以使用遞增金鑰，因為寫入效能較不重要。
- 如果完整資料表掃描不符合您的效能需求，請選擇更有效率的存取方法。在大多數情況下，這表示您使用的主索引鍵應符合查詢中最常見的聯結和查詢索引鍵。
- 主索引鍵中資料欄合併後的大小上限為 1 KiB。如需詳細資訊，請參閱 Aurora DSQL 資料庫限制和 Aurora DSQL 支援的資料類型。
- 您可以在主索引鍵或次要索引中包含最多 8 個資料欄。如需詳細資訊，請參閱 Aurora DSQL 資料庫限制和 Aurora DSQL 支援的資料類型。

序列和身分資料欄

序列和身分資料欄會產生整數值，並在需要精簡或人類可讀取的識別符時很有用。這些值涉及 CREATE SEQUENCE 文件中所述的配置和快取行為。

主題

- 序列處理函式
- 身分資料欄
- 使用序列和身分資料欄

序列處理函式

本節說明在序列物件上操作的函數，也稱為序列產生器或僅序列。序列物件是使用 建立的特殊單列資料表CREATE SEQUENCE。序列物件通常用於為資料表的資料列產生唯一識別符。序列函數提供簡單、多使用者安全的方法來從序列物件取得連續的序列值。

Important

使用序列時，應仔細考慮快取值。如需詳細資訊，請參閱 CREATE SEQUENCE 頁面上的重要標註。
如需如何根據工作負載模式以最佳方式使用序列的指導，請參閱 使用序列和身分資料欄。

函式	Description
nextval (regclass) # bigint	將序列物件推進到下一個值，並傳回該值。這是以原子方式完成的：即使多個工作階段nextval同時執行，每個工作階段都會安全地收到不同的序列值。如果序列物件已使用預設參數建立，後續nextval呼叫將傳回以 1 開頭的遞增值。您可以在 CREATE SEQUENCE命令中使用適當的參數來取得其他行為。此函數需要 序列上的 USAGE或UPDATE權限。
setval (regclass, bigint [, boolean]) # bigint	設定序列物件的目前值，並選擇性地設定其is_called 旗標。雙參數形式會將序列的 last_value 欄位設定為指定的值，並將其is_called 欄位設定為 true，這表示下一個 nextval會在傳回值之前推進序列。將報告的值currval也會設定為指定的值。在三參數形式中，is_called 可以設定為 true或 false。 true 具有與雙參數形式相同的效果。如果設定為 false，下一個nextval會傳回確切指定的值，而序列提升會從下列 開

函式	Description

	始nextval。此外，此處currval不會變更 回報的值。例如：
	`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`
	傳回的結果setval只是其第二個引數的值。此函數需要序列UPDATE的權限。

currval (regclass) # bigint	傳回 最近在目前工作階段中nextval針對此序列取得的值。（如果在此工作階段中nextval從未針對此序列呼叫，則會報告錯誤。）由於這會傳回工作階段本機值，因此無論其他工作階段是否nextval因為目前工作階段而執行，都會提供可預測的答案。此函數需要 序列上的 USAGE或 SELECT權限。
lastval () # bigint	傳回目前交易nextval中 最近傳回的值。此函數與 相同currval，除了不將序列名稱作為引數，而是參考目前交易中nextval最近套用到的任何序列。lastval 如果目前交易中nextval尚未呼叫 ，則呼叫 時發生錯誤。此函數對上次使用的序列需要 USAGE或 SELECT權限。

> **Warning**
>
> 如果呼叫交易稍後中止，nextval則不會回收 取得的值以供重複使用。這表示交易中止或資料庫當機可能會導致指派值序列的差距。這也可能在沒有交易中止的情況下發生。例如，INSERT具有 ON CONFLICT子句的 會先計算to-be-inserted元組，包括進行任何必要的nextval呼叫，然後再偵測任何可能導致其遵循ON CONFLICT規則的衝突。因此，Aurora DSQL 的序列物件無法用來取得「無間隙」序列。
> 同樣地，其他交易setval也會立即看到 所做的序列狀態變更，而且如果呼叫交易轉返，則不會復原。

序列函數要操作的序列由引數指定，該regclass引數只是pg_class系統目錄中序列的 OID。不過，您不需要手動查詢 OID，因為regclass資料類型的輸入轉換器會為您執行工作。如需詳細資訊，請參閱物件識別符類型的 PostgreSQL 文件。

身分資料欄

Important

使用身分資料欄時，應仔細考慮快取值。如需詳細資訊，請參閱 CREATE SEQUENCE 頁面上的重要標註。

如需如何根據工作負載模式最佳使用身分資料欄的指引，請參閱 使用序列和身分資料欄。

身分資料欄是從隱含序列自動產生的特殊資料欄。它可用於產生索引鍵值。若要建立身分資料欄，請使用 中的 GENERATED ... AS IDENTITY子句CREATE TABLE，例如：

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

或者：

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

如需詳細資訊，請參閱CREATE TABLE。

如果在具有身分資料欄的資料表上執行 INSERT命令，且身分資料欄未明確指定任何值，則會插入隱含序列產生的值。例如，使用上述定義並假設其他適當的資料欄，撰寫：

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

會從 1 開始產生資料id欄的值，並產生下列資料表資料：

```
 id | name | address
```

```
----+------+---------
  1 | A    | foo
  2 | B    | bar
```

或者，DEFAULT可以指定關鍵字來取代值，以明確請求序列產生的值：

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

同樣地， 關鍵字DEFAULT可用於 UPDATE 命令。

因此，在許多方面，身分資料欄的行為就像具有預設值的資料欄。

資料欄定義BY DEFAULT中的 子句 ALWAYS和 會決定在 INSERT和 UPDATE命令中明確處理使用者指定值的方式。在 INSERT命令中，如果選取 ALWAYS ，則只有在INSERT陳述式指定 時，才會接受使用者指定的值OVERRIDING SYSTEM VALUE。如果選取 BY DEFAULT ，則使用者指定的值優先。因此，使用 BY DEFAULT 會產生與預設值更相似的行為，其中預設值可以被明確值覆寫，而 ALWAYS 提供更多保護，防止意外插入明確值。

身分資料欄的資料類型必須是序列支援的其中一種資料類型。(請參閱 CREATE SEQUENCE)。建立身分資料欄 （請參閱 CREATE TABLE) 時或稍後變更 （請參閱）時，可能會指定相關序列的屬性ALTER TABLE。

身分資料欄會自動標記為 NOT NULL。不過，身分資料欄並不保證唯一性。（序列通常會傳回唯一值，但序列可以重設，也可以手動將值插入身分資料欄，如先前所述。）必須使用 PRIMARY KEY或 UNIQUE 限制條件強制執行唯一性。

使用序列和身分資料欄

本節協助您了解如何根據工作負載模式，以最佳方式使用序列和身分資料欄。

Important

如需配置和快取行為的詳細資訊，請參閱 CREATE SEQUENCE頁面上的重要標註。

選擇識別符類型

Amazon Aurora DSQL 支援 UUID 型識別符，以及使用序列或身分資料欄產生的整數值。這些選項的配置方式，以及它們在負載下擴展的方式有所不同。

UUID 值可在不協調的情況下產生，非常適合經常或在許多工作階段之間建立識別符的工作負載。由於 Amazon Aurora DSQL 專為分散式操作而設計，因此避免協調通常很有幫助。因此，建議使用 UUIDs 做為預設識別符類型，特別是對於可擴展性很重要且不需要嚴格排序識別符的工作負載中的主要金鑰。

序列和身分資料欄會產生小型整數值，方便人類讀取識別符、報告和外部界面。當基於可用性或整合原因而偏好數值識別符時，請考慮使用序列或身分資料欄搭配 UUID 型識別符。當需要整數序列或身分值時，選擇適當的快取大小會成為工作負載設計的重要部分。如需選擇快取大小的指引，請參閱下一節。

選擇快取大小

選取適當的快取值是有效使用序列和身分資料欄的重要部分。快取設定會決定識別碼配置在負載下的行為方式，同時影響系統輸送量和值反映配置順序的緊密程度。

較大的快取大小`CACHE >= 65536`非常適合下列情況：

- 以高頻率產生識別符
- 許多工作階段同時插入
- 工作負載可以容忍差距和可見的排序效果

例如，大量事件擷取工作負載 （例如 IoT 或遙測），以及作業執行 IDs、支援案例參考或內部順序號碼等操作識別符，通常會受益於較大的快取大小，其中經常產生識別符，而且不需要嚴格排序。

在下列情況下，快取大小為 1 會更好地對齊：

- 配置率相對較低
- 預期識別符會隨著時間更密切地遵循配置順序
- 將差距降至最低比最大輸送量更重要

指派帳戶或參考號碼等工作負載，其中識別符的產生頻率較低，而且需要更接近的排序，與快取大小 1 更一致。

Aurora DSQL 的非同步索引

`CREATE INDEX ASYNC` 命令會在指定資料表的一或多個資料欄上建立索引。此命令是一種非同步 DDL 操作，不會封鎖其他交易。當您執行 `CREATE INDEX ASYNC` 時，Aurora DSQL 會立即傳回 `job_id`。

您可以使用 `sys.jobs` 系統檢視監控此非同步作業的狀態。當索引正在建立作業時，您可以使用下列程序和命令：

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

封鎖目前的工作階段，直到指定的作業完成或失敗為止。傳回一個表示成功或失敗的布林值。

`DROP INDEX`

取消進行中的索引建置作業。

當非同步索引建立完成時，Aurora DSQL 會更新系統目錄，將索引標記為作用中。

> **Note**
> 請注意，在此更新期間，存取相同命名空間中物件的並行交易可能會發生並行錯誤。

當 Aurora DSQL 完成非同步索引任務時會更新系統目錄，以顯示該索引處於作用中狀態。如果其他交易目前參考相同命名空間中的物件，可能會顯示並行錯誤。

語法

`CREATE INDEX ASYNC` 使用下列語法。

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

Parameters

`UNIQUE`

指示 Aurora DSQL，在您每次新增資料時，檢查資料表中是否有重複的值。如果您指定此參數，具重複項目的插入和更新作業會產生錯誤。

`IF NOT EXISTS`

指示 Aurora DSQL，如果具相同名稱的索引已存在，不用擲回例外狀況。在這種情況下，Aurora DSQL 不會建立新的索引。請注意，您嘗試建立的索引，其結構可能與已存在的索引截然不同。如果您未指定此參數，則索引名稱為必要。

name

索引的名稱。您無法在此參數中包含結構描述的名稱。

Aurora DSQL 會以索引的父資料表相同結構描述建立索引。索引的名稱必須不同於相同結構描述中任何其他物件 (資料表或索引) 的名稱。

如果您未指定名稱，Aurora DSQL 會依據父資料表和索引資料欄的名稱自動產生名稱。例如，如果您執行 `CREATE INDEX ASYNC on table1 (col1, col2)`，Aurora DSQL 會自動將索引命名為 `table1_col1_col2_idx`。

NULLS FIRST | LAST

Null 和非 Null 資料欄的排序順序。`FIRST` 指示 Aurora DSQL 應該在非 Null 資料欄之前排序 Null 資料欄。`LAST` 指示 Aurora DSQL 應該在非 Null 資料欄之後排序 Null 資料欄。

INCLUDE

要以非索引鍵資料欄形式包含在索引中的資料欄清單。您不能在索引掃描搜尋資格中使用非索引鍵資料欄。Aurora DSQL 會基於索引的唯一性而忽略資料欄。

NULLS DISTINCT | NULLS NOT DISTINCT

指定 Aurora DSQL 是否應將 Null 值視為唯一索引中的不同值。預設值為 `DISTINCT`，表示唯一索引可以在資料欄中包含多個 Null 值。`NOT DISTINCT` 表示索引不能在資料欄中包含多個 Null 值。

使用須知

請考量下列準則：

- `CREATE INDEX ASYNC` 命令不會引發鎖定。也不會影響 Aurora DSQL 用來建立索引的基礎資料表。
- 在結構描述移轉作業期間，`sys.wait_for_job(job_id)`*`'your_index_creation_job_id'`* 程序非常實用。其可確保後續的 DDL 和 DML 操作以新建立的索引為目標。
- 每次 Aurora DSQL 執行新的非同步任務時，都會檢查 `sys.jobs` 檢視並刪除狀態為 `completed` 或因超過 30 分鐘而狀態為 `failed` 的任務。因此，`sys.jobs` 主要會顯示進行中的任務，而不會包含舊任務的相關資訊。
- 如果 Aurora DSQL 無法建立非同步索引，則索引會維持 `INVALID`。若是唯一索引，DML 作業會受到唯一性限制，直到您捨棄索引為止。建議您捨棄無效的索引並重新建立索引。

建立索引：範例

下列範例說明如何建立結構描述、資料表以及索引。

1. 建立名為 `test.departments` 的資料表。

```
CREATE SCHEMA test;

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

2. 在資料表中插入一個資料列。

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

3. 建立非同步索引。

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

`CREATE INDEX` 命令會傳回作業 ID，如下所示。

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

`job_id` 指出 Aurora DSQL 已提交新作業以建立索引。您可以使用 `sys.wait_for_job(job_id)`*'your_index_creation_job_id'* 程序封鎖工作階段的其他工作，直到作業完成或逾時為止。

查詢索引建立的狀態：範例

查詢 `sys.jobs` 系統檢視以檢查索引的建立狀態，如下列範例所示。

```
SELECT * FROM sys.jobs where job_id = 'wqhu6ewifze5xitg3umt24h5ua';
```

Aurora DSQL 會傳回類似如下的回應。

```
          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
```

狀態資料欄可以是下列其中一個值。

狀態	Description
submitted	任務已提交，但 Aurora DSQL 尚未開始處理。
processing	Aurora DSQL 正在處理任務。
failed	任務失敗。如需詳細資訊，請參閱詳細資料欄。如果 Aurora DSQL 無法建置索引，Aurora DSQL 不會自動移除索引定義。您必須使用 DROP INDEX 命令手動移除索引。
completed	Aurora DSQL 已成功完成任務。

您也可以透過目錄資料表 pg_index 和 pg_class 查詢索引的狀態。具體來說，您可以從屬性 indisvalid 和 indisimmediate 得知索引的狀態。當 Aurora DSQL 建立索引時，其初始狀態為 INVALID。索引的 indisvalid 旗標會傳回 FALSE 或 f，表示索引無效。如果旗標傳回 TRUE 或 t，表示索引已就緒。

```
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)
```

唯一索引建置失敗

如果您的非同步唯一索引建置作業顯示失敗狀態，詳細資料為 `Found duplicate key while validating index for UCVs`，這表示因違反唯一性限制而無法建置唯一索引。

解決唯一的索引建置失敗

1. 針對唯一次要索引中指定的索引鍵，移除主要資料表中具重複項目的任何資料列。
2. 捨棄失敗的索引。
3. 發出新的建立索引命令。

偵測主要資料表是否違反唯一性

下列 SQL 查詢有助您找出資料表指定欄中的重複值。此功能特別適合為下列資料欄強制執行唯一性：目前未設定為主索引鍵或不含唯一限制的資料欄，例如使用者資料表中的電子郵件地址。

以下範例說明如何建立範例使用者資料表、填入含已知重複項目的測試資料，然後執行偵測查詢。

定義資料表結構描述

```
-- 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
);
```

插入包含一組重複電子郵件地址的範例資料

```
-- 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');
```

執行重複偵測查詢

```
-- 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;
```

檢視具有重複電子郵件地址的所有記錄

```
 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)
```

如果我們現在嘗試索引建立陳述式，將會失敗：

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

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

Aurora DSQL 的系統資料表和命令

請參閱下列各節，以了解 Aurora DSQL 中支援的系統資料表和目錄，以及用於擷取系統相關資訊的實用查詢，例如 版本。

系統表

Aurora DSQL 與 PostgreSQL 相容，因此 Aurora DSQL 也有來自 PostgreSQL 的許多系統目錄資料表和檢視。

重要的 PostgreSQL 目錄資料表和檢視

下表列出可在 Aurora DSQL 中使用的最常見資料表和檢視。

名稱	描述
pg_namespace	所有結構描述的相關資訊
pg_tables	所有資料表的相關資訊
pg_attribute	所有屬性的相關資訊
pg_views	(預先) 定義檢視的相關資訊
pg_class	描述所有資料表、資料欄、索引和類似物件

名稱	描述
pg_stats	規劃器統計資料的檢視
pg_user	使用者的相關資訊
pg_roles	使用者和群組的相關資訊
pg_indexes	列出所有索引
pg_constraint	列出資料表的限制

支援和不支援的目錄資料表

下表指出 Aurora DSQL 支援和不支援哪些資料表。

名稱	適用於 Aurora DSQL
pg_aggregate	否
pg_am	是
pg_amop	否
pg_amproc	否
pg_attrdef	是
pg_attribute	是
pg_authid	否 (使用 pg_roles)
pg_auth_members	是
pg_cast	是
pg_class	是
pg_collation	是

名稱	適用於 Aurora DSQL
pg_constraint	是
pg_conversion	否
pg_database	否
pg_db_role_setting	是
pg_default_acl	是
pg_depend	是
pg_description	是
pg_enum	否
pg_event_trigger	否
pg_extension	否
pg_foreign_data_wrapper	否
pg_foreign_server	否
pg_foreign_table	否
pg_index	是
pg_inherits	是
pg_init_privs	否
pg_language	否
pg_largeobject	否
pg_largeobject_metadata	是
pg_namespace	是

名稱	適用於 Aurora DSQL
pg_opclass	否
pg_operator	是
pg_opfamily	否
pg_parameter_acl	是
pg_partitioned_table	否
pg_policy	否
pg_proc	否
pg_publication	否
pg_publication_namespace	否
pg_publication_rel	否
pg_range	是
pg_replication_origin	否
pg_rewrite	否
pg_seclabel	否
pg_sequence	否
pg_shdepend	是
pg_shdescription	是
pg_shseclabel	否
pg_statistic	是
pg_statistic_ext	否

名稱	適用於 Aurora DSQL
pg_statistic_ext_data	否
pg_subscription	否
pg_subscription_rel	否
pg_tablespace	否
pg_transform	否
pg_trigger	否
pg_ts_config	是
pg_ts_config_map	是
pg_ts_dict	是
pg_ts_parser	是
pg_ts_template	是
pg_type	是
pg_user_mapping	否

支援和不支援的系統檢視

下表指出 Aurora DSQL 支援和不支援哪些檢視。

名稱	適用於 Aurora DSQL
pg_available_extensions	否
pg_available_extension_versions	否
pg_backend_memory_contexts	是

名稱	適用於 Aurora DSQL
pg_config	否
pg_cursors	否
pg_file_settings	否
pg_group	是
pg_hba_file_rules	否
pg_ident_file_mappings	否
pg_indexes	是
pg_locks	否
pg_matviews	否
pg_policies	否
pg_prepared_statements	否
pg_prepared_xacts	否
pg_publication_tables	否
pg_replication_origin_status	否
pg_replication_slots	否
pg_roles	是
pg_rules	否
pg_seclabels	否
pg_sequences	否
pg_settings	是

名稱	適用於 Aurora DSQL
pg_shadow	是
pg_shmem_allocations	是
pg_stats	是
pg_stats_ext	否
pg_stats_ext_exprs	否
pg_tables	是
pg_timezone_abbrevs	是
pg_timezone_names	是
pg_user	是
pg_user_mappings	否
pg_views	是
pg_stat_activity	否
pg_stat_replication	否
pg_stat_replication_slots	否
pg_stat_wal_receiver	否
pg_stat_recovery_prefetch	否
pg_stat_subscription	否
pg_stat_subscription_stats	否
pg_stat_ssl	是
pg_stat_gssapi	否

名稱	適用於 Aurora DSQL
pg_stat_archiver	否
pg_stat_io	否
pg_stat_bgwriter	否
pg_stat_wal	否
pg_stat_database	否
pg_stat_database_conflicts	否
pg_stat_all_tables	否
pg_stat_all_indexes	否
pg_statio_all_tables	否
pg_statio_all_indexes	否
pg_statio_all_sequences	否
pg_stat_slru	否
pg_statio_user_tables	否
pg_statio_user_sequences	否
pg_stat_user_functions	否
pg_stat_user_indexes	否
pg_stat_progress_analyze	否
pg_stat_progress_basebackup	否
pg_stat_progress_cluster	否
pg_stat_progress_create_index	否

名稱	適用於 Aurora DSQL
pg_stat_progress_vacuum	否
pg_stat_sys_indexes	否
pg_stat_sys_tables	否
pg_stat_xact_all_tables	否
pg_stat_xact_sys_tables	否
pg_stat_xact_user_functions	否
pg_stat_xact_user_tables	否
pg_statio_sys_indexes	否
pg_statio_sys_sequences	否
pg_statio_sys_tables	否
pg_statio_user_indexes	否

sys.jobs 檢視

sys.jobs 提供非同步作業的狀態資訊。例如，在您建立非同步索引之後，Aurora DSQL 會傳回 job_uuid。您可以搭配使用此 job_uuid 與 sys.jobs 查詢作業的狀態。

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

Aurora DSQL 會傳回類似如下的回應。

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

```
fyopxjb6ovdn7po6lrkj63cyea | completed |          | DROP          |      1259 |      26422
|                    | 2025-09-25 22:05:57+00 | 2025-09-25 22:06:03+00
```

下表說明sys.jobs檢視中的資料欄。

sys.jobs 檢視資料欄

資料行	Type	說明
job_id	text	代表任務的 base-32 UUID。
status	text	任務的目前狀態。可能值為 submitted 、processing 、completed 和 failed。如需詳細資訊，請參閱sys.jobs 狀態值。
details	text	任務的任何相關詳細資訊。如果任務失敗，則會提供詳細原因。
job_type	text	非同步任務的類型。可能的值為： INDEX_BUILD – 非同步索引組建。 ANALYZE – 系統提交的自動分析任務。 DROP – 在 DROP TABLE或 DROP INDEX操作之後移除實體資料。
class_id	oid	目錄資料表的 OID，其中包含 物件。
object_id	oid	物件的 OID。
object_name	text	物件的完整名稱。 DROP任務無法參考已捨棄的物件。如果已捨棄參考的物件，則 object_name 可能是 NULL。
start_time	timestamp with time zone	提交任務的時間戳記。
update_time	timestamp with time zone	任務列上次更新的時間戳記。

sys.jobs 狀態值

狀態	Description
`submitted`	任務已提交，但 Aurora DSQL 尚未開始處理。
`processing`	Aurora DSQL 正在處理任務。
`failed`	任務失敗。如需詳細資訊，請參閱 `details` 欄。
`completed`	Aurora DSQL 已成功完成任務。

sys.iam_pg_role_mappings 檢視

檢視 `sys.iam_pg_role_mappings` 提供授予 IAM 使用者的許可相關資訊。例如，如果 `DQSLDBConnect` 是授予 Aurora DSQL 非管理員存取權的 IAM 角色，而名為 `testuser` 的使用者獲授予 `DQSLDBConnect` 角色和對應的許可，您可以查詢 `sys.iam_pg_role_mappings` 檢視以查看哪些使用者獲得哪些許可。

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

有用的系統中繼資料查詢

使用這些查詢來取得資料表統計資料和系統中繼資料，而無需執行昂貴的操作，例如完整資料表掃描。

取得資料表的預估資料列計數

若要在不執行完整資料表掃描的情況下取得資料表中資料列的大致計數，請使用下列查詢：

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

此命令會傳回類似以下的輸出：

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

此方法比 Aurora DSQL 中的`SELECT COUNT(*)`大型資料表更有效率。

取得目前的 Aurora DSQL 主要版本

若要取得 Aurora DSQL 叢集的目前主要版本，請使用下列查詢：

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

此命令會傳回類似以下的輸出：

```
 dsql_major_version
--------------------
                  1
```

這會傳回 SQL 連線在 Aurora DSQL 中的主要版本。

取得目前的 PostgreSQL 版本

若要取得 Aurora DSQL 叢集的目前 PostgreSQL 版本，請使用下列查詢：

```
SHOW server_version;
```

此命令會傳回類似以下的輸出：

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

這會傳回 SQL 連線在 Aurora DSQL 中的 PostgreSQL 版本。

ANALYZE 命令。

ANALYZE 命令會收集資料庫中資料表內容的統計資料，並將結果儲存在 pg_stats 系統檢視中。之後，查詢規劃器會使用這些統計資料，協助判斷最有效率的查詢執行計劃。

使用 Aurora DSQL 時，您無法在明確交易內執行 ANALYZE 命令。ANALYZE 不受資料庫交易逾時限制的約束。

為了減少手動介入的需求，並確保統計資料的最新狀態，Aurora DSQL 會自動以背景處理程序執行 ANALYZE。系統會依據資料表中觀察到的變更率自動觸發這項背景作業。其與自前次分析後已插入、更新或刪除的資料列 (元組) 數目連結。

ANALYZE 會以非同步方式在背景執行，其活動可透過下列查詢在系統檢視 sys.jobs 中進行監控：

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

重要考量事項

Note

ANALYZE 作業會按 Aurora DSQL 中的其他非同步作業一樣計費。當您修改資料表時，可能會間接觸發自動背景統計資料收集作業，進而因相關聯的系統層級活動而產生計量費用。

系統會自動觸發背景 ANALYZE 任務、收集與手動 ANALYZE 相同的統計資料類型，並依預設將其套用到使用者資料表。此自動化程序會排除系統和目錄資料表。

使用 Aurora DSQL EXPLAIN 計劃

Aurora DSQL 使用與 PostgreSQL 類似的 EXPLAIN 計劃結構，但具有反映其分散式架構和執行模型的新增金鑰。

在本文件中，我們將提供 Aurora DSQL EXPLAIN 計劃的概觀，重點介紹與 PostgreSQL 相比的相似性和差異。我們將涵蓋 Aurora DSQL 中可用的各種掃描操作類型，並協助您了解執行查詢的成本。

PostgreSQL VS Aurora DSQL EXPLAIN 計劃

Aurora DSQL 建立在 PostgreSQL 資料庫之上，並與 PostgreSQL 共用大多數計劃結構，但具有影響查詢執行和最佳化的關鍵架構差異：

功能	PostgreSQL	Aurora DSQL
資料儲存體	堆積儲存	沒有堆積，所有資料列都會以唯一識別符編製索引
主索引鍵	主索引鍵索引與資料表資料不同	主索引鍵索引是包含所有額外資料欄做為 INCLUDE 資料欄的資料表
次要索引	標準次要索引	運作方式與 PostgreSQL 相同，能夠包含非索引鍵資料欄

功能	PostgreSQL	Aurora DSQL
篩選功能	索引條件、堆積篩選條件	索引條件、儲存篩選條件、查詢處理器篩選條件
掃描類型	循序掃描、索引掃描、僅限索引掃描	完整掃描、僅限索引掃描、索引掃描
查詢執行	資料庫本機	分散式（運算和儲存是分開的）

Aurora DSQL 會以主索引鍵順序直接存放資料表資料，而不是以個別堆積存放。每一列都由唯一索引鍵識別，通常是主索引鍵，可讓資料庫更有效率地最佳化查詢。架構差異說明在 PostgreSQL 可能選擇循序掃描的情況下，Aurora DSQL 經常使用僅限索引掃描的原因。

另一個關鍵區別在於，Aurora DSQL 會將運算與儲存區分開，讓篩選條件能更早在執行路徑中套用，以減少資料移動並改善效能。

如需搭配 PostgreSQL 使用 EXPLAIN 計劃的詳細資訊，請參閱 PostgreSQL EXPLAIN 文件。

Aurora DSQL EXPLAIN 計畫中的關鍵元素

Aurora DSQL EXPLAIN 計畫提供如何執行查詢的詳細資訊，包括進行篩選的位置，以及從儲存體擷取哪些資料欄。了解此輸出可協助您最佳化查詢效能。

索引條件

用來導覽索引的條件。最有效率的篩選，可減少掃描的資料。在 Aurora DSQL 中，索引條件可以套用至執行計畫的多層。

投影

從儲存體擷取的資料欄。較少的投影表示效能更好。

儲存篩選條件

在儲存層級套用的條件。比查詢處理器篩選條件更有效率。

查詢處理器篩選條件

在查詢處理器層級套用的條件。篩選之前需要傳輸所有資料，這會導致更高的資料移動和處理額外負荷。

Aurora DSQL 中的篩選條件

Aurora DSQL 會將運算與儲存區分開，這表示在查詢執行期間套用篩選條件的點會對效能產生重大影響。在傳輸大量資料之前套用的篩選條件可減少延遲並改善效率。越早套用篩選條件，需要處理、移動和掃描的資料就越少，從而加快查詢速度。

Aurora DSQL 可以在查詢路徑的多個階段套用篩選條件。了解這些階段是解譯查詢計劃和最佳化效能的關鍵。

Level	篩選條件類型	描述
1	索引條件	掃描索引時套用。限制從儲存體讀取的資料量，並減少傳送至運算層的資料。
2	儲存篩選條件	從儲存體讀取資料之後，但在傳送至運算之前套用。這裡的範例是 索引包含資料欄上的篩選條件。減少資料傳輸，但不會減少讀取量。
3	查詢處理器篩選條件	在資料到達運算層後套用。所有資料都必須先傳輸，這會增加延遲和成本。目前，Aurora DSQL 無法在儲存體上執行所有篩選和投影操作，因此某些查詢可能會被迫回復到這種類型的篩選。

讀取 Aurora DSQL EXPLAIN 計劃

了解如何讀取 EXPLAIN 計劃是最佳化查詢效能的關鍵。在本節中，我們將逐步介紹 Aurora DSQL 查詢計畫的真實範例、顯示不同掃描類型的行為、說明套用篩選條件的位置，以及強調最佳化的機會。

這些範例中使用的範例資料表

以下範例參考兩個資料表： `transaction`和 `account`。

`transaction` 資料表沒有主索引鍵，這會導致 Aurora DSQL 在查詢資料表時執行完整資料表掃描。

`account` 資料表在 上有索引`customer_id`。此索引包含 `balance`和 `status`作為涵蓋欄，這允許直接從索引滿足某些查詢，而無需從基礎資料表讀取。不過，索引不包含 `created_at`，因此參考此欄的查詢需要額外的資料表存取。

```
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);
```

完整掃描範例

Aurora DSQL 具有序列掃描，其功能與 PostgreSQL 和完整掃描相同。兩者之間的唯一區別是完整掃描可以對儲存體使用額外的篩選。因此，幾乎一律會在連續掃描上方選取。由於相似性，我們只會涵蓋更有趣的完整掃描範例。

完整掃描主要用於沒有主索引鍵的資料表。由於 Aurora DSQL 主金鑰預設為完全涵蓋索引，因此在 PostgreSQL 會使用循序掃描的許多情況下，Aurora DSQL 很可能在主金鑰上使用僅限索引掃描。與大多數其他資料庫一樣，沒有索引的資料表會嚴重擴展。

```
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)
```

該計劃顯示在不同階段套用的兩個篩選條件。`transaction_date > '2025-01-01'` 條件會套用至儲存層，減少傳回的資料量。條件稍後`description LIKE '%external%'`會在傳輸資料後，於查詢處理器中套用，使其效率降低。將更多選擇性篩選條件推送至儲存體或索引層通常會改善效能。

僅限索引掃描範例

僅索引掃描是 Aurora DSQL 中最理想的掃描類型，因為它們導致到儲存層的往返次數最少，並且可以進行最多的篩選。但是，因為您看到僅限索引掃描並不表示您擁有最佳計劃。由於可能發生的所有不同層級的篩選，仍須注意可能發生篩選的不同位置。

```
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)
```

在此計劃中，會先在索引掃描期間評估索引條件 `customer_id = '4b18a761-5870-4d7c-95ce-0a48eca3fceb'`，這是最有效率的階段，因為它會限制從儲存體讀取的資料量。儲存篩選條件 `status = 'pending'`會在資料讀取後，但在傳送至運算層之前套用，以減少傳輸的資料量。最後，查詢處理器篩選條件 會在資料移動後最後`balance > 100`執行，使其效率最低。其中，索引條件可提供最大效能，因為它會直接控制掃描的資料量。

索引掃描範例

索引掃描類似於僅限索引掃描，但它們需要呼叫 基底資料表的額外步驟。由於 Aurora DSQL 可以指定儲存篩選條件，因此可以在索引呼叫和查詢呼叫上執行此操作。

為了清楚這一點，Aurora DSQL 會將計劃顯示為兩個節點。如此一來，您可以清楚地看到從儲存體傳回的資料列中新增包含資料欄將有多少幫助。

```
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)
```

此計劃顯示篩選如何跨多個階段進行：

- 上的索引條件會及早customer_id 篩選資料。
- 上的儲存篩選條件會status進一步縮小結果，然後再傳送至運算。
- 上的查詢處理器篩選條件balance稍後會在傳輸後套用。
- 從基礎資料表擷取其他資料欄時created_at，會評估 上的查詢篩選條件。

新增經常使用的資料欄，因為INCLUDE欄位通常可以消除此查詢並改善效能。

最佳實務

- 將篩選條件與索引資料欄對齊，以提早推送篩選。
- 使用 INCLUDE 資料欄允許僅限索引掃描並避免查詢。
- 調查效能問題時驗證資料列預估。Aurora DSQL 會根據資料變更率ANALYZE在背景執行，以自動管理統計資料。如果估計值看起來不準確，您可以ANALYZE手動執行 以立即重新整理統計資料。
- 避免大型資料表上的未索引查詢，以避免昂貴的完全掃描。

了解 EXPLAIN ANALYZE 中的 DPUs

Aurora DSQL 在EXPLAIN ANALYZE VERBOSE計劃輸出中提供陳述式層級分散式處理單元 (DPU) 資訊，讓您在開發期間更深入地了解查詢成本。本節說明什麼是 DPUs，以及如何在EXPLAIN ANALYZE VERBOSE輸出中解譯它們。

什麼是 DPU？

分散式處理單元 (DPU) 是 Aurora DSQL 所完成工作的標準化測量。它由以下項目組成：

- ComputeDPU – 執行 SQL 查詢所花費的時間
- ReadDPU – 用於從儲存體讀取資料的資源
- WriteDPU - 用來將資料寫入儲存體的資源
- MultiRegionWriteDPU – 用來將寫入複寫到多區域組態中對等叢集的資源。

EXPLAIN ANALYZE VERBOSE 中的 DPU 用量

Aurora DSQL 延伸`EXPLAIN ANALYZE VERBOSE`到包含陳述式層級的 DPU 用量預估，直到輸出結束。這可讓您立即了解查詢成本，協助您識別工作負載成本驅動因素、調整查詢效能，以及更好地預測資源用量。

下列範例示範如何解譯 EXPLAIN ANALYZE VERBOSE 輸出中包含的陳述式層級 DPU 預估值。

範例 1：SELECT 查詢

```
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
```

在此範例中，SELECT 陳述式會執行僅索引掃描，因此大部分成本來自讀取 DPU (0.04312)，代表從儲存和運算 DPU (0.01607) 擷取的資料，反映用於處理和傳回結果的運算資源。由於查詢不會修改資料，因此沒有寫入 DPU。總 DPU (0.05919) 是運算 + 讀取 + 寫入的總和。

範例 2：INSERT 查詢

```
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
```

此陳述式主要執行寫入，因此大多數成本都與寫入 DPU 相關聯。運算 DPU (0.01550) 代表處理和插入值所完成的工作。讀取 DPU (0.00307) 反映次要系統讀取（用於目錄查詢或索引檢查）。

請注意讀寫 DPUs旁顯示的交易下限。這些表示只有在操作包含讀取或寫入時才套用的每筆交易基準成本。並不表示每筆交易會自動產生 0.00375 讀取 DPU 或 0.05 寫入 DPU 費用。相反地，這些最小值會在成本彙總期間套用到交易層級，而且只有在該交易中發生讀取或寫入時才會套用。由於範圍的這種差異， 中的陳述式層級估算`EXPLAIN ANALYZE VERBOSE`可能不完全符合 CloudWatch 或帳單資料中報告的交易層級指標。

使用 DPU 資訊進行最佳化

每個陳述式 DPU 預估可讓您在執行時間之外最佳化查詢的強大方式。常用案例包括：

- 成本意識：了解查詢相對於其他查詢的成本。
- 結構描述最佳化：比較索引或結構描述變更對效能和資源效率的影響。

- 預算規劃：根據觀察到的 DPU 用量估計工作負載成本。
- 查詢比較：依其相對 DPU 耗用量評估替代查詢方法。

解譯 DPU 資訊

使用來自 的 DPU 資料時，請記住下列最佳實務`EXPLAIN ANALYZE VERBOSE`：

- 以方向使用：將報告的 DPU 視為了解查詢相對成本的方式，而不是與 CloudWatch 指標或帳單資料完全相符。由於`EXPLAIN ANALYZE VERBOSE`報告陳述式層級成本，而 CloudWatch 彙總交易層級活動，因此預期會有差異。CloudWatch 也包含`EXPLAIN ANALYZE VERBOSE`刻意排除的背景操作（例如 ANALYZE 或壓縮）和交易額外負荷 (`BEGIN`/`COMMIT`)。
- 在分散式系統中，跨執行的 DPU 變異性是正常的，不表示錯誤。快取、執行計畫變更、並行或資料分佈中的轉移等因素都可能導致相同的查詢從一個執行消耗不同的資源。
- 批次小型操作：如果您的工作負載發出許多小型陳述式，請考慮將它們批次處理成較大的操作（不超過 10MB)。這可減少四捨五入的開銷，並產生更有意義的成本預估。
- 用於調校，而非計費： 中的 DPU 資料`EXPLAIN ANALYZE VERBOSE`專為成本感知、查詢調校和最佳化而設計。它不是帳單等級指標。一律依賴 CloudWatch 指標或每月帳單報告，以取得授權成本和用量資料。

管理 Aurora 資料庫叢集

Aurora DSQL 提供多種組態選項，可協助建立符合您需求的正確資料庫基礎架構。若要設定 Aurora DSQL 叢集基礎架構，請檢閱下列各節。

主題

- 設定單一區域叢集
- 設定多區域叢集
- 使用 AWS CloudFormation 設定 Aurora DSQL 叢集
- Aurora DSQL 叢集生命週期

本指南中所討論的特徵與功能，可確保您的 Aurora DSQL 環境具有更高的彈性、回應能力，並能夠在應用程式成長和發展時提供支援。

設定單一區域叢集

使用 AWS CLI 或您偏好的程式設計語言，設定及管理 AWS 區域 的叢集，包括 Python、C++、JavaScript、Java、Rust、Ruby、.NET 和 Golang。AWS CLI 可透過 Shell 命令提供快速存取，而 AWS 軟體開發套件 (SDK) 可透過原生語言支援，啟用程式設計控制能力。

主題

- 使用 AWS SDK
- 使用 AWS CLI

使用 AWS SDK

AWS SDK 以您偏好的程式設計語言，提供 Aurora DSQL 的程式設計存取。下列各節說明如何使用不同的程式設計語言，執行常見的叢集作業。

建立叢集

下列範例說明如何使用不同的程式設計語言建立單一區域叢集。

Python

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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++

下列範例可讓您在單一 AWS 區域 中建立叢集。

```
#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

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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

使用下列範例在單一 AWS 區域 中建立叢集。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;

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

public class CreateCluster {

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

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

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

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

Rust

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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

若要在單一 AWS 區域 中建立叢集，請使用下列範例。

```
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)
 }

}
```

取得叢集

下列範例說明如何取得以不同程式設計語言建立單一區域叢集的相關資訊。

Python

若要取得單一區域叢集的相關資訊，請使用下列範例。

```
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++

使用下列範例取得單一區域叢集的相關資訊。

```
#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

若要取得單一區域叢集的相關資訊，請使用下列範例。

```
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

您可透過下列範例取得單一區域叢集的相關資訊。

```
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

您可透過下列範例取得單一區域叢集的相關資訊。

```
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

您可透過下列範例取得單一區域叢集的相關資訊。

```
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

您可透過下列範例取得單一區域叢集的相關資訊。

```
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

您可透過下列範例取得單一區域叢集的相關資訊。

```
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)
 }
}
```

更新叢集

下列範例說明如何使用不同的程式設計語言更新單一區域叢集。

Python

若要更新單一區域叢集，請使用下列範例。

```
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++

使用以下範例更新單一區域叢集。

```
#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

若要更新單一區域叢集，請使用下列範例。

```
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

使用以下範例更新單一區域叢集。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsql.model.UpdateClusterResponse;

public class UpdateCluster {

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

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            UpdateClusterRequest request = UpdateClusterRequest.builder()
                    .identifier(clusterId)
                    .deletionProtectionEnabled(false)
                    .build();
            UpdateClusterResponse cluster = client.updateCluster(request);
            System.out.println("Updated " + cluster.arn());
```

```
        }
    }
}
```

Rust

使用以下範例更新單一區域叢集。

```
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

使用以下範例更新單一區域叢集。

```
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

使用以下範例更新單一區域叢集。

```
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

使用以下範例更新單一區域叢集。

```
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)
 }
}
```

刪除叢集

下列範例說明如何使用不同的程式設計語言刪除單一區域叢集。

Python

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
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++

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
#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

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
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

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

import java.time.Duration;

public class DeleteCluster {

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


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

            // The DSQL SDK offers a built-in waiter to poll for deletion.
```

```
            System.out.println("Waiting for cluster to finish deletion");
            client.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );
            System.out.println("Deleted " + cluster.arn());
        } catch (ResourceNotFoundException e) {
            System.out.printf("Cluster %s not found in %s%n", clusterId, region);
        }
    }
}
```

Rust

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
use aws_config::load_defaults;
use aws_sdk_dsql::client::Waiters;
use aws_sdk_dsql::{
    Client, Config,
    config::{BehaviorVersion, Region},
};

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

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

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

    Client::from_conf(config)
}
```

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

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

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

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

    Ok(())
}
```

Ruby

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
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

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

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

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

```
            {
                RegionEndpoint = region
            };
            return new AmazonDSQLClient(awsCredentials, clientConfig);
        }

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

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

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

            await Delete(region, clusterId);
        }
    }
}
```

Golang

若要刪除單一 AWS 區域 中的叢集，請使用下列範例。

```
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)
 }

}
```

如需更多程式碼樣本和範例，請參閱 Aurora DSQL 範例 GitHub 儲存庫。

使用 AWS CLI

AWS CLI 提供命令列界面，用於管理 Aurora DSQL 叢集。下列範例說明常見的叢集管理作業。

建立叢集

使用 create-cluster 命令建立叢集。

Note

建立叢集為非同步作業。呼叫 GetCluster API，直到狀態變更為 `ACTIVE` 為止。在叢集變成作用中狀態之後，您可以連線到叢集。

Example命令

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

Note

若要在建立期間停用刪除保護，請包含 `--no-deletion-protection-enabled` 標記。

Example回應

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "CREATING",
    "creationTime": "2024-05-25T16:56:49.784000-07:00",
    "deletionProtectionEnabled": true,
    "tag": {},
    "encryptionDetails": {
        "encryptionType": "AWS_OWNED_KMS_KEY",
        "encryptionStatus": "ENABLED"
    }
}
```

描述叢集

使用 get-cluster 命令取得叢集的相關資訊。

Example命令

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

Example回應

```
{
    "identifier": "abc0def1baz2quux3quuux4",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/abc0def1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2024-11-27T00:32:14.434000-08:00",
    "deletionProtectionEnabled": false,
    "encryptionDetails": {
        "encryptionType": "CUSTOMER_MANAGED_KMS_KEY",
        "kmsKeyArn": "arn:aws:kms:us-east-1:111122223333:key/123a456b-c789-01de-2f34-
g5hi6j7k8lm9",
        "encryptionStatus": "ENABLED"
    }
}
```

更新叢集

使用 update-cluster 命令更新現有的叢集。

> **Note**
>
> 更新為非同步作業。呼叫 GetCluster API，直到狀態變更為 `ACTIVE` 為止，以查看您的變更。

Example命令

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

Example回應

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

刪除叢集

使用 delete-cluster 命令刪除現有的叢集。

Note

您只能刪除已停用刪除保護的叢集。當您建立叢集時，預設會啟用刪除保護。

Example命令

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

Example回應

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

列出叢集

使用 list-clusters 命令列出您的叢集。

Example命令

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

Example回應

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

```
            "identifier": "abc0def1baz2quux3quux5quuuux",
            "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/
abc0def1baz2quux3quux5quuuux"
        }
    ]
}
```

設定多區域叢集

使用 AWS CLI 或您偏好的程式設計語言，設定及管理多個 AWS 區域 的叢集，包括 Python、C++、JavaScript、Java、Rust、Ruby、.NET 和 Golang。AWS CLI 可透過 Shell 命令提供快速存取，而 AWS SDK 可透過原生語言支援，啟用程式設計控制能力。

主題

- 使用 AWS SDK
- 使用 AWS CLI

使用 AWS SDK

AWS SDK 以您偏好的程式設計語言，提供 Aurora DSQL 的程式設計存取。下列各節說明如何使用不同的程式設計語言執行常見的叢集作業。

建立叢集

下列範例說明如何使用不同的程式設計語言建立多區域叢集。

Python

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
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++

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
#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

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
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

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.DsqlClientBuilder;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;

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

public class CreateMultiRegionCluster {

    public static void main(String[] args) {
        Region region1 = Region.US_EAST_1;
        Region region2 = Region.US_EAST_2;
        Region witnessRegion = Region.US_WEST_2;

        DsqlClientBuilder clientBuilder = DsqlClient.builder()
                .credentialsProvider(DefaultCredentialsProvider.create());

        try (
            DsqlClient client1 = clientBuilder.region(region1).build();
            DsqlClient client2 = clientBuilder.region(region2).build()
        ) {
            // We can only set the witness region for the first cluster
            System.out.println("Creating cluster in " + region1);
            CreateClusterRequest request1 = CreateClusterRequest.builder()
                    .deletionProtectionEnabled(true)
                    .multiRegionProperties(mrp ->
 mrp.witnessRegion(witnessRegion.toString()))
                    .tags(Map.of("Name", "java multi region cluster"))
                    .build();
            CreateClusterResponse cluster1 = client1.createCluster(request1);
            System.out.println("Created " + cluster1.arn());
```

```
            // For the second cluster we can set the witness region and designate
            // cluster1 as a peer.
            System.out.println("Creating cluster in " + region2);
            CreateClusterRequest request2 = CreateClusterRequest.builder()
                    .deletionProtectionEnabled(true)
                    .multiRegionProperties(mrp ->

 mrp.witnessRegion(witnessRegion.toString()).clusters(cluster1.arn())
                    )
                    .tags(Map.of("Name", "java multi region cluster"))
                    .build();
            CreateClusterResponse cluster2 = client2.createCluster(request2);
            System.out.println("Created " + cluster2.arn());

            // Now that we know the cluster2 ARN we can set it as a peer of cluster1
            UpdateClusterRequest updateReq = UpdateClusterRequest.builder()
                    .identifier(cluster1.identifier())
                    .multiRegionProperties(mrp ->

 mrp.witnessRegion(witnessRegion.toString()).clusters(cluster2.arn())
                    )
                    .build();
            client1.updateCluster(updateReq);
            System.out.printf("Added %s as a peer of %s%n", cluster2.arn(),
 cluster1.arn());

            // Now that MultiRegionProperties is fully defined for both clusters
 they'll begin
            // the transition to ACTIVE.
            System.out.printf("Waiting for cluster %s to become ACTIVE%n",
 cluster1.arn());
            GetClusterResponse activeCluster1 =
 client1.waiter().waitUntilClusterActive(
                    getCluster -> getCluster.identifier(cluster1.identifier()),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            ).matched().response().orElseThrow();

            System.out.printf("Waiting for cluster %s to become ACTIVE%n",
 cluster2.arn());
            GetClusterResponse activeCluster2 =
 client2.waiter().waitUntilClusterActive(
```

```
                    getCluster -> getCluster.identifier(cluster2.identifier()),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            ).matched().response().orElseThrow();

            System.out.println("Created multi region clusters:");
            System.out.println(activeCluster1);
            System.out.println(activeCluster2);
        }
    }
}
```

Rust

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
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

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

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

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

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

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

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

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

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

  [ cluster_1, cluster_2 ]
rescue Aws::Errors::ServiceError => e
```

```
  abort "Failed to create multi-region clusters: #{e.message}"
end

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

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

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

main if $PROGRAM_NAME == __FILE__
```

Golang

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

```
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

若要建立多區域叢集，請使用下列範例。建立多區域叢集可能需要一些時間。

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

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

        /// <summary>
```

```
        /// Create multi-region clusters with a witness region.
        /// </summary>
        public static async Task<(CreateClusterResponse, CreateClusterResponse)>
 Create(
            RegionEndpoint region1,
            RegionEndpoint region2,
            RegionEndpoint witnessRegion)
        {
            using (var client1 = await CreateDSQLClient(region1))
            using (var client2 = await CreateDSQLClient(region2))
            {
                var tags = new Dictionary<string, string>
                {
                    { "Name", "csharp multi region cluster" }
                };

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

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

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

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

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

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

                return (cluster1, cluster2);
            }
        }

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

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

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

取得叢集

下列範例說明如何以不同程式設計語言取得多區域叢集的相關資訊。

Python

若要取得多區域叢集的相關資訊，請使用下列範例。

```
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++

使用下列範例取得多區域叢集的相關資訊。

```
#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

若要取得多區域叢集的相關資訊，請使用下列範例。

```
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

您可透過下列範例取得多區域叢集的相關資訊。

```
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

您可透過下列範例取得多區域叢集的相關資訊。

```
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

您可透過下列範例取得多區域叢集的相關資訊。

```
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

您可透過下列範例取得多區域叢集的相關資訊。

```
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

您可透過下列範例取得多區域叢集的相關資訊。

```
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)
 }
}
```

更新叢集

下列範例說明如何使用不同的程式設計語言更新多區域叢集。

Python

若要更新多區域叢集，請使用下列範例。

```
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++

使用以下範例更新多區域叢集。

```
#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

若要更新多區域叢集，請使用下列範例。

```
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

使用以下範例更新多區域叢集。

```
package org.example;

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

```
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsql.model.UpdateClusterResponse;

public class UpdateCluster {

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

        try (
            DsqlClient client = DsqlClient.builder()
                    .region(region)
                    .credentialsProvider(DefaultCredentialsProvider.create())
                    .build()
        ) {
            UpdateClusterRequest request = UpdateClusterRequest.builder()
                    .identifier(clusterId)
                    .deletionProtectionEnabled(false)
                    .build();
            UpdateClusterResponse cluster = client.updateCluster(request);
            System.out.println("Updated " + cluster.arn());
        }
    }
}
```

Rust

使用以下範例更新多區域叢集。

```
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

使用以下範例更新多區域叢集。

```
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

使用以下範例更新多區域叢集。

```
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

使用以下範例更新多區域叢集。

```
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)
 }
}
```

刪除叢集

下列範例說明如何使用不同的程式設計語言刪除多區域叢集。

Python

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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++

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
#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

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.api.BackoffStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.DsqlClientBuilder;
import software.amazon.awssdk.services.dsql.model.DeleteClusterRequest;

import java.time.Duration;

public class DeleteMultiRegionClusters {

    public static void main(String[] args) {
        Region region1 = Region.US_EAST_1;
        String clusterId1 = "<your cluster id 1>";
        Region region2 = Region.US_EAST_2;
        String clusterId2 = "<your cluster id 2>";

        DsqlClientBuilder clientBuilder = DsqlClient.builder()
                .credentialsProvider(DefaultCredentialsProvider.create());

        try (
```

```
            DsqlClient client1 = clientBuilder.region(region1).build();
            DsqlClient client2 = clientBuilder.region(region2).build()
        ) {
            System.out.printf("Deleting cluster %s in %s%n", clusterId1, region1);
            DeleteClusterRequest request1 = DeleteClusterRequest.builder()
                    .identifier(clusterId1)
                    .build();
            client1.deleteCluster(request1);

            // cluster1 will stay in PENDING_DELETE until cluster2 is deleted
            System.out.printf("Deleting cluster %s in %s%n", clusterId2, region2);
            DeleteClusterRequest request2 = DeleteClusterRequest.builder()
                    .identifier(clusterId2)
                    .build();
            client2.deleteCluster(request2);

            // Now that both clusters have been marked for deletion they will
 transition
            // to DELETING state and finalize deletion.
            System.out.printf("Waiting for cluster %s to finish deletion%n",
 clusterId1);
            client1.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId1),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );

            System.out.printf("Waiting for cluster %s to finish deletion%n",
 clusterId2);
            client2.waiter().waitUntilClusterNotExists(
                    getCluster -> getCluster.identifier(clusterId2),
                    config -> config.backoffStrategyV2(

 BackoffStrategy.fixedDelayWithoutJitter(Duration.ofSeconds(10))
                    ).waitTimeout(Duration.ofMinutes(5))
            );

            System.out.printf("Deleted %s in %s and %s in %s%n", clusterId1,
 region1, clusterId2, region2);
        }
    }
```

```
}
```

Rust

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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

若要刪除多區域叢集，請使用下列範例。刪除多區域叢集可能需要一些時間。

```
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)
 }
}
```

如需更多程式碼樣本和範例，請參閱 Aurora DSQL 範例 GitHub 儲存庫。

使用 AWS CLI

CLI AWS 提供用於管理多區域 Aurora DSQL 叢集的命令列界面。下列範例說明如何建立、設定和刪除多區域叢集。

連線至多區域叢集

多區域對等叢集提供兩個區域端點，每個對等叢集 AWS 區域各一個。這兩個端點都提供單一邏輯資料庫，支援具有強大資料一致性的並行讀取和寫入作業。除了對等叢集之外，多區域叢集還具有見證區域，會存放一段有限的加密交易日誌，用於改善多區域的耐久性和可用性。多區域見證區域不具備端點。

建立多區域叢集

若要建立多區域叢集，您必須先建立具有見證區域的叢集。然後，將此叢集與第二個叢集對等互連，第二個叢集與第一個叢集共用相同的見證區域。下列範例說明如何在美國東部 (維吉尼亞北部) 和美國東部 (俄亥俄) 建立叢集，並以美國西部 (奧勒岡) 做為見證區域。

步驟 1：在美國東部 (維吉尼亞北部) 建立一個叢集

若要在美國東部（維吉尼亞北部） AWS 區域 使用多區域屬性建立叢集，請使用下列命令。

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

Example回應：

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

> **Note**
>
> 當 API 作業成功時，叢集會進入 PENDING_SETUP 狀態。建立的叢集會維持在 PENDING_SETUP 狀態，直到您使用對等叢集的 ARN 為其更新。

步驟 2：在美國東部 (俄亥俄) 建立叢集二

若要在美國東部（俄亥俄）AWS 區域 使用多區域屬性建立叢集，請使用下列命令。

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

Example回應：

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

```
        "witnessRegion": "us-west-2",
        "clusters": [
            "arn:aws:dsql:us-east-2:111122223333:cluster/foo0bar1baz2quux3quuxquux5"
        ]
    }
}
```

當 API 作業成功時，叢集會轉換為 PENDING_SETUP 狀態。建立的叢集會維持在 PENDING_SETUP 狀態，直到您使用另一個對等叢集的 ARN 為其更新。

步驟 3：將美國東部 (維吉尼亞北部) 與美國東部 (俄亥俄) 叢集對等互連

若要將美國東部 (維吉尼亞北部) 叢集與美國東部 (俄亥俄) 叢集對等互連，請使用 update-cluster 命令。使用美國東部 (俄亥俄) 叢集的 ARN，指定美國東部 (維吉尼亞北部) 叢集名稱和 JSON 字串。

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

Example回應

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

步驟 4：將美國東部 (俄亥俄) 與美國東部 (維吉尼亞北部) 叢集對等互連

若要將美國東部 (俄亥俄) 叢集與美國東部 (維吉尼亞北部) 叢集對等互連，請使用 update-cluster 命令。使用美國東部 (維吉尼亞北部) 叢集的 ARN，指定您的美國東部 (俄亥俄) 叢集名稱和 JSON 字串。

Example

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

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

Example回應

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

> **Note**
>
> 成功對等互連後，兩個叢集都會從 "PENDING_SETUP" 轉換為 "CREATING"，最後在準備就緒時轉換為 "ACTIVE" 狀態。

檢視多區域叢集屬性

當您描述叢集時，您可以檢視不同 中叢集的多區域屬性 AWS 區域。

Example

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

Example回應

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

在叢集建立期間進行對等互連

在叢集建立時，您可以將對等資訊包含在內，以減少步驟。在美國東部 (維吉尼亞北部) 建立第一個叢集 (步驟 1) 之後，您可以在美國東部 (俄亥俄) 建立第二個叢集，同時透過含有第一個叢集的 ARN，啟動對等互連程序。

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"]}'
```

此處結合了步驟 2 和 4，但您仍然需要完成步驟 3 (使用第二個叢集的 ARN 更新第一個叢集) 以建立對等互連關係。完成所有步驟後，兩個叢集都會轉換到與標準程序中相同的狀態：從 PENDING_SETUP 到 CREATING，最後在準備就緒時轉換為 ACTIVE。

刪除多區域叢集

若要刪除多區域叢集，您需要完成兩個步驟。

1. 關閉每個叢集的刪除保護。
2. 在各自的 中分別刪除每個對等叢集 AWS 區域

在美國東部 (維吉尼亞北部) 更新和刪除叢集

1. 使用 `update-cluster` 命令關閉刪除保護。

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

2. 使用 `delete-cluster` 命令刪除叢集。

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

該命令會傳回下列回應。

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

> **Note**
>
> 叢集會轉換為 PENDING_DELETE 狀態。在您刪除美國東部 (俄亥俄) 的對等叢集之前，不會完成刪除程序。

在美國東部 (俄亥俄) 更新和刪除叢集

1. 使用 `update-cluster` 命令關閉刪除保護。

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

2. 使用 `delete-cluster` 命令刪除叢集。

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

該命令會傳回下列回應：

```
{
```

```
    "identifier": "foo0bar1baz2quux3quuxquux5",
    "arn": "arn:aws:dsql:us-east-2:111122223333:cluster/
foo0bar1baz2quux3quuxquux5",
    "status": "PENDING_DELETE",
    "creationTime": "2025-05-06T06:46:10.745000-07:00"
}
```

> **Note**
>
> 叢集會轉換為 PENDING_DELETE 狀態。幾秒鐘後，系統會在驗證後自動將兩個對等叢集轉換為 DELETING 狀態。

使用 AWS CloudFormation 設定 Aurora DSQL 叢集

您可以使用相同的 CloudFormation 資源 AWS::DSQL::Cluster 來部署和管理單一區域和多區域 Aurora DSQL 叢集。

請參閱 Amazon Aurora DSQL 資源類型參考，進一步了解如何使用 AWS::DSQL::Cluster 資源建立、修改和管理叢集。

建立初始叢集組態

首先，建立 AWS CloudFormation 範本來定義多區域叢集：

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

使用下列 AWS CLI 命令在兩個區域中建立堆疊：

```
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
```

尋找叢集識別碼

擷取叢集的實體資源 ID：

```
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"
]
```

更新叢集組態

更新您的 AWS CloudFormation 範本以納入兩個叢集 ARN：

```
---
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
```

將更新的組態套用至兩個區域：

```
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 叢集生命週期

了解 Aurora DSQL 叢集生命週期可協助您有效管理叢集。本節涵蓋叢集狀態定義，以及可將成本最佳化的擴展至零功能。

定義 Aurora DSQL 叢集狀態

Aurora DSQL 叢集狀態可提供有關叢集運作狀態和連線能力的重要資訊。您可以使用 AWS CLI、AWS 管理主控台或 Aurora DSQL API 檢視叢集和叢集執行個體的狀態。

下表說明 Aurora DSQL 叢集的各種可能狀態，以及每個狀態的意義。

狀態	Description
正在建立	Aurora DSQL 正在嘗試建立或設定叢集的資源。當叢集處於此狀態時，任何連線嘗試都會失敗。
Active (作用中)	叢集可作業且隨時可供使用。
閒置	當叢集閒置的時間夠長，讓 Aurora DSQL 縮減執行中的資源以降低容量和成本時，就會變成閒置。當您連線到閒置叢集時，Aurora DSQL 會將叢集轉換回作用中狀態。
非作用中	當叢集上長時間沒有活動時，閒置叢集會變成非作用中。在此暫停狀態下，在保留資料時，執行中的資源會擴展至零。當您嘗試連線到非作用中叢集時，Aurora DSQL 會自動將叢集轉換回作用中狀態。還原時間取決於叢集大小。
更新中	當您變更叢集組態時，叢集會轉換為正在更新狀態。
正在刪除	當您提交刪除請求時，叢集會轉換為正在刪除狀態。

狀態	Description
Deleted (已刪除)	叢集已成功刪除。
失敗	Aurora DSQL 因為遇到錯誤而無法建立叢集。
待設定	僅適用於多區域叢集。當您在具有見證區域的首要區域中建立多區域叢集，多區域叢集會進入待設定狀態。此時會暫時停止建立叢集，直到您在次要區域中建立另一個叢集，並將兩個叢集對等互連。
待刪除	僅適用於多區域叢集。當您從多區域叢集中刪除叢集時，多區域叢集會進入待刪除狀態。在您刪除最後一個對等叢集之後，叢集會移至刪除狀態。

使用閒置和非作用中叢集

當 Aurora DSQL 在叢集上偵測到一段時間沒有連線活動時，它會將叢集轉換為閒置狀態，減少執行中的資源，以將容量和成本降至最低。如果連線活動長時間不存在，閒置叢集會自動轉換為非作用中狀態，其中在保留資料時，執行中的資源會擴展為零。

若要恢復正常操作，只需照常連線到叢集即可。當您成功連線至叢集時，Aurora Aurora DSQL 會自動將叢集轉換為作用中狀態。

> Note
> 對閒置或非作用中叢集的第一次連線嘗試會比平常慢。

需要作用中叢集狀態的操作

有些操作需要您的叢集處於作用中狀態。若要在閒置或非作用中叢集上執行這些操作，您需要連線至叢集，將叢集轉換回作用中。

備份操作

進行備份需要作用中叢集狀態。如果您的叢集是閒置或非作用中，則備份會失敗，並顯示下列錯誤：

```
"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."
}
```

若要繼續備份：

1. 使用您偏好的資料庫用戶端或 Aurora DSQL 主控台連線至叢集以將其喚醒。
2. 等待自動轉換為作用中狀態。
3. 在叢集完全運作後啟動備份。

> **Note**
>
> 在叢集閒置或非作用中之前取得的現有備份會保持有效且不受影響。叢集上的新備份嘗試將會失敗，直到叢集連線至 進行自動喚醒為止。

檢視 Aurora DSQL 叢集狀態

若要檢視叢集的狀態，請使用 AWS 管理主控台 AWS CLI或 Aurora DSQL API。

主控台

請遵循下列步驟，在 AWS 管理主控台中檢視叢集狀態：

在主控台中檢視叢集狀態

1. 在 https://console.aws.amazon.com/dsql 開啟 Aurora DSQL 主控台。
2. 在導覽窗格中，選擇 Clusters (叢集)。
3. 在儀表板中檢視每個叢集的狀態。

AWS CLI

使用下列 AWS CLI 命令來檢查單一叢集的狀態。

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

執行下列命令，表列所有叢集狀態。

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

  cluster_status=$(aws dsql get-cluster --identifier "$id" --query 'status' --output
 text)
  echo "$id    $cluster_status"
done
```

此範例輸出會顯示兩個作用中的叢集，以及一個正在刪除的叢集。

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

使用 Aurora DSQL 進行程式設計

Aurora DSQL 為您提供下列工具，以程式設計方式管理您的 Aurora DSQL 資源。

AWS Command Line Interface (AWS CLI)

您可以使用命令列 shell AWS CLI 中的 來建立和管理 資源。 AWS CLI 可讓您直接存取 APIs AWS 服務，例如 Aurora DSQL。如需 Aurora DSQL 命令的語法和範例，請參閱 AWS CLI 命令參考中的 dsql。

AWS 軟體開發套件 SDKs)

AWS 提供適用於許多熱門技術和程式設計語言SDKs。它們可讓您更輕鬆地 AWS 服務 從應用程式內以該語言或技術呼叫 。如需這些 SDK 的詳細資訊，請參閱 AWS上的應用程式開發和管理工具。

Aurora DSQL API

此 API 是 Aurora DSQL 的另一個程式設計界面。使用此 API 時，您必須正確格式化每個 HTTPS 要求，並為每個請求新增有效的數位簽章。如需詳細資訊，請參閱API 參考。

CloudFormation

AWS::DSQL::Cluster 是一項 CloudFormation 資源，可讓您建立和管理 Aurora DSQL 叢集做為基礎設施的一部分做為程式碼。 CloudFormation 可協助您在程式碼中定義整個 AWS 環境，讓您更輕鬆地以一致且可靠的方式佈建、更新和複寫基礎設施。

當您在 CloudFormation 範本中使用 AWS::DSQL::Cluster 資源時，您可以與其他雲端資源一起宣告佈建 Aurora DSQL 叢集。有助於確保資料基礎架構，能與其餘的應用程式堆疊，一起進行部署和管理。

Aurora DSQL 連接器

Aurora DSQL 提供特殊的連接器，可延伸現有的資料庫驅動程式，以啟用無縫的 IAM 身分驗證和與 AWS 服務的整合。這些連接器的用途是使用熱門的程式設計語言和架構，並同時保持與現有 PostgreSQL 工作流程的相容性。

未來版本會規劃其他連接器。如需連接器可用性的最新資訊，請參閱 Aurora DSQL 範例儲存庫。

使用 JDBC 連接器連線至 Aurora DSQL 叢集

Aurora DSQL Connector for JDBC 設計為身分驗證外掛程式，可延伸 PostgreSQL JDBC 驅動程式的功能，讓應用程式使用 IAM 憑證向 Aurora DSQL 進行身分驗證。連接器不會直接連線至資料庫，而是以 PostgreSQL JDBC 驅動器為基礎，提供無縫的 IAM 身分驗證。

Aurora DSQL Connector for JDBC 旨在與 PostgreSQL JDBC 驅動程式搭配使用，並提供與 Aurora DSQL IAM 身分驗證需求的無縫整合。

搭配 PostgreSQL JDBC 驅動程式，適用於 JDBC 的 Aurora DSQL 連接器可啟用 Aurora DSQL 的 IAM 型身分驗證。它引入了與 AWS 身分驗證服務的深度整合，例如 AWS Identity and Access Management(IAM)。

關於連接器

Aurora DSQL 是一種分散式 SQL 資料庫服務，可為 PostgreSQL 相容應用程式提供高度可用性和可擴展性。Aurora DSQL 需要 IAM 型身分驗證，其具備現有 JDBC 驅動器未原生支援的時間限制權杖。

Aurora DSQL Connector for JDBC 背後的主要概念是在處理產生 IAM 字符的 PostgreSQL JDBC 驅動程式上新增身分驗證層，讓使用者在不變更現有 JDBC 工作流程的情況下連線到 Aurora DSQL。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：使用 AWS 登入資料產生身分驗證權杖，並具有可設定的生命週期

Aurora DSQL Connector for JDBC 旨在了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

Aurora DSQL Connector for JDBC 的優點

雖然 Aurora DSQL 提供 PostgreSQL 相容介面，但現有 PostgreSQL 驅動程式尚不支援 Aurora DSQL 的 IAM 身分驗證要求。Aurora DSQL Connector for JDBC 可讓客戶繼續使用其現有的 PostgreSQL 工作流程，同時透過下列方式啟用 IAM 身分驗證：

- 自動產生字符：使用 AWS 登入資料自動產生 IAM 字符
- 無縫整合：適用於現有的 JDBC 連線模式
- AWS 登入資料支援：支援各種 AWS 登入資料提供者（預設、設定檔型等）

使用適用於 JDBC 的 Aurora DSQL 連接器搭配連線集區

Aurora DSQL Connector for JDBC 適用於連線集區程式庫，例如 HikariCP。連接器會在連線建立期間，處理 IAM 權杖的產生，讓連線集區能正常運作。

主要功能

自動產生權杖

IAM 字符會使用 AWS 登入資料自動產生。

無縫整合

使用現有的 JDBC 連線模式，無需改變工作流程。

AWS 登入資料支援

支援各種 AWS 登入資料提供者 （預設、設定檔型等）。

連線集區相容性

可無縫搭配使用連線集區程式庫，例如 HikariCP。

先決條件

在開始前，請確定您具有以下先決條件：

- 已在 Aurora DSQL 中建立叢集。
- 已安裝 Java 開發套件 (JDK)。請確定您的版本是 17 或更新版本。
- 設定適當的 IAM 權限，以允許應用程式連線到 Aurora DSQL。
- AWS 設定的登入資料 AWS CLI（透過環境變數或 IAM 角色）。

使用適用於 JDBC 的 Aurora DSQL 連接器

若要在 Java 應用程式中使用適用於 JDBC 的 Aurora DSQL Connector，請遵循下列步驟：

1. 將以下相依性新增至您的 Maven 專案：

```
<dependencies>
    <!-- Aurora DSQL Connector for JDBC -->
    <dependency>
        <groupId>software.amazon.dsql</groupId>
```

```
        <artifactId>aurora-dsql-jdbc-connector</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

對於 Gradle 專案，新增此相依性：

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

2. 使用 DSQL PostgreSQL 連接器格式建立 Aurora AWS DSQL 叢集的基本連線：

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

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

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

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

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

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

組態屬性

Aurora DSQL Connector for JDBC 支援下列連線屬性：

user

決定連線的使用者，以及使用權杖的產生方式。範例：`admin`

token-duration-secs

權杖有效性的持續時間，以秒為單位。如需更多有關權杖限制的詳細資訊，請參閱在 Amazon Aurora DSQL 中產生身分驗證權杖。

profile

用於將 ProfileCredentialsProvider 執行個體化，以使用所提供的設定檔名稱產生權杖。

region

AWS Aurora DSQL 連線的區域。這是選用的。在提供時，會覆寫從 URL 擷取的區域。

資料庫

要連線的資料庫名稱。預設值為 `postgres`。

日誌

啟用記錄功能，針對 Aurora DSQL JDBC 連接器可能遇到的任何使用問題進行故障診斷。

連接器使用 Java 的內建記錄系統 (java.util.logging)。您可以透過建立 `logging.properties` 檔案設定記錄層級：

```
# 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
```

範例

如需更完整的範例和使用案例，請參閱 Aurora DSQL Connector for JDBC 儲存庫

適用於 Python 的 Aurora DSQL 連接器

Aurora DSQL Connector for Python 整合 IAM 身分驗證，以將 Python 應用程式連線至 Amazon Aurora DSQL 叢集。在內部，它會利用 psycopg、psycopg2 和非同步用戶端程式庫。

Aurora DSQL Connector for Python 設計為身分驗證外掛程式，可延伸 psycopg、psycopg2 和非同步 cpg 用戶端程式庫的功能，讓應用程式能夠使用 IAM 憑證向 Amazon Aurora DSQL 進行身分驗證。連接器不會直接連線至資料庫，而是在基礎用戶端程式庫之上提供無縫的 IAM 身分驗證。

關於 連接器

Amazon Aurora DSQL 是一種分散式 SQL 資料庫服務，可為 PostgreSQL 相容應用程式提供高可用性和可擴展性。Aurora DSQL 需要使用現有 Python 程式庫原生不支援的時間限制字符進行 IAM 型身分驗證。

Aurora DSQL Connector for Python 背後的概念是在處理 IAM 字符產生的 psycopg、psycopg2 和非yncpg 用戶端程式庫上新增身分驗證層，讓使用者在不變更現有工作流程的情況下連線到 Aurora DSQL。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：使用 AWS 登入資料產生身分驗證權杖，並具有可設定的生命週期

Aurora DSQL Connector for Python 旨在了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

功能

- 自動 IAM 身分驗證 - 使用 AWS 登入資料自動產生 IAM 字符
- 以 psycopg、psycopg2 和 asyncpg 為基礎 - 利用 psycopg、psycopg2 和非yncpg 用戶端程式庫
- 無縫整合 - 使用現有的 psycopg、psycopg2 和非同步連線模式，而不需要工作流程變更
- Region Auto-Discovery - 從 DSQL 叢集主機名稱擷取 AWS 區域
- AWS 登入資料支援 - 支援各種 AWS 登入資料提供者（預設、設定檔型、自訂）
- 連線集區相容性 - 適用於 psycopg、psycopg2 和非yncpg 內建連線集區

快速入門指南

要求

- Python 3.10 或更新版本
- 存取 Aurora DSQL 叢集
- 設定適當的 IAM 權限，以允許應用程式連線到 Aurora DSQL。
- 已設定的 AWS 登入資料（透過 AWS CLI、環境變數或 IAM 角色）

安裝

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

分別安裝 psycopg 或 psycopg2 或非同步

Aurora DSQL Connector for Python 安裝程式不會安裝基礎程式庫。它們需要單獨安裝，例如：

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

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

```
# Install asyncpg
```

```
pip install asyncpg
```

請注意：

只需要安裝所需的程式庫。因此，如果用戶端要使用 psycopg，則只需要安裝 psycopg。如果用戶端將使用 psycopg2，則只需要安裝 psycopg2。如果用戶端將使用 asyncpg，則只需要安裝 asyncpg。

如果用戶端需要多個程式庫，則需要安裝所有必要的程式庫。

基本使用

psycopg

```
import aurora_dsql_psycopg as dsql

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

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

psycopg2

```
import aurora_dsql_psycopg2 as dsql

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

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

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

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

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

僅使用主機

psycopg

```
import aurora_dsql_psycopg as dsql

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

psycopg2

```
import aurora_dsql_psycopg2 as dsql

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

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

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

僅使用叢集 ID

psycopg

```
import aurora_dsql_psycopg as dsql
```

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

psycopg2

```
import aurora_dsql_psycopg2 as dsql

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

asyncpg

```
import asyncio
import aurora_dsql_asyncpg as dsql

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

請注意：

在「僅使用叢集 ID」案例中，會使用先前在機器上設定的區域，例如：

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

如果尚未設定區域，或指定的叢集 ID 位於不同的區域，連線將會失敗。若要讓它正常運作，請提供區域做為參數，如以下範例所示：

psycopg

```
import aurora_dsql_psycopg as dsql

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

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

psycopg2

```
import aurora_dsql_psycopg2 as dsql
```

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

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

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

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

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

連線字串

psycopg

```
    import aurora_dsql_psycopg as dsql

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

psycopg2

```
    import aurora_dsql_psycopg2 as dsql

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

asyncpg

```
    import asyncio
    import aurora_dsql_asyncpg as dsql

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

進階組態

psycopg

```
import aurora_dsql_psycopg as dsql

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

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

psycopg2

```
import aurora_dsql_psycopg2 as dsql

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

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

asyncpg

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

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

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

組態選項

選項	Type	必要	描述
host	string	是	DSQL 叢集主機名稱或叢集 ID
user	string	否	DSQL 使用者名稱。預設：admin
dbname	string	否	資料庫名稱。預設：Postgres
region	string	否	AWS 區域（若未提供，則從主機名稱自動偵測）
port	int	否	預設為 5432
custom_credentials_provider	CredentialProvider	否	自訂 AWS 登入資料提供者
profile	string	否	IAM 設定檔名稱。預設：預設。
token_duration_secs	int	否	字符過期時間，以秒為單位

也支援基礎 psycopg、psycopg2 和 asyncpg 程式庫的所有標準連線選項，但 DSQL 不支援的 asyncpg 參數 krbsrvname 和 gsslib 除外。

使用適用於 Python 的 Aurora DSQL 連接器搭配連線集區

Aurora DSQL Connector for Python 適用於 psycopg、psycopg2 和非同步內建連線集區。連接器會在連線建立期間，處理 IAM 權杖的產生，讓連線集區能正常運作。

psycopg

對於 psycopg，連接器會實作名為 DSQLConnection 的連線類別，可直接傳遞至 psycopg_pool.ConnectionPool 建構函數。對於非同步操作，也有名為 DSQLAsyncConnection 之類別的非同步版本。

```
from psycopg_pool import ConnectionPool as PsycopgPool

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

注意：連線 max_lifetime 組態

max_lifetime 參數應設定為小於 3600 秒（一小時），因為這是 Aurora DSQL 資料庫允許的最長連線持續時間。設定較低的 max_lifetime 可讓連線集區主動管理連線回收，這比處理資料庫的連線逾時錯誤更有效率。

psycopg2

對於 psycopg2，連接器提供名為 AuroraDSQLThreadedConnectionPool 的類別，繼承自 psycopg2.pool.ThreadedConnectionPool。AuroraDSQLThreadedConnectionPool 類別只會覆寫內部 _connect 方法。其餘實作由 psycopg2.pool.ThreadedConnectionPool 提供，保持不變。

```
import aurora_dsql_psycopg2 as dsql

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

```
)
```

asyncpg

對於 asyncpg，連接器提供 create_pool 函數，其會傳回 asyncpg.Pool 的執行個體。

```
import asyncio
import os

import aurora_dsql_asyncpg as dsql

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

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

身分驗證

連接器會自動處理 DSQL 身分驗證，方法是使用 DSQL 用戶端字符產生器產生字符。如果未提供 AWS 區域，則會自動從提供的主機名稱剖析。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 使用者指南。

管理員與一般使用者

- 名為 的使用者"admin"會自動使用管理員身分驗證字符
- 所有其他使用者都使用非管理員身分驗證字符
- 權杖會為每個連線動態產生

範例

如需完整的範例程式碼，請參閱以下各節中所示的範例。如需如何執行範例的說明，請參閱範例 READMDE 檔案。

psycopg

範例 README

Description	範例
使用適用於 Python 的 Aurora DSQL 連接器進行基本連線	基本連線範例
使用適用於 Python 的 Aurora DSQL 連接器進行基本非同步連線	基本非同步連線範例
搭配連線集區使用適用於 Python 的 Aurora DSQL 連接器	使用連線集區的基本連線範例
	與連線集區的並行連線範例
將適用於 Python 的 Aurora DSQL 連接器與非同步連線集區搭配使用	使用非同步連線集區的基本連線範例

psycopg2

範例 README

Description	範例
使用適用於 Python 的 Aurora DSQL 連接器進行基本連線	基本連線範例
搭配連線集區使用適用於 Python 的 Aurora DSQL 連接器	使用連線集區的基本連線範例
	與連線集區的並行連線範例

asyncpg

範例 README

Description	範例
使用適用於 Python 的 Aurora DSQL 連接器進行基本連線	基本連線範例
搭配連線集區使用適用於 Python 的 Aurora DSQL 連接器	使用連線集區的基本連線範例
	與連線集區的並行連線範例

使用 Go 連接器連線至 Aurora DSQL 叢集

Aurora DSQL Connector for Go 使用自動 IAM 身分驗證包裝 pgx。連接器會處理字符產生、SSL 組態和連線管理，讓您可以專注於應用程式邏輯。

關於連接器

Aurora DSQL 需要使用現有 Go PostgreSQL 驅動程式原生不支援的時間限制權杖進行 IAM 型身分驗證。Aurora DSQL Connector for Go 在處理 IAM 字符產生的 pgx 驅動程式上新增了身分驗證層，可讓您連線到 Aurora DSQL，而無需變更現有的 pgx 工作流程。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：連接器使用 AWS 登入資料產生身分驗證權杖，這些權杖具有可設定的生命週期

Aurora DSQL Connector for Go 旨在了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

Aurora DSQL Connector for Go 的優點

Aurora DSQL Connector for Go 可讓您繼續使用現有的 pgx 工作流程，同時透過下列方式啟用 IAM 身分驗證：

- 自動產生權杖：連接器會為每個連線自動產生 IAM 權杖
- 連線集區： 內建支援`pgxpool`，每個連線可自動產生字符
- 彈性組態：支援具有區域自動偵測功能的完整端點或叢集 IDs
- AWS 登入資料支援：支援 AWS 設定檔和自訂登入資料供應商

主要功能

自動字符管理

連接器會使用預先解析的登入資料，為每個新連線自動產生 IAM 字符。

連線集區

透過 進行連線集區`pgxpool`，並為每個連線自動產生字符。

彈性主機組態

透過自動區域偵測支援完整叢集端點和叢集 IDs。

SSL 安全性

SSL 一律會啟用完全驗證模式和直接 TLS 交涉。

先決條件

- Go 1.24 或更新版本
- AWS 已設定的登入資料
- Aurora DSQL 叢集

連接器使用AWS 適用於 Go v2 的 SDK 預設登入資料鏈，依下列順序解析登入資料：

1. 環境變數 (AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY)
2. 共用登入資料檔案 (~/.aws/credentials)
3. 共用組態檔案 (~/.aws/config)
4. Amazon EC2/ECS/Lambda 的 IAM 角色

安裝

使用 Go 模組安裝連接器：

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

快速入門

下列範例示範如何建立連線集區並執行查詢：

```
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)
}
```

組態選項

連接器支援下列組態選項：

欄位	Type	預設	Description
主機	string	（必要）	叢集端點或叢集 ID
區域	string	（自動偵測）	AWS region；如果主機是叢集 ID，則為必要
使用者	string	「管理員」	資料庫使用者
資料庫	string	「postgres」	資料庫名稱
站點	int	5432	Database port (資料庫連線埠)
設定檔	string	""	AWS 登入資料的設定檔名稱
TokenDurationSecs	int	900 (15 分鐘）	字符有效性持續時間，以秒為單位（允許上限：1 週，預設值：15 分鐘）
MaxConns	int32	0	集區連線上限 (0 = pgxpool 預設）
MinConns	int32	0	最低集區連線數 (0 = pgxpool 預設）
MaxConnLifetime	time.Duration	55 分鐘	最大連線生命週期

連線字串格式

連接器支援 PostgreSQL 和 DSQL 連線字串格式：

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

支援的查詢參數：

- `region` - AWS 區域
- `profile` - AWS 設定檔名稱
- `tokenDurationSecs` - 字符有效持續時間，以秒為單位

範例：

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

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

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

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

進階用量

主機組態

連接器支援兩種主機格式：

完整端點（自動偵測區域）：

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

叢集 ID（需要區域）：

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

集區組態調校

為您的工作負載設定連線集區：

```
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,
})
```

單一連線用量

對於簡單的指令碼或不需要連線集區時：

```
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")
```

使用 AWS 設定檔

指定登入資料的 AWS 設定檔：

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

OCC 重試

Aurora DSQL 使用樂觀並行控制 (OCC)。當兩個交易修改相同的資料時，第一個遞交獲勝，第二個收到 OCC 錯誤。

`occretry` 套件為具有指數退避和抖動的自動重試提供了協助程式。使用 進行安裝：

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

`WithRetry` 用於交易寫入：

```
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
})
```

對於 DDL 或單一陳述式，請使用 `ExecWithRetry`：

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

Important

`WithRetry` 會在內部管理 BEGIN/COMMIT/ROLLBACK。您的回呼會收到交易，並且應該只包含資料庫操作，並且可以安全地重試。

範例

如需更完整的範例和使用案例，請參閱 Aurora DSQL Connector for Go 範例。

範例	Description
example_preferred	建議：具有並行查詢的連線集區
交易	使用 BEGIN/COMMIT/ROLLBACK 處理交易
occ_retry	處理 OCC 與指數退避衝突
connection_string	使用連線字串進行組態

適用於 Node.js 的 Aurora DSQL 連接器

Aurora DSQL Connector for node-postgres 和 Aurora DSQL Connector for Postgres.js 是身分驗證外掛程式，可延伸 node-postgres 和 Postgres.js 用戶端的功能，讓應用程式使用 IAM 登入資料向 Aurora DSQL 進行身分驗證。

適用於節點後置的 Aurora DSQL 連接器

Aurora DSQL Connector for node-postgres 是建置在節點-postgres 上的 Node.js 連接器，整合 IAM 身分驗證，以將 JavaScript/TypeScript 應用程式連線至 Amazon Aurora DSQL 叢集。

Aurora DSQL Connector 設計為身分驗證外掛程式，可延伸節點後綴的用戶端和集區功能，讓應用程式使用 IAM 憑證向 Amazon Aurora DSQL 進行身分驗證。

關於 連接器

Amazon Aurora DSQL 是與 PostgreSQL 相容的雲端原生分散式資料庫。雖然它需要 IAM 身分驗證和時間限制權杖，但傳統 Node.js 資料庫驅動程式缺少此內建支援。

Aurora DSQL Connector for node-postgres 透過實作與節點-postgres 無縫搭配的身分驗證中介軟體來彌補此差距。此方法可讓開發人員維護現有的節點後置程式碼，同時透過自動化權杖管理取得 Aurora DSQL 叢集的安全 IAM 型存取。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：使用 AWS 登入資料產生身分驗證權杖，並具有可設定的生命週期

Aurora DSQL Connector for node-postgres 旨在了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

功能

- 自動 IAM 身分驗證 - 處理 DSQL 字符產生和重新整理
- 建置在 Node-postgres 上 - 利用 Node.js 的熱門 PostgreSQL 用戶端
- 無縫整合 - 適用於現有的節點後置連線模式
- 區域自動探索 - 從 DSQL 叢集主機名稱擷取 AWS 區域

- 完整 TypeScript 支援 - 提供完整類型安全
- AWS 登入資料支援 - 支援各種 AWS 登入資料提供者 （預設、設定檔型、自訂）
- 連線集區相容性 - 可與內建連線集區無縫搭配使用

範例應用程式

範例中包含一個範例應用程式，示範如何使用 Aurora DSQL Connector for node-postgres。若要執行包含的範例，請參閱範例 README。

快速入門指南

要求

- Node.js 20+
- 存取 Aurora DSQL 叢集
- 設定適當的 IAM 權限，以允許應用程式連線到 Aurora DSQL。
- 已設定的 AWS 登入資料 （透過 AWS CLI、環境變數或 IAM 角色）

安裝

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

對等相依性

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

Usage

用戶端連線

```
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();
```

集區連線

```
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()");
```

進階使用方式

```
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();
```

組態選項

選項	Type	必要	描述
host	string	是	DSQL 叢集主機名稱
username	string	是	DSQL 使用者名稱
database	string	否	資料庫名稱

選項	Type	必要	描述
`region`	`string`	否	AWS 區域（若未提供，則從主機名稱自動偵測）
`port`	`number`	否	預設為 5432
`customCredentialsProvider`	`AwsCredentialIdentity / AwsCredentialIdentityProvider`	否	自訂 AWS 登入資料提供者
`profile`	`string`	否	IAM 設定檔名稱。預設為「預設」
`tokenDurationSecs`	`number`	否	字符過期時間，以秒為單位

用戶端/集區的所有其他參數都受到支援。

身分驗證

連接器會使用 DSQL 用戶端字符產生器產生字符，以自動處理 DSQL 身分驗證。如果未提供 AWS 區域，則會自動從提供的主機名稱剖析。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 使用者指南。

管理員與一般使用者

- 名為「admin」的使用者會自動使用管理員身分驗證字符
- 所有其他使用者都使用一般身分驗證字符
- 權杖會為每個連線動態產生

適用於 Postgres.js 的 Aurora DSQL 連接器

Aurora DSQL Connector for Postgres.js 是建置在 Postgres.js 上的 Node.js 連接器，整合 IAM 身分驗證，以將 JavaScript 應用程式連線至 Amazon Aurora DSQL 叢集。

Aurora DSQL Connector for Postgres.js 設計為身分驗證外掛程式，可延伸 Postgres.js 用戶端的功能，讓應用程式使用 IAM 憑證向 Amazon Aurora DSQL 進行身分驗證。連接器不會直接連線至資料庫，而是在基礎 Postgres.js 驅動程式上提供無縫的 IAM 身分驗證。

關於 連接器

Amazon Aurora DSQL 是一種分散式 SQL 資料庫服務，可為 PostgreSQL 相容應用程式提供高可用性和可擴展性。Aurora DSQL 需要具有現有 Node.js 驅動程式原生不支援的時間限制權杖的 IAM 型身分驗證。

Aurora DSQL Connector for Postgres.js 背後的概念是在處理產生 IAM 字符的 Postgres.js 用戶端上新增身分驗證層，允許使用者連線到 Aurora DSQL，而無需變更現有的 Postgres.js 工作流程。

Aurora DSQL Connector for Postgres.js 適用於大多數版本的 Postgres.js。使用者透過直接安裝 Postgres.js 來提供自己的版本。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：使用 AWS 登入資料產生身分驗證權杖，並具有可設定的生命週期

Aurora DSQL Connector for Postgres.js 旨在了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

功能

- 自動 IAM 身分驗證 - 處理 DSQL 字符產生和重新整理
- 在 Postgres.js 上建置 - 利用 Node.js 的快速 PostgreSQL 用戶端
- 無縫整合 - 適用於現有的 Postgres.js 連線模式
- Region Auto-Discovery - 從 DSQL 叢集主機名稱擷取 AWS 區域
- 完整 TypeScript 支援 - 提供完整類型安全
- AWS 登入資料支援 - 支援各種 AWS 登入資料提供者 （預設、設定檔型、自訂）
- 連線集區相容性 - 可與 Postgres.js 的內建連線集區無縫搭配使用

快速入門指南

要求

- Node.js 20+
- 存取 Aurora DSQL 叢集

- 設定適當的 IAM 權限，以允許應用程式連線到 Aurora DSQL。
- 已設定的 AWS 登入資料（透過 AWS CLI、環境變數或 IAM 角色）

安裝

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

基本使用

```
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();
```

使用叢集 ID 而非主機

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

});
```

連線字串

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

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

進階組態

```
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
});
```

組態選項

選項	Type	必要	描述
host	string	是	DSQL 叢集主機名稱或叢集 ID
database	string?	否	資料庫名稱
username	string?	否	資料庫使用者名稱（若未提供，則使用 admin)
region	string?	否	AWS 區域（若未提供，則從主機名稱自動偵測）
customCredentialsProvider	AwsCredentialIdentityProvider?	否	自訂 AWS 登入資料提供者
tokenDurationSecs	number?	否	字符過期時間，以秒為單位

也支援所有標準 Postgres.js 選項。

身分驗證

連接器會自動處理 DSQL 身分驗證，方法是使用 DSQL 用戶端字符產生器產生字符。如果未提供 AWS 區域，則會自動從提供的主機名稱剖析。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 使用者指南。

管理員與一般使用者

- 名為「admin」的使用者會自動使用管理員身分驗證字符
- 所有其他使用者都使用一般身分驗證字符
- 權杖會為每個連線動態產生

範例用量

使用 Aurora DSQL Connector for Postgres.js 的 JavaScript 範例可在 GitHub 上取得。如需如何執行範例的說明，請參閱範例目錄。

Description	範例
具有並行查詢的連線集區，包括跨多個工作者建立、插入和讀取資料表	連線集區範例（偏好）
無連線集區的 CRUD 操作（建立資料表、插入、選取、刪除）	沒有連線集區的範例

使用 Ruby 連接器連接到 Aurora DSQL 叢集

Aurora DSQL Connector for Ruby 是建置在 pg 上的 Ruby 連接器，整合將 Ruby 應用程式連線至 Amazon Aurora DSQL 叢集的 IAM 身分驗證。

連接器會處理字符產生、SSL 組態和連線集區，讓您可以專注於應用程式邏輯。

關於連接器

Amazon Aurora DSQL 需要使用現有 Ruby PostgreSQL 驅動程式原生不支援的時間限制字符進行 IAM 身分驗證。Aurora DSQL Connector for Ruby 在處理 IAM 字符產生的 pg Gem 上新增了身分驗證層，可讓您連線到 Aurora DSQL，而無需變更現有的 pg 工作流程。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：連接器使用 AWS 登入資料產生身分驗證權杖，這些權杖具有可設定的生命週期

Aurora DSQL Connector for Ruby 了解這些要求，並在建立連線時自動產生 IAM 身分驗證字符。

功能

- 自動 IAM 身分驗證 - 處理 Aurora DSQL 字符產生和重新整理
- 以 pg 建置 - 包裝 Ruby 的熱門 PostgreSQL Gem 套件
- 無縫整合 - 適用於現有的 pg Gem 套件工作流程
- 連線集區 - 透過具有 max_lifetime 強制執行的 `connection_pool` Gem 套件內建支援
- 區域自動偵測 - 從 Aurora DSQL 叢集主機名稱擷取 AWS 區域
- AWS 憑證支援 - 支援 AWS 設定檔和自訂憑證提供者
- OCC 重試 - 選擇加入具有指數退避的樂觀並行控制重試

範例應用程式

如需完整範例，請參閱 GitHub 上的範例應用程式。

快速入門指南

要求

- Ruby 3.1 或更新版本
- 存取 Aurora DSQL 叢集
- AWS 設定的登入資料（透過 AWS CLI、環境變數或 IAM 角色）

安裝

將 新增至 Gemfile：

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

或直接安裝：

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

Usage

集區連線

```
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
```

單一連接

對於簡單的指令碼或不需要連線集區時：

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

進階用量

主機組態

連接器同時支援完整叢集端點（自動偵測區域）和叢集 IDs（需要區域）：

```
# 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 設定檔

指定登入資料的 AWS 設定檔：

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

連線字串格式

連接器支援 PostgreSQL 連線字串格式：

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

支援的查詢參數：`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 重試

Aurora DSQL 使用樂觀並行控制 (OCC)。當兩個交易修改相同的資料時，第一個遞交獲勝，第二個收到 OCC 錯誤。

OCC 重試是選擇加入。在建立集區`occ_max_retries`時設定 ，以在 上啟用具有指數退避和抖動的自動重試`pool.with`：

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

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

> **Warning**
>
> `pool.with` 不會在交易中自動包裝您的區塊。您必須呼叫`conn.transaction`自己進行寫入操作。在 OCC 衝突時，連接器會重新執行整個區塊，因此應該只包含資料庫操作，而且可以安全地重試。

若要略過個別呼叫的重試，請傳遞 `retry_occ: false`：

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

組態選項

欄位	Type	預設	Description
託管	String	（必要）	叢集端點或叢集 ID
region	String	（自動偵測）	AWS region；如果主機是叢集 ID，則為必要
user	String	「管理員」	資料庫使用者
資料庫	String	「postgres」	資料庫名稱
port	Integer	5432	Database port (資料庫連線埠)
profile	String	nil	AWS 登入資料的設定檔名稱
token_duration	Integer	900 (15 分鐘）	字符有效性持續時間，以秒為單位（允許上限：1 週，預設值：15 分鐘）
credentials_provider	Aws：：Credentials	nil	自訂登入資料提供者
max_lifetime	Integer	3300 (55 分鐘）	連線生命週期上限，以秒為單位
application_name	String	nil	application_name 的 ORM 字首
記錄器	Logger	nil	OCC 重試警告的記錄器
occ_max_retries	Integer	nil （已停用）	上的最大 OCC 重試次數`pool.with`；設定時啟用重試次數

`create_pool` 也接受`pool:`關鍵字，其中包含您直接傳遞給 的選項雜湊`ConnectionPool.new`。如果您省略 `pool:`，連接器會預設為 `{size: 5, timeout: 5}`。您提供的金鑰只會覆寫這些特定預設值。

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

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

身分驗證

連接器會使用 AWS 登入資料產生字符，自動處理 Aurora DSQL 身分驗證。如果您未提供區域，連接器會從主機名稱剖析該 AWS 區域。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 Aurora DSQL 的身分驗證和授權。

管理員與一般使用者

- 名為「admin」的使用者會自動使用管理員身分驗證字符
- 所有其他使用者都使用一般身分驗證字符
- 連接器會為每個連線動態產生字符

使用 .NET 連接器連接到 Aurora DSQL 叢集

Amazon Aurora DSQL Connector for .NET 是建置在 Npgsql 上的 .NET 連接器，整合 IAM 身分驗證以將 .NET 應用程式連線至 Amazon Aurora DSQL 叢集。

連接器會處理字符產生、SSL 組態和連線集區，讓您可以專注於應用程式邏輯。

關於連接器

Amazon Aurora DSQL 需要使用現有 .NET PostgreSQL 驅動程式原生不支援的時間限制權杖進行 IAM 身分驗證。Aurora DSQL Connector for .NET 在處理產生 IAM 字符的 Npgsql 上新增了身分驗證層，可讓您連線到 Aurora DSQL，而無需變更現有的 Npgsql 工作流程。

什麼是 Aurora DSQL 身分驗證？

在 Aurora DSQL 中，身分驗證涉及：

- IAM 身分驗證：所有連線都使用具有時間限制權杖的 IAM 型身分驗證
- 權杖產生：連接器使用 AWS 登入資料產生身分驗證權杖，這些權杖具有可設定的生命週期

Aurora DSQL Connector for .NET 了解這些需求，並在建立連線時自動產生 IAM 身分驗證字符。

功能

- 自動 IAM 身分驗證 - 處理 Aurora DSQL 字符產生和重新整理

- 建置於 Npgsql - 包裝適用於 .NET 的熱門 PostgreSQL 驅動程式
- 無縫整合 - 適用於現有的 Npgsql 工作流程
- 連線集區 - 透過 的內建支援`NpgsqlDataSource`與最長生命週期強制執行
- 區域自動偵測 - 從 Aurora DSQL 叢集主機名稱擷取 AWS 區域
- AWS 憑證支援 - 支援 AWS 設定檔和自訂憑證提供者
- OCC 重試 - 選擇加入具有指數退避的樂觀並行控制重試
- SSL 強制執行 - 一律使用 SSL 搭配 `verify-full` 模式和直接 TLS 交涉

範例應用程式

如需完整範例，請參閱 GitHub 上的範例應用程式。

快速入門指南

要求

- .NET 8.0 或更新版本
- 存取 Aurora DSQL 叢集
- AWS 設定的登入資料（透過 AWS CLI、環境變數或 IAM 角色）

安裝

將套件新增至您的專案：

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

Usage

集區連線

```
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();
});
```

單一連接

對於簡單的指令碼或當您不需要連線集區時：

```
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 重試

Aurora DSQL 使用樂觀並行控制 (OCC)。當兩個交易修改相同的資料時，第一個遞交獲勝，第二個收到 OCC 錯誤。

OCC 重試是選擇加入。在組態`OccMaxRetries`中設定 ，以啟用具有指數退避和抖動的自動重試。`WithTransactionRetryAsync` 用於交易寫入：

```
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();
});
```

對於 DDL 或單一陳述式，請使用 `ExecWithRetryAsync`：

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

Important

`WithTransactionRetryAsync` 會在內部管理 `BEGIN/COMMIT/ROLLBACK`，並在每次嘗試時開啟新的連線。您的回呼應僅包含資料庫操作，並且可以安全地重試。

組態選項

連接器也接受具有 `postgres://`和 `profile`查詢參數的 `region`和 `postgresql://`連線字串。

欄位	Type	預設	Description
`Host`	`string`	（必要）	叢集端點或 26 個字元的叢集 ID
`Region`	`string?`	（自動偵測）	AWS region；如果 `Host` 是叢集 ID，則為必要
`User`	`string`	`"admin"`	資料庫使用者
`Database`	`string`	`"postgres"`	資料庫名稱
`Port`	`int`	`5432`	Database port (資料庫連線埠)

欄位	Type	預設	Description
Profile	string?	null	AWS 登入資料的設定檔名稱
CustomCredentialsProvider	AWSCredentials?	null	自訂 AWS 登入資料供應商
TokenDurationSecs	int?	null (SDK 預設，900s)	字符有效性持續時間，以秒為單位
OccMaxRetries	int?	null（已停用）	資料來源上重試方法的預設 OCC 重試上限
OrmPrefix	string?	null	前綴為 的 ORM 字首 application_name
LoggerFactory	ILoggerFactory?	null	用於重試警告和診斷的記錄器工廠
ConfigureConnectionString	Action<NpgsqlConnectionStringBuilder>?	null	回呼以覆寫集區設定或設定其他 Npgsql 連線字串屬性。SSL 和 Enlist是安全性變異，無法覆寫。

身分驗證

連接器會使用 AWS 登入資料產生字符，自動處理 Aurora DSQL 身分驗證。如果您未提供區域，連接器會從主機名稱剖析該 AWS 區域。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 Aurora DSQL 的身分驗證和授權。

管理員與一般使用者

- 名為「admin」的使用者會自動使用管理員身分驗證字符
- 所有其他使用者都使用一般身分驗證字符
- 連接器會為每個連線動態產生字符

使用 PostgreSQL 相容用戶端存取 Aurora DSQL

Aurora DSQL 使用 PostgreSQL 線路通訊協定。您可以使用各種工具和用戶端連線至 PostgreSQL，例如 AWS CloudShell、psql、DBeaver 和 DataGrip。下表摘要說明 Aurora DSQL 如何映射常見的 PostgreSQL 連線參數：

PostgreSQL	Aurora DSQL	備註
角色 (也稱為使用者或群組)	資料庫角色	Aurora DSQL 會為您建立名為 `admin` 的角色。建立自訂資料庫角色時，您必須使用 `admin` 角色，將其與 IAM 角色關聯，以便在連線至叢集時進行身分驗證。如需詳細資訊，請參閱使用資料庫角色和 IAM 身分驗證。
主機 (也稱為主機名稱或 hostspec)	叢集端點	Aurora DSQL 單一區域叢集提供單一受管理端點，若該區域內發生無法使用的情況，系統會自動重新導向流量。
站點	不適用 – 使用預設 `5432`	這是 PostgreSQL 的預設值。
資料庫 (dbname)	使用 `postgres`	當您建立叢集時，Aurora DSQL 會自動建立此資料庫。
SSL 模式	SSL 一律在伺服器端啟用	在 Aurora DSQL 中，系統支援 `require` SSL 模式。Aurora DSQL 會拒絕未使用 SSL 的連線。
密碼	身分驗證權杖	Aurora DSQL 使用暫時性身分驗證權杖，而非長期密碼。如需詳細資訊，請參閱 在 Amazon Aurora DSQL 產生身分驗證記號。

連線時，Aurora DSQL 需要簽章的 IAM 身分驗證字符來取代傳統密碼。這些臨時字符是使用 AWS Signature 第 4 版產生，並且僅在建立連線時使用。連線後，工作階段會保持作用中狀態，直到結束或用戶端中斷連線為止。

如果您嘗試使用過期的權杖開啟新的工作階段，連線請求會失敗，而且必須產生新的權杖。如需詳細資訊，請參閱在 Amazon Aurora DSQL 產生身分驗證記號。

使用 SQL 用戶端存取 Aurora DSQL

Aurora DSQL 支援多個 PostgreSQL 相容用戶端來連線至您的叢集。下列各節說明如何使用 PostgreSQL 搭配 AWS CloudShell 或本機命令列進行連線，以及 GUI 型工具，例如 DBeaver 和 JetBrains DataGrip。每個用戶端都需要有效的身分驗證字符，如上節所述。

主題

- 使用 DBeaver 存取 Aurora DSQL
- 使用 JetBrains DataGrip 存取 Aurora DSQL
- 使用 PostgreSQL 互動式終端機 (psql) 存取 Aurora DSQL
- 使用適用於 SQLTools 的 Aurora DSQL 驅動程式
- 疑難排解

使用 DBeaver 存取 Aurora DSQL

DBeaver 是通用 SQL 用戶端，可用於管理具有 JDBC 驅動程式的任何資料庫。它在開發人員和資料庫管理員之間廣泛使用，因為其強大的資料檢視、編輯和管理功能。使用 DBeaver 的雲端連線選項，您可以將 DBeaver 原生連線至 Aurora DSQL。

DBeaver Pro

DBeaver PRO 產品提供自 25.3 版起與 Aurora DSQL 的原生整合。遵循 DBeaver 文件中的指示，連線到您的 Aurora DSQL 叢集。

DBeaver Community Edition

DBeaver Community Edition 是免費且開放原始碼的版本。如需安裝說明，請參閱下載頁面。若要從 DBeaver Community Edition 連線至 DSQL，您需要安裝適用於 DBeaver 的 Aurora DSQL 外掛程式。

適用於 DBeaver 的 Aurora DSQL 外掛程式建置在適用於 JDBC 的 Aurora DSQL 連接器之上，並啟用對 Aurora DSQL 叢集的 IAM 身分驗證。它透過 DBeaver UI 方便安裝，無需撰寫字符產生程式碼或手動提供有效的 IAM 字符，可簡化身分驗證，同時消除與傳統使用者產生密碼相關的安全風險。

功能

- IAM 身分驗證支援：使用 AWS IAM 登入資料連線至 Aurora DSQL 叢集，以實現安全、無密碼的身分驗證

- 自動驅動程式管理：無縫安裝和設定適用於 JDBC 的 Aurora DSQL 連接器
- 彈性連線選項：選擇主機型或 JDBC URL 型連線組態

用於 DBeaver 安裝的 Aurora DSQL 外掛程式

1. 開啟 DBeaver 後，前往下拉式功能表說明 → 安裝新軟體
2. 按一下新增以新增新的儲存庫
3. 輸入：
   - 名稱: `Aurora DSQL Plugin`
   - 位置： `https://awslabs.github.io/aurora-dsql-dbeaver-plugin/update-site/`
4. 檢查適用於 JDBC 的 Aurora DSQL 連接器
5. 按一下下一步，接受授權並完成安裝
6. 出現提示時重新啟動 DBeaver

建立 Aurora DSQL 連線

1. 按一下新的資料庫連線
2. 選取 Aurora DSQL
3. 在伺服器下，透過設定 為 Connect 選取下列其中一項
   - Host
     - 為下列欄位啟用使用者介面文字輸入：
       - 端點：DSQL 叢集端點
       - 使用者名稱：DSQL 使用者名稱 （例如 admin)
       - AWS 設定檔：例如預設 - 未指定特定設定檔時使用的標準設定檔
       - AWS 區域 （選用）： 必須符合 DSQL 叢集所在的區域，否則身分驗證會失敗
   - URL
     - 此格式的 JDBC URL：

       ```
       jdbc:aws-dsql:postgresql://{cluster_endpoint}/{database}?
       user=admin&profile=default&region=us-east-1
       ```

     - 注意：在此模式中，只會啟用 URL 輸入。若要將參數新增至 JDBC 連線字串，請使用以 ? 開頭的 URL 查詢參數格式做為第一個參數，並針對後續參數附加 &。
4. 按一下測試連線以驗證 Aurora DSQL 連線是否正常運作
5. 按一下完成

疑難排解

Windows 信任存放區問題

Windows 使用者從 Maven Central 下載 Aurora DSQL Connector for JDBC 驅動程式時可能會遇到問題。

原因：Windows Trust Store 可能不會包含存取 Maven Central 儲存庫所需的憑證。

解決方案：

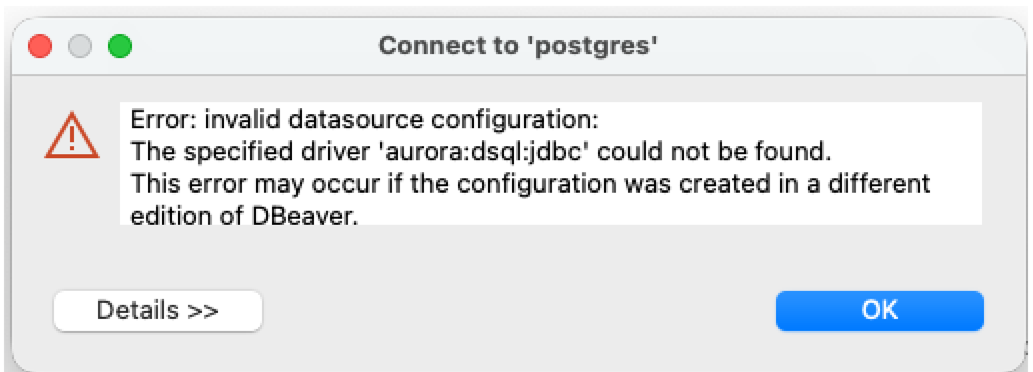
1. 將 DBeaver 執行為 "Administrator"
2. 取消核取此設定 - Windows > 偏好設定 > 連線 > "使用 Windows Trust Store"

缺少驅動程式錯誤

如果您看到缺少驅動程式圖示或連線錯誤，Aurora DSQL （社群外掛程式） 可能不會安裝在您目前的 DBeaver 版本中。請參閱以下一些錯誤範例以及如何修正錯誤：

- 使用缺少的驅動程式建立新的連線：

- 嘗試在沒有驅動程式的情況下連線：



原因：安裝多個 DBeaver 版本時，會共用連線設定，但每個應用程式都會安裝驅動程式。

解決方案：請依照上述安裝步驟重新安裝 Aurora DSQL （社群外掛程式）。

> **Important**
>
> DBeaver for PostgreSQL 資料庫 （例如 Session Manager 和 Lock Manager) 提供的管理功能不適用於 Aurora DSQL 資料庫，因為其唯一的架構。可存取時，這些畫面不會提供有關資料庫運作狀態或狀態的可靠資訊。

使用 JetBrains DataGrip 存取 Aurora DSQL

JetBrains DataGrip 是一款跨平台的 IDE，可用於處理 SQL 與各類資料庫 (包含 PostgreSQL)。DataGrip 提供功能強大的圖形介面 (GUI)，並內建智慧型 SQL 編輯器。若要下載 DataGrip，請前往 JetBrains 網站上的下載頁面。

在 JetBrains DataGrip 中設定新的 Aurora DSQL 連線

1. 選擇新增資料來源，然後選擇 PostgreSQL。
2. 在資料來源/一般索引標籤中，輸入下列資訊：
    - 主機 – 使用叢集端點。

      連接埠 – Aurora DSQL 使用 PostgreSQL 預設：`5432`

      資料庫 – Aurora DSQL 使用 PostgreSQL 預設值 `postgres`

      身分驗證 – 選擇 `User & Password` 。

      使用者名稱 – 輸入 `admin`。

      密碼 – 產生權杖並將其貼入此欄位。

      URL – 請勿修改此欄位。系統會根據其他欄位自動填入此值。
3. 密碼 – 透過產生驗證權杖填入此欄位。複製權杖產生器的輸出結果，並貼入密碼欄位。

> **Note**
>
> 必須在用戶端連線中設定 SSL 模式。Aurora DSQL 支援 `PGSSLMODE=require and PGSSLMODE=verify-full`。Aurora DSQL 在伺服器端強制啟用 SSL 通訊，並拒絕非

SSL 連線。對於 `verify-full`選項，您需要在本機安裝 SSL 憑證。如需詳細資訊，請參閱 SSL/TLS 憑證。

4. 成功連線至叢集後，即可執行 SQL 陳述式：

> **Important**
>
> DataGrip for PostgreSQL 資料庫 （例如工作階段） 提供的某些檢視不適用於 Aurora DSQL 資料庫，因為它們是唯一的架構。可存取時，這些畫面不會提供有關連線到資料庫之實際工作階段的可靠資訊。

使用 PostgreSQL 互動式終端機 (psql) 存取 Aurora DSQL

使用 AWS CloudShell 以 PostgreSQL 互動式終端機 (psql) 存取 Aurora DSQL

使用下列程序，從 使用 PostgreSQL 互動式終端機存取 Aurora DSQL AWS CloudShell。如需詳細資訊，請參閱什麼是 AWS CloudShell 。

使用 連線 AWS CloudShell

1. 登入 Aurora DSQL 主控台。
2. 選擇您要在 CloudShell 中開啟的叢集。如果您尚未建立叢集，請遵循 步驟 1：建立 Aurora DSQL 單一區域叢集 或 建立多區域叢集 中的步驟。
3. 選擇與查詢編輯器連線，然後選擇與 CloudShell 連線。
4. 選擇您要以管理員身分或使用自訂資料庫角色進行連線。
5. 選擇在 CloudShell 中啟動，然後在下列 CloudShell 對話方塊中選擇執行。

使用本機 CLI 透過 PostgreSQL 互動式終端機 (psql) 存取 Aurora DSQL

使用`psql`終端機型前端到 PostgreSQL 公用程式 ，以互動方式輸入查詢、將查詢發行到 PostgreSQL，以及檢視查詢結果。

Note

若要提升查詢回應效能，請使用 PostgreSQL 17 版客戶端。如果您在不同的環境中使用 CLI，請務必手動設定 Python 3.8+ 版和 psql 14+ 版。

請從 PostgreSQL Downloads 頁面下載適用於您作業系統的安裝程式。如需 的詳細資訊psql，請參閱 PostgreSQL 網站上的 PostgreSQL 用戶端應用程式。 PostgreSQL

如果您已 AWS CLI 安裝 ，請使用下列範例來連接至您的叢集。

```
# Aurora DSQL requires a valid IAM token as the password when connecting.
# Aurora DSQL provides tools for this and here we're using Python.
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token \
  --region us-east-1 \
  --expires-in 3600 \
  --hostname your_cluster_endpoint)

# Aurora DSQL requires SSL and will reject your connection without it.
export PGSSLMODE=require

# Connect with psql, which automatically uses the values set in PGPASSWORD and
 PGSSLMODE.
# Quiet mode suppresses unnecessary warnings and chatty responses but still outputs
 errors.
psql --quiet \
  --username admin \
  --dbname postgres \
  --host your_cluster_endpoint
```

使用適用於 SQLTools 的 Aurora DSQL 驅動程式

Aurora DSQL Driver for SQLTools 是 Amazon Aurora DSQL 的 Visual Studio 程式碼延伸模組，與 SQLTools 整合。它可讓開發人員直接從 VS Code 連線至和查詢 Aurora DSQL 資料庫。驅動程式可從 Visual Studio Marketplace 和 Open VSX Registry 安裝。Kiro、Cursor 和其他 VSCode 型 IDEs可以使用 Open VSX 登錄檔，依照本頁所述的標準安裝程序安裝驅動程式。

功能

- 自動 IAM 身分驗證

- 標準資料庫操作，例如瀏覽結構描述、資料表和執行 SQL 查詢。

安裝

1. 開啟延伸項目檢視。
2. 搜尋「Aurora DSQL Driver for SQLTools」。
3. 按一下「安裝」。

請注意：

如果 SQLTools 延伸模組尚未存在，則會自動安裝。

身分驗證

在 Aurora DSQL 中，所有連線都使用以 IAM 為基礎的身分驗證搭配限時權杖。驅動程式會自動使用 Aurora DSQL Connector for node-postgres 處理 Aurora DSQL 身分驗證。

如需 Aurora DSQL 中身分驗證的詳細資訊，請參閱 使用者指南。

建立 Aurora DSQL 連線

先決條件

- 已設定的 AWS 登入資料 （透過 AWS CLI、環境變數或 IAM 角色）

步驟

1. 按一下左側邊欄中的 SQLTools 圖示。
2. 在 SQLTools 窗格中，將滑鼠游標暫留在 CONNECTIONS 上，然後按一下新增連線圖示。
3. 在 SQLTools 設定索引標籤中，從清單中選擇 Aurora DSQL 驅動程式。
4. 填寫連線參數。
   - AWS 區域
     - 選用 - 區域將從 Aurora DSQL 叢集端點剖析。
     - 在 DSQL 叢集欄位中僅指定叢集 ID 時為必要。
   - AWS Profile (AWS 設定檔)
     - 用於產生字符。
     - 如果未指定，則使用預設設定檔。
5. 按一下「測試連線」按鈕來測試連線。

6. 按一下儲存連線。

疑難排解

SQL 用戶端的身分驗證憑證過期

已建立的工作階段將保持驗證狀態，最長 1 小時，或直到明確中斷連線或用戶端逾時。如果需要建立新的連線，則必須在連線的密碼欄位中產生並提供新的身分驗證字符。嘗試開啟新的工作階段（例如，列出新的資料表或開啟新的 SQL 主控台）會強制新的身分驗證嘗試。若連線設定中的驗證權杖已失效，則新工作階段將建立失敗，且所有先前開啟的工作階段皆會失效。使用 `expires-in`選項選擇 IAM 身分驗證字符的持續時間時，請記住這一點，該選項預設為 15 分鐘，並可設定為七天的最大值。

此外，請參閱 Aurora DSQL 文件的故障診斷一節。

Amazon Aurora DSQL 叢集連線工具

Aurora DSQL 與許多第三方資料庫驅動程式和 ORM 程式庫相容。 AWS 提供兩種類型的工具，可簡化 Aurora DSQL 的使用：

- 連接器 – 延伸資料庫驅動程式以自動處理 IAM 字符產生的身分驗證外掛程式。直接使用資料庫驅動程式時使用連接器。
- 轉接器和方言 – 特定 ORM 架構的延伸，可提供 IAM 身分驗證並改善 Aurora DSQL 相容性。使用支援的 ORM 架構時，請使用轉接器。

Aurora DSQL 轉接器和方言

下表顯示 Aurora DSQL 可用的轉接器和方言。

程式設計語言	ORM/架構	儲存庫連結
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/java/hibernate
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django

程式設計語言	ORM/架構	儲存庫連結
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm

資料庫驅動程式範例

下表顯示使用第三方資料庫驅動程式連線至 Aurora DSQL 的範例程式碼。

程式設計語言	驅動程式	範例儲存庫連結
C++	libpq	https://github.com/aws-samples/aurora-dsql-samples/tree/main/cpp/libpq
C# (.NET)	Npgsql	https://github.com/aws-samples/aurora-dsql-samples/tree/main/dotnet/npgsql
Go	pgx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/go/pgx
Java	HikariCP + pgJDBC	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/pgjdbc
JavaScript	node-postgres (AWS Lambda)	https://github.com/aws-samples/aurora-dsql-samples/tree/main/lambda
JavaScript	node-postgres	https://github.com/aws-samples/aurora-dsql-samples/

程式設計語言	驅動程式	範例儲存庫連結
		tree/main/javascript/node-postgres
JavaScript	Postgres.js	https://github.com/aws-samples/aurora-dsql-samples/tree/main/javascript/postgres-js
Python	asyncpg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/asyncpg
Python	Psycopg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg
Python	Psycopg2	https://github.com/aws-samples/aurora-dsql-samples/tree/main/python/psycopg2
Ruby	pg	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/ruby-pg
Rust	SQLx	https://github.com/aws-samples/aurora-dsql-samples/tree/main/rust/sqlx

ORM 和架構範例

下表顯示搭配 Aurora DSQL 使用第三方 ORM 程式庫和架構的範例程式碼。

程式設計語言	ORM/架構	範例儲存庫連結
Java	Hibernate	https://github.com/awslabs/aurora-dsql-orms/tree/main/

程式設計語言	ORM/架構	範例儲存庫連結
		java/hibernate/examples/pet-clinic-app
Java	Liquibase	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/liquibase
Java	Spring Boot	https://github.com/aws-samples/aurora-dsql-samples/tree/main/java/spring_boot
Python	Django	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/django/examples/pet-clinic-app
Python	SQLAlchemy	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/sqlalchemy/examples/pet-clinic-app
Python	Tortoise ORM	https://github.com/awslabs/aurora-dsql-orms/tree/main/python/tortoise-orm/example
Ruby	Rails	https://github.com/aws-samples/aurora-dsql-samples/tree/main/ruby/rails
TypeScript	Prisma	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/prisma
TypeScript	Sequelize	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/sequelize

程式設計語言	ORM/架構	範例儲存庫連結
TypeScript	TypeORM	https://github.com/aws-samples/aurora-dsql-samples/tree/main/typescript/type-orm

將資料載入 Aurora DSQL

無論您是從現有的資料庫遷移、從 Amazon Simple Storage Service 匯入檔案，還是從本機系統載入資料，Aurora DSQL 都會提供多種方法來取得您的資料。本節涵蓋所有大小資料載入的建議工具和技術，從 GB 到數百 TB。

選擇載入方法

Aurora DSQL 支援標準 PostgreSQL 資料載入命令，但大規模有效率地載入資料需要處理平行處理、連線管理和錯誤復原。下表摘要說明您的選項：

方法	最適合	考量事項
Aurora DSQL Loader - 開放原始碼公用程式，可讓您在使用 Aurora DSQL 時輕鬆平行化插入	大多數資料載入案例，特別是遷移和大量匯入	自動處理平行化、連線集區、衝突解決和 IAM 身分驗證。以原始碼或二進位檔的形式提供。
PostgreSQL **\copy** - `psql` 用戶端中繼命令	已透過 連線的簡單載入 `psql`	讀取用戶端上的檔案，並透過連線串流資料；您可以自行管理平行處理
`INSERT` 交易 - 標準 `SQL DML`	小型資料集或應用程式驅動的插入	最簡單的方法，但大量資料速度最慢

對於大多數資料載入任務，請使用 Aurora DSQL Loader。它會處理將資料載入分散式資料庫的操作複雜性，包括跨多個連線的平行執行，以及自動重試失敗的操作。

Aurora DSQL Loader

Aurora DSQL Loader 是一種開放原始碼命令列公用程式，旨在有效地將資料載入 Aurora DSQL 叢集。它可管理連線集區、平行處理跨多個工作者的資料傳輸，並自動處理衝突和重試。

主要功能

Aurora DSQL Loader 提供下列功能：

平行載入

可設定的工作者執行緒可在多個連線間同時載入資料，以提升效能。

連線集區

管理 Aurora DSQL 叢集的連線集區，自動處理 IAM 身分驗證和連線生命週期。

支援多個檔案格式

支援 CSV （逗號分隔值）、TSV （標籤分隔值） 和 Apache Parquet 單欄式格式。載入器會根據來源 URI 延伸項目自動偵測檔案格式。

自動結構描述推論

與 `--if-not-exists`旗標搭配使用時，載入器可以根據資料自動建立具有適當資料欄類型的資料表。

衝突處理

當您的目標資料表具有唯一限制條件時，請使用 `--on-conflict`選項設定載入器處理衝突的方式：略過重複項目、upsert 記錄或傳回錯誤。

容錯能力

自動重試和任務恢復功能可確保中斷的負載可以從其停止點繼續，而不是完全重新啟動。

本機和 S3 來源

使用 Amazon S3 S3 儲存貯體載入資料。 URIs

先決條件

使用 Aurora DSQL Loader 之前，請確定您有下列項目：

- 具有有效端點的作用中 Aurora DSQL 叢集。

- AWS 透過 AWS CLI (aws configure)、 AWS 單一登入 (aws sso login) 或 IAM 角色設定的憑證。
- IAM 許可： `dsql:DbConnectAdmin`或 Aurora DSQL 叢集`dsql:DbConnect`上的 。
- 對於 S3 來源，從來源儲存貯體讀取的適當許可。

安裝

從 GitHub 版本頁面下載最新版本。預先建置的二進位檔適用於常見的平台。如需從來源建置的指示，請參閱 Aurora DSQL Loader 儲存庫。

使用範例

下列範例示範 Aurora DSQL Loader 的常見使用案例。

Example載入本機 CSV 檔案

此範例會將 CSV 檔案從本機檔案系統載入現有資料表：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table
```

Example從 Amazon S3 載入資料

此範例會從 Amazon S3 儲存貯體載入 Parquet 檔案：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/data.parquet \
  --table my_table
```

Example自動建立資料表

此範例會根據資料結構描述自動建立新的資料表：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
```

```
  --if-not-exists
```

Example載入前驗證

此範例會驗證您的組態，而不會實際載入資料：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri data.csv \
  --table my_table \
  --dry-run
```

Example繼續中斷的載入

如果載入操作中斷，您可以使用先前執行的任務 ID 來繼續執行：

```
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**
> 恢復時，載入器會略過大部分已完成的工作，但可能會重試一些記錄。如果您的目標資料表有唯一的限制條件，請使用 `--on-conflict`選項來處理重複項目，例如`DO NOTHING`略過或`DO UPDATE` upsert。

命令列選項

Aurora DSQL Loader 支援下列命令列選項：

`--endpoint`

（必要）Aurora DSQL 叢集端點。範例：`cluster-id.dsql.region.on.aws`

`--source-uri`

（必要）資料檔案的路徑。可以是本機檔案路徑或 S3 URI（例如 `s3://bucket-name/file.parquet`）。

`--table`

（必要）Aurora DSQL 資料庫中目標資料表的名稱。

`--if-not-exists`

（選用）如果目標資料表不存在，則自動建立目標資料表。載入器會從資料推斷結構描述。

`--dry-run`

（選用）驗證組態和資料，而不實際將其載入資料庫。

`--resume-job-id`

（選用）使用指定的任務 ID 恢復先前中斷的載入操作。

`--manifest-dir`

（選用）用於儲存任務狀態和資訊清單的目錄，用於恢復任務。

`--on-conflict`

（選用）指定插入違反目標資料表唯一限制的資料列時如何處理衝突。有效值為 `error`（傳回錯誤）、 `do-nothing`（略過重複的資料列）或 `do-update`（使用新值更新現有資料列）。

如需選項和其他組態參數的完整清單，請執行：

```
aurora-dsql-loader load --help
```

最佳實務

- 使用試轉進行驗證 – 在將資料載入生產資料表`--dry-run`之前，請務必使用 測試您的負載組態。
- 定義恢復的唯一限制條件 – 如果您需要繼續中斷的載入，請定義目標資料表的唯一限制條件，並使用 `--on-conflict`選項來處理已載入的記錄。
- 對大型資料集使用 Parquet – 相較於 CSV 或 TSV，Parquet 的單欄格式通常會為大型資料集提供更好的壓縮和更快的載入。
- 保留資訊清單目錄 – 保留載入任務的資訊清單目錄，直到您確認載入成功完成，以便在需要時恢復。
- 盡可能預先建立資料表 – 在載入資料之前，使用明確的資料欄資料類型和主索引鍵定義目標資料表。預先建立的結構描述可讓您控制類型精確度和索引，相較於自動推斷結構描述，這通常會產生更好的查詢效能。

疑難排解

身分驗證錯誤

確認您的 AWS 登入資料設定正確，且您的 IAM 身分在目標叢集上具有必要的 `dsql:DbConnect`或 `dsql:DbConnectAdmin`許可。

S3 存取錯誤

確保您的 IAM 身分具有來源儲存貯體和物件的適當 S3 讀取許可。

結構描述推論錯誤

使用 時`--if-not-exists`，請確定您的資料檔案具有一致的資料欄類型。欄中的混合類型可能會導致結構描述推論失敗。

恢復時重複的金鑰錯誤

如果您在繼續載入時遇到重複的金鑰錯誤，請將唯一限制新增至目標資料表，以便載入器可以`ON CONFLICT DO NOTHING`用來略過已載入的記錄。

如需其他故障診斷資訊，請參閱 Aurora DSQL Loader GitHub 儲存庫。

遷移途徑

下列各節說明如何將資料從常見的來源系統遷移至 Aurora DSQL。

從 PostgreSQL 遷移

若要將資料從現有的 PostgreSQL 資料庫遷移至 Aurora DSQL：

1. 將您的資料從 PostgreSQL 匯出到 CSV 或 Parquet 格式。您可以使用 PostgreSQL `COPY`命令匯出每個資料表：

```
COPY my_table TO '/path/to/my_table.csv' WITH (FORMAT csv, HEADER true);
```

2. 在 Aurora DSQL 中建立目標資料表。您可以手動建立結構描述，或使用載入器的`--if-not-exists`旗標從資料推斷結構描述。
3. 使用 Aurora DSQL Loader 載入匯出的資料：

```
aurora-dsql-loader load \
```

```
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

> **Tip**
>
> 對於大型遷移，請考慮匯出為 Parquet 格式，以獲得更好的壓縮和更快的載入速度。DuckDB 等工具可以有效地將 CSV 檔案轉換為 Parquet。

從 MySQL 遷移

若要將資料從 MySQL 遷移至 Aurora DSQL：

1. 使用 `SELECT INTO OUTFILE`或 等工具mysqldump搭配 `--tab`選項，將您的資料從 MySQL 匯出為 CSV 格式：

```
SELECT * FROM my_table
INTO OUTFILE '/path/to/my_table.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

2. 使用適當的 PostgreSQL 相容資料類型，在 Aurora DSQL 中建立目標資料表。
3. 使用 Aurora DSQL Loader 載入匯出的資料：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri /path/to/my_table.csv \
  --table my_table
```

> **Note**
>
> MySQL 和 PostgreSQL 具有不同的資料類型系統。在 Aurora DSQL 中建立資料表時，請檢閱您的結構描述並視需要調整資料類型。

從 Amazon S3 載入

如果您的資料已在 Amazon S3 中，您可以直接載入資料，而無需下載到本機系統。Aurora DSQL Loader 原生支援 S3 URIs：

```
aurora-dsql-loader load \
  --endpoint cluster-id.dsql.region.on.aws \
  --source-uri s3://my-bucket/path/to/data.parquet \
  --table my_table
```

確保您的 IAM 身分具有來源物件的`s3:GetObject`許可。

使用 PostgreSQL \copy

如果您已透過處理 IAM 身分驗證的`psql`工作階段連線至 Aurora DSQL，您可以使用用戶端中`\copy`繼命令從本機檔案系統載入資料。與伺服器端`COPY`陳述式不同， 會`\copy`讀取用戶端機器上的檔案，並透過現有連線串流資料，因此不需要伺服器端檔案存取。此方法適用於簡單的單執行緒負載。

Example使用 \copy 載入 CSV 檔案

```
\copy my_table FROM '/path/to/data.csv' WITH (FORMAT csv, HEADER true);
```

`\copy` 直接使用 時，您必須負責：

- 在載入多個檔案或大型資料集時管理平行處理
- 處理連線管理和身分驗證字符重新整理
- 實作失敗操作的重試邏輯

INSERT 交易的最佳實務

使用`INSERT`陳述式將資料載入 Aurora DSQL 時，請遵循下列實務來改善輸送量和可靠性：

- 將資料列批次成多列 INSERTs將多列分組成單一`INSERT`陳述式，以減少往返。例如， `INSERT INTO my_table VALUES (1, 'a'), (2, 'b'), (3, 'c')`比三個單獨的陳述式更有效率。
- 使用參數化查詢 – 使用預備陳述式搭配參數繫結，而非字串串連。這可避免 SQL 注入風險，並允許資料庫重複使用查詢計劃。

- 保持交易較小 – Aurora DSQL 使用樂觀並行控制，因此接觸許多資料列的大型交易更可能會遇到衝突。以數百列而非數千列的交易為目標。
- 實作重試邏輯 – 分散式系統中預期會發生暫時性錯誤，例如樂觀並行控制 (OCC) 衝突。實作指數退避，並重試失敗的交易。
- 跨連線平行化 – 開啟多個連線，並在其中分配插入。每個連線可以同時處理不同的資料子集。

對於大多數使用案例，Aurora DSQL Loader 提供更簡單且更強大的資料載入方法。

其他資源

- GitHub 上的 Aurora DSQL Loader – 原始程式碼、文件和問題追蹤
- 在 Amazon Aurora DSQL 產生身分驗證記號 – 了解 Aurora DSQL 的 IAM 身分驗證字符
- 使用 PostgreSQL 相容用戶端存取 Aurora DSQL – 使用各種用戶端和工具連線至 Aurora DSQL

Aurora DSQL 的生成式 AI

本節提供如何使用生成式 AI 工具搭配 Aurora DSQL 的詳細說明

AWS Labs Aurora DSQL MCP 伺服器

適用於 Aurora DSQL 的 AWS 實驗室模型內容通訊協定 (MCP) 伺服器

功能

- 將人類可讀的問題和命令轉換為結構化 Postgres 相容 SQL 查詢，並根據設定的 Aurora DSQL 資料庫執行它們。
- 根據預設為唯讀，使用 啟用的交易 `--allow-writes`
- 請求之間的連線重複使用以提升效能
- 內建存取 Aurora DSQL 文件、搜尋和最佳實務建議

可用的工具

資料庫操作

- readonly_query - 針對 DSQL 叢集執行唯讀 SQL 查詢
- 交易 - 在交易中執行寫入操作（需要 `--allow-writes`)
- get_schema - 擷取資料表結構描述資訊

文件和建議

- dsql_search_documentation - 搜尋 Aurora DSQL 文件
  - 參數： `search_phrase`（必要）、 `limit`（選用）
- dsql_read_documentation - 讀取特定 DSQL 文件頁面
  - 參數： `url` （必要）、 `start_index`（選用）、 `max_length`（選用）
- dsql_recommend - 取得 DSQL 最佳實務的建議
  - 參數：`url`（必要）

先決條件

1. 具有 Aurora DSQL 叢集的 AWS 帳戶
2. 此 MCP 伺服器只能在與 LLM 用戶端相同的主機上本機執行。
3. 設定可存取 AWS 服務的 AWS 登入資料
   - 您需要具有包含這些許可之角色的 AWS 帳戶：
     - `dsql:DbConnectAdmin` - 以管理員使用者身分連線至 DSQL 叢集
     - `dsql:DbConnect` - 使用自訂資料庫角色連線至 DSQL 叢集 （只有在使用非管理員使用者時才需要）
   - 使用 `aws configure`或 環境變數設定 AWS 登入資料

安裝

對於大多數工具，遵循預設安裝指示更新組態應該已足夠。

已針對 Claude Code 和 Codex 概述個別指示。

預設安裝：更新相關的 MCP Config 檔案

使用 **uv**

1. uv 從 Astral 或 GitHub README 安裝
2. 使用 安裝 Python uv `python install 3.10`

在 MCP 用戶端組態中設定 MCP 伺服器 （尋找 MCP Config 檔案)

```
{
  "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 安裝

對於 Windows 使用者，MCP 伺服器組態格式略有不同：

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

尋找 MCP 用戶端組態檔案

對於一些最常見的代理程式開發工具，您可以在下列檔案路徑找到 MCP 用戶端組態：

- Kiro：
  - 使用者組態： `~/.kiro/settings/mcp.json`
  - 工作區組態： `/path/to/workspace/.kiro/settings/mcp.json`
- Claude Code：請參閱 Claude Code Installation 以取得詳細的設定說明
  - 使用者組態： `~/.claude.json` 中的 `"mcpServers"`
  - 專案組態： `/path/to/project/.mcp.json`
  - 本機組態： `~/.claude.json` 中的 `"projects" -> "path/to/project" -> "mcpServers"`
- 游標：
  - 全域：`~/.cursor/mcp.json`
  - 專案： `/path/to/project/.cursor/mcp.json`
- Codex： `~/.codex/config.toml`
  - 每個 MCP 伺服器都會在組態檔案中設定`[mcp_servers.<server-name>]`資料表。請參閱自訂 Codex 安裝說明
- 扭曲：
  - 檔案編輯： `~/.warp/mcp_settings.json`
  - 應用程式編輯器：`Settings > AI > Manage MCP Servers`並貼上 json
- Amazon Q 開發人員 CLI： `~/.aws/amazonq/mcp.json`
- Cline：通常是巢狀 VS 程式碼路徑 - `~/.vscode-server/path/to/cline_mcp_settings.json`

Claude 程式碼

先決條件

重要：MCP 伺服器管理只能透過 Claude Code CLI 終端機體驗使用，而非 VS Code 原生面板模式。

遵循 Claude 的原生安裝建議程序，先安裝 Claude Code CLI。

選擇正確的範圍

Claude Code 提供 3 種不同的範圍：本機 （預設）、專案和使用者和詳細資訊，以及根據憑證敏感度選擇範圍，且需要共用的詳細資訊。如需詳細資訊，請參閱 MCP 安裝範圍上的 Claude Code 文件。

1. 本機範圍伺服器代表預設組態層級，並存放在專案路徑`~/.claude.json`下的 中。它們都是私有的，並且只能在目前的專案目錄中存取。這是建立 MCP 伺服器`scope`時的預設值。
2. 專案範圍伺服器可啟用團隊協同合作，同時仍可在專案目錄中存取。專案範圍伺服器會在專案的根目錄中新增`.mcp.json`檔案。此檔案旨在檢查版本控制，確保所有團隊成員都能存取相同的 MCP 工具和服務。當您新增專案範圍的伺服器時，Claude Code 會自動以適當的組態結構建立或更新此檔案。
3. 使用者範圍伺服器存放在 中`~/.claude.json`，並提供跨專案存取性，讓它們可在機器上的所有專案中使用，同時保持使用者帳戶的私有性。

使用 Claude CLI （建議）

使用互動式 `claude` CLI 工作階段可改善故障診斷體驗，因此這是建議的路徑。

```
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]"
```

故障診斷：在不同 AWS 帳戶上使用 Claude Code 搭配 Bedrock

如果您已使用與連線至 dsql 叢集所需的設定檔不同的 Bedrock AWS 帳戶或設定檔來設定 Claude Code，則需要提供額外的環境引數：

```
  --env AWS_PROFILE="[dsql profile, eg. default]" \
  --env AWS_REGION="[dsql cluster region, eg. us-east-1]" \
```

組態檔案中的直接修改

Claude Code 需要英數字元命名，因此我們建議您為伺服器命名：`aurora-dsql-mcp-server`。

Local-Scope

~/.claude.json 在專案特定mcpServers欄位中更新：

```
{
  "projects": {
    "/path/to/project": {
      "mcpServers": {}
    }
  }
}
```

專案範圍

/path/to/project/root/.mcp.json 在 mcpServers 欄位中更新：

```
{
  "mcpServers": {}
}
```

使用者範圍

~/.claude.json 在專案特定mcpServers欄位中更新：

```
{
  "mcpServers": {}
}
```

Codex

選項 1：Codex CLI

如果您已安裝 Codex CLI，您可以使用 codex mcp 命令來設定 MCP 伺服器。

```
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]"
```

選項 2：config.toml

如需更精細地控制 MCP 伺服器選項，您可以手動編輯`~/.codex/config.toml`組態檔案。每個 MCP 伺服器都會在組態檔案中設定`[mcp_servers.<server-name>]`資料表。

```
[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"
```

驗證安裝

對於 Amazon Q Developer CLI、Kiro CLI、Claude CLI/TUI 或 Codex CLI/TUI，請執行 `/mcp`以查看 MCP 伺服器的狀態。

對於 Kiro IDE，您也可以導覽至 Kiro 面板的`MCP SERVERS`索引標籤，其中顯示所有設定的 MCP 伺服器及其連線狀態指示燈。

伺服器組態選項

`--allow-writes`

根據預設，dsql mcp 伺服器不允許寫入操作（「唯讀模式」）。任何交易工具的調用都會在此模式下失敗。若要使用交易工具，請傳遞 `--allow-writes` 參數以允許寫入。

我們建議您在連線至 DSQL 時使用最低權限存取。例如，使用者應盡可能使用唯讀的角色。唯讀模式具有最佳用戶端強制執行來拒絕變動。

`--cluster_endpoint`

這是指定要連線之叢集的強制性參數。這應該是您叢集的完整端點，例如，`01abc2ldefg3hijklmnopqurstu.dsql.us-east-1.on.aws`

--database_user

這是強制性參數，用於指定要連線的使用者。例如 admin 或 my_user。請注意，您使用的 AWS 登入資料必須具有以該使用者身分登入的許可。如需在 DSQL 中設定和使用資料庫角色的詳細資訊，請參閱搭配 IAM 角色使用資料庫角色。

--profile

您可以指定要用於登入資料的 aws 設定檔。請注意，Docker 安裝不支援此功能。

也支援在 MCP 組態中使用 AWS_PROFILE環境變數：

```
"env": {
  "AWS_PROFILE": "your-aws-profile"
}
```

如果兩者皆未提供，則 MCP 伺服器會預設為使用 AWS 組態檔案中的「預設」設定檔。

--region

這是強制性參數，用於指定 DSQL 資料庫的區域。

--knowledge-server

為 DSQL 知識工具指定遠端 MCP 伺服器端點的選用參數（文件搜尋、讀取和建議）。預設會預先設定。

範例：

```
--knowledge-server https://custom-knowledge-server.example.com
```

注意：為了安全起見，請僅使用信任的知識伺服器端點。伺服器應該是 HTTPS 端點。

--knowledge-timeout

選用參數，以秒為單位指定對知識伺服器的請求逾時。

預設：30.0

範例：

```
--knowledge-timeout 60.0
```

如果您在存取慢速網路上的文件時遇到逾時，請提高此值。

Aurora DSQL 轉向：技能和能力

本節說明如何使用技能和能力設定 Aurora DSQL 的 AI 轉向。這些 Markdown 型組態檔案提供 AI 助理在產生程式碼時自動套用的內容和指引，以改善代理程式開發的品質。

概觀

技能和能力是模組化功能，可擴展 Aurora DSQL 的 AI 助理功能。它們封裝了 AI 助理在使用 Aurora DSQL 資料庫時自動使用的指示、中繼資料和資源。

為什麼要使用技能和能力

技能和能力為 Aurora DSQL 開發提供了幾個關鍵優勢：

- 專業 AI 助理 - 為 Aurora DSQL 提供特定領域的專業知識，包括最佳實務、Postgres 相容的 SQL 模式和分散式資料庫最佳化。
- 減少重複 - 建立一次，自動使用 。無需在多個對話之間重複提供相同的指引。
- 內容效率 - 技能隨需載入，而不是預先取用內容。AI 會視需要分階段載入資訊。
- 持續學習 - 隨著 Aurora DSQL 功能的演進，AI 助理會在技能更新時自動存取更新模式。

建議的設定路徑

選擇符合您開發環境的設定路徑：

- the section called “技能 CLI” (Agent-Agnostic)
- the section called “Kiro Power”
- the section called “Claude 技能”
- the section called “Gemini 技能”
- the section called “Codex 技能”

DSQL 技能也可以透過將技能資料夾複製到工具的 `rules`或 `skills`目錄，與其他 AI 編碼代理器搭配使用。

技能 CLI

您可以使用 Skills CLI 安裝 DSQL 技能。此與代理程式無關的設定方法適用於大多數 AI 編碼助理，並可讓您一次將技能安裝到多個代理程式。

設定

執行下列命令來安裝 Aurora DSQL 技能：

```
npx skills add awslabs/mcp --skill dsql
```

CLI 將引導您完成：

- 選取代理程式 - 選擇要安裝的代理程式 (Kiro、Claude Code、Cursor、Copilot、Gemini、Codex、Roo、Cline、OpenCode、Windsurf 等）
- 安裝範圍 - 選擇：
  - 專案：在目前的目錄中安裝（與您的專案一起遞交）
  - 全域：在主目錄中安裝（適用於所有專案）
- 安裝方法 - 選擇：
  - Symlink（建議）：單一事實來源，輕鬆更新
  - 複製到所有客服人員：每個客服人員的獨立副本

管理技能

隨時使用 檢查和更新技能：

```
npx skills check
npx skills update
```

Kiro Power

Kiro Powers 是統一套件，將 MCP 工具與架構專業知識和轉向指示綁定在一起。每個功能都包含一個進入點文件，說明可用的 MCP 工具和啟用觸發條件、MCP 伺服器組態，以及隨需載入的其他工作流程特定指引。

電源會根據使用者內容動態啟用。電源不會預先載入所有工具，而是維持接近零的基準用量，直到相關關鍵字觸發啟用為止。

設定

若要設定 Aurora DSQL 的 Kiro 電源：

1. 直接從 Kiro Powers 登錄檔安裝
2. 重新導向至 IDE 中的 Power 後，請執行下列其中一項操作：
   - 選取嘗試電源按鈕。建議希望 AI 引導 MCP 伺服器設定或互動入門體驗的使用者使用 Aurora DSQL 建立新的叢集。
   - 開啟新的 Kiro 聊天，並詢問與 Aurora DSQL 相關的任何內容。選擇性地使用您現有的叢集詳細資訊更新 MCP Config，以測試 MCP 伺服器連線，使其能夠立即使用。如果 Kiro 代理程式將電源識別為對完成使用者任務有價值，則其將自動啟用電源。

Claude 技能

Claude 技能是擴展 Claude 功能的模組化功能。Claude 會在相關時自動使用的每個技能套件指示、中繼資料和選用資源。技能是以檔案系統為基礎，並隨需載入，以將內容用量降至最低。

使用技能 CLI 進行簡單設定

您可以使用 安裝技能到 Claude Codethe section called “技能 CLI”。若要將 Claude Code 指定為要安裝的代理程式，請使用：

```
npx skills add awslabs/mcp --skill dsql --agent claude-code
```

替代方案：使用 Git 複製直接設定

替代設定會取得 dsql-skill 目錄的稀疏複製，並將此複製符號連結至 `~/.claude/skills/` 資料夾。這可讓每當需要更新技能時提取技能的變更。

先決條件

- 已安裝 Git

設定步驟

1。建立基礎儲存庫目錄

```
mkdir -p .dsql_skill_repos
```

2. 稀疏複製來自 MCP 儲存庫的技能

僅複製 `dsql-skill` 資料夾（沒有其他檔案）：

```
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. 將技能 Symlink 至技能目錄

新增技能目錄（預設：全域/使用者範圍）：

```
mkdir -p ~/.claude/skills
```

> **Note**
>
> 如果您想要將此設為專案範圍的技能，請改用專案根`.claude/skills/`目錄。

新增符號連結：

```
ln -s "$(pwd)/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/dsql-skill"
 ~/.claude/skills/dsql-skill
```

4. 驗證設定

```
# Should show SKILL.md and other skill files
ls -la ~/.claude/skills/dsql-skill/
```

5. 驗證技能使用

設定技能後，您應該有新的技能命令：`/dsql`。新增技能後，您可能需要重新啟動 Claude Code，才能偵測到它。您可以視需要從 Claude Code CLI 或面板使用此命令。

更新技能

若要從儲存庫提取最新的變更：

```
cd .dsql_skill_repos/mcp
git pull
```

目錄結構

設定全域技能之後，您應該會看到這些目錄：

```
.dsql_skill_repos/
### mcp/                              # Sparse git checkout
    ### src/
        ### aurora-dsql-mcp-server/
            ### skills/
                ### dsql-skill/
                    ### SKILL.md
                    ### ...

~/.claude/
### skills/
    ### dsql-skill -> /path/to/.dsql_skill_repos/mcp/src/aurora-dsql-mcp-server/skills/
dsql-skill
```

> **Note**
>
> `.gitignore` 如果您不想追蹤，請將 `.dsql_skill_repos/`新增至您的 。稀疏結帳只會保留技能資料夾，將磁碟用量降至最低。

Gemini 技能

若要直接在 Gemini 中新增 Aurora DSQL 技能，請決定範圍： `workspace`（包含專案） 或 `user`（預設、全域），並使用技能安裝程式。

設定

```
gemini skills install https://github.com/awslabs/mcp.git --path src/aurora-dsql-mcp-
server/skills/dsql-skill --scope $SCOPE
```

將 取代`$SCOPE`為 `workspace`或 `user`。

然後，您可以將`/dsql`技能命令與 Gemini 搭配使用，Gemini 會自動偵測何時應使用技能。

Codex 技能

使用技能從 Codex CLI 或 TUI 使用`$skill-installer`技能安裝程式。

設定

```
$skill-installer install dsql skill: https://github.com/awslabs/mcp/tree/main/src/
aurora-dsql-mcp-server/skills/dsql-skill
```

重新啟動 Codex 以取得技能。然後，您可以使用 來啟用技能`$dsql`。

Aurora DSQL 查詢編輯器入門

使用 Aurora DSQL 查詢編輯器，您可以安全地連線至 Aurora DSQL 叢集，並直接從 AWS 管理主控台執行 SQL 查詢，而無需安裝或設定外部用戶端。它提供直覺式工作區，其中包含內建語法反白、自動完成和智慧型程式碼協助。您可以在單一界面中快速探索結構描述物件、開發和執行 SQL 查詢，以及檢視結果。

本主題會逐步引導您連線至叢集、執行查詢、檢視結果，以及探索執行計畫等進階功能。

> **Note**
>
> 查詢編輯器可在支援 Aurora DSQL 的所有區域中使用。如需區域可用性的詳細資訊，請參閱AWS 區域服務。

先決條件

開始之前，請確定符合下列要求：

- 您至少有一個可用的 Aurora DSQL 叢集。如需建立叢集的詳細資訊，請參閱 步驟 1：建立 Aurora DSQL 單一區域叢集。
- 您的叢集端點可公開存取。查詢編輯器不支援由資源型政策封鎖公開存取的叢集，或透過 VPC 端點管理的叢集。如需存取限制的詳細資訊，請參閱 使用 Aurora DSQL 中的資源型政策封鎖公開存取和使用 管理和連線至 Amazon Aurora DSQL 叢集 AWS PrivateLink。
- 您的 IAM 使用者或角色具有存取和連線至叢集所需的許可。如需許可的詳細資訊，請參閱 使用資料庫角色和 IAM 身分驗證。

使用查詢編輯器

開啟查詢編輯器

開啟查詢編輯器

1. 開啟 Aurora DSQL 主控台。
2. 在導覽窗格中，選擇 Query Editor (查詢編輯器)。

或者，從叢集頁面，選取要查詢的叢集，然後選擇與查詢編輯器連線以直接啟動編輯器。

> **Note**
>
> 工作和連線狀態不會儲存。如果您離開 Aurora DSQL 主控台、關閉瀏覽器索引標籤或登出，您的連線、查詢文字和結果都會遺失。

連接至叢集

連線至叢集

1. 如果沒有叢集連線，編輯器會顯示未連線任何叢集。在 Cluster Explorer 窗格中選擇連線或選取 +（新增）以連線至現有的叢集。
2. （選用）使用不同的角色連線到多個叢集或相同的叢集。

探索叢集物件

Cluster Explorer 會顯示所有可用的叢集連線，並可讓您瀏覽資料庫、結構描述、資料表和檢視等物件。它還提供常見的動作，例如重新整理、建立資料表和其他內容特定選項。

執行查詢

若要執行查詢

1. 在查詢編輯器索引標籤窗格中，輸入您的 SQL 陳述式。例如：

```
SELECT * FROM public.orders LIMIT 10;
```

2. 驗證顯示在查詢索引標籤右上角的作用中叢集內容。這表示與目前查詢索引標籤相關聯的叢集連線。
3. （選用） 使用連線下拉式清單來檢閱所有可用的連線，或切換到不同的叢集。變更執行該標籤中查詢的連線更新。
4. 選擇執行以執行查詢。

> **Note**
>
> 每個查詢最多可傳回結果窗格中的 10，000 個資料列。對於較大的資料集，請使用篩選條件或限制來精簡查詢。

檢閱結果和執行計畫

查詢執行後，請在編輯器底部的結果面板中檢閱輸出。根據預設，每個查詢執行會顯示結果 （資料表） 索引標籤，顯示表格式查詢輸出。

若要取得查詢執行計畫，請執行 `EXPLAIN ANALYZE`或 `EXPLAIN ANALYZE VERBOSE` 以取得查詢效能的其他洞見。如需執行計劃的詳細資訊，請參閱 [讀取 Aurora DSQL EXPLAIN 計劃](#)。

> **Tip**
>
> `EXPLAIN ANALYZE VERBOSE` 命令顯示 DPU 用量預估，包括運算、讀取、寫入和總 DPU 值，可讓您立即查看個別 SQL 陳述式耗用的資源。

查詢編輯器：搭配 Aurora DSQL 使用 JupyterLab

本指南提供step-by-step指示，說明如何使用 JupyterLab 搭配 Python 來連接和查詢 Amazon Aurora DSQL。JupyterLab 是一種熱門的互動式運算環境，將程式碼、文字和視覺化結合在單一文件中。它廣泛用於資料科學和研究應用程式。

以下說明將涵蓋 JupyterLab 的本機安裝以及使用 Amazon SageMaker AI 的 Aurora DSQL 使用基本知識，Amazon SageMaker AI 是一種全受管的機器學習服務，可為資料工作流程提供具有 UI 的託管環境。

開始使用

要求

- Aurora DSQL 叢集
- 已設定的 AWS 登入資料（僅限本機安裝）
- Python 3.9 版或更新版本（僅限本機安裝）

使用本機 JupyterLab

若要開始使用 JupyterLab，使用者必須先使用 Python 的 pip 安裝應用程式：

```
pip install jupyterlab
```

然後，執行 即可開啟 JupyterLab**jupyter lab**。這將在 localhost：8888 開啟 JupyterLab 應用程式，可在瀏覽器中存取。在繼續之前，請確定已在本機環境中設定 AWS 登入資料。

使用 Amazon SageMaker AI

在 AWS 主控台中，繼續前往 Amazon SageMaker AI 主控台頁面，然後前往應用程式和 IDEs 下的筆記本區段。您可以在該處選取建立筆記本執行個體，以開始建立 SageMaker 環境。選取執行個體類型和平台，再按一下建立筆記本執行個體。

如需設定和執行個體選項的詳細資訊，請參閱 Amazon SageMaker AI 設定文件。

> **Note**
>
> 警告：使用 Amazon SageMaker AI 可能會導致您的 AWS 帳戶產生費用。

一旦 SageMaker 執行個體變成作用中，您就可以使用 Open JupyterLab 從筆記本執行個體區段開啟它。在筆記本中開始使用 Aurora DSQL 之前，您必須在 SageMaker 執行個體的 IAM 角色中提供 DSQL 叢集的存取權。最簡單的方法是遵循筆記本執行個體頁面中 IAM 角色的連結。您可以在該處編輯連接到 SageMaker IAM 角色的政策。如需設定 IAM 政策以允許存取 Aurora DSQL 的詳細資訊，請參閱身分驗證和授權。

使用 JupyterLab 連線至 Aurora DSQL

設定 JupyterLab 執行個體後，連線至 Aurora DSQL 的步驟會在本機和 SageMaker AI 中相同。建立空的 Python 3 筆記本，您可以在其中使用 Python 程式碼新增儲存格。

在 Python 儲存格中，從官方信任存放區下載 Amazon 根憑證：

```
import urllib.request
urllib.request.urlretrieve('https://www.amazontrust.com/repository/AmazonRootCA1.pem',
 'root.pem')
```

若要連線至 Aurora DSQL，請先在 Python 儲存格中安裝適用於 Python 的 Aurora DSQL Connector 和 Psycopg 驅動程式，然後將其匯入：

```
pip install aurora_dsql_python_connector psycopg
```

```
import aurora_dsql_psycopg as dsql
```

匯入連接器後，您就可以建立 DSQL 組態並進行連線。Aurora DSQL Python Connector 會自動在每個連線上處理身分驗證字符的建立。

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

conn = dsql.connect(**config)
```

執行程式碼時，您現在應該擁有與 Aurora DSQL 的 Psycopg 連線。然後，您可以使用 Psycopg 游標執行查詢並提供 SQL 查詢。如需搭配 Postgres 相容資料庫使用 Psycopg 的詳細資訊，請參閱 Psycopg 文件。此查詢將產生 中的元組清單`results_list`。

```
with conn:
    with conn.cursor() as cur:
        cur.execute("SELECT * FROM table")
        results_list = cur.fetchall()
```

然後，您可以使用 Pandas 等 Python 架構來分析或視覺化查詢結果，例如：

```
pip install pandas

import pandas as pd
```

```
df = pd.DataFrame(tuples_list)
print(df)
print(f"Total records: {len(df)}")
```

筆記本範例

使用 Aurora DSQL 的範例筆記本可在 Aurora DSQL 範例儲存庫中使用。

深入閱讀

Amazon SageMaker AI 設定文件

適用於 Python 的 Aurora DSQL 連接器

Pandas 文件

Amazon Aurora DSQL 的備份與還原

Amazon Aurora DSQL 透過與 AWS Backup 整合，協助您滿足法規遵循與業務持續性需求。這項全受管的資料保護服務可讓您輕鬆集中並自動化跨 AWS 服務、雲端及內部部署環境的備份作業。此服務簡化了單一區域與多區域 Aurora DSQL 叢集的備份建立、管理與還原作業。

重要功能如下所示：

- 透過 AWS 管理主控台 主控台、SDK 或 AWS CLI 進行集中式備份管理
- 完整叢集備份
- 自動化備份排程與保留原則
- 跨區域與跨帳戶功能
- WORM (一次寫入、多次讀取) 組態，適用於您儲存於備份保存庫中的所有備份

如需了解 AWS Backup 保存庫鎖定 (Vault Lock) 功能及其在 Aurora DSQL 上的完整可用 AWS Backup 功能清單，請參閱 AWS Backup 開發人員指南中的保存庫鎖定優點與 AWS Backup 功能可用性。

AWS Backup 入門

AWS Backup 會建立您 Aurora DSQL 叢集的完整備份副本。您可以依照 AWS Backup 入門指南中的步驟，開始使用 AWS Backup for Aurora DSQL：

1. 建立隨需備份以立即保護資料。
2. 建立備份計劃，以啟用自動化與排程備份。
3. 設定保留期限與跨區域複製。
4. 設定備份活動的監控與通知。

還原您的備份

當您還原 Aurora DSQL 叢集時，AWS Backup 會建立新的叢集以確保來源資料的完整性。

還原單一區域叢集

若要還原 Aurora DSQL 單一區域叢集，請使用主控台：https://console.aws.amazon.com/backup 或 CLI，選取您要還原的復原點 (備份)。設定將從備份建立的新叢集之各項設定。如需詳細步驟，請參閱還原單一區域 Aurora DSQL 叢集。

還原多區域叢集

Aurora DSQL 多區域叢集的還原作業可透過主控台 https://console.aws.amazon.com/backup 或 AWS CLI 進行。如需詳細步驟，請參閱還原多區域 Aurora DSQL 叢集。

若要將資料還原至多區域 Aurora DSQL 叢集，您可以使用在單一 AWS 區域 中建立的備份。不過，在啟動還原程序前，您必須確保多區域叢集所使用的所有 AWS 區域 中皆有相同的備份副本。若尚未建立這些副本，請先將備份複製至另一個支援多區域叢集的 AWS 區域。

建議您在主要 AWS 區域 中建立備份副本，以啟用強大的災難復原功能並符合法規要求。若要查看 Aurora DSQL 可用的 AWS 區域，請參閱 the section called “AWS 區域 可用性”。

如需這些步驟的詳細資訊，請參閱 Amazon Aurora DSQL 還原文件。

監控與法規遵循

AWS Backup 透過下列資源，提供備份與還原作業的完整可視性。

- 集中式儀表板，用於追蹤備份與還原作業。
- 可與 CloudWatch 及 CloudTrail 整合。
- 使用 AWS Backup Audit Manager 以進行法規報告與稽核。

如需深入了解如何在使用 Aurora DSQL 時記錄使用者、角色或 AWS 服務 執行的操作，請參閱 使用 AWS CloudTrail 記錄 Aurora DSQL 作業。

其他資源

若要進一步了解 AWS Backup 的功能及其與 Aurora DSQL 的整合使用方式，請參閱下列資源：

- AWS Backup 的受管原則
- Amazon Aurora DSQL 還原
- AWS 區域 支援的服務

- AWS Backup 中的備份加密

透過使用 AWS Backup for Aurora DSQL，您可建置穩健、符合法規且自動化的備份策略，既能保護關鍵資料庫資源，也能將管理負擔降至最低。無論您管理的是單一叢集或複雜的多區域部署，AWS Backup 都提供您所需的工具，確保資料安全且可復原。

Aurora DSQL 的監控和記錄

監控和記錄是維護 Amazon Aurora DSQL 資源可靠性、可用性與效能的重要環節。您應該全面監控並收集 Aurora DSQL 資源記錄資料，以更輕鬆的進行多點故障偵錯。

- Amazon CloudWatch AWS 會即時監控您的 AWS 資源和您在 上執行的應用程式。您可以收集和追蹤指標、建立自訂儀板表，以及設定警示，在特定指標達到您指定的閾值時通知您或採取動作。例如，您可以讓 CloudWatch 追蹤 CPU 使用量或其他 Amazon EC2 執行個體指標，並在需要時自動啟動新的執行個體。如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南。
- AWS CloudTrail 會擷取由 發出或代表發出的 API 呼叫和相關事件， AWS 帳戶 並將日誌檔案交付至您指定的 Amazon S3 儲存貯體。您可以識別呼叫的使用者和帳戶 AWS、進行呼叫的來源 IP 地址，以及呼叫的時間。如需詳細資訊，請參閱「AWS CloudTrail 使用者指南」。

使用 Amazon CloudWatch 監控 Aurora DSQL

使用 CloudWatch 監控 Aurora DSQL 時，其會收集原始資料並將資料處理成可讀且近乎即時的指標。CloudWatch 會保留這些統計資料 15 個月，協助您更深入了解 Web 應用程式或服務效能。設定警示以監控特定閾值，並在符合條件時傳送通知或採取動作。檢閱下列適用於 Aurora DSQL 的用量和可觀測性指標。

如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南。

可觀測性和效能

此表格概述 Aurora DSQL 的可觀測性指標。其包含用於追蹤唯讀及總交易的指標，以提供整體工作負載特徵分析。查詢逾時和 OCC 衝突率等可操作的指標，協助識別效能問題及並行衝突。與工作階段相關的指標，包括作用中及總體指標，可深入了解系統目前的負載。

CloudWatch 指標名稱	指標	單位	描述
ReadOnlyTransactions	Read-only transactions	none	The number of read-only transactions
TotalTransactions	Total transactions	none	The total number of transactions executed on the system,

CloudWatch 指標名稱	指標	單位	描述
			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

用量指標

Aurora DSQL 使用稱為分散式處理單元 (DPU) 的單一標準化計費單位，測量所有請求型的活動，例如查詢處理、讀取及寫入。

CloudWatch 指標名稱	指標	維度：ResourceId	單位	描述
WriteDPU	Write Units	<cluster-id>	DPU	Approximates the write active-

CloudWatch 指標名稱	指標	維度：ResourceId	單位	描述
				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 DSQL cluster DPU usage.

CloudWatch 指標名稱	指標	維度：ResourceId	單位	描述
TotalDPU	Total Units	<cluster-id>	DPU	Approximates the total active-use component of your Aurora DSQL cluster DPU usage.

使用 AWS CloudTrail 記錄 Aurora DSQL 作業

Amazon Aurora DSQL 與 AWS CloudTrail 整合，這項服務可提供由使用者、角色或 AWS 服務所採取之動作的記錄。CloudTrail 中包含兩種類型的事件：管理事件及資料事件。管理事件發生時，會稽核 AWS 資源組態變更。資料事件通常會擷取服務資料平面中的 AWS 資源用量。

CloudTrail 會將 Aurora DSQL 的所有 API 呼叫擷取為事件。Aurora DSQL 會將主控台活動記錄為管理事件。也會將針對叢集之已驗證連線嘗試擷取為資料事件。

您可以利用 CloudTrail 所收集的資訊，判斷向 Aurora DSQL 發出的請求，以及發出請求的 IP 位址、時間、使用者身分和其他詳細資訊。

當您建立帳戶時，CloudTrail 會預設在 AWS 帳戶 中啟用，而且您將自動取得 CloudTrail 事件歷史記錄的存取權。CloudTrail 事件歷史記錄為 AWS 區域中過去 90 天記錄的管理事件，提供可檢視、可搜尋、可下載且不可變的記錄。如需詳細資訊，請參閱 AWS CloudTrail 使用者指南中的使用 CloudTrail 事件歷史記錄。記錄事件歷史記錄不會產生 CloudTrail 費用。

若要在 AWS 帳戶中建立事件的持續記錄，包括 Aurora DSQL 的事件，請建立追蹤記錄或 AWS CloudTrail Lake 事件資料存放區 (AWS CloudTrail事件的集中儲存空間及分析解決方案)。如需進一步了解如何建立追蹤記錄，請參閱使用 CloudTrail 追蹤記錄。若要了解如何設定和管理事件資料存放區，請參閱 CloudTrail Lake 事件資料存放區。

CloudTrail 中的管理事件

CloudTrail 管理事件可提供您 AWS 帳戶資源上所執行之管理作業的相關資訊。這些也稱為控制平面操作。根據預設，CloudTrail 會擷取事件歷史記錄中的管理事件。

Amazon Aurora DSQL 會將所有控制平面作業，記錄為管理事件。如需由 Aurora DSQL 記錄到 CloudTrail 的 Amazon Aurora DSQL 控制平面作業記錄清單，請參閱 Aurora DSQL API 參考。

控制平面日誌

Amazon Aurora DSQL 會將下列 Aurora DSQL 控制平面作業歸類為管理事件，記錄到 CloudTrail。

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

備份和還原日誌

Amazon Aurora DSQL 會將下列 Aurora DSQL 備份和還原作業歸類為管理事件，記錄到 CloudTrail。

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

如需進一步了解如何使用 AWS Backup 保護 Aurora DSQL 叢集，請參閱 Amazon Aurora DSQL 的備份與還原。

AWS KMS 日誌

Amazon Aurora DSQL 會將下列 AWS KMS 作業歸類為管理事件，記錄到 CloudTrail。

- `GenerateDataKey`
- `Decrypt`

若要進一步了解 CloudTrail 日誌如何追蹤 Aurora DSQL 代表您傳送到 AWS KMS 的請求，請參閱 監控 Aurora DSQL 與 的互動 AWS KMS。

CloudTrail 中的 Aurora DSQL 資料事件

CloudTrail 資料事件通常提供於資源上或於資源中執行之資源作業相關資訊。此外，也會用來擷取服務的資料平面作業。資料事件通常是大量資料的活動。根據預設，CloudTrail 不會記錄資料事件。CloudTrail 事件歷史記錄不會記錄資料事件。

如需如何記錄資料事件的詳細資訊，請參閱 AWS CloudTrail 使用者指南中的使用 AWS 管理主控台 記錄資料事件和使用 AWS Command Line Interface 記錄資料事件。

資料事件需支付額外的費用。如需 CloudTrail 定價的詳細資訊，請參閱 AWS CloudTrail 定價。

對於 Aurora DSQL，CloudTrail 會將任何對 Aurora DSQL 叢集所做的連線嘗試都擷取為資料事件。下表列出可記錄資料事件的 Aurora DSQL 資源類型。資源類型 (主控台) 欄會顯示從 CloudTrail 主控台上的資源類型清單中選擇的值。resources.type 值資料行會顯示 `resources.type` 值，您需在使用 AWS CLI 或 CloudTrail API 設定進階事件選取器時指定該值。記錄到 CloudTrail 的資料 API 資料行會針對資源類型顯示記錄到 CloudTrail 的 API 呼叫。

資源類型 (主控台)	resources.type 值	記錄到 CloudTrail 的資料 API
Amazon Aurora DSQL	`AWS::DSQL::Cluster`	• `DbConnect` • `DbConnectAdmin`

您可以設定進階事件選取器以篩選 `eventName` 和 `resources.ARN` 欄位，僅記錄對您重要的事件。如需這些欄位的詳細資訊，請參閱 AWS CloudTrail API 參考中的 AdvancedFieldSelector。

下列範例說明如何使用 AWS CLI 設定 `dsql-data-events-trail` 來接收 Aurora DSQL 的資料事件。

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

Amazon Aurora DSQL 的安全性

的雲端安全性 AWS 是最高優先順序。身為 AWS 客戶，您可以受益於資料中心和網路架構，這些架構是為了滿足最安全敏感組織的需求而建置。

安全性是 AWS 與您之間的共同責任。共同責任模式將其描述為雲端的安全性，和雲端中的安全性：

- 雲端的安全性 – AWS 負責保護在 中執行 AWS 服務的基礎設施 AWS 雲端。 AWS 也為您提供可安全使用的服務。作為AWS 合規計畫的一部分，第三方稽核人員會定期測試和驗證我們安全的有效性。若要了解適用於 Amazon Aurora DSQL 的合規計劃，請參閱AWS 合規計劃的 服務範圍。
- 雲端的安全性 – 您的責任取決於您使用 AWS 的服務。您也必須對其他因素負責，包括資料的機密性、您公司的要求和適用法律和法規。

本文件有助於您瞭解如何在使用 Aurora DSQL 時套用共同責任模型。下列主題將示範如何設定 Aurora DSQL 以達到您的安全與合規目標。您也會了解如何使用其他 AWS 服務來協助您監控和保護 Aurora DSQL 資源。

主題

- AWS Amazon Aurora DSQL 的 受管政策
- Amazon Aurora DSQL 中的資料保護
- 適用於 Amazon Aurora DSQL 的資料加密
- Aurora DSQL 的身分與存取管理
- Aurora DSQL 的資源型政策
- 在 Aurora DSQL 中使用服務連結角色
- 使用 IAM 條件索引鍵搭配 Amazon Aurora DSQL
- Amazon Aurora DSQL 中的事件回應
- Amazon Aurora DSQL 的合規驗證
- Amazon Aurora DSQL 的恢復能力
- Amazon Aurora DSQL 中的基礎結構安全性
- Amazon Aurora DSQL 中的組態與漏洞分析
- 預防跨服務混淆代理人
- Aurora DSQL 的安全最佳實務

AWS Amazon Aurora DSQL 的 受管政策

AWS 受管政策是由 AWS AWS 受管政策建立和管理的獨立政策旨在為許多常用案例提供許可，以便您可以開始將許可指派給使用者、群組和角色。

請記住， AWS 受管政策可能不會授予特定使用案例的最低權限許可，因為這些許可可供所有 AWS 客戶使用。我們建議您定義特定於使用案例的客戶管理政策，以便進一步減少許可。

您無法變更 AWS 受管政策中定義的許可。如果 AWS 更新受 AWS 管政策中定義的許可，則更新會影響政策連接的所有委託人身分（使用者、群組和角色）。當新的 AWS 服務 啟動或新的 API 操作可供現有服務使用時， AWS 最有可能更新 AWS 受管政策。

如需詳細資訊，請參閱 IAM 使用者指南中的 AWS 受管政策。

AWS 受管政策：AmazonAuroraDSQLFullAccess

您可以將 `AmazonAuroraDSQLFullAccess` 連接至使用者、群組與角色。

此政策會授予許可權限，藉此提供 Aurora DSQL 完整的管理存取權限。具有這些許可權限的主體可以：

- 建立、刪除和更新 Aurora DSQL 叢集 (包括多區域叢集)
- 管理叢集內嵌政策（建立、檢視、更新和刪除政策）
- 從叢集新增和移除標籤
- 列出叢集並檢視個別叢集的相關資訊
- 請參閱連接到 Aurora DSQL 叢集的標籤
- 以任何使用者身分 (包括管理員) 連線至資料庫
- 執行 Aurora DSQL 叢集的備份和還原作業，包括啟動、停止和監控備份和還原作業
- 使用客戶受管 AWS KMS 金鑰進行叢集加密
- 從 CloudWatch 檢視其帳戶的任何指標
- 使用 AWS Fault Injection Service (AWS FIS) 將故障注入 Aurora DSQL 叢集以進行容錯能力測試
- 建立 `dsql.amazonaws.com` 服務的服務連結角色 (建立叢集所需的角色)

許可詳細資訊

此政策包含以下許可。

- `dsql`：授予權限允許主體完整存取 Aurora DSQL。
- `cloudwatch`：授予許可權限允許將指標資料點發佈至 Amazon CloudWatch。
- `iam`：授予許可權限允許建立服務連結角色。
- `backup and restore`：授予許可權限以允許啟動、停止和監控 Aurora DSQL 叢集的備份和還原作業。
- `kms`：在建立、更新或連線至叢集時，授予必要許可權限以驗證存取用於 Aurora DSQL 叢集加密的客戶受管金鑰。
- `fis`—授予使用 AWS Fault Injection Service (AWS FIS) 將失敗注入 Aurora DSQL 叢集以進行容錯能力測試的許可。

您可在 IAM 主控台和AWS 受管政策參考指南找到 `AmazonAuroraDSQLFullAccess` 政策。

AWS 受管政策：AmazonAuroraDSQLReadOnlyAccess

您可以將 `AmazonAuroraDSQLReadOnlyAccess` 連接至使用者、群組與角色。

允許讀取存取 Aurora DSQL。具有這類許可權限的主體可以列出叢集，並檢視個別叢集的相關資訊。他們可以查看連接到 Aurora DSQL 叢集的標籤，並檢視叢集內嵌政策。他們可以擷取及查看 CloudWatch 監控您帳戶的任何指標。

許可詳細資訊

此政策包含以下許可。

- `dsql`：授予 Aurora DSQL 中所有資源的唯讀許可權限。
- `cloudwatch`：授予許可權限以允許批次擷取 CloudWatch 指標資料，並依據擷取資料執行指標數學運算。

您可在 IAM 主控台和AWS 受管政策參考指南找到 `AmazonAuroraDSQLReadOnlyAccess` 政策。

AWS 受管政策：AmazonAuroraDSQLConsoleFullAccess

您可以將 `AmazonAuroraDSQLConsoleFullAccess` 連接至使用者、群組與角色。

允許透過 AWS 管理主控台對 Amazon Aurora DSQL 進行完整管理存取。具有這些許可權限的主體可以：

- 使用主控台建立、刪除和更新 Aurora DSQL 叢集 (包括多區域叢集)
- 透過主控台管理叢集內嵌政策（建立、檢視、更新和刪除政策）
- 列出叢集並檢視個別叢集的相關資訊
- 查看您帳戶任何資源的標籤
- 以任何使用者身分 (包括管理員) 連線至資料庫
- 執行 Aurora DSQL 叢集的備份和還原作業，包括啟動、停止和監控備份和還原作業
- 使用客戶受管 AWS KMS 金鑰進行叢集加密
- AWS CloudShell 從 啟動 AWS 管理主控台
- 從您帳戶的 CloudWatch 檢視任何指標
- 使用 AWS Fault Injection Service (AWS FIS) 將故障注入 Aurora DSQL 叢集以進行容錯能力測試
- 建立 `dsql.amazonaws.com` 服務的服務連結角色 (建立叢集所需的角色)

您可以在 IAM 主控台和《 AWS 受管`AmazonAuroraDSQLConsoleFullAccess`政策參考指南》中的 AmazonAuroraDSQLConsoleFullAccess 中找到政策。

許可詳細資訊

此政策包含以下許可。

- `dsql`：透過 AWS 管理主控台授予 Aurora DSQL 中所有資源的完整管理許可權限。
- `cloudwatch`：授予許可權限以允許批次擷取 CloudWatch 指標資料，並依據擷取資料執行指標數學運算。
- `tag`- 授予許可，以傳回目前在 AWS 區域 為呼叫帳戶指定的 中使用的標籤索引鍵和值。
- `backup and restore`：授予許可權限以允許啟動、停止和監控 Aurora DSQL 叢集的備份和還原作業。
- `kms`：在建立、更新或連線至叢集時，授予必要許可權限以驗證存取用於 Aurora DSQL 叢集加密的客戶受管金鑰。
- `cloudshell`- 授予啟動 AWS CloudShell 以與 Aurora DSQL 互動的許可。
- `ec2`：授予許可權限以允許檢視 Aurora DSQL 連線所需的 Amazon VPC 端點資訊。

- `fis-` 授予許可，以使用 AWS FIS 將故障注入 Aurora DSQL 叢集以進行容錯能力測試。
- `access-analyzer:ValidatePolicy` 在政策編輯器中授予 linter 的許可，該政策可提供目前政策中錯誤、警告和安全問題的即時意見回饋。
- `fis`—授予使用 AWS Fault Injection Service (AWS FIS) 將失敗注入 Aurora DSQL 叢集以進行容錯能力測試的許可。

您可在 IAM 主控台和AWS 受管政策參考指南找到 `AmazonAuroraDSQLConsoleFullAccess` 政策。

AWS 受管政策：AuroraDSQLServiceRolePolicy

您無法將 AuroraDSQLServiceRolePolicy 連接至 IAM 實體。此政策會連接至服務連結角色，可讓 Aurora DSQL 存取帳戶資源。

您可以在 IAM 主控台和 AWS 受管`AuroraDSQLServiceRolePolicy`政策參考指南中的 AuroraDSQLServiceRolePolicy 中找到政策。

AWS 受管政策的 Aurora DSQL 更新

檢視自此服務開始追蹤這些變更以來，Aurora DSQL AWS 受管政策更新的詳細資訊。如需自動收到有關此頁面變更的提醒，請前往 Aurora DSQL 文件歷史記錄頁面上訂閱 RSS 摘要。

變更	描述	Date
AmazonAuroraDSQLFullAccess 和 AmazonAuroraDSQLConsoleFullAccess 更新	新增對 AWS Fault Injection Service (AWS FIS) 與 Aurora DSQL 整合的支援。這可讓您將故障注入單一區域和多區域 Aurora DSQL 叢集，以測試應用程式的容錯能力。您可以在 AWS FIS 主控台中建立實驗範本，以定義故障案例並鎖定特定 Aurora DSQL 叢集進行測試。	2025 年 8 月 19 日

變更	描述	Date
	如需這些政策的詳細資訊，請參閱 AmazonAuroraDSQLFullAccess 和 AmazonAuroraDSQLConsoleFullAccess。	
AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess 更新	新增具有新許可的資源型政策 (RBP) 支援：`PutClusterPolicy`、`GetClusterPolicy` 和 `DeleteClusterPolicy`。這些許可允許管理連接到 Aurora DSQL 叢集的內嵌政策，以進行精細存取控制。 如需詳細資訊，請參閱 AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess。	2025 年 10 月 15 日
AmazonAuroraDSQLFullAccess 更新	新增可執行 Aurora DSQL 叢集備份和還原操作的功能，包括啟動、停止和監控任務。此外也新增可使用客戶受管 KMS 金鑰進行叢集加密的功能。 如需更多詳細資訊，請參閱 AmazonAuroraDSQLFullAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 21 日

變更	描述	Date
AmazonAuroraDSQLConsoleFullAccess 更新	新增透過 AWS Console Home 執行 Aurora DSQL 叢集備份和還原操作的功能。其中包括啟動、停止和監控任務。此外還支援使用客戶受管 KMS 金鑰進行叢集加密和啟動 AWS CloudShell。 如需更多詳細資訊，請參閱 AmazonAuroraDSQLConsoleFullAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 21 日
AmazonAuroraDSQLFullAccess 更新	政策會新增四個新許可，以跨多個 建立和管理資料庫叢集 AWS 區域：`PutMultiRegionProperties` 、`AddPeerCluster` 、 `PutWitnessRegion` 和 `RemovePeerCluster` 。這些許可權限包括資源層級控制和條件索引鍵，讓您可以控制自己能夠修改的叢集使用者。 政策也新增了 `GetVpcEndpointServiceName` 許可權限，協助您透過 AWS PrivateLink連線至 Aurora DSQL 叢集。 如需更多詳細資訊，請參閱 AmazonAuroraDSQLFullAccess 和在 Aurora DSQL 中使用服務連結角色。	2025 年 5 月 13 日

變更	描述	Date
AmazonAuroraDSQLReadOnlyAccess 更新	包括透過 Aurora DSQL 連線至 Aurora DSQL 叢集時判斷正確 VPC AWS PrivateLink 端點服務名稱的能力，可為每個儲存格建立唯一的端點，因此此 API 有助於確保您可以識別叢集的正確端點，並避免連線錯誤。 如需更多詳細資訊，請參閱 AmazonAuroraDSQLReadOnlyAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 13 日
AmazonAuroraDSQLConsoleFullAccess 更新	將新許可權限新增至 Aurora DSQL，以支援多區域叢集管理和 VPC 端點連線。新的許可權限包括：`PutMultiRegionProperties` `PutWitnessRegion` `AddPeerCluster` `RemovePeerCluster` `GetVpcEndpointServiceName` 如需更多詳細資訊，請參閱 AmazonAuroraDSQLConsoleFullAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 13 日

變更	描述	Date
AuroraDsqlServiceLinkedRolePolicy 更新	新增將指標發佈至 AWS/AuroraDSQL 和 AWS/Usage CloudWatch 命名空間至政策的功能。這可讓相關聯的服務或角色，將更全面的使用和效能資料傳送到 CloudWatch 環境。 如需更多詳細資訊，請參閱 AuroraDsqlServiceLinkedRolePolicy 和在 Aurora DSQL 中使用服務連結角色。	2025 年 5 月 8 日
頁面已建立	開始追蹤與 Amazon Aurora DSQL 相關的 AWS 受管政策	2024 年 12 月 3 日

Amazon Aurora DSQL 中的資料保護

共同責任模型套用至之中的資料保護。如此模型所述，負責保護執行所有 AWS 雲端的全球基礎結構。您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱資料隱私權常見問答集。如需有關歐洲資料保護的相關資訊，請參閱安全性部落格上的共同責任模型和 GDPR 部落格文章。

基於資料保護目的，我們建議您保護登入資料，並使用 AWS IAM Identity Center 或 設定個別使用者 AWS Identity and Access Management。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與資源通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用 設定 API 和使用者活動記錄 AWS CloudTrail。如需使用追蹤功能擷取活動的更多詳細資訊，請參閱使用者指南中的使用追蹤功能。
- 使用加密解決方案，以及 AWS 服務內的所有預設安全控制。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱欄位。這包括當您使用 或使用 主控台、API AWS CLI、 或 AWS SDKs的其他 時。您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

資料加密

Amazon Aurora DSQL 提供高度耐用的儲存基礎結構，專為關鍵任務及主要資料儲存所設計。資料會以備援方式儲存在 Aurora DSQL 區域中多個設施的多部裝置上。

傳輸中加密

預設會為您設定傳輸中加密。Aurora DSQL 使用 TLS 加密 SQL 用戶端與 Aurora DSQL 之間的所有流量。

在 AWS CLI SDK 或 API 用戶端與 Aurora DSQL 端點之間加密和簽署傳輸中的資料：

- Aurora DSQL 提供 HTTPS 端點用於加密傳輸中的資料。
- 為了保護向 Aurora DSQL 發出 API 請求的完整性，必須由發起人簽署 API 呼叫。根據簽章第 4 版簽署程序 (Sigv4)，呼叫由 X.509 憑證或客戶的 AWS 私密存取金鑰進行簽署。如需詳細資訊，請參閱 AWS 一般參考 中的 Signature 第 4 版簽署程序。
- 使用 AWS CLI 或其中一個 AWS SDKs 向 提出請求 AWS。這些工具會自動使用您設定工具時指定的存取金鑰，替您簽署請求。

FIPS 合規

Aurora DSQL 資料平面端點 （用於資料庫連線的叢集端點） 預設會使用 FIPS 140-2 驗證的密碼編譯模組。叢集連線不需要單獨的 FIPS 端點。

對於控制平面操作，Aurora DSQL 在支援的區域提供專用 FIPS 端點。如需控制平面 FIPS 端點的詳細資訊，請參閱《》中的 Aurora DSQL 端點和配額AWS 一般參考。

靜態加密請參閱 Aurora DSQL 的靜態加密。

網際網路流量隱私權

Aurora DSQL 與內部部署應用程式之間，以及 Aurora DSQL 與相同資源內其他 AWS 資源之間的連線都會受到保護 AWS 區域。

您的私有網路與 之間有兩個連線選項 AWS：

- An AWS Site-to-Site VPN 連接。如需詳細資訊，請參閱什麼是 AWS Site-to-Site VPN？
- Direct Connect 連線。如需詳細資訊，請參閱什麼是 Direct Connect？

您可以使用 AWS發佈的 API 操作透過網路存取 Aurora DSQL。使用者端必須支援下列專案：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

見證區域中的資料保護

您建立多區域叢集時，見證區域會參與加密交易的同步複寫，以協助啟用自動故障復原。如果對等叢集無法使用，見證區域仍可用於驗證和處理資料庫寫入，確保不會喪失可用性。

見證區域透過以下設計功能保護資料安全無虞：

- 見證區域只會接收和儲存加密的交易日誌。其中絕對不會託管、儲存或傳輸您的加密金鑰。
- 見證區域僅專注於寫入交易日誌和仲裁函數。見證區域設計為無法讀取您的資料。
- 見證區域是在沒有叢集連線端點或查詢處理器的情況下運作。這可防止存取使用者資料庫。

如需關於見證區域的詳細資訊，請參閱 設定多區域叢集。

為 Aurora DSQL 連線設定 SSL/TLS 憑證

Aurora DSQL 要求所有連線使用 Transport Layer Security (TLS) 加密。若要建立安全連線，您的用戶端系統必須信任 Amazon 根憑證認證機構 (Amazon 根 CA 1)。此憑證已預先安裝在許多作業系統。本節說明如何驗證各種作業系統預先安裝的 Amazon 根 CA 1 憑證，並在沒有安裝憑證的情況下，引導您完成手動安裝憑證的程序。

我們建議使用 PostgreSQL 17 版。

> **Important**
>
> 對於正式作業環境，我們建議使用 `verify-full` SSL 模式以確保最高層級的連線安全性。此模式會驗證伺服器憑證是否由信任的憑證認證機構簽署，以及伺服器主機名稱是否與憑證相符。

驗證預先安裝的憑證

在大多數作業系統中，Amazon 根 CA 1 已預先安裝。若需驗證，您可以遵循下列步驟。

Linux (RedHat/CentOS/Fedora)

在終端中執行下列命令：

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

如果已安裝憑證，您會看到下列輸出內容：

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

macOS

1. 開啟 Spotlight 搜尋 (命令 + 空格)
2. 搜尋鑰匙圈存取
3. 選擇系統鑰匙圈之下的系統根目錄
4. 在憑證清單中尋找 Amazon 根 CA 1

Windows

Note

由於 psql Windows 用戶端的已知問題，使用系統根憑證 (`sslrootcert=system`) 可能會傳回下列錯誤：`SSL error: unregistered scheme`。您可以遵循 從 Windows 進行連線 作為使用 SSL 連線至叢集的替代方法。

如果您的作業系統並未安裝 Amazon 根 CA 1，請遵循下列步驟。

安裝憑證

如果 `Amazon Root CA 1` 憑證未預先安裝在您的作業系統上，您將需要手動安裝憑證，才能與 Aurora DSQL 叢集建立安全連線。

Linux 憑證安裝

請依照下列步驟在 Linux 系統安裝 Amazon 根 CA 憑證。

1. 下載根憑證：

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

2. 將憑證複製到信任存放區：

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

3. 更新 CA 信任存放區：

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

4. 驗證安裝：

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

macOS 憑證安裝

這些憑證安裝步驟為選用。Linux 憑證安裝 也適用於 macOS。

1. 下載根憑證：

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

2. 將憑證新增至系統鑰匙圈：

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

3. 驗證安裝：

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

以 SSL/TLS 驗證連線

在設定 SSL/TLS 憑證以安全連線至 Aurora DSQL 叢集之前，請先確定您符合下列先決條件。

- 已安裝 PostgreSQL 第 17 版

- AWS CLI 使用適當的登入資料設定
- Aurora DSQL 叢集端點資訊

從 Linux 連線

1. 產生並設定身分驗證記號：

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

2. 使用系統憑證進行連線 (如果預先安裝)：

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

3. 或者使用下載的憑證進行連線：

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

> **Note**
>
> 如需 PGSSLMODE 設定的詳細資訊，請參閱 PostgresQL 17 資料庫連線控制功能文件中的 sslmode。

從 macOS 連線

1. 產生並設定身分驗證記號：

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

2. 使用系統憑證進行連線 (如果預先安裝)：

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

3. 或者下載根憑證並將其儲存為 `root.pem` (如果未預先安裝憑證)

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

4. 使用 psql 進行連線：

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

從 Windows 進行連線

使用命令提示

1. 產生身分驗證記號：

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

2. 設定密碼環境變數：

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

3. 設定 SSL 組態：

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

4. 連線至資料庫：

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

使用 PowerShell

1. 產生並設定身分驗證記號：

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

2. 設定 SSL 組態：

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

3. 連線至資料庫：

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

其他資源

- PostgreSQL SSL 文件
- Amazon Trust Services

適用於 Amazon Aurora DSQL 的資料加密

Amazon Aurora DSQL 會加密所有使用者的靜態資料。為了增強安全性，此加密使用 AWS Key Management Service (AWS KMS)。此功能協助降低了保護敏感資料所涉及的操作負擔和複雜性。靜態加密可協助您：

- 減少保護敏感資料的作業負擔
- 建立對安全性要求甚高的應用程式，以符合嚴格的加密合規或管制需求。
- 一律保護加密叢集中的資料，以增加多一層的資料保護
- 遵守組織政策、產業或政府法規，以及合規要求

Aurora DSQL 可協助您建立對安全性要求甚高的應用程式，以符合嚴格加密合規和法規要求。下列各節說明如何為新的和現有 Aurora DSQL 資料庫設定加密，並管理您的加密金鑰。

主題

- 適用於 Aurora DSQL 的 KMS 金鑰類型
- Aurora DSQL 的靜態加密
- 搭配 Aurora DSQL 使用 AWS KMS 和 資料金鑰
- 授權使用 AWS KMS key 適用於 Aurora DSQL 的
- Aurora DSQL 加密內容
- 監控 Aurora DSQL 與 的互動 AWS KMS
- 建立加密的 Aurora DSQL 叢集
- 移除或更新 Aurora DSQL 叢集的金鑰
- 使用 Aurora DSQL 進行加密的考量事項

適用於 Aurora DSQL 的 KMS 金鑰類型

Aurora DSQL 與 整合 AWS KMS ，以管理叢集的加密金鑰。若要進一步瞭解金鑰類型和狀態，請參閱 AWS Key Management Service 開發人員指南中的AWS Key Management Service 概念。建立新叢集時，您可選擇以下列 KMS 金鑰類型加密叢集：

AWS 擁有的金鑰

預設的加密類型。Aurora DSQL 擁有金鑰，您不需支付額外費用。您存取加密叢集時，Amazon Aurora DSQL 會透明地解密叢集資料。您不需要變更程式碼或應用程式以使用或管理加密叢集，而且所有 Aurora DSQL 查詢都適用於您的加密資料。

客戶自管金鑰

您可以在 中建立、擁有和管理金鑰 AWS 帳戶。您可以完全控制 KMS 金鑰。需支付 AWS KMS 費用。

使用 AWS 擁有的金鑰 進行靜態加密無需額外費用。不過， AWS KMS 費用適用於客戶受管金鑰。如需詳細資訊，請參閱 AWS KMS 定價頁面。

您可以隨時在這些金鑰類型之間切換。如需金鑰類型的詳細資訊，請參閱 AWS Key Management Service 開發人員指南中的客戶自管金鑰和 AWS 擁有的金鑰。

Aurora DSQL 靜態加密可在可使用 Aurora DSQL 的所有 AWS 區域中使用。

Aurora DSQL 的靜態加密

Amazon Aurora DSQL 使用 256 位元進階加密標準 (AES-256) 加密您的靜態資料。此加密有助於保護您的資料免於未經授權存取基礎 storage。 會 AWS KMS 管理叢集的加密金鑰。您可以使用預設 AWS 擁有的金鑰，或選擇使用您自己的 AWS KMS 客戶自管金鑰。若要進一步瞭解如何指定和管理 Aurora DSQL 叢集的金鑰，請參閱 建立加密的 Aurora DSQL 叢集 和 移除或更新 Aurora DSQL 叢集的金鑰。

主題

- AWS 擁有的金鑰
- 客戶自管金鑰

AWS 擁有的金鑰

Aurora DSQL 預設會使用 加密所有叢集 AWS 擁有的金鑰。這些金鑰可免費使用，並會每年輪換，以保護您的帳戶資源。您不需要檢視、管理、使用或稽核這些金鑰，因此不需採取任何動作進行資料保護。如需 的詳細資訊 AWS 擁有的金鑰，請參閱《 AWS Key Management Service 開發人員指南AWS 擁有的金鑰》中的 。

客戶自管金鑰

您可以在自己的 AWS 帳戶建立、擁有及管理客戶自管金鑰。您可以完全控制這些 KMS 金鑰，包括其政策、加密資料、標籤和別名。如需有關管理許可權限的詳細資訊，請參閱 AWS Key Management Service 開發人員指南中的客戶自管金鑰。

您指定客戶自管金鑰用於叢集層級加密時，Aurora DSQL 會使用該金鑰加密叢集及其所有區域資料。為了防止資料遺失和維護叢集存取，Aurora DSQL 需要存取您的加密金鑰。如果您停用客戶自管金鑰、排程刪除金鑰，或訂定政策以限制服務存取，叢集加密狀態會變更為

KMS_KEY_INACCESSIBLE。Aurora DSQL 無法存取金鑰時，使用者就無法連線至叢集，而叢集的加密狀態會變更為 KMS_KEY_INACCESSIBLE，且服務無法存取叢集資料。

對於多區域叢集，客戶可以分別設定每個區域的 AWS KMS 加密金鑰，而每個區域叢集會使用自己的叢集層級加密金鑰。如果 Aurora DSQL 無法存取多區域叢集中對等的加密金鑰，則對等狀態會變為 KMS_KEY_INACCESSIBLE，且無法用於進行讀取和寫入作業。其他對等會繼續正常作業。

> **Note**
>
> 如果 Aurora DSQL 無法存取您的客戶自管金鑰，您的叢集加密狀態會變更為 KMS_KEY_INACCESSIBLE。還原金鑰存取後，服務會在 15 分鐘內自動偵測還原。如需更多詳細資訊，請參閱叢集閒置。
> 對於多區域叢集，如果長時間喪失金鑰存取，則叢集還原時間取決於無法存取金鑰期間寫入的資料量。

搭配 Aurora DSQL 使用 AWS KMS 和 資料金鑰

Aurora DSQL 靜態加密功能使用 AWS KMS key 和資料金鑰階層來保護您的叢集資料。

我們建議您先規劃加密策略，再於 Aurora DSQL 中實作叢集。如果您將敏感或機密資料儲存在 Aurora DSQL，請考慮在計劃中加入用戶端加密。如此一來，您就能盡量靠近資料來源進行加密，並確保資料在整個生命週期受到保護。

主題

- AWS KMS key搭配 Aurora DSQL 使用
- 使用叢集金鑰搭配 Aurora DSQL
- 叢集金鑰快取

AWS KMS key搭配 Aurora DSQL 使用

靜態加密可保護 AWS KMS key之下的 Aurora DSQL 叢集。根據預設，Aurora DSQL 會使用 AWS 擁有的金鑰在 Aurora DSQL 服務帳戶中建立和管理的多租用戶加密金鑰。但您可在 AWS 帳戶中加密客戶自管金鑰之下的 Aurora DSQL 叢集。您可為每個叢集選取不同的 KMS 金鑰，即使叢集參與多區域設定也沒問題。

您在建立或更新叢集時，可以選取叢集的 KMS 金鑰。您可隨時在 Aurora DSQL 主控台或使用 UpdateCluster 操作變更叢集的 KMS 金鑰。切換金鑰的程序不需要停機時間或降低服務效能。

> **Important**
>
> Aurora DSQL 僅支援對稱 KMS 金鑰。您無法使用非對稱 KMS 金鑰加密 Aurora DSQL 叢集。

客戶自管金鑰具有以下優點：

- 您可以建立和管理 KMS 金鑰，包括設定金鑰政策和 IAM 政策以控制對 KMS 金鑰的存取。您可以啟用和停用 KMS 金鑰、啟用和停用自動金鑰輪換，以及於不再使用時刪除 KMS 金鑰。
- 您可以搭配匯入的金鑰材料使用客戶自管金鑰，或是使用位於您所擁有及管理自訂金鑰存放區中的客戶自管金鑰。
- 您可以在 AWS CloudTrail 日誌 AWS KMS 中檢查對 的 Aurora DSQL API 呼叫，以稽核 Aurora DSQL 叢集的加密和解密。

不過， AWS 擁有的金鑰 是免費的，其使用方式不會計入 AWS KMS 資源或請求配額。客戶自管金鑰會對每個 API 呼叫產生費用，且這些金鑰都適用 AWS KMS 配額。

使用叢集金鑰搭配 Aurora DSQL

Aurora DSQL 使用叢集 AWS KMS key 的 來產生和加密叢集的唯一資料金鑰，稱為叢集金鑰。

叢集金鑰用作金鑰加密金鑰。Aurora DSQL 使用此叢集金鑰保護用於加密叢集資料的資料加密金鑰。Aurora DSQL 會為叢集中的每個基礎結構產生唯一的資料加密金鑰，但多個叢集項目可能是由相同的資料加密金鑰保護。

若要解密叢集金鑰，Aurora DSQL 會在您第一次存取加密的叢集 AWS KMS 時傳送請求至 。為了讓叢集保持可用，Aurora DSQL 會定期驗證對 KMS 金鑰的解密存取，即使您未主動存取叢集也一樣。

Aurora DSQL 會在 外部存放和使用叢集金鑰和資料加密金鑰 AWS KMS。它使用進階加密標準 (AES) 加密和 256 位元加密金鑰來保護所有金鑰。然後會將加密金鑰與加密資料儲存在一起，以便用來隨需解密叢集資料。

如果您變更叢集的 KMS 金鑰，Aurora DSQL 會使用新的 KMS 金鑰重新加密現有叢集金鑰。

叢集金鑰快取

為了避免 AWS KMS 針對每個 Aurora DSQL 操作呼叫 ，Aurora DSQL 會快取記憶體中每個呼叫者的純文字叢集金鑰。如果 Aurora DSQL 在閒置 15 分鐘後收到快取叢集金鑰的請求，它會將新的請求傳送至 AWS KMS 以解密叢集金鑰。在上次請求解密叢集金鑰之後，此呼叫將擷取對 AWS KMS 或 AWS Identity and Access Management (IAM) AWS KMS key 中 存取政策所做的任何變更。

授權使用 AWS KMS key 適用於 Aurora DSQL 的

如果您在帳戶中使用客戶自管金鑰保護 Aurora DSQL 叢集，該金鑰的政策必須授予 Aurora DSQL 許可權限以代您使用該金鑰。

您可以完全控制客戶自管金鑰的政策。Aurora DSQL 不需要額外的授權，即可使用預設值 AWS 擁有的金鑰 來保護您 中的 Aurora DSQL 叢集 AWS 帳戶。

客戶自管金鑰的金鑰政策

當您選取客戶受管金鑰來保護 Aurora DSQL 叢集時，Aurora DSQL 需要 AWS KMS key 代表進行選取之主體使用 的許可。該委託人、使用者或角色必須擁有 AWS KMS key Aurora DSQL 所需的 許可。您可以在金鑰政策或 IAM 政策中提供這些許可權限。

Aurora DSQL 在客戶自管金鑰至少需要具備下列許可權限：

- `kms:Encrypt`
- `kms:Decrypt`
- `kms:ReEncrypt*` (適用於 kms:ReEncryptFrom 和 kms:ReEncryptTo)
- `kms:GenerateDataKey`
- `kms:DescribeKey`

例如，以下範例金鑰政策只會提供必要許可。政策具有下列效果：

- 允許 Aurora DSQL 在密碼編譯操作 AWS KMS key 中使用 ，但僅限於代表帳戶中有權使用 Aurora DSQL 的主體時。如果政策陳述式中指定的主體沒有許可權限可以使用 Aurora DSQL，呼叫便會失敗，即使呼叫是來自 Aurora DSQL 服務也一樣。
- `kms:ViaService` 條件索引鍵只會在請求來自 Aurora DSQL 且代表政策陳述式中列出的主體時，才會允許提供許可權限。這些主體無法直接呼叫這些操作。

使用範例金鑰政策之前，請將範例主體取代為來自您 的實際主體 AWS 帳戶。

```
{
  "Sid": "Enable dsql IAM User Permissions",
  "Effect": "Allow",
  "Principal": {
    "Service": "dsql.amazonaws.com"
  },
```

```
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey",
    "kms:Encrypt",
    "kms:ReEncryptFrom",
    "kms:ReEncryptTo"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:aws:dsql:ClusterId": "w4abucpbwuxx",
      "aws:SourceArn": "arn:aws:dsql:us-east-2:111122223333:cluster/w4abucpbwuxx"
    }
  }
},
{
  "Sid": "Enable dsql IAM User Describe Permissions",
  "Effect": "Allow",
  "Principal": {
    "Service": "dsql.amazonaws.com"
  },
  "Action": "kms:DescribeKey",
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "aws:SourceArn": "arn:aws:dsql:us-east-2:111122223333:cluster/w4abucpbwuxx"
    }
  }
}
```

Aurora DSQL 加密內容

加密內容是一組金鑰/值對，其中包含任意非私密資料。當您在加密資料的請求中包含加密內容時， 會以 AWS KMS 加密方式將加密內容繫結至加密的資料。若要解密資料，您必須傳遞相同的加密內容。

Aurora DSQL 在所有 AWS KMS 密碼編譯操作中使用相同的加密內容。如果您使用客戶受管金鑰來保護 Aurora DSQL 叢集，您可以使用加密內容來識別在稽核記錄和日誌 AWS KMS key 中使用 。加密內容也會以純文字顯示在日誌中，例如 AWS CloudTrail中的加密內容。

加密內容也可以用作政策中的授權條件。

在對 的請求中 AWS KMS，Aurora DSQL 會使用具有金鑰/值對的加密內容：

```
"encryptionContext": {
  "aws:dsql:ClusterId": "w4abucpbwuxx"
},
```

鍵值對可用於識別 Aurora DSQL 加密的叢集。金鑰為 `aws:dsql:ClusterId`。值是叢集的識別符。

監控 Aurora DSQL 與 的互動 AWS KMS

如果您使用客戶受管金鑰來保護 Aurora DSQL 叢集，您可以使用 AWS CloudTrail 日誌來追蹤 Aurora DSQL 代表您傳送到 AWS KMS 的請求。

展開下列各節，以了解 Aurora DSQL 如何使用 AWS KMS 操作`GenerateDataKey`和 `Decrypt`。

GenerateDataKey

您在叢集上啟用靜態加密時，Aurora DSQL 會建立唯一的叢集金鑰。它會將`GenerateDataKey`請求傳送至 AWS KMS ，以指定叢集的 AWS KMS key 。

記錄 `GenerateDataKey` 操作的事件類似於以下範例事件。使用者是 Aurora DSQL 服務帳戶。參數包括 的 Amazon Resource Name (ARN) AWS KMS key、需要 256 位元金鑰的金鑰指標，以及識別叢集的加密內容。

```
{
    "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"
}
```


解密

您存取加密的 Aurora DSQL 叢集時，Aurora DSQL 需要解密叢集金鑰以便解密階層下方的金鑰。接著會解密叢集中的資料。若要解密叢集金鑰，Aurora DSQL 會傳送`Decrypt`請求至 AWS KMS ，以指定叢集 AWS KMS key 的 。

記錄 `Decrypt` 操作的事件類似於以下範例事件。使用者是 中存取叢集的委託 AWS 帳戶 人。這些參數包括加密的叢集金鑰（做為加密文字 Blob) 和可識別叢集的加密內容。 會從加密文字 AWS KMS 衍生 的 AWS KMS key ID。


```
{
  "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"
}
```


建立加密的 Aurora DSQL 叢集

所有 Aurora DSQL 叢集都會靜態加密。根據預設，叢集 AWS 擁有的金鑰 可免費使用 ，或者您可以指定自訂 AWS KMS 金鑰。請依照下列步驟，從 AWS 管理主控台 或 建立您的加密叢集 AWS CLI。

Console

在 中建立加密叢集 AWS 管理主控台

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。

2. 在主控台左側的導覽窗格中選擇叢集。
3. 選擇右上角的建立叢集，然後選取單一區域。
4. 在叢集加密設定中選擇下列其中一個選項。
   - 接受預設設定以使用 AWS 擁有的金鑰 加密，無需額外費用。
   - 選擇自訂加密設定 (進階) 以指定自訂 KMS 金鑰。然後請搜尋或輸入 KMS 金鑰的 ID 或別名。或者，選擇建立 AWS KMS 金鑰以在 AWS KMS 主控台中建立新的金鑰。
5. 選擇 Create Cluster (建立叢集)。

若要確認叢集的加密類型，請導覽至叢集頁面，然後選取叢集 ID 以檢視叢集詳細資訊。檢閱叢集設定索引標籤，叢集 KMS 金鑰設定會顯示 Aurora DSQL 預設金鑰，適用於使用 AWS 擁有的金鑰或其他加密類型的金鑰 ID 的叢集。

> **Note**
> 如果您選擇擁有和管理自己的金鑰，請確保已正確設定 KMS 金鑰政策。範例和詳細資訊請參閱 the section called "客戶自管金鑰的金鑰政策"。

CLI

建立使用預設 加密的叢集 AWS 擁有的金鑰

- 使用下列命令建立 Aurora DSQL 叢集。

```
aws dsql create-cluster
```

如下列加密詳細資訊所示，叢集加密狀態預設為啟用，預設加密類型為 AWS 擁有的金鑰。叢集現在已使用 Aurora DSQL 服務帳戶中的預設 AWS 擁有金鑰進行加密。

```
"encryptionDetails": {
  "encryptionType" : "AWS_OWNED_KMS_KEY",
  "encryptionStatus" : "ENABLED"
}
```

建立以客戶自管金鑰加密的叢集

- 使用下列命令建立 Aurora DSQL 叢集，以客戶自管金鑰 ID 取代紅色文字的金鑰 ID。

```
aws dsql create-cluster \
--kms-encryption-key d41d8cd98f00b204e9800998ecf8427e
```

如下列加密詳細資訊所示，叢集加密狀態預設為啟用，加密類型是客戶受管 KMS 金鑰。叢集現在已使用您的金鑰加密。

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:kms:us-east-1:111122223333:key/
d41d8cd98f00b204e9800998ecf8427e",
  "encryptionStatus" : "ENABLED"
}
```

移除或更新 Aurora DSQL 叢集的金鑰

您可以使用 AWS 管理主控台 或 AWS CLI 來更新或移除 Amazon Aurora DSQL 中現有叢集上的加密金鑰。如果您在未取代金鑰的情況下移除金鑰，Aurora DSQL 會使用預設的 AWS 擁有的金鑰。請依照下列步驟從 Aurora DSQL 主控台或 AWS CLI更新現有叢集的加密金鑰。

Console

若要更新或移除 中的加密金鑰 AWS 管理主控台

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。
2. 在主控台左側的導覽窗格中選擇叢集。
3. 從清單檢視中尋找並選取您要更新的叢集列。
4. 選擇動作功能表，然後選擇修改。
5. 在叢集加密設定中選擇下列其中一個選項以修改加密設定。
    - 如果您想要從自訂金鑰切換至 AWS 擁有的金鑰，請取消選取自訂加密設定 （ 進階 ） 選項。預設設定將 AWS 擁有的金鑰 免費套用和加密 叢集。

- 如果您想要從自訂 KMS 金鑰切換到另一個金鑰，或從 AWS 擁有的金鑰 切換到 KMS 金鑰，請選取自訂加密設定 (進階) 選項 (如果尚未選取)。然後搜尋並選取您要使用的金鑰 ID 或別名。或者，選擇建立 AWS KMS 金鑰以在主控台中 AWS KMS 建立新的金鑰。

6. 選擇儲存。

CLI

下列範例示範如何使用 AWS CLI 更新加密的叢集。

使用預設值更新加密叢集 AWS 擁有的金鑰

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key "AWS_OWNED_KMS_KEY"
```

叢集說明的 `EncryptionStatus` 狀態會設為 `ENABLED`，且 `EncryptionType` 為 `AWS_OWNED_KMS_KEY`。

```
"encryptionDetails": {
  "encryptionType" : "AWS_OWNED_KMS_KEY",
  "encryptionStatus" : "ENABLED"
}
```

此叢集現在已使用 AWS 擁有的金鑰 Aurora DSQL 服務帳戶中的預設值加密。

使用 Aurora DSQL 的客戶自管金鑰更新加密叢集

如下列範例所示更新加密叢集：

```
aws dsql update-cluster \
--identifier aiabtx6icfp6d53snkhseduiqq \
--kms-encryption-key arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-ab1234a1b234
```

叢集說明的 EncryptionStatus 會轉換為 UPDATING，且 EncryptionType 為 CUSTOMER_MANAGED_KMS_KEY。Aurora DSQL 透過平台完成傳播新金鑰後，加密狀態將轉換為 ENABLED

```
"encryptionDetails": {
  "encryptionType" : "CUSTOMER_MANAGED_KMS_KEY",
  "kmsKeyArn" : "arn:aws:us-east-1:kms:key/abcd1234-abcd-1234-a123-ab1234a1b234",
  "encryptionStatus" : "ENABLED"
}
```

> **Note**
>
> 如果您選擇擁有和管理自己的金鑰，請確保已正確設定 KMS 金鑰政策。範例和詳細資訊請參閱 the section called "客戶自管金鑰的金鑰政策"。

使用 Aurora DSQL 進行加密的考量事項

- Aurora DSQL 會加密所有叢集靜態資料。您無法停用此加密或僅加密叢集中的部分項目。
- AWS Backup 會加密您的備份，以及從這些備份還原的任何叢集。您可以使用 AWS 擁有 AWS Backup 的金鑰或客戶受管金鑰，在 中加密備份資料。
- Aurora DSQL 已啟用下列資料保護狀態：
  - 靜態資料：Aurora DSQL 會加密持久性儲存媒體上的所有靜態資料
  - 傳輸中的資料：Aurora DSQL 預設使用 Transport Layer Security (TLS) 加密所有通訊
- 當您轉換到不同的金鑰時，我們建議您保持啟用原始金鑰，直到轉換完成為止。 AWS 需要原始金鑰來解密資料，才能使用新的金鑰加密您的資料。叢集的 encryptionStatus 為 ENABLED 且您看到新客戶自管金鑰的 kmsKeyArn 時即代表流程完成。
- 您停用客戶自管金鑰或撤銷 Aurora DSQL 使用金鑰的存取權限時，您的叢集將進入 IDLE 狀態。
- AWS 管理主控台 和 Amazon Aurora DSQL API 對加密類型使用不同的術語：
  - AWS 主控台 – 在主控台中，您會KMS在使用客戶受管金鑰和使用 DEFAULT時看到 AWS 擁有的金鑰。
  - API：Amazon Aurora DSQL API 使用 CUSTOMER_MANAGED_KMS_KEY 用於客戶自管金鑰，並使用 AWS_OWNED_KMS_KEY 用於 AWS 擁有的金鑰。

- 如果您在叢集建立期間未指定加密金鑰，Aurora DSQL 會使用 自動加密您的資料 AWS 擁有的金鑰。
- 您可以隨時在 AWS 擁有的金鑰 和客戶受管金鑰之間切換。使用 AWS 管理主控台 AWS CLI或 Amazon Aurora DSQL API 進行此變更。

Aurora DSQL 的身分與存取管理

AWS Identity and Access Management (IAM) 是一種 AWS 服務 ，可協助管理員安全地控制對 AWS 資源的存取。IAM 管理員可以控制誰能完成身分驗證 (登入) 和獲得授權 (取得許可權限)，而得以使用 Aurora DSQL 資源。IAM 是 AWS 服務 您可以免費使用的 。

主題

- 目標對象
- 使用身分驗證
- 使用政策管理存取權
- Amazon Aurora DSQL 如何搭配使用 IAM
- Amazon Aurora DSQL 的身分型政策範例
- 對 Amazon Aurora DSQL 身分與存取進行故障診斷

目標對象

使用方式 AWS Identity and Access Management (IAM) 會根據您的角色而有所不同：

- 服務使用者 — 若無法存取某些功能，請向管理員申請所需許可 (請參閱 對 Amazon Aurora DSQL 身分與存取進行故障診斷)
- 服務管理員 — 負責設定使用者存取權並提交相關許可請求 (請參閱 Amazon Aurora DSQL 如何搭配使用 IAM)
- IAM 管理員 — 撰寫政策以管理存取控制 (請參閱 Amazon Aurora DSQL 的身分型政策範例)

使用身分驗證

身分驗證是您 AWS 使用身分憑證登入 的方式。您必須以 AWS 帳戶根使用者、IAM 使用者或擔任 IAM 角色身分進行身分驗證。

您可以使用身分來源的登入資料，例如 AWS IAM Identity Center (IAM Identity Center)、單一登入身分驗證或 Google/Facebook 登入資料，以聯合身分的形式登入。如需有關登入的詳細資訊，請參閱《AWS 登入 使用者指南》中的如何登入您的 AWS 帳戶。

對於程式設計存取， AWS 提供 SDK 和 CLI 以密碼編譯方式簽署請求。如需詳細資訊，請參閱《IAM 使用者指南》中的 API 請求的AWS 第 4 版簽署程序。

AWS 帳戶 根使用者

當您建立 時 AWS 帳戶，您會從一個名為 AWS 帳戶 theroot 使用者的登入身分開始，該身分可完整存取所有 AWS 服務 和 資源。強烈建議不要使用根使用者來執行日常任務。有關需要根使用者憑證的任務，請參閱《IAM 使用者指南》中的需要根使用者憑證的任務。

聯合身分

最佳實務是要求人類使用者使用聯合身分提供者，以 AWS 服務 使用臨時憑證存取 。

聯合身分是來自您企業目錄、Web 身分提供者的使用者，或使用來自身分來源的 AWS 服務 憑證存取 Directory Service 的使用者。聯合身分會擔任角色，而該角色會提供臨時憑證。

若需集中化管理存取權限，建議使用 AWS IAM Identity Center。如需詳細資訊，請參閱 AWS IAM Identity Center 使用者指南中的什麼是 IAM Identity Center？。

IAM 使用者和群組

IAM 使用者https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html是一種身分具備單人或應用程式的特定許可權。建議以臨時憑證取代具備長期憑證的 IAM 使用者。如需詳細資訊，請參閱《IAM 使用者指南》中的要求人類使用者使用聯合身分提供者來 AWS 使用臨時憑證存取 。

IAM 群組會指定 IAM 使用者集合，使管理大量使用者的許可權更加輕鬆。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 使用者的使用案例。

IAM 角色

IAM 角色https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html的身分具有特定許可權，其可以提供臨時憑證。您可以透過從使用者切換到 IAM 角色 （ 主控台 ）或呼叫 AWS CLI 或 AWS API 操作來擔任角色。如需詳細資訊，請參閱《IAM 使用者指南》中的擔任角色的方法。

IAM 角色適用於聯合身分使用者存取、臨時 IAM 使用者許可、跨帳戶存取權與跨服務存取，以及在 Amazon EC2 執行的應用程式。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 中的快帳戶資源存取。

使用政策管理存取權

您可以透過建立政策並將其連接到身分或資源 AWS 來控制 AWS 中的存取。政策定義與身分或資源相關聯的許可。當委託人提出請求時 AWS ， 會評估這些政策。大多數政策會以 JSON 文件 AWS 形式存放在 中。如需進一步了解 JSON 政策文件，請參閱《IAM 使用者指南》中的 JSON 政策概觀。

管理員會使用政策，透過定義哪些主體可在哪些條件下對哪些資源執行動作，以指定可存取的範圍。

預設情況下，使用者和角色沒有許可。IAM 管理員會建立 IAM 政策並將其新增至角色，供使用者後續擔任。IAM 政策定義動作的許可，無論採用何種方式執行。

身分型政策

身分型政策是附加至身分 (使用者、使用者群組或角色) 的 JSON 許可政策文件。這類政策控制身分可對哪些資源執行哪些動作，以及適用的條件。如需了解如何建立身分型政策，請參閱《IAM 使用者指南》中的透過客戶管理政策定義自訂 IAM 許可。

身分型政策可分為內嵌政策 (直接內嵌於單一身分) 與受管政策 (可附加至多個身分的獨立政策)。如需了解如何在受管政策及內嵌政策之間做選擇，請參閱《IAM 使用者指南》中的在受管政策與內嵌政策之間選擇。

資源型政策

資源型政策是附加到資源的 JSON 政策文件。範例包括 IAM 角色信任政策與 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。您必須在資源型政策中指定主體。

資源型政策是位於該服務中的內嵌政策。您無法在以資源為基礎的政策中使用來自 IAM 的 AWS 受管政策。

其他政策類型

AWS 支援其他政策類型，可設定更多常見政策類型授予的最大許可：

- 許可界限 — 設定身分型政策可授與 IAM 實體的最大許可。如需詳細資訊，請參閱《 IAM 使用者指南》中的 IAM 實體許可界限。
- 服務控制政策 (SCP) — 為 AWS Organizations中的組織或組織單位指定最大許可。如需詳細資訊，請參閱《AWS Organizations 使用者指南》中的服務控制政策。
- 資源控制政策 (RCP) — 設定您帳戶中資源可用許可的上限。如需詳細資訊，請參閱《AWS Organizations 使用者指南》中的資源控制政策 (RCP)。

- 工作階段政策 — 在以程式設計方式為角色或聯合身分使用者建立臨時工作階段時，以參數形式傳遞的進階政策。如需詳細資訊，請參《IAM 使用者指南》中的工作階段政策。

多種政策類型

當多種類型的政策套用到請求時，產生的許可會更複雜而無法理解。若要了解如何 AWS 在涉及多個政策類型時決定是否允許請求，請參閱《IAM 使用者指南》中的政策評估邏輯。

Amazon Aurora DSQL 如何搭配使用 IAM

在您使用 IAM 管理 Aurora DSQL 的存取權之前，請瞭解哪些 IAM 功能可搭配使用 Aurora DSQL。

您可以搭配 Amazon Aurora DSQL 使用的 IAM 功能

IAM 功能	Aurora DSQL 支援
身分型政策	是
資源型政策	是
政策動作	是
政策資源	是
政策條件索引鍵	是
ACL	否
ABAC (政策中的標籤)	是
臨時憑證	是
主體許可	是
服務角色	是
服務連結角色	是

若要全面了解 Aurora DSQL 和其他 AWS 服務如何與大多數 IAM 功能搭配使用，請參閱《IAM 使用者指南》中的AWS 與 IAM 搭配使用的 服務。

適用於 Aurora DSQL 的身分型政策

支援身分型政策：是

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。如需了解如何建立身分型政策，請參閱《IAM 使用者指南》中的透過客戶管理政策定義自訂 IAM 許可。

使用 IAM 身分型政策，您可以指定允許或拒絕的動作和資源，以及在何種條件下允許或拒絕動作。如要了解您在 JSON 政策中使用的所有元素，請參閱 IAM 使用者指南中的 IAM JSON 政策元素參考。

Aurora DSQL 的身分型政策範例

如需檢視 Aurora DSQL 身分型政策範例，請參閱 Amazon Aurora DSQL 的身分型政策範例。

Aurora DSQL 內的資源型政策

支援資源型政策：是

資源型政策是附加到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。主體可以包括帳戶、使用者、角色、聯合身分使用者或 AWS 服務。資源型政策是位於該服務中的內嵌政策。您無法在以資源為基礎的原則中使用 IAM 的 AWS 受管原則。

若要了解如何建立和管理 Aurora DSQL 叢集的資源型政策，請參閱 Aurora DSQL 的資源型政策。

Aurora DSQL 的政策動作

支援政策動作：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

JSON 政策的 `Action` 元素描述您可以用來允許或拒絕政策中存取的動作。政策會使用動作來授予執行相關聯動作的許可。

如要查看 Aurora DSQL 動作清單，請參閱服務授權參考中的 Amazon Aurora DSQL 定義動作。

Aurora DSQL 中的政策動作會在動作之前使用以下字首：

```
dsql
```

若要在單一陳述式中指定多個動作，請用逗號分隔。

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

如需檢視 Aurora DSQL 身分型政策範例，請參閱 Amazon Aurora DSQL 的身分型政策範例。

Aurora DSQL 的政策資源

支援政策資源：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

`Resource` JSON 政策元素可指定要套用動作的物件。最佳實務是使用其 Amazon Resource Name (ARN) 來指定資源。若動作不支援資源層級許可，使用萬用字元 (*) 表示該陳述式適用於所有資源。

```
"Resource": "*"
```

若要查看 Aurora DSQL 資源類型清單及其 ARN，請參閱服務授權參考中的 Amazon Aurora DSQL 定義資源。若要瞭解您可以使用哪些動作指定每個資源的 ARN，請參閱 Amazon Aurora DSQL 定義的動作。

如需檢視 Aurora DSQL 身分型政策範例，請參閱 Amazon Aurora DSQL 的身分型政策範例。

Aurora DSQL 的政策條件索引鍵

支援服務特定政策條件金鑰：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

`Condition` 元素會根據定義的條件，指定陳述式的執行時機。您可以建立使用條件運算子的條件運算式 (例如等於或小於)，來比對政策中的條件和請求中的值。若要查看所有 AWS 全域條件索引鍵，請參閱《IAM 使用者指南》中的AWS 全域條件內容索引鍵。

若要查看 Aurora DSQL 條件索引鍵清單，請參閱服務授權參考中的 Amazon Aurora DSQL 條件索引鍵。若要瞭解哪些動作和資源可搭配使用條件索引鍵，請參閱 Amazon Aurora DSQL 定義的動作。

如需檢視 Aurora DSQL 身分型政策範例，請參閱 Amazon Aurora DSQL 的身分型政策範例。

Aurora DSQL 中的 ACL

支援 ACL：否

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

ABAC 搭配 Aurora DSQL

支援 ABAC (政策中的標籤)：是

屬性型存取控制 (ABAC) 是一種授權策略，依據稱為標籤的屬性來定義許可。您可以將標籤連接至 IAM 實體 AWS 和資源，然後設計 ABAC 政策，以便在委託人的標籤符合資源上的標籤時允許操作。

如需根據標籤控制存取，請使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 條件索引鍵，在政策的條件元素中，提供標籤資訊。

如果服務支援每個資源類型的全部三個條件金鑰，則對該服務而言，值為 Yes。如果服務僅支援某些資源類型的全部三個條件金鑰，則值為 Partial。

如需 ABAC 的詳細資訊，請參閱《IAM 使用者指南》中的使用 ABAC 授權定義許可。如要查看含有設定 ABAC 步驟的教學課程，請參閱《IAM 使用者指南》中的使用屬性型存取控制 (ABAC)。

使用臨時憑證搭配 Aurora DSQL

支援臨時憑證：是

臨時登入資料提供 AWS 資源的短期存取權，並在您使用聯合或切換角色時自動建立。 AWS 建議您動態產生臨時登入資料，而不是使用長期存取金鑰。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 中的臨時安全憑證與可與 IAM 搭配運作的AWS 服務。

Aurora DSQL 的跨服務主體許可權限

支援轉寄存取工作階段 (FAS)：是

轉送存取工作階段 (FAS) 使用呼叫 的委託人許可 AWS 服務，並結合 請求 AWS 服務 向下游服務提出請求。如需提出 FAS 請求時的政策詳細資訊，請參閱轉發存取工作階段。

Aurora DSQL 的服務角色

支援服務角色：是

服務角色是服務擔任的 IAM 角色，可代您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱 IAM 使用者指南中的建立角色以委派許可權給 AWS 服務。

Warning

變更服務角色的許可權限有可能會讓 Aurora DSQL 功能出現故障。只有在 Aurora DSQL 提供指引時，才能編輯服務角色。

適用於 Aurora DSQL 的服務連結角色

支援服務連結角色：是

服務連結角色是連結至 的一種服務角色 AWS 服務。服務可以擔任代表您執行動作的角色。服務連結角色會出現在您的 中 AWS 帳戶 ，並由服務擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。

如需建立或管理 Aurora DSQL 服務連結角色的詳細資訊，請參閱 在 Aurora DSQL 中使用服務連結角色。

Amazon Aurora DSQL 的身分型政策範例

根據預設，使用者和角色不具備建立或修改 Aurora DSQL 資源的許可權限。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。

如需了解如何使用這些範例 JSON 政策文件建立 IAM 身分型政策，請參閱 IAM 使用者指南中的建立 IAM 政策 (主控台)。

如需 Aurora DSQL 所定義之動作和資源類型的詳細資訊，包括每種資源類型的 ARN 格式，請參閱服務授權參考的適用於 Amazon Aurora DSQL 的動作、資源和條件索引鍵。

主題

- 政策最佳實務
- 使用 Aurora DSQL 主控台
- 允許使用者檢視他們自己的許可

- 允許叢集管理和資料庫連線
- 根據標籤的 Aurora DSQL 資源存取

政策最佳實務

身分型政策會判斷您帳戶中的人員是否可以建立、存取或刪除 Aurora DSQL 資源。這些動作可能會讓您的 AWS 帳戶產生費用。當您建立或編輯身分型政策時，請遵循下列準則及建議事項：

- 開始使用 AWS 受管政策並邁向最低權限許可 – 若要開始將許可授予您的使用者和工作負載，請使用將許可授予許多常見使用案例的 AWS 受管政策。它們可在您的 中使用 AWS 帳戶。我們建議您定義特定於使用案例 AWS 的客戶受管政策，進一步減少許可。如需更多資訊，請參閱《IAM 使用者指南》中的 AWS 受管政策或任務職能的AWS 受管政策。
- 套用最低權限許可 – 設定 IAM 政策的許可時，請僅授予執行任務所需的許可。為實現此目的，您可以定義在特定條件下可以對特定資源採取的動作，這也稱為最低權限許可。如需使用 IAM 套用許可的更多相關資訊，請參閱《IAM 使用者指南》中的 IAM 中的政策和許可。
- 使用 IAM 政策中的條件進一步限制存取權 – 您可以將條件新增至政策，以限制動作和資源的存取。例如，您可以撰寫政策條件，指定必須使用 SSL 傳送所有請求。如果透過特定 例如 使用服務動作 AWS 服務，您也可以使用條件來授予其存取權 CloudFormation。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素：條件。
- 使用 IAM Access Analyzer 驗證 IAM 政策，確保許可安全且可正常運作 – IAM Access Analyzer 驗證新政策和現有政策，確保這些政策遵從 IAM 政策語言 (JSON) 和 IAM 最佳實務。IAM Access Analyzer 提供 100 多項政策檢查及切實可行的建議，可協助您撰寫安全且實用的政策。如需詳細資訊，請參閱《IAM 使用者指南》中的使用 IAM Access Analyzer 驗證政策。
- 需要多重要素驗證 (MFA) – 如果您的案例需要 IAM 使用者或 中的根使用者 AWS 帳戶，請開啟 MFA 以提高安全性。如需在呼叫 API 操作時請求 MFA，請將 MFA 條件新增至您的政策。如需詳細資訊，請參閱《IAM 使用者指南》中的透過 MFA 的安全 API 存取。

如需 IAM 中最佳實務的相關資訊，請參閱《IAM 使用者指南》中的 IAM 安全最佳實務。

使用 Aurora DSQL 主控台

若要存取 Amazon Aurora DSQL 主控台，您必須擁有最基本的一組許可權限。這些許可必須允許您列出和檢視 中 Aurora DSQL 資源的詳細資訊 AWS 帳戶。如果您建立比最基本必要許可更嚴格的身分型政策，則對於具有該政策的實體 (使用者或角色) 而言，主控台就無法如預期運作。

對於僅呼叫 AWS CLI 或 AWS API 的使用者，您不需要允許最低主控台許可。反之，只需允許存取符合他們嘗試執行之 API 操作的動作就可以了。

為了確保使用者和角色仍然可以使用 Aurora DSQL 主控台，請將 Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess`或 `AmazonAuroraDSQLReadOnlyAccess` AWS 受管政策連接到實體。如需詳細資訊，請參閱 IAM 使用者指南中的新增許可到使用者。

允許使用者檢視他們自己的許可

此範例會示範如何建立政策，允許 IAM 使用者檢視附加到他們使用者身分的內嵌及受管政策。此政策包含在主控台或使用 或 AWS CLI AWS API 以程式設計方式完成此動作的許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

允許叢集管理和資料庫連線

下列政策會授予 IAM 使用者許可，以管理和連線至特定的 Aurora DSQL 叢集。政策會將叢集管理和連線動作範圍限定為單一叢集 Amazon Resource Name (ARN)，同時允許所有資源`dsql:ListClusters`，因為此動作不支援資源層級許可。

此範例使用 `dsql:DbConnectAdmin` 與 `admin`角色連線。若要改為使用自訂資料庫角色連線，請將取代`dsql:DbConnectAdmin`為 `dsql:DbConnect`。如需詳細資訊，請參閱Aurora DSQL 的身分驗證和授權。

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": "*"
        }
    ]
}
```

根據標籤的 Aurora DSQL 資源存取

您可以在身分型政策中使用條件，根據標籤控制對 Aurora DSQL 資源的存取。下列範例示範如何建立允許檢視叢集的政策。不過，只有在叢集標籤Owner具有該使用者名稱的值時，政策才會授予許可。此政策也會授予在主控台完成此動作的必要許可。

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}"
                }
            }
        }
    ]
}
```

您可以將此政策連接到您帳戶中的 IAM 使用者。如果名為 的使用者richard-roe嘗試檢視 Aurora DSQL 叢集，則叢集必須加上標籤 Owner=richard-roe或 owner=richard-roe。否則 IAM 會拒絕存取。條件標籤鍵 Owner 符合 Owner 和 owner，因為條件索引鍵名稱不區分大小寫。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素：條件。

下列政策允許使用者建立叢集，前提是叢集使用自己的使用者名稱標記為 Owner。它也僅允許在使用者已擁有的叢集上進行標記。

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}"
                }
            }
        }
    ]
}
```

對 Amazon Aurora DSQL 身分與存取進行故障診斷

請使用以下資訊協助您診斷和修正使用 Aurora DSQL 和 IAM 時的常見問題。

主題

- 我未獲授權，不得在 Aurora DSQL 中執行動作
- 我未獲得執行 iam:PassRole 的授權
- 我想要允許 以外的人員 AWS 帳戶 存取我的 Aurora DSQL 資源

我未獲授權，不得在 Aurora DSQL 中執行動作

如果您收到錯誤，告知您未獲授權執行動作，您的政策必須更新，允許您執行動作。

下列範例錯誤會在 mateojackson 嘗試使用主控台檢視 *my-dsql-cluster* 資源詳細資訊，但並無 *GetCluster* 許可權限時發生。

```
User: iam:::user/mateojackson is not authorized to perform: GetCluster on resource: my-dsql-cluster
```

在此情況下，必須更新 mateojackson 使用者的政策，允許使用 *GetCluster* 動作存取 *my-dsql-cluster* 資源。

如需任何協助，請聯絡您的 管理員。您的管理員提供您的簽署憑證。

我未獲得執行 iam:PassRole 的授權

如果錯誤訊息告知您未獲得授權，無法執行 iam:PassRole 動作，您的政策就必須更新，允許您將角色傳遞給 Aurora DSQL。

有些 AWS 服務 可讓您將現有角色傳遞給該服務，而不是建立新的服務角色或服務連結角色。如需執行此作業，您必須擁有將角色傳遞至該服務的許可。

名為 marymajor 的 IAM 使用者嘗試使用主控台在 Aurora DSQL 中執行動作時，發生下列範例錯誤。但是，動作請求服務具備服務角色授予的許可。Mary 沒有將角色傳遞給服務的許可。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在這種情況下，Mary 的政策必須更新，允許她執行 iam:PassRole 動作。

如果您需要協助，請聯絡您的 AWS 管理員。您的管理員提供您的簽署憑證。

我想要允許 以外的人員 AWS 帳戶 存取我的 Aurora DSQL 資源

您可以建立一個角色，讓其他帳戶中的使用者或您組織外部的人員存取您的資源。您可以指定要允許哪些信任物件取得該角色。針對支援基於資源的政策或存取控制清單 (ACL) 的服務，您可以使用那些政策來授予人員存取您的資源的許可。

如需進一步了解，請參閱以下內容：

- 若要瞭解 Aurora DSQL 是否支援這些功能，請參閱 Amazon Aurora DSQL 如何搭配使用 IAM。
- 若要了解如何提供您擁有 AWS 帳戶 的資源存取權，請參閱《IAM 使用者指南》中的在您擁有 AWS 帳戶 的另一個 中為 IAM 使用者提供存取權。
- 若要了解如何將資源的存取權提供給第三方 AWS 帳戶，請參閱《IAM 使用者指南》中的將存取權提供給第三方 AWS 帳戶 擁有。
- 如需了解如何透過聯合身分提供存取權，請參閱《IAM 使用者指南》中的將存取權提供給在外部進行身分驗證的使用者 (聯合身分)。
- 如需了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱《IAM 使用者指南》中的 IAM 中的跨帳戶資源存取。

Aurora DSQL 的資源型政策

使用 Aurora DSQL 的資源型政策，透過直接連接到叢集資源的 JSON 政策文件來限制或授予對叢集的存取。這些政策可精細控制誰可以存取您的叢集，以及在哪些條件下存取您的叢集。

預設可從公有網際網路存取 Aurora DSQL 叢集，並以 IAM 身分驗證做為主要安全控制。以資源為基礎的政策可讓您新增存取限制，尤其是封鎖來自公有網際網路的存取。

以資源為基礎的政策可與 IAM 身分為基礎的政策搭配使用。 會 AWS 評估這兩種類型的政策，以判斷對叢集的任何存取請求的最終許可。根據預設，可在 帳戶中存取 Aurora DSQL 叢集。如果 IAM 使用者或角色具有 Aurora DSQL 許可，他們可以存取未連接資源型政策的叢集。

> **Note**
>
> 資源型政策的變更最終會一致，且通常會在一分鐘內生效。

如需身分型政策與資源型政策之間差異的詳細資訊，請參閱《IAM 使用者指南》中的身分型政策和資源型政策。

何時使用資源型政策

以資源為基礎的政策在這些案例中特別有用：

- 網路型存取控制 — 根據請求來源的 VPC 或 IP 地址來限制存取，或完全封鎖公有網際網路存取。使用 `aws:SourceVpc`和 等條件金鑰`aws:SourceIp`來控制網路存取。

- 多個團隊或應用程式 — 授予多個團隊或應用程式的相同叢集存取權。您可以在叢集上定義存取規則一次，而不是管理每個委託人的個別 IAM 政策。
- 複雜條件式存取 — 根據多個因素控制存取，例如網路屬性、請求內容和使用者屬性。您可以在單一政策中結合多個條件。
- 集中式安全控管 — 讓叢集擁有者使用與現有安全實務整合的熟悉 AWS 政策語法來控制存取。

> **Note**
>
> Aurora DSQL 資源型政策尚未支援跨帳戶存取，但將在未來版本中提供。

當有人嘗試連線到您的 Aurora DSQL 叢集時， 會 AWS 評估您的資源型政策，做為授權內容的一部分，以及任何相關的 IAM 政策，以判斷是否應允許或拒絕請求。

以資源為基礎的政策可以將存取權授予與叢集相同 AWS 帳戶內的主體。對於多區域叢集，每個區域叢集都有自己的資源型政策，允許在需要時進行區域特定的存取控制。

> **Note**
>
> 條件內容索引鍵可能因區域（例如 VPC IDs) 而異。

主題

- 使用資源型政策建立叢集
- 新增和編輯叢集的資源型政策
- 檢視資源型政策
- 移除資源型政策
- 常見的資源型政策範例
- 使用 Aurora DSQL 中的資源型政策封鎖公開存取
- Aurora DSQL API 操作和資源型政策

使用資源型政策建立叢集

您可以在建立新叢集時連接資源型政策，以確保存取控制從一開始就就緒。每個叢集可以有一個直接連接到叢集的內嵌政策。

AWS 管理主控台

在叢集建立期間新增資源型政策

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。
2. 選擇 Create Cluster (建立叢集)。
3. 視需要設定叢集名稱、標籤和多區域設定。
4. 在叢集設定區段中，找到以資源為基礎的政策選項。
5. 開啟新增以資源為基礎的政策。
6. 在 JSON 編輯器中輸入您的政策文件。例如，若要封鎖公有網際網路存取：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

7. 您可以使用編輯陳述式或新增陳述式來建置您的政策。
8. 完成剩餘的叢集組態，然後選擇建立叢集。

AWS CLI

建立叢集時，請使用 `--policy` 參數來連接內嵌政策：

```
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 SDKs

Python

```
import boto3
import json

client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "StringNotEquals": { "aws:SourceVpc": "vpc-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());
```

新增和編輯叢集的資源型政策

AWS 管理主控台

將資源型政策新增至現有叢集

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。
2. 從叢集清單中選擇叢集，以開啟叢集詳細資訊頁面。
3. 選擇許可索引標籤。

4. 在資源型政策區段中，選擇新增政策。
5. 在 JSON 編輯器中輸入您的政策文件。您可以使用編輯陳述式或新增陳述式來建置您的政策。
6. 選擇 Add Policy (新增政策)。

編輯現有的資源型政策

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。
2. 從叢集清單中選擇叢集，以開啟叢集詳細資訊頁面。
3. 選擇許可索引標籤。
4. 在資源型政策區段中，選擇編輯。
5. 在 JSON 編輯器中修改政策文件。您可以使用編輯陳述式或新增陳述式來更新您的政策。
6. 選擇儲存變更。

AWS CLI

使用 `put-cluster-policy`命令來連接新政策或更新叢集上的現有政策：

```
aws dsql put-cluster-policy --identifier your_cluster_id --policy '{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": { "aws:SourceVpc": "true" }
        }
    }]
}'
```

AWS SDKs

Python

```
import boto3
import json
```

```
client = boto3.client('dsql')

policy = {
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Principal": {"AWS": "*"},
        "Resource": "*",
        "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
        "Condition": {
            "Null": {"aws:SourceVpc": "true"}
        }
    }]
}

response = client.put_cluster_policy(
    identifier='your_cluster_id',
    policy=json.dumps(policy)
)
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.PutClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

String policy = """
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Principal": {"AWS": "*"},
    "Resource": "*",
    "Action": ["dsql:DbConnect", "dsql:DbConnectAdmin"],
    "Condition": {
      "Null": {"aws:SourceVpc": "true"}
    }
  }]
}
```

```
""";

PutClusterPolicyRequest request = PutClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .policy(policy)
    .build();

client.putClusterPolicy(request);
```

檢視資源型政策

您可以檢視連接至叢集的資源型政策，以了解目前的存取控制。

AWS 管理主控台

檢視資源型政策

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。
2. 從叢集清單中選擇叢集，以開啟叢集詳細資訊頁面。
3. 選擇許可索引標籤。
4. 在以資源為基礎的政策區段中檢視附加的政策。

AWS CLI

使用 `get-cluster-policy`命令來檢視叢集的資源型政策：

```
aws dsql get-cluster-policy --identifier your_cluster_id
```

AWS SDKs

Python

```
import boto3
import json

client = boto3.client('dsql')
```

```
response = client.get_cluster_policy(
    identifier='your_cluster_id'
)

# Parse and pretty-print the policy
policy = json.loads(response['policy'])
print(json.dumps(policy, indent=2))
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterPolicyResponse;

DsqlClient client = DsqlClient.create();

GetClusterPolicyRequest request = GetClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

GetClusterPolicyResponse response = client.getClusterPolicy(request);
System.out.println("Policy: " + response.policy());
```

移除資源型政策

您可以從叢集移除資源型政策，以變更存取控制。

Important

當您從叢集移除所有資源型政策時，存取權將完全由 IAM 身分型政策控制。

AWS 管理主控台

移除以資源為基礎的政策

1. 登入 AWS 管理主控台，並在 https：//https://console.aws.amazon.com/dsql/ 開啟 Aurora DSQL 主控台。

2. 從叢集清單中選擇叢集，以開啟叢集詳細資訊頁面。
3. 選擇許可索引標籤。
4. 在資源型政策區段中，選擇刪除。
5. 在確認對話方塊中，輸入 **confirm** 以確認刪除。
6. 選擇 刪除。

AWS CLI

使用 `delete-cluster-policy`命令從叢集中移除政策：

```
aws dsql delete-cluster-policy --identifier your_cluster_id
```

AWS SDKs

Python

```
import boto3

client = boto3.client('dsql')

response = client.delete_cluster_policy(
    identifier='your_cluster_id'
)

print("Policy deleted successfully")
```

Java

```
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterPolicyRequest;

DsqlClient client = DsqlClient.create();

DeleteClusterPolicyRequest request = DeleteClusterPolicyRequest.builder()
    .identifier("your_cluster_id")
    .build();

client.deleteClusterPolicy(request);
System.out.println("Policy deleted successfully");
```

常見的資源型政策範例

這些範例顯示控制 Aurora DSQL 叢集存取的常見模式。您可以結合和修改這些模式，以符合您的特定存取需求。

封鎖公有網際網路存取

此政策會封鎖從公有網際網路（非 VPC) 連線至 Aurora DSQL 叢集。政策不會指定客戶可以從哪些 VPC 連線，只有他們必須從 VPC 連線。若要限制對特定 VPC 的存取，請使用 `aws:SourceVpc`搭配 `StringEquals`條件運算子。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect",
        "dsql:DbConnectAdmin"
      ],
      "Condition": {
        "Null": {
          "aws:SourceVpc": "true"
        }
      }
    }
  ]
}
```

> **Note**
>
> 此範例僅使用 `aws:SourceVpc` 來檢查 VPC 連線。`aws:VpcSourceIp` 和 `aws:SourceVpce`條件金鑰提供額外的精細度，但對於僅限 VPC 的基本存取控制則不需要。

若要為特定角色提供例外狀況，請改用此政策：

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

限制對 AWS Organization 的存取

此政策限制對 AWS 組織內主體的存取：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect",
```

```
        "dsql:DbConnectAdmin"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalOrgID": "o-exampleorgid"
        }
      }
    }
  ]
}
```

限制對特定組織單位的存取

此政策限制對 AWS 組織中特定組織單位 (OU) 內主體的存取，提供比整個組織更精細的控制：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "dsql:DbConnect"
      ],
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/
mydsqlclusterid0123456789a",
      "Condition": {
        "StringNotLike": {
          "aws:PrincipalOrgPaths": "o-exampleorgid/r-examplerootid/ou-exampleouid/*"
        }
      }
    }
  ]
}
```

多區域叢集政策

對於多區域叢集，每個區域叢集會維護自己的資源政策，允許區域特定的控制項。以下是每個區域具有不同政策的範例：

us-east-1 政策：

```
{
  "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 政策：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*",
      "Action": [
        "dsql:DbConnect"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": "vpc-east2-id"
```

```
            }
        }
      }
    ]
}
```

> **Note**
>
> 條件內容索引鍵可能不同 AWS 區域（例如 VPC IDs)。

使用 Aurora DSQL 中的資源型政策封鎖公開存取

封鎖公開存取 (BPA) 是一種功能，可識別和防止連接以資源為基礎的政策，以授予您 AWS 帳戶間 Aurora DSQL 叢集的公開存取權。使用 BPA，您可以防止公開存取您的 Aurora DSQL 資源。BPA 會在建立或修改資源型政策期間執行檢查，並協助改善 Aurora DSQL 的安全狀態。

BPA 會使用自動推理技術分析資源型政策授予的存取權，若在管理政策時偵測到此類權限，系統會發出警示通知。分析會驗證所有以資源為基礎的政策陳述、動作，以及您政策中所使用的條件金鑰集合的存取權限。

> **Important**
>
> BPA 可防止透過直接連接到叢集等 Aurora DSQL 資源的資源型政策授予公開存取，以協助保護您的資源。除了使用 BPA 外，請仔細檢查下列政策，確認這些政策未授予公開存取權：
>
> - 連接到相關聯 AWS 主體的身分型政策（例如 IAM 角色）
> - 連接至相關聯 AWS 資源的資源型政策（例如 AWS ，Key Management Service (KMS) 金鑰）

您必須確保主體不包含 * 項目，或由指定的條件金鑰之一限制主體對資源的存取。如果以資源為基礎的政策跨 AWS 帳戶授予對叢集的公有存取權，Aurora DSQL 會阻止您建立或修改政策，直到政策中的規格更正並視為非公有。

您可以在 `Principal` 區塊內指定一個或多個主體，將政策設為非公開。下列資源型政策範例透過指定兩個主體來阻止公開存取。

```
{
  "Effect": "Allow",
```

```
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id"
}
```

指定特定條件金鑰以限制存取的政策，也不會被視為公開。除了評估資源型政策中指定的主體外，下列受信任條件金鑰也用於完成非公開存取的政策評估。

- `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`

此外，若要使資源型政策為非公開，Amazon Resource Name (ARN) 與字串金鑰的值不得包含萬用字元或變數。若您的資源型政策使用 `aws:PrincipalIsAWSService` 金鑰，您必須確保已將該金鑰值設為 true。

下列政策限制指定帳戶中使用者 `Ben` 的存取權。該條件使 `Principal` 受限，因此不被視為公開。

```
{
```

```
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dsql:*",
  "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/Ben"
    }
  }
}
```

下列非公開的資源型政策範例使用 `StringEquals` 運算子來限制 `sourceVPC`。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dsql:*",
      "Resource": "arn:aws:dsql:us-east-1:123456789012:cluster/cluster-id",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
      }
    }
  ]
}
```

Aurora DSQL API 操作和資源型政策

Aurora DSQL 中的資源型政策會控制對特定 API 操作的存取。下列各節會列出依類別組織的所有 Aurora DSQL API 操作，並指出哪些操作支援以資源為基礎的政策。

支援 RBP 欄指出當政策連接到叢集時，API 操作是否受限於資源型政策評估。

標籤 APIs

API 作業	Description	支援 RBP
ListTagsForResource	列出 Aurora DSQL 資源的標籤	是
TagResource	將標籤新增至 Aurora DSQL 資源	是
UntagResource	從 Aurora DSQL 資源移除標籤	是

叢集管理 APIs

API 作業	Description	支援 RBP
CreateCluster	建立新叢集	否
DeleteCluster	刪除叢集	是
GetCluster	擷取叢集的相關資訊	是
GetVpcEndpointServiceName	擷取叢集的 VPC 端點服務名稱	是
ListClusters	列出您帳戶中的叢集	否
UpdateCluster	更新叢集的組態	是

多區域屬性 APIs

API 作業	Description	支援 RBP
AddPeerCluster	將對等叢集新增至多區域組態	是
PutMultiRegionProperties	設定叢集的多區域屬性	是
PutWitnessRegion	設定多區域叢集的見證區域	是

資源型政策 API

API 作業	Description	支援 RBP
DeleteClusterPolicy	從叢集刪除資源型政策	是
GetClusterPolicy	擷取叢集的資源型政策	是
PutClusterPolicy	建立或更新叢集的資源型政策	是

AWS Fault Injection Service APIs

API 作業	Description	支援 RBP
InjectError	注入錯誤以進行錯誤注入測試	否

備份和還原 APIs

API 作業	Description	支援 RBP
GetBackupJob	擷取備份任務的相關資訊	否
GetRestoreJob	擷取還原任務的相關資訊	否
StartBackupJob	啟動叢集的備份任務	是
StartRestoreJob	從備份啟動還原任務	否
StopBackupJob	停止執行中的備份任務	否
StopRestoreJob	停止執行中的還原任務	否

在 Aurora DSQL 中使用服務連結角色

Aurora DSQL 使用 AWS Identity and Access Management (IAM) 服務連結角色。服務連結角色是直接連結至 Aurora DSQL 的一種特殊 IAM 角色類型。服務連結角色由 Aurora DSQL 預先定義，並包含服務 AWS 服務 代表 Aurora DSQL 叢集呼叫 所需的所有許可。

服務連結角色可簡化設定流程，因為您不必手動新增必要的許可權限以使用 Aurora DSQL。您建立叢集時，Aurora DSQL 會自動為您建立服務連結角色。只有在刪除所有叢集後，您才可以刪除服務連結角色。這樣您就不會不小心移除存取資源所需的許可權限，以保護您的 Aurora DSQL 資源。

如需關於支援服務連結角色的其他服務資訊，請參閱可搭配 IAM 運作的AWS 服務，尋找 Service-Linked Role (服務連結角色) 欄中顯示為 Yes (是) 的服務。選擇具有連結的是，以檢視該服務的服務連結角色文件。

服務連結角色可在所有支援的 Aurora DSQL 區域中使用。

Aurora DSQL 的服務連結角色許可權限

Aurora DSQL 使用名為 的服務連結角色 `AWSServiceRoleForAuroraDsql` – 允許 Amazon Aurora DSQL 代表您建立和管理 AWS 資源。此服務連結角色連接至下列受管政策：AuroraDsqlServiceLinkedRolePolicy。

> **Note**
>
> 您必須設定許可，IAM 實體 (如使用者、群組或角色) 才可建立、編輯或刪除服務連結角色。您可能會遇到下列錯誤訊息：`You don't have the permissions to create an Amazon Aurora DSQL service-linked role`。如果您看到此訊息，請確認您已啟用以下許可：
>
> JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateDsqlServiceLinkedRole",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "dsql.amazonaws.com"
                }
            }
        }
    ]
```

```
}
```

如需更多資訊，請參閱服務連結角色許可權限。

建立服務連結角色

您不需要手動建立 AuroraDSQLServiceLinkedRolePolicy 服務連結角色。Aurora DSQL 會為您建立服務連結角色。如果已從您的帳戶刪除 AuroraDSQLServiceLinkedRolePolicy 服務連結角色，Aurora DSQL 會在您建立新的 Aurora DSQL 叢集時建立角色。

編輯服務連結角色

Aurora DSQL 不允許您編輯 AuroraDSQLServiceLinkedRolePolicy 服務連結角色。因為可能有各種實體會參考服務連結角色，所以您無法在建立角色之後變更其名稱。不過，您可以使用 IAM 主控台、AWS Command Line Interface (AWS CLI) 或 IAM API 編輯角色的描述。

刪除服務連結角色

若您不再使用需要服務連結角色的功能或服務，我們建議您刪除該角色。如此一來，您就沒有未主動監控或維護的未使用實體。

您必須先刪除帳戶中的任何叢集，才能刪除帳戶的服務連結角色。

您可以使用 IAM 主控台 AWS CLI、 或 IAM API 來刪除服務連結角色。如需詳細資訊，請參閱《IAM 使用者指南》中的建立服務連結角色。

Aurora DSQL 服務連結角色的支援區域

Aurora DSQL 支援在所有提供服務的區域中使用服務連結角色。如需詳細資訊，請參閱AWS 區域與端點。

使用 IAM 條件索引鍵搭配 Amazon Aurora DSQL

您在 Aurora DSQL 中授予許可權限時，可以指定條件以決定許可權限政策的生效方式。以下是如何在 Aurora DSQL 許可權限政策中使用條件索引鍵的範例。

範例 1：授予在特定 中建立叢集的許可 AWS 區域

下列政策授予許可權限在美國東部 (維吉尼亞北部) 和美國東部 (俄亥俄) 區域中建立叢集。此政策使用資源 ARN 限制允許的區域，因此 Aurora DSQL 只有在政策 `Resource` 區段中指定該 ARN 時才能建立叢集。

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"
        }
    ]
}
```

範例 2：授予在特定 AWS 區域中建立多區域叢集的許可

下列政策授予許可權限在美國東部 (維吉尼亞北部) 和美國東部 (俄亥俄) 區域中建立多區域叢集。此政策使用資源 ARN 限制允許的區域，因此 Aurora DSQL 只有在政策 `Resource` 區段中指定此 ARN 時才能建立多區域叢集。請注意，建立多區域叢集也需要在每個指定區域中具備 `PutMultiRegionProperties`、`PutWitnessRegion` 和 `AddPeerCluster` 許可權限。

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/*"
        ]
      }
    ]
}
```

範例 3：授予許可權限以建立具有特定見證區域的多區域叢集

下列政策使用 Aurora DSQL `dsql:WitnessRegion` 條件索引鍵，可讓使用者在美國西部 (奧勒岡) 建立包含見證區域的多區域叢集。如果您未指定 `dsql:WitnessRegion` 條件，您可以使用任何區域做為見證區域。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dsql:CreateCluster",
                "dsql:PutMultiRegionProperties",
                "dsql:AddPeerCluster"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dsql:PutWitnessRegion"
            ],
            "Resource": "arn:aws:dsql:*:123456789012:cluster/*",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": [
                        "us-west-2"
```

```
                    ]
                }
            }
        }
    ]
}
```

Amazon Aurora DSQL 中的事件回應

安全是 AWS最重視的一環。作為 AWS 雲端共同責任模型的一部分， 會 AWS 管理符合最安全敏感組織需求的資料中心、網路和軟體架構。 AWS 負責任何與 Amazon Aurora DSQL 服務本身相關的事件回應。此外，身為 AWS 客戶，您需共同負責維護雲端的安全性。這表示您可以從可存取的 AWS 工具和功能控制您選擇實作的安全性。此外您也需要負責處理自己這部分的共同責任模型事件回應。

建立安全基準以達到讓應用程式在雲端中執行的目標，您就可以偵測偏差並加以回應。為了協助您瞭解事件回應和您的選擇對公司目標的影響，建議您檢閱下列資源：

- AWS 安全事件回應指南
- AWS 安全性、身分和合規的最佳實務
- AWS 雲端採用架構 (CAF) 的安全觀點白皮書

Amazon GuardDuty 是一種受管威脅偵測服務，會持續監控惡意或未經授權的行為，以協助客戶保護 AWS 帳戶 和工作負載，並在可疑活動可能升級到事件之前識別。此服務會監控活動，例如異常 API 呼叫或可能未經授權的部署，指出帳戶或資源可能遭惡意行為者入侵或偵察。例如 Amazon GuardDuty 能夠偵測 Amazon Aurora DSQL API 中的可疑活動，例如使用者從新位置登入和建立新叢集。

Amazon Aurora DSQL 的合規驗證

若要了解 是否 AWS 服務 在特定合規計劃範圍內，請參閱AWS 服務 合規計劃範圍內然後選擇您感興趣的合規計劃。如需一般資訊，請參閱AWS 合規計劃。

您可以使用 下載第三方稽核報告 AWS Artifact。如需詳細資訊，請參閱在 中下載報告 AWS Artifact。

您使用 時的合規責任 AWS 服務 取決於資料的敏感度、您公司的合規目標，以及適用的法律和法規。如需使用 時合規責任的詳細資訊 AWS 服務，請參閱 AWS 安全文件。

Amazon Aurora DSQL 的恢復能力

AWS 全球基礎設施是以 AWS 區域 和可用區域 (AZ) 為基礎建置。 AWS 區域 提供多個實體隔離和隔離的可用區域，這些可用區域以低延遲、高輸送量和高備援聯網連接。透過可用區域，您可以設計與操作的應用程式和資料庫，在可用區域之間自動容錯移轉而不會發生中斷。可用區域的可用性、容錯能力和擴展能力，均較單一或多個資料中心的傳統基礎設施還高。Aurora DSQL 旨在讓您充分利用 AWS 區域基礎設施，同時提供最高的資料庫可用性。根據預設，Aurora DSQL 中的單一區域叢集具有多可用區域可用性，能夠容忍可能影響完整可用區域存取的主要元件故障和基礎結構中斷。多區域叢集提供多可用區域恢復能力的所有優點，同時仍提供高度一致的資料庫可用性，即使應用程式用戶端無法存取 AWS 區域 也一樣。

如需 AWS 區域 和可用區域的詳細資訊，請參閱 AWS 全球基礎設施。

除了 AWS 全球基礎設施之外，Aurora DSQL 還提供數種功能，以協助支援您的資料彈性和備份需求。

備份與還原

Aurora DSQL 支援使用 備份和還原 AWS Backup 主控台。您可為單一區域和多區域叢集執行完整備份和還原。如需詳細資訊，請參閱Amazon Aurora DSQL 的備份與還原。

複寫

Aurora DSQL 的設計是將所有寫入交易遞交至分散式交易日誌，並將所有遞交的日誌資料同步複寫至三個 AZ 中的使用者儲存複本。多區域叢集可在讀取和寫入區域之間提供完整的跨區域複寫功能。

指定的見證區域可支援僅交易日誌寫入，不會耗用儲存。見證區域沒有端點。這表示見證區域只會儲存加密的交易日誌，不需要管理或設定，而且使用者無法存取。如果見證區域受損，則不會影響叢集可用性。在見證區域復原之前，寫入交易的延遲可能會小幅增加。

Aurora DSQL 交易日誌和使用者儲存會與呈現給 Aurora DSQL 查詢處理器的所有資料一起發佈，做為單一邏輯磁碟區。Aurora DSQL 會根據資料庫主索引鍵範圍和存取模式自動分割、合併和複寫資料。Aurora DSQL 會根據讀取存取頻率自動擴展和縮減讀取複本。

叢集儲存複本會分散在多租用戶儲存機群中。如果元件或可用區域受損，Aurora DSQL 會自動將存取重新導向至仍在運作的元件，並以非同步方式修復遺失的複本。Aurora DSQL 修正受損複本後，就會自動將複本新增回到儲存仲裁，並將其提供給叢集。

高可用性

根據預設，Aurora DSQL 中的單一區域和多區域叢集為雙主動設計，您不需要手動佈建、設定或重新設定任何叢集。Aurora DSQL 可完全自動化叢集復原，無須傳統的主要次要容錯移轉作業。複寫一律是同步的，並在多個可用區域完成，因此在故障復原期間，不會因為複寫延遲或容錯移轉至非同步次要資料庫而導致資料遺失。

單一區域叢集提供多可用區域備援端點，可自動啟用並行存取，並跨三個可用區域提供強大的資料一致性。這表示這三個可用區域的使用者儲存複本，都會將相同的結果傳回給一或多個讀取器，並且永遠可用於接收寫入。Aurora DSQL 多區域叢集的所有區域都可享有這種強大的一致性和多可用區域恢復能力。這表示多區域叢集提供兩個高度一致的區域端點，因此用戶端可以無差別地讀取或寫入至任一區域，且遞交時沒有複寫延遲。

Aurora DSQL 為單一區域叢集提供 99.99% 的可用性，多區域叢集則為 99.999% 的可用性。

故障注入測試

Amazon Aurora DSQL 與 AWS Fault Injection Service (AWS FIS) 整合，這是一種全受管服務，用於執行受控故障注入實驗，以改善應用程式的彈性。使用 AWS FIS，您可以：

- 建立可定義特定故障情境的實驗範本
- 注入故障 (提升叢集連線錯誤率) 以驗證應用程式錯誤的處理和復原機制
- 測試多區域應用程式行為，以驗證應用程式流量在 AWS 區域 發生高連線錯誤率 AWS 區域 之間的轉移

例如在跨越美國東部 (維吉尼亞北部) 和美國東部 (俄亥俄) 的多區域叢集中，您可以在美國東部 (俄亥俄) 執行實驗以測試故障，同時美國東部 (維吉尼亞北部) 則繼續正常運作。此項受控測試可協助您事先識別和解決潛在問題，以免影響正式作業工作負載。

如需 AWS FIS 支援動作的完整清單，請參閱 使用者指南中的動作目標。 AWS FIS

如需 中可用 Amazon Aurora DSQL 動作的相關資訊 AWS FIS，請參閱AWS FIS 《 使用者指南》中的 Aurora DSQL 動作參考。

若要開始執行故障注入實驗，請參閱 AWS FIS 使用者指南中的規劃 AWS FIS 實驗。

Amazon Aurora DSQL 中的基礎結構安全性

Amazon Aurora DSQL 是受管服務，受到安全性、身分和合規最佳實務中所述的 AWS 全球網路安全程序的保護。

您可以使用 AWS 發佈的 API 呼叫，透過網路存取 Aurora DSQL。用戶端必須支援 Transport Layer Security (TLS) 1.2 或更新版本。用戶端也必須支援具備完美轉送私密 (PFS) 的密碼套件，例如臨時 Diffie-Hellman (DHE) 或橢圓曲線臨時 Diffie-Hellman (ECDHE)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

此外，請求必須使用存取金鑰 ID 和與 IAM 主體相關聯的私密存取金鑰來簽署。或者，您可以透過 AWS Security Token Service (AWS STS) 來產生暫時安全憑證來簽署請求。

使用 管理和連線至 Amazon Aurora DSQL 叢集 AWS PrivateLink

使用 AWS PrivateLink for Amazon Aurora DSQL，您可以在 Amazon Virtual Private Cloud 中佈建界面 Amazon VPC 端點 （界面端點）。這些端點可直接透過 Amazon VPC 和 內部部署的應用程式存取 Direct Connect，或 AWS 區域 是透過 Amazon VPC 對等互連在不同的 中存取。使用 AWS PrivateLink 和 介面端點，您可以簡化從應用程式到 Aurora DSQL 的私有網路連線。

Amazon VPC 內的應用程式可以使用 Amazon VPC 介面端點存取 Aurora DSQL，不需要使用公共 IP 位址。

介面端點是由一個或多個彈性網路介面 (ENI) 表示；這些介面是從 Amazon VPC 子網路所指派的私有 IP 位址。透過介面端點對 Aurora DSQL 的請求會保留在 AWS 網路上。如需如何將 Amazon VPC 與內部部署網路連線的詳細資訊，請參閱 Direct Connect 使用者指南和AWS Site-to-Site VPN VPN 使用者指南。

如需介面端點的一般資訊，請參閱AWS PrivateLink《 使用者指南》中的使用介面 Amazon VPC 端點存取 AWS 服務。

適用於 Aurora DSQL 的 Amazon VPC 端點類型

Aurora DSQL 需要兩種不同類型的 AWS PrivateLink 端點。

1. 管理端點 — 此端點用於管理作業，例如 Aurora DSQL 叢集上的 `get`、`create`、`update`、`delete` 和 `list`。請參閱 使用 管理 Aurora DSQL 叢集 AWS PrivateLink。
2. 連線端點 — 此端點用於透過 PostgreSQL 用戶端連線至 Aurora DSQL 叢集。請參閱 使用 連線至 Aurora DSQL 叢集 AWS PrivateLink。

使用 AWS PrivateLink for Aurora DSQL 時的考量事項

Amazon VPC 考量適用於 Aurora DSQL AWS PrivateLink 的 。如需詳細資訊，請參閱《 AWS PrivateLink 指南》中的使用介面 VPC 端點和配額存取 AWS 服務。 AWS PrivateLink

使用 管理 Aurora DSQL 叢集 AWS PrivateLink

您可以使用 AWS Command Line Interface 或 AWS 軟體開發套件 (SDKs) 透過 Aurora DSQL 介面端點管理 Aurora DSQL 叢集。

建立 Amazon VPC 端點

若要建立 Amazon VPC 介面端點，請參閱《 AWS PrivateLink 指南》中的建立 Amazon VPC 端點。

```
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 \
```

若要使用 Aurora DSQL API 請求的預設區域 DNS 名稱，請勿在建立 Aurora DSQL 介面端點時停用私有 DNS。啟用私有 DNS 時，從 Amazon VPC 內部對 Aurora DSQL 服務提出的請求，會自動解析至 Amazon VPC 端點的私有 IP 位址，而不是公共 DNS 名稱。啟用私有 DNS 時，Amazon VPC 內部提出的 Aurora DSQL 請求會自動解析至您的 Amazon VPC 端點。

如果未啟用私有 DNS，請使用 `--region`和 `--endpoint-url` 參數搭配 AWS CLI 命令，透過 Aurora DSQL 介面端點管理 Aurora DSQL 叢集。

使用端點 URL 列出叢集

在下列範例中，將 AWS 區域 `us-east-1`和 Amazon VPC 端點 ID 的 DNS 名稱取代`vpce-1a2b3c4d-5e6f.dsql.us-east-1.vpce.amazonaws.com`為您自己的資訊。

```
aws dsql --region us-east-1 --endpoint-url https://vpce-1a2b3c4d-5e6f.dsql.us-
east-1.vpce.amazonaws.com list-clusters
```

API 操作

如需在 Aurora DSQL 中管理資源的文件，請參閱 Aurora DSQL API 參考。

管理端點政策

透過徹底測試和設定 Amazon VPC 端點政策，您可以協助確保 Aurora DSQL 叢集安全、合規，並符合組織的特定存取控制和控管要求。

範例：完整 Aurora DSQL 存取政策

下列政策會透過指定的 Amazon VPC 端點，授予所有 Aurora DSQL 動作和資源的完整存取權限。

```
aws ec2 modify-vpc-endpoint \
    --vpc-endpoint-id vpce-xxxxxxxxxxxxxxxxx \
    --region region \
    --policy-document '{
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": "*",
          "Action": "dsql:*",
          "Resource": "*"
        }
      ]
    }'
```

範例：受限制的 Aurora DSQL 存取政策

下列政策僅允許這些 Aurora DSQL 動作。

- `CreateCluster`
- `GetCluster`
- `ListClusters`

所有其他 Aurora DSQL 動作都會遭到拒絕。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": "*",
      "Action": [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:ListClusters"
      ],
      "Resource": "*"
    }
  ]
}
```

使用 連線至 Aurora DSQL 叢集 AWS PrivateLink

設定並啟用 AWS PrivateLink 端點後，您可以使用 PostgreSQL 用戶端連線至 Aurora DSQL 叢集。以下連線指示概述建構適當的主機名稱以透過 AWS PrivateLink 端點連線的步驟。

設定 AWS PrivateLink 連線端點

步驟 1：取得叢集的服務名稱

建立端點以 AWS PrivateLink 連線至叢集時，您必須先擷取叢集特定的服務名稱。

AWS CLI

```
aws dsql get-vpc-endpoint-service-name \
--region us-east-1 \
--identifier your-cluster-id
```

回應範例

```
{
    "serviceName": "com.amazonaws.us-east-1.dsql-fnh4"
}
```

服務名稱包含識別符，例如範例中的 `dsql-fnh4`。建構主機名稱以連線至叢集時，也需要此識別符。

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);
```

步驟 2：建立 Amazon VPC 端點

使用上一個步驟中取得的服務名稱建立 Amazon VPC 端點。

Important

以下連線指示僅適用於在啟用私有 DNS 時連線至叢集。建立端點時請勿使用 `--no-private-dns-enabled` 旗標，因為這會使下列連線指示無法正常運作。如果您停用私有 DNS，則需要建立自己的萬用字元私有 DNS 記錄指向建立的端點。

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
```

回應範例

```
{
    "VpcEndpoint": {
        "VpcEndpointId": "vpce-0123456789abcdef0",
        "VpcEndpointType": "Interface",
        "VpcId": "vpc-0123456789abcdef0",
        "ServiceName": "com.amazonaws.us-east-1.dsql-fnh4",
        "State": "pending",
        "RouteTableIds": [],
        "SubnetIds": [
            "subnet-0123456789abcdef0",
            "subnet-0123456789abcdef1"
        ],
        "Groups": [
            {
                "GroupId": "sg-0123456789abcdef0",
                "GroupName": "default"
            }
        ],
        "PrivateDnsEnabled": true,
        "RequesterManaged": false,
        "NetworkInterfaceIds": [
            "eni-0123456789abcdef0",
            "eni-0123456789abcdef1"
        ],
        "DnsEntries": [
            {
                "DnsName": "*.dsql-fnh4.us-east-1.vpce.amazonaws.com",
                "HostedZoneId": "Z7HUB22UULQXV"
            }
        ],
        "CreationTimestamp": "2025-01-01T00:00:00.000Z"
```

```
    }
}
```

SDK for Python

```
import boto3

ec2_client = boto3.client('ec2', region_name='us-east-1')
response = ec2_client.create_vpc_endpoint(
    VpcEndpointType='Interface',
    VpcId='your-vpc-id',
    ServiceName='com.amazonaws.us-east-1.dsql-fnh4',  # Use the service name from
 previous step
    SubnetIds=[
        'subnet-id-1',
        'subnet-id-2'
    ],
    SecurityGroupIds=[
        'security-group-id'
    ]
)

vpc_endpoint_id = response['VpcEndpoint']['VpcEndpointId']
print(f"VPC Endpoint created with ID: {vpc_endpoint_id}")
```

SDK for Java 2.x

使用適用於 Aurora DSQL API 的端點 URL

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.ec2.Ec2Client;
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointRequest;
import software.amazon.awssdk.services.ec2.model.CreateVpcEndpointResponse;
import software.amazon.awssdk.services.ec2.model.VpcEndpointType;

String region = "us-east-1";
String serviceName = "com.amazonaws.us-east-1.dsql-fnh4";  // Use the service name
 from previous step
String vpcId = "your-vpc-id";

Ec2Client ec2Client = Ec2Client.builder()
    .region(Region.of(region))
```

```
        .credentialsProvider(DefaultCredentialsProvider.create())
        .build();

CreateVpcEndpointRequest request = CreateVpcEndpointRequest.builder()
        .vpcId(vpcId)
        .serviceName(serviceName)
        .vpcEndpointType(VpcEndpointType.INTERFACE)
        .subnetIds("subnet-id-1", "subnet-id-2")
        .securityGroupIds("security-group-id")
        .build();

CreateVpcEndpointResponse response = ec2Client.createVpcEndpoint(request);
String vpcEndpointId = response.vpcEndpoint().vpcEndpointId();
System.out.println("VPC Endpoint created with ID: " + vpcEndpointId);
```

透過 Direct Connect 或 Amazon VPC 對等互連連線時的其他設定

使用來自內部部署裝置的 AWS PrivateLink 連線端點，透過 Amazon VPC 對等互連或 連線到 Aurora DSQL 叢集時，可能需要一些額外的設定 Direct Connect。如果您的應用程式在與 AWS PrivateLink 端點相同的 Amazon VPC 中執行，則不需要此設定。上述建立的私有 DNS 項目無法在端點的 Amazon VPC 之外正確解析，但您可以建立自己的私有 DNS 記錄，以解析至您的 AWS PrivateLink 連線端點。

建立指向 AWS PrivateLink 端點完整網域名稱的私有 CNAME DNS 記錄。所建立 DNS 記錄的網域名稱應該從下列元件建構：

1. 服務名稱的服務識別符。例如：`dsql-fnh4`
2. 的 AWS 區域

使用下列格式的網域名稱建立 CNAME DNS 記錄： `*.`*`service-identifier`*`.`*`region`*`.on.aws`

網域名稱的格式很重要的原因有兩個：

1. 使用 `verify-full` SSL 模式時，用於連線至 Aurora DSQL 的主機名稱必須符合 Aurora DSQL 的伺服器憑證。這可確保最高層級的連線安全性。
2. Aurora DSQL 會使用用於連線至 Aurora DSQL 之主機名稱的叢集 ID 部分來識別連線叢集。

如果無法建立私有 DNS 記錄，您仍然可以連線到 Aurora DSQL。請參閱 使用不含私有 DNS 的 AWS PrivateLink 端點連線至 Aurora DSQL 叢集。

使用連線 AWS PrivateLink 端點連線至 Aurora DSQL 叢集

設定 AWS PrivateLink 端點並處於作用中狀態（檢查 `State`是否為 `available`) 之後，您可以使用 PostgreSQL 用戶端連線至 Aurora DSQL 叢集。如需有關使用 AWS SDK 的說明，您可以遵循使用 Aurora DSQL 進行程式設計中的指南。您必須變更叢集端點以符合主機名稱格式。

建構主機名稱

透過 連線的主機名稱與公有 DNS 主機名稱 AWS PrivateLink 不同。您需要使用下列元件建構該主機名稱。

1. `Your-cluster-id`
2. 服務名稱的服務識別符。例如：`dsql-fnh4`
3. AWS 區域。例如：`us-east-1`

使用下列格式：*`cluster-id`*`.`*`service-identifier`*`.`*`region`*`.on.aws`

範例：使用 PostgreSQL 進行連線

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export SERVICE_IDENTIFIER=dsql-fnh4  # This should match the identifier in your service
 name

# Construct the hostname
export HOSTNAME="$CLUSTERID.$SERVICE_IDENTIFIER.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $HOSTNAME)

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

使用不含私有 DNS 的 AWS PrivateLink 端點連線至 Aurora DSQL 叢集

上述連線指示依賴私有 DNS 記錄。如果您的應用程式在與 AWS PrivateLink 端點相同的 Amazon VPC 中執行，則會為您建立 DNS 記錄。或者，如果您透過 Amazon VPC 對等互連從內部部署裝置連線 Direct Connect，則可以建立自己的私有 DNS 記錄。不過，由於安全團隊實施的網路限制，DNS 記錄設定並非總是可行的。如果您的應用程式必須使用對等 Amazon VPC Direct Connect 或從對等 Amazon VPC 連線，且無法進行 DNS 記錄設定，您仍然可以連線至 Aurora DSQL。

Aurora DSQL 會使用您主機名稱的叢集 ID 部分來識別連線叢集，但如果無法設定 DNS 記錄，Aurora DSQL 支援使用`amzn-cluster-id`連線選項指定目標叢集。透過此選項，您可以在連線時使用 AWS PrivateLink 端點的完整網域名稱做為主機名稱。

Important

連線至 AWS PrivateLink 端點的完整網域名稱或 IP 地址時，不支援 `verify-full` SSL 模式。因此，偏好設定私有 DNS。

範例：使用 PostgreSQL 指定叢集 ID 連線選項

```
# Set environment variables
export CLUSTERID=your-cluster-id
export REGION=us-east-1
export HOSTNAME=vpce-04037adb76c111221-d849uc2p.dsql-fnh4.us-east-1.vpce.amazonaws.com
 # This should match your endpoint's fully-qualified domain name

# Construct the hostname used to generate the authentication token
export AUTH_HOSTNAME="$CLUSTERID.dsql.$REGION.on.aws"

# Generate authentication token
export PGPASSWORD=$(aws dsql --region $REGION generate-db-connect-admin-auth-token --
hostname $AUTH_HOSTNAME)

# Specify the amzn-cluster-id connection option
export PGOPTIONS="-c amzn-cluster-id=$CLUSTERID"

# Connect using psql
psql -d postgres -h $HOSTNAME -U admin
```

對 的問題進行故障診斷 AWS PrivateLink

常見問題與解決方案

下表列出有關 AWS PrivateLink 搭配 Aurora DSQL 時的常見問題和解決方案。

問題	可能原因	解決方案
連線逾時	安全群組未正確設定	使用 Amazon VPC Reachability Analyzer 確保您的聯網設定允許連接埠 5432 的流量。
DNS 解析失敗	未啟用私有 DNS	確認在啟用私有 DNS 的情況下建立 Amazon VPC 端點。
身分驗證失敗	憑證不正確或記號過期	產生新的身分驗證記號並驗證使用者名稱。
找不到服務名稱	不正確的叢集 ID	擷取服務名稱 AWS 區域 時，請再次檢查您的叢集 ID 和 。

相關資源

如需詳細資訊，請參閱下列資源：

- Amazon Aurora DSQL 使用者指南
- AWS PrivateLink 文件
- 透過 存取 AWS 服務 AWS PrivateLink

Amazon Aurora DSQL 中的組態與漏洞分析

AWS 處理基本安全任務，例如訪客作業系統 (OS) 和資料庫修補、防火牆組態和災難復原。這些程序已由適當的第三方進行檢閱並認證。如需詳細資訊，請參閱以下資源：

- 共同責任模型
- Amazon Web Services：安全程序概觀 (白皮書)

預防跨服務混淆代理人

混淆代理人問題屬於安全性問題，其中沒有執行動作許可的實體可以強制具有更多許可的實體執行該動作。在 中 AWS，跨服務模擬可能會導致混淆代理人問題。在某個服務 (呼叫服務) 呼叫另一個服務 (被呼叫服務) 時，可能會發生跨服務模擬。可以操縱呼叫服務來使用其許可，以其不應有存取許可的方式對其他客戶的資源採取動作。為了預防這種情況， AWS 提供的工具可協助您保護所有服務的資料，而這些服務主體已獲得您帳戶中資源的存取權。

若要限制 Amazon Aurora DSQL 為資源提供另一項服務的許可權限，我們建議在資源政策中使用 `aws:SourceArn` 和 `aws:SourceAccount` 全域條件內容索引鍵。如果您想要僅允許一個資源與跨服務存取相關聯，則請使用 `aws:SourceArn`。如果您想要允許該帳戶中的任何資源與跨服務使用相關聯，請使用 `aws:SourceAccount`。

防範混淆代理人問題的最有效方法是使用 `aws:SourceArn` 全域條件內容索引鍵，以及資源的完整 ARN。如果不知道資源的完整 ARN，或者如果您指定了多個資源，請使用 `aws:SourceArn` 全域內容條件索引鍵搭配萬用字元 (*) 來表示 ARN 的未知部分。例如 `arn:aws:dsql:*:123456789012:*`。

如果 `aws:SourceArn` 值不包含帳戶 ID (例如 Amazon S3 儲存貯體 ARN)，您必須使用這兩個全域條件內容索引鍵來限制許可。

`aws:SourceArn` 的值必須是 ResourceDescription。

下列範例示範如何使用 Aurora DSQL 中的 `aws:SourceArn` 和 `aws:SourceAccount` 全域條件內容索引鍵，以預防混淆代理人問題。

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 DSQL 的安全最佳實務

開發和實作自己的安全性政策時，不妨考慮使用 Aurora DSQL 提供的多種安全性功能。以下最佳實務為一般準則，並不代表完整的安全解決方案。這些最佳實務可能不適用或無法滿足您的環境需求，因此請將其視為實用建議就好，而不要當作是指示。

主題

- Aurora DSQL 的偵測性安全最佳實務
- Aurora DSQL 預防性安全最佳實務

Aurora DSQL 的偵測性安全最佳實務

除了下列安全使用 Aurora DSQL 的方式之外，請參閱 中的安全性 AWS Well-Architected Tool ，以了解雲端技術如何改善您的安全性。

Amazon CloudWatch 警示

使用 Amazon CloudWatch 警示，您可在自己指定的一段時間內監看單一指標。如果指標超過指定的閾值，則會傳送通知至 Amazon SNS 主題或 AWS Auto Scaling 政策。CloudWatch 警示不會因為處於特定狀態而調用動作。必須是狀態已變更並維持了所指定的時間長度，才會呼叫動作。

為 Aurora DSQL 資源加上標籤，以便進行識別和自動化

您可以將中繼資料以標籤形式指派給 AWS 資源。每個標記都是由客戶定義金鑰和選用值組成的簡單標籤，能夠更輕鬆地管理、搜尋和篩選資源。

標記允許實現分組控制。雖然標籤不具固有類型，但能讓您依用途、擁有者、環境或其他條件分類資源。下列是一些範例：

- 安全性：用於確定加密等需求。

- 機密性：資源支援的特定資料機密等級識別符。
- 環境：用來區分開發、測試和正式作業基礎結構。

您可以將中繼資料以標籤形式指派給 AWS 資源。每個標記都是由客戶定義金鑰和選用值組成的簡單標籤，能夠更輕鬆地管理、搜尋和篩選資源。

標記允許實現分組控制。雖然標籤不具固有類型，但能讓您依用途、擁有者、環境或其他條件分類資源。下列是一些範例。

- 安全性：用於確定加密等需求。
- 機密性：資源支援的特定資料機密等級識別符。
- 環境：用來區分開發、測試和正式作業基礎結構。

如需詳細資訊，請參閱標記 AWS 資源的最佳實務。

Aurora DSQL 預防性安全最佳實務

除了下列安全使用 Aurora DSQL 的方式之外，請參閱 中的安全性 AWS Well-Architected Tool ，以了解雲端技術如何改善您的安全性。

使用 IAM 角色驗證對 Aurora DSQL 的存取權限。

存取 Aurora DSQL 的使用者、應用程式和其他 AWS 服務 必須在 AWS API 和 AWS CLI 請求中包含有效的 AWS 登入資料。您不應將 AWS 登入資料直接存放在應用程式或 EC2 執行個體中。這些是不會自動輪換的長期憑證。如果這些憑證遭到入侵，會對業務產生重大影響。IAM 角色可讓您取得可用來存取 AWS 服務 和資源的臨時存取金鑰。

如需詳細資訊，請參閱Aurora DSQL 的身分驗證和授權。

使用 IAM 政策進行 Aurora DSQL 基礎授權。

您在授予許可權限時會決定誰可以取得許可權限、可以取得哪些 Aurora DSQL API 的許可權限，以及可以對這些資源進行的特定動作。對降低錯誤或惡意意圖所引起的安全風險和影響而言，實作最低權限是其中關鍵。

將許可權限政策連接至 IAM 角色，並授予許可權限在 Aurora DSQL 資源上執行作業。也提供 IAM 實體的許可界限，可讓您設定身分型政策可授予 IAM 實體的最大許可權限。

與 的根使用者最佳實務 AWS 帳戶類似，請勿使用 Aurora DSQL 中的 `admin`角色來執行日常操作。反之，我們建議您建立自訂資料庫角色以管理和連線至您的叢集。如需詳細資訊，請參閱存取 Aurora DSQL 和瞭解 Aurora DSQL 的身分驗證和授權。

在正式作業環境中使用 **`verify-full`**。

此設定會驗證伺服器憑證是否由信任的憑證認證機構簽署，以及伺服器主機名稱是否與憑證相符。

更新 PostgreSQL 用戶端

定期將 PostgreSQL 用戶端更新為最新版本，以享有提升安全性的效益。我們建議使用 PostgreSQL 17 版。

在 Aurora DSQL 中標記資源

在 AWS 中，標籤是您定義並與 Aurora DSQL 資源 (例如叢集) 建立關聯的使用者定義鍵值對。標籤是選擇性的。若提供索引鍵，對應值為選用項。

您可以使用 AWS 管理主控台、AWS CLI 或 AWS SDK 來新增、列出及刪除 Aurora DSQL 叢集上的標籤。您可以使用 AWS 主控台在建立叢集期間或之後新增標籤。若要在建立叢集後標記叢集，請使用 AWS CLI 的 `TagResource` 操作。

使用名稱標籤標記叢集

Aurora DSQL 建立叢集時，會自動分配全域唯一識別符作為 Amazon Resource Name (ARN)。若想為叢集指派易於辨識的名稱，建議使用標籤。

若使用 Aurora DSQL 主控台建立叢集，Aurora DSQL 會自動建立標籤。該標籤索引鍵為名稱，自動生成的值代表叢集名稱。該值可設定，因此可為叢集指派更易辨識的名稱。若叢集具有名稱標籤及其關聯值，可在 Aurora DSQL 主控台中查看該值。

標記需求

標籤均擁有以下要求：

- 索引鍵字首不能是 `aws:`。
- 索引鍵在標籤集內必須是唯一的。
- 索引鍵必須介於 1 到 128 個允許的字元之間。
- 值必須介於 0 到 256 個允許的字元之間。
- 值在每個標籤集中不需要是唯一的。
- 索引鍵和值允許使用字母、數字、空格及下列符號：_ . : / = + - @。
- 金鑰和值會區分大小寫。

標籤使用注意事項

使用 Aurora DSQL 標籤時，請注意以下事項。

- 使用 AWS CLI 或 Aurora DSQL API 操作時，請務必提供 Aurora DSQL 資源的 Amazon Resource Name (ARN)。如需更多資訊，請參閱 Aurora DSQL 資源的 Amazon Resource Name (ARNs) 格式。
- 每個資源皆有一個標籤集，此為指派給該資源之一或多個標籤的集合。
- 每個資源每個標籤集最多可擁有 50 個標籤。
- 如果您刪除資源，任何關聯的標籤也會遭到刪除。
- 可在建立資源時新增標籤，並可使用以下 API 操作檢視及修改標籤：`TagResource`、`UntagResource` 和 `ListTagsForResource`。
- 可將標籤用於 IAM 政策。可透過 IAM 政策管理 Aurora DSQL 叢集存取權限，並控制可對這些資源執行的操作。想進一步了解，請參閱使用標籤控制對 AWS 資源的存取。
- 可將標籤用於跨 AWS 的各種其他活動。想進一步了解，請參閱常見標記策略。

使用 Amazon Aurora DSQL 的考量事項

當您使用 Amazon Aurora DSQL 時，請考慮下列行為。如需進一步了解 PostgreSQL 相容性及支援，請參閱 Aurora DSQL 中的 SQL 功能相容性。如需了解配額和限制，請參閱 Amazon Aurora DSQL 中的叢集配額與資料庫限制。

- 執行DROP TABLE命令後，儲存限制計算可能需要一些時間來反映釋放的儲存。如果您需要額外的儲存容量，請參閱 叢集配額以請求配額更新。
- 對於 Aurora DSQL 中的大型資料表，請使用系統目錄來擷取資料表資料列計數，而非 COUNT(*) 操作。如需詳細資訊，請參閱在 Aurora DSQL 中使用系統資料表和命令。
- Aurora DSQL 透過結構描述層級授予來管理許可。管理員使用者使用 建立結構描述CREATE SCHEMA，並使用 授予其他角色的存取權GRANT USAGE ON SCHEMA。管理員使用者管理公有結構描述中的物件，而非管理員使用者則在使用者建立的結構描述中建立物件。管理員角色可以授予自己任何其他角色，以取得使用者建立物件的許可。如需詳細資訊，請參閱授權資料庫角色在您的資料庫使用 SQL。
- 當驅動程式呼叫 時PG_PREPARED_STATEMENTS，Aurora DSQL 會提供快取預備陳述式的全叢集檢視。對於相同的叢集和 IAM 角色，每個連線可能會看到比預期更多的預備陳述式。Aurora DSQL 會在準備期間動態管理陳述式名稱。
- 從IPv4-only執行個體連線時，請確定您的用戶端已設定為 IPv4 連線。有些 PostgreSQL 用戶端會在雙堆疊模式下嘗試 IPv4 和 IPv6 連線。如果 IPv4 連線遇到限流，用戶端可能會嘗試 IPv6，並在 IPv4-only 的主機上傳回NetworkUnreachable錯誤。將用戶端設定為明確使用 IPv4 以避免此行為。
- 管理員使用者建立新的結構描述，GRANT並在連線生命週期 （ 最多一小時 ） 內將REVOKE變更傳播到現有的連線。若要立即生效，請在許可變更後建立新的連線。
- 在極少數的多區域連結叢集復原案例中，自動化叢集復原操作會維持高可用性，但您可能會遇到暫時性並行控制或連線錯誤。在大多數情況下，只會影響您工作負載的百分比。當您遇到這些暫時性錯誤時，請重試您的交易或重新與您的用戶端連線。
- 有些 SQL 用戶端，例如 Datagrip，請求廣泛的系統中繼資料來填入結構描述資訊。Aurora DSQL 提供 SQL 查詢功能的核心中繼資料。與完整功能集相比，這些用戶端中的結構描述顯示可能會顯示有限的資訊。
- 為了確保查詢可辨識新建立的結構描述和資料表，請在建立或捨棄資料庫物件後重新整理連線。這包括您在捨棄結構描述或查詢在另一個連線中建立的物件時看到Schema Already Exists錯誤的案例。中斷連線並重新連線，或SET search_path再次執行 以重新整理目錄快取。

- 對於複雜的查詢，請使用 `EXPLAIN ANALYZE VERBOSE` 來識別高延遲操作並最佳化查詢計劃。覆蓋索引可透過啟用僅索引掃描而非完整資料表掃描，大幅降低 DPU 成本。如需詳細資訊，請參閱使用 Aurora DSQL EXPLAIN 計劃。
- 連線限制是在叢集層級管理。請參閱 叢集配額 以請求配額更新。

Amazon Aurora DSQL 中的叢集配額與資料庫限制

下列章節說明 Aurora DSQL 的叢集配額與資料庫限制。

叢集配額

您的 在 Aurora DSQL 中 AWS 帳戶 具有下列叢集配額。若要請求增加特定內單一區域和多區域叢集的服務配額 AWS 區域，請使用 Service Quotas 主控台頁面。如需提高其他配額，請聯絡 AWS 支援。

Description	預設值限制	可設定？	Aurora DSQL 錯誤代碼
每個 的單一區域叢集上限 AWS 帳戶	20 個叢集	是	API 錯誤代碼 `ServiceQuotaExceededException`
每個 的多區域叢集上限 AWS 帳戶	5 個叢集	是	API 錯誤代碼 `ServiceQuotaExceededException`
每個叢集的最大儲存容量	10 TiB 預設限制，最多 256 TiB，並核准增加限制	是	`DISK_FULL(53100)`
每個叢集的最大連線數量	10,000 個連線	是	`TOO_MANY_CONNECTIONS(53300)`
每個叢集的最大連線率	每秒 100 個連線	否	`CONFIGURED_LIMIT_EXCEEDED(53400)`
每個叢集的最大高載容量	1,000 個連線	否	無錯誤代碼。
最大同時還原作業數量	4	否	無錯誤代碼。

Description	預設值限制	可設定？	Aurora DSQL 錯誤代碼
連線補充速率	每秒 100 個連線	否	無錯誤代碼。

Aurora DSQL 資料庫限制

下表說明 Aurora DSQL 的資料庫限制。

Description	預設值限制	可設定？	Aurora DSQL 錯誤代碼	錯誤訊息
主索引鍵中資料欄合併後的總大小上限	1 KiB	否	`54000`	`ERROR: key size too large`
次要索引中資料欄合併後的總大小上限	1 KiB	否	`54000`	`ERROR: key size too large`
資料表中單筆資料列大小上限	2 MiB	否	`54000`	`ERROR: maximum row size excee`
非索引資料欄的大小上限	1 MiB	否	`54000`	`ERROR: maximum column size ex`
主索引鍵或次要索引中的資料欄數量上限	8	否	`54011`	`ERROR: more than 8 column key are not supported`
每個資料表的欄位數量上限	255	否	`54011`	`ERROR: tables can have at mos`
每個資料表的索引數量上限	24	否	`54000`	`ERROR: more than 24 indexes p allowed`

Description	預設值限制	可設定？	Aurora DSQL 錯誤代碼	錯誤訊息
單次寫入交易可修改之資料總量上限	10 MiB	否	54000	`ERROR: transaction size limit DETAIL: Current transaction s 10mb`
交易區塊中可變更的資料表資料列數目上限	每筆交易 3,000 列。請參閱 Aurora DSQL 的 PostgreSQL 相容性考量。	否	54000	`ERROR: transaction row limit`
查詢作業可使用的基礎記憶體上限	每筆交易 128 MiB	否	53200	`ERROR: query requires too muc out of memory.`
資料庫內可定義的結構描述數量上限	10	否	54000	`ERROR: more than 10 schemas n`
資料庫中可建立的資料表數量上限	1,000 個資料表	否	54000	`ERROR: creating more than 100 allowed`
叢集中可建立的資料庫數量上限	1	否	無錯誤代碼。	`ERROR: unsupported statement`
單筆交易時間上限	5 分鐘	否	54000	`ERROR: transaction age limit exceeded`
連線持續時間上限	60 分鐘	否	無錯誤代碼。	無錯誤訊息。

Description	預設值限制	可設定？	Aurora DSQL 錯誤代碼	錯誤訊息
資料庫中可建立的檢視數量上限	5,000	否	54000	ERROR: creating more than 500 allowed
檢視定義的最大大小上限	2 MiB	否	54000	ERROR: view definition too la
序列數目上限	5,000	否	54000	ERROR: creating more than 500 not allowed

如需針對 Aurora DSQL 的資料型別限制，請參閱 Aurora DSQL 支援的資料類型。

Aurora DSQL API 參考

除了 AWS 管理主控台 與 AWS Command Line Interface (AWS CLI) 之外，Aurora DSQL 也提供 API 介面。您可以透過 API 作業管理 Aurora DSQL 中的資源。

如需依字母排序的 API 操作清單，請參閱動作。

如需依字母排序的資料類型清單，請參閱資料類型。

如需常用查詢參數的清單，請參閱常用參數。

如需錯誤碼的說明，請參閱常見錯誤。

如需 AWS CLI 的詳細資訊，請參閱 AWS Command Line Interface 參考 Aurora DSQL。

Aurora DSQL 問題故障診斷

Note

下列主題提供在使用 Aurora DSQL 時，針對可能發生的錯誤與問題之故障排除建議。若您發現此處未列出的問題，請聯絡 AWS 支援團隊。

主題

- 連線錯誤故障排除
- 身分驗證錯誤故障排除
- 授權錯誤故障排除
- SQL 錯誤疑難排解
- OCC 錯誤疑難排解
- SSL/TLS 連線疑難排解

連線錯誤故障排除

錯誤：無法辨識的 SSL 錯誤碼：6 或無法接受連線，未收到 SNI

您可能正在使用 psql 版本 14 之前的版本，這不支援伺服器名稱指示 (SNI)。連線至 Aurora DSQL 時必須啟用 SNI。

您可以使用 `psql --version` 檢查用戶端版本。

錯誤：NetworkUnreachable

連線嘗試期間出現的 `NetworkUnreachable` 錯誤可能表示您的用戶端不支援 IPv6 連線，而非真正的網路問題。此錯誤通常發生於僅支援 IPv4 的執行個體，原因在於 PostgreSQL 用戶端處理雙堆疊連線的方式。當伺服器支援雙堆疊模式時，用戶端會先將主機名稱解析為 IPv4 與 IPv6 位址。用戶端會先嘗試 IPv4 連線，若初始連線失敗再改用 IPv6。若系統不支援 IPv6，畫面將顯示一般性 `NetworkUnreachable` 錯誤，而非明確的「IPv6 不支援」訊息。

身分驗證錯誤故障排除

使用者 "..." 的 IAM 身分驗證失敗

產生 Aurora DSQL IAM 驗證權杖時，可設定的最長有效期為 1 週。一週後，該權杖將失效，無法再用於身分驗證。

此外，若您所假設的角色已過期，Aurora DSQL 將拒絕您的連線請求。例如，即使驗證權杖仍有效，若您使用臨時 IAM 角色嘗試連線，Aurora DSQL 仍會拒絕請求。

若要深入了解 IAM 與 Aurora DSQL 的整合機制，請參閱了解 Aurora DSQL 的身分驗證與授權及 Aurora DSQL 中的AWS Identity and Access Management。

呼叫 GetObject 操作時發生錯誤 (InvalidAccessKeyId)：您提供的 AWS 存取金鑰 ID 不存在於我們的記錄中

IAM 拒絕了您的請求。如需詳細資訊，請參閱為什麼要簽署請求。

IAM 角色 <role> 不存在

Aurora DSQL 找不到您的 IAM 角色。如需詳細資訊，請參閱 IAM 角色。

IAM 角色必須符合 IAM ARN 格式

如需詳細資訊，請參閱 IAM 識別碼 - IAM ARN。

錯誤的使用者對動作映射

當身分驗證字符類型不符合資料庫角色時，會發生此錯誤。Aurora DSQL 使用兩種字符類型：`DbConnectAdmin` 用於`admin`角色， `DbConnect`用於自訂資料庫角色。

- 如果您看到 `Wrong user to action mapping. user: admin, action: DbConnect`，請使用 `generate-db-connect-admin-auth-token`而非 `generate-db-connect-auth-token`。
- 如果您看到 `Wrong user to action mapping. user: *myusername*, action: DbConnectAdmin`，請使用 `generate-db-connect-auth-token`而非 `generate-db-connect-admin-auth-token`。

授權錯誤故障排除

角色 <role> 不受支援

Aurora DSQL 不支援 `GRANT` 操作。請參閱 Aurora DSQL 中支援的 SQL 命令子集。

無法使用角色 <role> 建立信任

Aurora DSQL 不支援 GRANT 操作。請參閱 Aurora DSQL 中支援的 SQL 命令子集。

角色 <role> 不存在

Aurora DSQL 找不到指定的資料庫使用者帳號。請參閱授權自訂資料庫角色以連線至叢集。

錯誤：拒絕以角色 <role> 授予 IAM 信任的權限

若要授與資料庫角色存取權限，您必須以管理員角色連線至叢集。若要進一步了解，請參閱授權資料庫角色以在資料庫中使用 SQL。

錯誤：角色 <role> 必須具有 LOGIN 屬性

您建立的任何資料庫角色都必須具有 LOGIN 權限。

若要解決此錯誤，請確定您已建立具備 LOGIN 權限的 PostgreSQL 角色。如需詳細資訊，請參閱 PostgreSQL 文件中的 CREATE ROLE 與 ALTER ROLE。

錯誤：無法刪除角色 <role>，因為部分物件仍依賴該角色

如果您刪除具有 IAM 關聯的資料庫角色，Aurora DSQL 會傳回錯誤，直到您使用 AWS IAM REVOKE 撤銷關聯為止。若要深入了解，請參閱撤銷授權。

SQL 錯誤疑難排解

錯誤：不支援的查詢

Aurora DSQL 並未支援所有以 PostgreSQL 為基礎的方言。若要了解支援項目，請參閱 Aurora DSQL 所支援的 PostgreSQL 功能。

錯誤：請改用 **CREATE INDEX ASYNC** 替代。

若要在包含現有資料列的資料表上建立索引，您必須使用 CREATE INDEX ASYNC 指令。若要進一步了解，請參閱在 Aurora DSQL 中非同步建立索引。

OCC 錯誤疑難排解

OC000「錯誤：變更與另一筆交易發生衝突，請視需要重試」

此交易嘗試修改與另一個並行交易相同的元組。這表示修改後的元組出現爭用。若要進一步了解，請參閱 Aurora DSQL 中的並行控制

OC001「錯誤：結構描述已被另一筆交易更新，請視需要重試」

您的 PostgreSQL 工作階段中含有結構描述目錄的快取副本。該快取副本在載入時是有效的。我們將該時間稱為 T1，版本稱為 V1。

另一筆交易在時間 T2 更新目錄。我們將此版本稱為 V2。

當原始工作階段在時間 T2 嘗試從儲存體讀取資料時，仍使用目錄版本 V1。Aurora DSQL 的儲存層會拒絕該請求，因為時間 T2 的最新目錄版本為 V2。

當您在原始工作階段於時間 T3 重新嘗試時，Aurora DSQL 會更新目錄快取。時間 T3 的交易使用目錄版本 V2。只要自時間 T2 以來未發生其他目錄變更，Aurora DSQL 即可完成該交易。

SSL/TLS 連線疑難排解

SSL 錯誤：憑證驗證失敗

此錯誤表示用戶端無法驗證伺服器憑證的有效性。請確認以下事項：

1. Amazon 根 CA 1 憑證已正確安裝。如需驗證與安裝此憑證的相關步驟，請參閱 為 Aurora DSQL 連線設定 SSL/TLS 憑證。
2. `PGSSLROOTCERT` 環境變數應指向正確的憑證檔案。
3. 憑證檔案已設定正確的權限。

無法識別的 SSL 錯誤碼：6

PostgreSQL 用戶端版本低於 14 時，會出現此錯誤。將 PostgreSQL 用戶端升級至版本 17 以解決此問題。

SSL 錯誤：未註冊的通訊協定 (Windows)

這是 Windows psql 用戶端在使用系統憑證時的已知問題。請依 從 Windows 進行連線 說明中的指示，使用下載憑證檔案的方法。

提供有關 Amazon Aurora DSQL 的意見回饋

如果您遇到對遷移至關重要但 Aurora DSQL 目前不支援的功能，AWS 會提供多個意見回饋管道：

意見回饋管道

Aurora DSQL Discord 伺服器

加入 Aurora DSQL Discord 伺服器以與 AWS 團隊和社群連線。共用功能請求、討論遷移挑戰，以及取得即時意見回饋。

AWS Support

如果您有 AWS Support 計畫，請建立支援案例來討論您的特定需求和時間表需求。

AWS re：Post

使用 AWS re：Post 提出問題，並與社群和 AWS 專家分享意見回饋。

有效的功能請求

請求功能時，請提供：

- 使用案例描述：說明您嘗試完成的內容和原因
- 目前的解決方法：描述您嘗試過的任何替代方案
- 業務影響：說明缺少的功能如何影響遷移時間表或應用程式功能
- 優先順序層級：指出這是否封鎖您的遷移，還是nice-to-have

Amazon Aurora DSQL 使用者指南的文件歷史記錄

下表說明 Aurora DSQL 的文件版本。

變更	描述	日期
新內容：適用於 .NET Npgsql 的 Aurora DSQL Connector	新增適用於 .NET Npgsql 的 Aurora DSQL Connector 文件，以自動 IAM 身分驗證包裝 Npgsql。連接器可處理 .NET 應用程式的字符產生、SSL 組態和連線管理。如需詳細資訊，請參閱使用 .NET Npgsql 連接器連線至 Aurora DSQL 叢集。	2026 年 3 月 20 日
更新內容：SQL 命令參考和系統查詢	將 START TRANSACTION 和 ROLLBACK 新增至交易控制命令參考，並以 END 和 ABORT 做為別名。新增了用於擷取 Aurora DSQL 和 PostgreSQL 版本資訊的實用系統查詢。如需詳細資訊，請參閱 PostgreSQL 相容性參考。	2026 年 3 月 13 日
更新內容：將資料載入 Aurora DSQL	更新資料載入指南，其中包含用戶端\copy用量、INSERT 最佳實務，以及在載入前預先建立資料表的指引。如需詳細資訊，請參閱將資料載入 Aurora DSQL。	2026 年 3 月 13 日
新內容：Aurora DSQL Connector for Ruby pg	新增 Aurora DSQL Connector for Ruby pg 的文件，該文件使用自動 IAM 身分驗證包裝 pg Gem。連接器可處理 Ruby 應用程式的字符產生、SSL 組態	2026 年 3 月 12 日

	和連線管理。如需詳細資訊，請參閱使用 Ruby pg 連接器連線至 Aurora DSQL 叢集。	
已更新內容：非同步 DDL 任務	更新sys.jobs文件，增加監控和管理非同步 DDL 操作的詳細資訊。如需詳細資訊，請參閱 PostgreSQL 相容性參考。	2026 年 3 月 6 日
更新內容：PHP 身分驗證字符產生	已將 PHP SDK 標籤新增至身分驗證字符產生頁面。如需詳細資訊，請參閱產生身分驗證字符。	2026 年 3 月 5 日
新內容：將資料載入 Aurora DSQL	新增將資料載入 Aurora DSQL 叢集的指南，包括 Aurora DSQL 載入器公用程式的使用。如需詳細資訊，請參閱將資料載入 Aurora DSQL。	2026 年 3 月 5 日
新內容：序列和身分資料欄	新增對序列和身分資料欄的支援。CREATE SEQUENCE、ALTER SEQUENCE、DROP SEQUENCE 和序列操作函數的新 SQL 命令參考頁面。更新 CREATE TABLE 和 ALTER TABLE 以包含身分資料欄語法。新增了選擇識別符類型和快取大小的新指南。如需詳細資訊，請參閱序列和身分資料欄。	2026 年 2 月 11 日

新內容：Aurora DSQL Connector for Go	新增 Aurora DSQL Connector for Go 的文件，該文件使用自動 IAM 身分驗證包裝 pgx。連接器會處理 Go 應用程式的字符產生、SSL 組態和連線管理。如需詳細資訊，請參閱使用 Go 連接器連線至 Aurora DSQL 叢集。	2026 年 2 月 5 日
更新內容：Amazon Aurora DSQL 叢集連線工具	重新組織叢集連線工具文件，以釐清 AWS所提供連接器、轉接器和第三方工具之間的差異。新增程式碼範例的遺失連結。如需詳細資訊，請參閱 Amazon Aurora DSQL 叢集連線工具。	2026 年 1 月 26 日
新內容：適用於 DBeaver Community Edition 的 Aurora DSQL 外掛程式	新增適用於 DBeaver Community Edition 的 Aurora DSQL 外掛程式文件，可啟用 Aurora DSQL 叢集的 IAM 身分驗證和簡化連線設定。包括安裝指示、連線組態和疑難排解指引。如需詳細資訊，請參閱使用 DBeaver 存取 Aurora DSQL。	2026 年 1 月 26 日
新內容：Aurora DSQL Driver for SQLTools	新增 Aurora DSQL Driver for SQLTools 的文件，這是一種 Visual Studio Code 延伸模組，可讓開發人員使用自動 IAM 身分驗證直接從 VS Code 連線和查詢 Aurora DSQL 資料庫。如需詳細資訊，請參閱使用 SQLTools 的 Aurora DSQL 驅動程式。	2026 年 1 月 26 日

更新內容：Aurora DSQL 轉向：技能和能力	新增支援使用 Skills CLI 進行代理程式無關支援安裝的文件。Skills CLI 提供簡化的設定方法，可用於多個 AI 編碼助理，包括 Claude Code、Cursor、Copilot、Gemini 等。如需詳細資訊，請參閱 Aurora DSQL 轉向：技能和能力。	2026 年 1 月 23 日
新內容：適用於 Tortoise ORM 的 Aurora DSQL 轉接器	新增對 Python 非同步 ORM 架構 Tortoise ORM 的支援。Aurora DSQL Adapter for Tortoise ORM 可讓開發人員搭配 Aurora DSQL 叢集使用 Tortoise ORM。如需詳細資訊，請參閱 Aurora DSQL 轉接器和方言。	2026 年 1 月 23 日
新內容：Aurora DSQL 轉向：技能和能力	新增了使用技能和能力使用 Aurora DSQL 設定 AI 轉向的新文件。包含 Kiro Powers、Claude Skills、Gemini Skills 和 Codex Skills 的設定指示。如需詳細資訊，請參閱 Aurora DSQL 轉向：技能和能力。	2026 年 1 月 16 日
數值資料類型索引支援	新增 Aurora DSQL 中`numeric`資料類型的索引支援。您現在可以在次要索引中使用`numeric`資料欄做為主索引鍵和 。如需詳細資訊，請參閱 Aurora DSQL 中支援的資料類型。	2026 年 1 月 13 日

更新內容：SQL 命令支援的子集	將 SQL 命令文件重新組織為不同的頁面，以改善導覽和清晰度。每個命令 (CREATE TABLE、ALTER TABLE、CREATE VIEW、ALTER VIEW、DROP VIEW) 現在都有自己的專用頁面。如需詳細資訊，請參閱 SQL 命令支援的子集。	2026 年 1 月 6 日
更新內容： AWS Labs Aurora DSQL MCP 伺服器	使用 Claude Code 和 Codex 的詳細安裝方法更新 MCP Server 文件，包括以 CLI 為基礎的設定和組態檔案範例。新增了跨不同開發工具尋找 MCP 用戶端組態檔案的完整指引。如需詳細資訊，請參閱 AWS Labs Aurora DSQL MCP Server。	2025 年 12 月 19 日
更新內容：從 PostgreSQL 遷移至 Aurora DSQL	重新起草 PostgreSQL 相容性區段，做為完整的遷移指南。包括架構相容性資訊、常見的遷移模式、架構差異和 AI 輔助遷移指導。新增提供 Aurora DSQL 意見回饋的新章節。如需詳細資訊，請參閱從 PostgreSQL 遷移至 Aurora DSQL。	2025 年 12 月 16 日

更新內容：使用 連線至 Aurora DSQL AWS PrivateLink	新增私有 DNS 設定和叢集 ID 連線選項的文件，以支援使用 AWS PrivateLink 搭配 Direct Connect 或 Amazon VPC 對等互連的客戶。包括使用連線選項不使用私有 DNS 進行`amzn-cluster-id` 連線的指引。如需詳細資訊，請參閱使用 管理和連線至 Aurora DSQL 叢集 AWS PrivateLink。	2025 年 12 月 11 日
更新內容：Aurora DSQL 叢集生命週期	更新 Aurora DSQL 叢集生命週期管理的文件。說明閒置和非作用中狀態期間可用的叢集狀態定義、狀態轉換和操作。如需詳細資訊，請參閱 Aurora DSQL 叢集生命週期。	2025 年 12 月 4 日
新內容：適用於 Python 和 Node.js 的 Aurora DSQL 連接器	新增適用於 Python (psycopg、psycopg2、asyncpg) 和 Node.js (node-postgres、Postgres.js) 的 Aurora DSQL 連接器的文件。這些連接器整合了 IAM 身分驗證，可將應用程式連線至 Aurora DSQL 叢集。如需詳細資訊，請參閱 Aurora DSQL 連接器。	2025 年 11 月 21 日
新內容：搭配 Aurora DSQL 使用 JupyterLab	新增使用 JupyterLab 搭配 Python 連接和查詢 Aurora DSQL step-by-step指南。包括本機 JupyterLab 安裝和 Amazon SageMaker AI 環境的說明。如需詳細資訊，請參閱搭配使用 JupyterLab 與 Aurora DSQL。	2025 年 11 月 20 日

更新內容：Aurora DSQL 的配額	將叢集儲存配額上限從 128 TiB 更新為 256 TiB。如需詳細資訊，請參閱 Aurora DSQL 的配額。	2025 年 11 月 19 日
新內容：Aurora DSQL 查詢編輯器入門	新增在 AWS 管理主控台中使用 Aurora DSQL 查詢編輯器的文件。包括先決條件、連線設定和查詢執行指示。如需詳細資訊，請參閱 Aurora DSQL 查詢編輯器入門。	2025 年 11 月 18 日
Amazon Aurora DSQL 的資源型政策支援	新增具有新許可的資源型政策 (RBP) 支援：`PutClusterPolicy` 、 `GetClusterPolicy` 和 `DeleteClusterPolicy` 。這些許可允許管理連接到 Aurora DSQL 叢集的內嵌政策，以進行精細存取控制。已更新 受管政策 AmazonAuroraDSQLFullAccess、AmazonAuroraDSQLReadOnlyAccess 和 AmazonAuroraDSQLConsoleFullAccess，以包含 RBP 功能。如需詳細資訊，請參閱 AWS Amazon Aurora DSQL 的 受管政策。	2025 年 10 月 15 日
Aurora DSQL JDBC 連接器	新增 Aurora DSQL JDBC Connector 的文件，這是整合 IAM 身分驗證的 PgJDBC 連接器，可將 Java 應用程式連線至 Amazon Aurora DSQL 叢集。如需詳細資訊，請參閱使用 JDBC 連接器連線至 Aurora DSQL 叢集。	2025 年 9 月 2 日

AWSAWS FIS 整合的 受管政策更新	已更新 `AmazonAuroraDSQLFullAccess` 和 `AmazonAuroraDSQLConsoleFullAccess` 政策，以支援與 Aurora DSQL 的 AWS Fault Injection Service 整合。這可讓您將故障注入單一區域和多區域 Aurora DSQL 叢集，以測試應用程式的容錯能力。如需這些政策的詳細資訊，請參閱 AWS 受管政策更新。	2025 年 8 月 19 日
Amazon Aurora DSQL 的一般可用性 (GA)	Amazon Aurora DSQL 現在已全面推出，並新增對 CloudWatch 監控、增強型資料保護功能和 AWS Backup 整合的支援。如需更多詳細資訊，請參閱使用 CloudWatch 監控 Aurora DSQL、Amazon Aurora DSQL 的備份和還原，以及 Amazon Aurora DSQL 的資料加密。	2025 年 5 月 27 日
AmazonAuroraDSQLFullAccess 更新	新增可執行 Aurora DSQL 叢集備份和還原操作的功能，包括啟動、停止和監控任務。此外也新增可使用客戶受管 KMS 金鑰進行叢集加密的功能。如需更多詳細資訊，請參閱 AmazonAuroraDSQLFullAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 21 日

AmazonAuroraDSQLConsoleFullAccess 更新	新增透過 AWS Console Home 執行 Aurora DSQL 叢集備份和還原操作的功能。其中包括啟動、停止和監控任務。此外還支援使用客戶受管 KMS 金鑰進行叢集加密和啟動 AWS CloudShell。如需詳細資訊，請參閱 AmazonAuroraDSQLConsoleFullAccess 和在 Aurora DSQL 中使用服務連結角色。	2025 年 5 月 21 日
AmazonAuroraDSQLReadOnlyAccess 更新	包括透過 Aurora DSQL 連線至 Aurora DSQL 叢集時判斷正確 VPC AWS PrivateLink 端點服務名稱的能力，可為每個儲存格建立唯一的端點，因此此 API 有助於確保您可以識別叢集的正確端點，並避免連線錯誤。如需更多詳細資訊，請參閱 AmazonAuroraDSQLReadOnlyAccess 及在 Aurora DSQL 使用服務連結角色。	2025 年 5 月 13 日

AmazonAuroraDSQLFullAccess 更新	政策會新增四個新許可，以跨多個 建立和管理資料庫叢集 AWS 區域：PutMultiRegionProperties 、AddPeerCluster 、 PutWitnessRegion 和 RemovePeerCluster 。這些許可權限包括資源層級控制和條件索引鍵，讓您可以控制自己能夠修改的叢集使用者。此政策也會新增 GetVpcEndpointServiceName 許可，協助您透過 連線至 Aurora DSQL 叢集 AWS PrivateLink。如需詳細資訊，請參閱 AmazonAuroraDSQLConsoleFullAccess 和在 Aurora DSQL 中使用服務連結角色。	2025 年 5 月 13 日
AmazonAuroraDSQLConsoleFullAccess 更新	將新許可權限新增至 Aurora DSQL，以支援多區域叢集管理和 VPC 端點連線。新的許可權限包括：PutMultiRegionProperties PutWitnessRegion AddPeerCluster RemovePeerCluster GetVpcEndpointServiceName 。請參閱 Aurora DSQL 中的 AmazonAuroraDSQLConsoleFullAccess 和使用服務連結角色。	2025 年 5 月 13 日

AuroraDsqlServiceLinkedRolePolicy 更新	新增將指標發佈至 AWS/AuroraDSQL 和 AWS/Usage CloudWatch 命名空間至政策的功能。這可讓相關聯的服務或角色，將更全面的使用和效能資料傳送到 CloudWatch 環境。如需更多詳細資訊，請參閱 AuroraDsqlServiceLinkedRolePolicy 和在 Aurora DSQL 中使用服務連結角色。	2025 年 5 月 8 日
AWS PrivateLink for Amazon Aurora DSQL	Aurora DSQL 現在支援 AWS PrivateLink。透過 AWS PrivateLink，您可以使用界面 Amazon VPC 端點和私有 IP 地址，簡化虛擬私有雲端 (VPCs)、Aurora DSQL 和內部部署資料中心之間的私有網路連線。如需更多詳細資訊，請參閱使用 AWS PrivateLink 管理和連線至 Amazon Aurora DSQL 叢集。	2025 年 5 月 8 日
初始版本	《Amazon Aurora DSQL 使用者指南》的初始版本。	2024 年 12 月 3 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。