Microsoft SQL Server
To Amazon Aurora with MySQL Compatibility

Migration Playbook

Version 1.8, September 2018

© 2018 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Table of Contents

Introduction **8**

Tables of Feature Compatibility **11**

AWS Schema and Data Migration Tools **23**

AWS Schema Conversion Tool (SCT) 24

SCT Action Code Index 38

AWS Database Migration Service (DMS) 54

ANSI SQL **57**

Migrate from SQL Server Constraints 58

Migrate to Aurora MySQL Constraints 62

Migrate from SQL Server Creating Tables 67

Migrate to Aurora MySQL Creating Tables 71

Migrate from SQL Server Common Table Expressions 77

Migrate to Aurora MySQL Common Table Expressions 80

Migrate from SQL Server Data Types 84

Migrate to Aurora MySQL Data Types 86

Migrate from SQL Server GROUP BY 96

Migrate to Aurora MySQL GROUP BY 100

Migrate from SQL Server Table JOIN 105

Migrate to Aurora MySQL Table JOIN 110

Migrate from SQL Server Views 114

Migrate to Aurora MySQL Views 117

Migrate from SQL Server Window Functions 122

Migrate to Aurora MySQL Window Functions 125

T-SQL **128**

Migrate from SQL Server Collations 129

Migrate to Aurora MySQL Collations 132

Migrate from SQL Server Cursors 141

Migrate to Aurora MySQL Cursors 143

Migrate from SQL Server Date and Time Functions 148
Migrate to Aurora MySQL Date and Time Functions 150
Migrate from SQL Server String Functions 153
Migrate to Aurora MySQL String Functions 156
Migrate from SQL Server Databases and Schemas 160
Migrate to Aurora MySQL Databases and Schemas 163
Migrate from SQL Server Transactions 166
Migrate to Aurora MySQL Transactions 170
Migrate from SQL Server DELETE and UPDATE FROM 174
Migrate to Aurora MySQL DELETE and UPDATE FROM 177
Migrate from SQL Server Stored Procedures 180
Migrate to Aurora MySQL Stored Procedures 184
Migrate from SQL Server Error Handling 189
Migrate to Aurora MySQL Error Handling 193
Migrate from SQL Server Flow Control 199
Migrate to Aurora MySQL Flow Control 202
Migrate from SQL Server Full-Text Search 207
Migrate to Aurora MySQL Full-Text Search 211
Migrate from SQL Server JSON and XML 216
Migrate to Aurora MySQL JSON and XML 219
Migrate from SQL Server MERGE 223
Migrate to Aurora MySQL MERGE 226
Migrate from SQL Server PIVOT and UNPIVOT 231
Migrate to Aurora MySQL PIVOT and UNPIVOT 235
Migrate from SQL Server Synonyms 238
Migrate to Aurora MySQL Synonyms 240
Migrate from SQL Server TOP and FETCH 241
Migrate to Aurora MySQL LIMIT (TOP and FETCH Equivalent) 244
Migrate from SQL Server Triggers 247
Migrate to Aurora MySQL Triggers 250

Migrate from SQL Server User Defined Functions 254
Migrate to Aurora MySQL User Defined Functions 257
Migrate from SQL Server User Defined Types 260
Migrate to Aurora MySQL User Defined Types 263
Migrate from SQL Server Sequences and Identity 266
Migrate to Aurora MySQL Sequences and Identity 271
Migrate from SQL Server Managing Statistics 276
Migrate to Aurora MySQL Managing Statistics 278
Configuration **281**
Migrate from SQL Server Session Options 282
Migrate to Aurora MySQL Session Options 285
Migrate from SQL Server Database Options 289
Migrate to Aurora MySQL Database Options 290
Migrate from SQL Server Server Options 291
Migrate to Aurora MySQL Server Options 293
High Availability and Disaster Recovery (HADR) **296**
Migrate from SQL Server Backup and Restore 297
Migrate to Aurora MySQL Backup and Restore 301
Migrate from SQL Server High Availability Essentials 309
Migrate to Aurora MySQL High Availability Essentials 313
Indexes **322**
Migrate from SQL Server Clustered and Non Clustered Indexes 323
Migrate to Aurora MySQL Clustered and Non Clustered Indexes 327
Management **331**
Migrate from SQL Server SQL Server Agent 332
Migrate to Aurora MySQL Agent 333
Migrate from SQL Server Alerting 336
Migrate to Aurora MySQL Alerting 338
Migrate from SQL Server Database Mail 342
Migrate to Aurora MySQL Database Mail 345

Migrate from SQL Server ETL 346
Migrate to Aurora MySQL ETL 348
Migrate from SQL Server Viewing Server Logs 374
Migrate to Aurora MySQL Viewing Server Logs 376
Migrate from SQL Server Maintenance Plans 379
Migrate to Aurora MySQL Maintenance Plans 382
Migrate from SQL Server Monitoring 388
Migrate to Aurora MySQL Monitoring 391
Migrate from SQL Server Resource Governor 392
Migrate to Aurora MySQL Resource Governor 394
Migrate from SQL Server Linked Servers 396
Migrate to Aurora MySQL Linked Servers 399
Migrate from SQL Server Scripting 400
Migrate to Aurora MySQL Scripting 402
Performance Tuning **407**
Migrate from SQL Server Execution Plans 408
Migrate to Aurora MySQL Execution Plans 410
Migrate from SQL Server Query Hints and Plan Guides 412
Migrate to Aurora MySQL Query Hints and Plan Guides 415
Physical Storage **419**
Migrate from SQL Server Partitioning 420
Migrate to Aurora MySQL Partitioning 422
Security **428**
Migrate from SQL Server Column Encryption 429
Migrate to Aurora MySQL Column Encryption 431
Migrate from SQL Server Data Control Language 433
Migrate to Aurora MySQL Data Control Language 434
Migrate from SQL Server Transparent Data Encryption 437
Migrate to Aurora MySQL Transparent Data Encryption 438
Migrate from SQL Server Users and Roles 443

Migrate to Aurora MySQL Users and Roles 445

Appendix A: SQL Server 2018 Deprecated Feature List 448

Appendix B: Migration Quick Tips 449

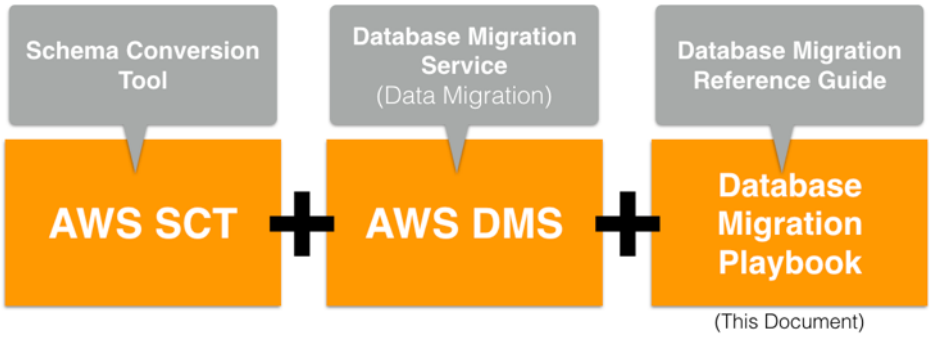
Management 449

SQL 449

Glossary 453

Introduction

The migration process from SQL Server to Amazon Aurora MySQL typically involves several stages. The first stage is to use the AWS Schema Conversion Tool (SCT) and the AWS Database Migration Service (DMS) to convert and migrate the schema and data. While most of the migration work can be automated, some aspects require manual intervention and adjustments to both database schema objects and database code.



The purpose of this Playbook is to assist administrators tasked with migrating SQL Server databases to Aurora MySQL with the aspects that can't be automatically migrated using the Amazon Web Services Schema Conversion Tool (AWS SCT). It focuses on the differences, incompatibilities, and similarities between SQL Server and Aurora MySQL in a wide range of topics including T-SQL, Configuration, High Availability and Disaster Recovery (HADR), Indexing, Management, Performance Tuning, Security, and Physical Storage.

The first section of this document provides an overview of AWS SCT and the AWS Data Migration Service (DMS) tools for automating the migration of schema, objects and data. The remainder of the document contains individual sections for SQL Server features and their Aurora MySQL counterparts. Each section provides a short overview of the feature, examples, and potential workaround solutions for incompatibilities.

You can use this playbook either as a reference to investigate the individual action codes generated by the AWS SCT tool, or to explore a variety of topics where you expect to have some incompatibility issues. When using the AWS SCT, you may see a report that lists *Action codes* , which indicates some manual conversion is required, or that a manual verification is recommended. For your convenience, this Playbook includes an SCT Action Code Index section providing direct links to the relevant topics that discuss the manual conversion tasks needed to address these action codes. Alternatively, you can explore the Tables of Feature Compatibility section that provides high-level graphical indicators and descriptions of the feature compatibility between SQL Server and Aurora MySQL. It also includes a graphical compatibility indicator and links to the actual sections in the playbook.

There are two appendices at the end of this playbook: Appendix A: SQL Server 2008 Deprecated Feature List provides focused links on features that were deprecated in SQL Server 2008R2. Appendix B: Migration Quick Tips provides a list of tips for SQL Server administrators or developers who have little experience with MySQL. It briefly highlights key differences between SQL Server and Aurora MySQL that they are likely to encounter.

Note that not all SQL Server features are fully compatible with Aurora MySQL, or have simple work-arounds. From a migration perspective, this document does not yet cover all SQL Server features and capabilities. This first release focuses on some of the most important features and will be expanded over time.

Disclaimer

The various code snippets, commands, guides, best practices, and scripts included in this document should be used for reference only and are provided as-is without warranty. Test all of the code, commands, best practices, and scripts outlined in this document in a non-production environment first. Amazon and its affiliates are not responsible for any direct or indirect damage that may occur from the information contained in this document.

Tables of Feature Compatibility

Feature Compatibility Legend

Compatibility Score Symbol	Description
	Very high compatibility: None or minimal low-risk and low-effort rewrites needed
	High compatibility: Some low-risk rewrites needed, easy work-arounds exist for incompatible features
	Medium compatibility: More involved low-medium risk rewrites needed, some redesign may be needed for incom-patible features
	Low compatibility: Medium to high risk rewrites needed, some incompatible features require redesign and reasonable-effort workarounds exist
	Very low compatibility: High risk and/or high-effort rewrites needed, some features require redesign and workarounds are challenging
	Not compatible: No practical workarounds yet, may require an application level architectural solution to work around incom-patibilities

SCT Automation Level Legend

SCT Automation Level Symbol	Description
	Full Automation SCT performs fully automatic conversion, no manual conversion needed.
	High Automation: Minor, simple manual conversions may be needed.
	Medium Automation: Low-medium complexity manual con-versions may be needed.
	Low Automation: Medium-high complexity manual con-versions may be needed.

SCT Automation Level Symbol	Description
	Very Low Automation: High risk or complex manual conversions may be needed.
	No Automation: Not currently supported by SCT, manual conversion is required for this feature.

ANSI SQL

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Constraints	Constraints	• Unsupported CHECK • Indexing require-ments for UNIQUE	
Creating Tables	Creating Tables	• IDENTITY vs. AUTO_INCREMENT • Primary key always clustered • CREATE TEMPORARY TABLE syntax • Unsupported @table variables	
Common Table Expres-sions	Common Table Expres-sions	• Rewrite non- recurs-ive CTE to use views and derived tables • Redesign recursive CTE code	
Data Types	Data Types	• Minor syntax and handling differences • No special UNICODE data types	
GROUP BY	GROUP BY	• Basic syntax com-patible • Advanced options like ALL, CUBE, GROUPING SETS will require rewrites to use multiple queries with UNION	
Table JOIN	Table JOIN	• Basic syntax com-patible • FULL OUTER, APPLY,	

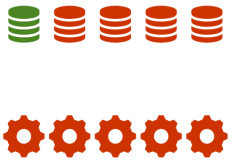
SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
		and ANSI SQL 89 outer joins will need to be rewritten	
Views	Views	• Minor syntax and handling differences • Indexes, Triggers, and temporary views not supported	
Windowed Functions	Windowed Functions	• Rewrite window functions to use alternative SQL syntax	

T-SQL

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Collations	Collations	• UNICODE uses CHARACTER SET property instead of NCHAR/NVARCHAR data types	
Cursors	Cursors	• Only static, forward only, read-only cursors are supported in Aurora MySQL	
Date and Time Functions	Date and Time Functions	• Timezone handling • Syntax differences	
String Functions	String Functions	• UNICODE paradigm (See Collations) • Syntax and option differences	
Databases and Schemas	Databases and Schemas	• SCHEMA and DATABASE are synonymous	
Transactions	Transactions	• Default isolation level REPEATABLE READ • Default mechanism CONSISTENT SNAPSHOT is similar to SQL Server's READ COMMITTED SNAPSHOT isolation • Syntax and option	

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
		differences	
DELETE and UPDATE FROM	DELETE and UPDATE FROM	• Rewrite to use sub-queries	
Stored Procedures	Stored Procedures	• No support for Table Valued Parameters • Syntax and option differences	
Error Handling	Error Handling	• Different paradigm and syntax requires rewrite of error handling code	
Flow Control	Flow Control	• Syntax and option differences, similar functionality	
Full Text Search	Full Text Search	• Syntax and option differences, less comprehensive but simpler • Most common basic functionality is similar • Requires rewrite of administration logic and queries	
JSON and XML	JSON and XML	• Minimal XML support, extensive JSON support • No XQUERY support, optionally convert to	

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
		JSON	
MERGE	MERGE	• Rewrite to use REPLACE and ON DUPLICATE KEY, or individual constituent DML statements	
PIVOT	PIVOT	• Straight forward rewrite to use traditional SQL syntax	
Triggers	Triggers	• Only FOR EACH ROW processing • No DDL or EVENT triggers • BEFORE triggers replace INSTEAD OF triggers	
User Defined Functions	User Defined Functions	• Scalar functions only, rewrite inline TVF as views or derived tables, and multi-statememt TVF as stored procedures	
User Defined Types	User Defined Types	• Replace scalar UDT with base types • Rewrite Stored Procedures that use table-type input parameters to use strings with CSV, XML, or JSON, or to process row-by-row	

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Sequences and Identity	Sequences and Identity	• SEQUENCE objects not supported • Rewrite IDENTITY to AUTO_INCREMENT • Last value evaluated as MAX(Existing Value) + 1 on every restart!	
Synonyms	Synonyms	• Use stored procedures and functions to abstract instance-wide objects	
TOP and FETCH	LIMIT (TOP alternative)	• Syntax rewrite, very similar functionality • Convert PERCENT and TIES to subqueries	

Configuration

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility
Session Options	Session Options	• SET options are significantly different, except for transaction isolation control	
Database Options	Database Options	• SQL Server's database options are inapplicable to Aurora MySQL	
Server Options	Server Options	• Use Cluster and Database Parameter Groups	

High Availability and Disaster Recovery (HADR)

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility
Backup and Restore	Backup and Restore	• Storage level backup managed by Amazon RDS	
High Availability Essentials	High Availability Essentials	• Multi replica, scale out solution using Amazon Aurora clusters and Availability Zones	

Indexes

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Clustered and Non Clustered Indexes	Clustered and Non Clustered Indexes	• Clustered primary keys only • Filtered indexes and included columns not supported	

Management

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility
SQL Server Agent	SQL Agent	• See Alerting and Maintenance Plans	
Alerting	Alerting	• Use Event Notifications Subscription with Amazon Simple Notification Service (SNS)	
Database Mail	Database Mail	• Use Lambda Integration	
ETL	ETL	• Use Amazon Glue for ETL	
Viewing Server Logs	Viewing Server Logs	• View logs from the Amazon RDS console, the Amazon	

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility
		RDS API, the AWS CLI, or the AWS SDKs	
Maintenance Plans	Maintenance Plans	• Backups via the RDS services • Table maintenance via SQL	
Monitoring	Monitoring	• Use Amazon Cloud Watch service	
Resource Governor	Resource Governor	• Use Per User Resource limit	
Linked Servers	Linked Servers	• Data transfer across schemas only, use a custom application solution to access remote instances	
Scripting	Scripting	• Non compatible tool sets and scripting languages • Use MySQL Work-bench, Amazon RDS API, AWS Man-agement Console, and Amazon CLI	

Performance Tuning

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Execution Plans	Execution Plans	• Syntax differences • Completely different optimizer with different operators and rules	
Query Hints and Plan Guides	Query Hints and Plan Guides	• Very limited set of hints - Index hints and optimizer hints as comments • Syntax differences	
Managing Statistics	Managing Statistics	• Statistics contain only density information, and only for index key columns	

Physical Storage

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility SCT Automation Level
Partitioning	Partitioning	• More partition types in Aurora MySQL with more restrictions on partitioned tables	

Security

SQL Server	Aurora MySQL	Key Differences	Feature Compatibility
Column Encryption	Column Encryption	• Syntax • Encryption hierarchy much simpler	
Data Control Language	Data Control Language	• Simpler permission hierarchy	
Transparent Data Encryption	Transparent Data Encryption	• Enable encryption when creating the database instance	
Users and Roles	Users and Roles	• No native role support in the database • Use AWS IAM accounts with the AWS Authentication Plugin	

AWS Schema and Data Migration Tools

AWS Schema Conversion Tool (SCT)

Overview

The AWS Schema Conversion Tool (SCT) is a stand alone tool that connects to source and target databases, scans the source database schema objects (tables, views, indexes, procedures, etc.), and converts them to target database objects.

This section provides a step-by-step process for using AWS SCT to migrate an SQL Server database to an Aurora MySQL database cluster. Since AWS SCT can automatically migrate most of the database objects, it greatly reduces manual effort.

It is recommended to start every migration with the process outlined in this section and then use the rest of the Playbook to further explore manual solutions for objects that could not be migrated automatically. Even though AWS SCT can automatically migrate most schema objects, it is highly recommended that you allocate sufficient resources to perform adequate testing, and performance tuning due to the differences between the SQL Server engine and the Aurora MySQL engine. For more information, see http://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/Welcome.html

Migrating a Database

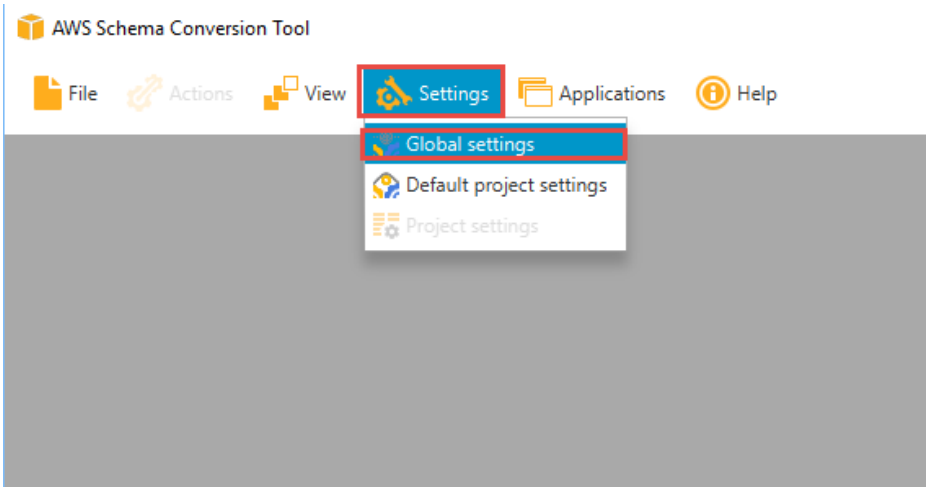
> **Note:** This walkthrough uses the AWS DMS Sample Database. You can download it from https://github.com/aws-samples/aws-database-migration-samples.

Download the Software and Drivers

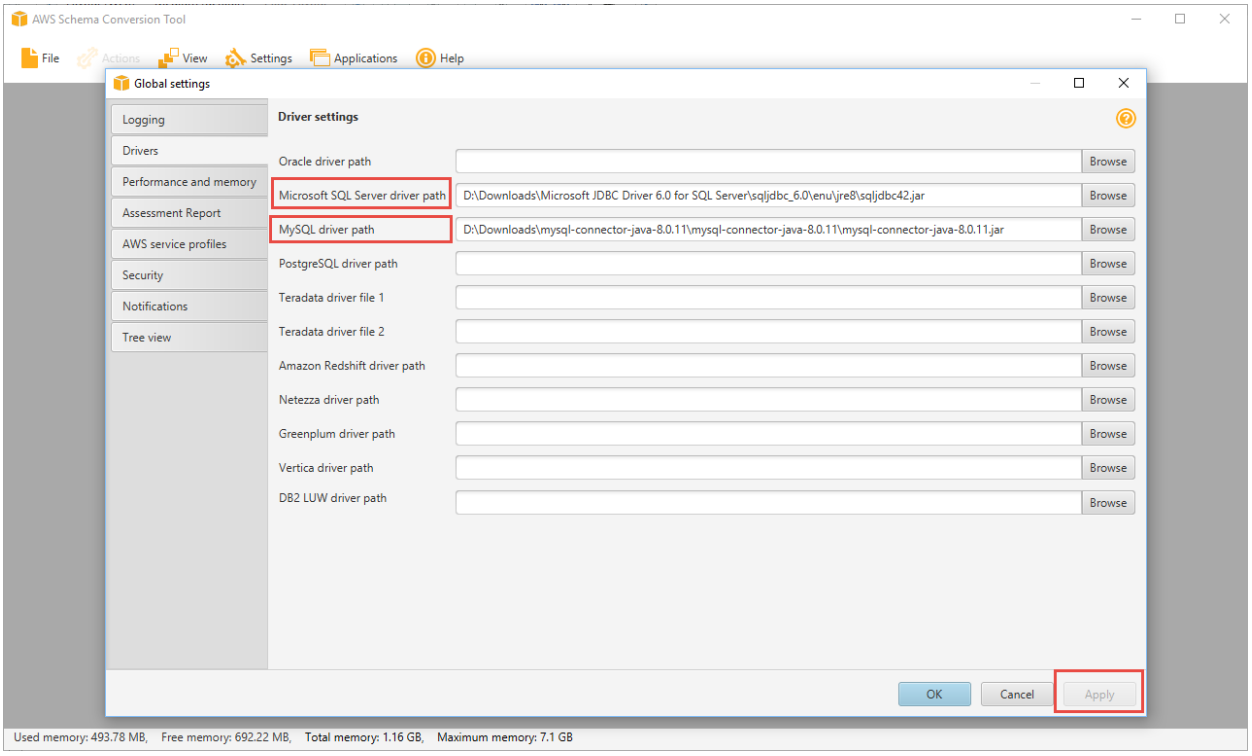
1. Download and install the AWS SCT from https://-docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Installing.html.
2. Download the SQL Server driver from https://www.microsoft.com/en-us/-download/details.aspx?displaylang=en&id=11774
3. Download the MySQL driver from https://www.mysql.com/products/connector/

Configure SCT

Launch SCT. Click the **Settings** button and select **Global Settings**.

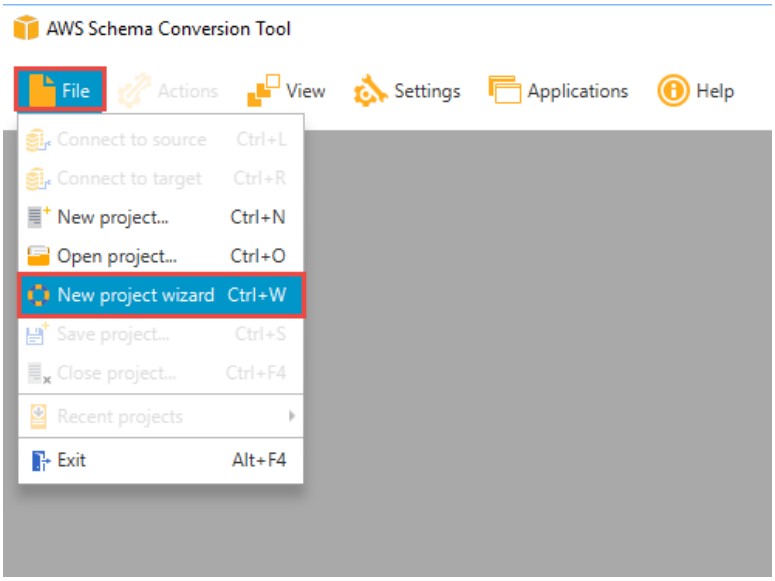


On the left navigation bar, click **Drivers**. Enter the paths for the SQL Server and MySQL drivers downloaded in the first step. Click **Apply** and then **OK**.

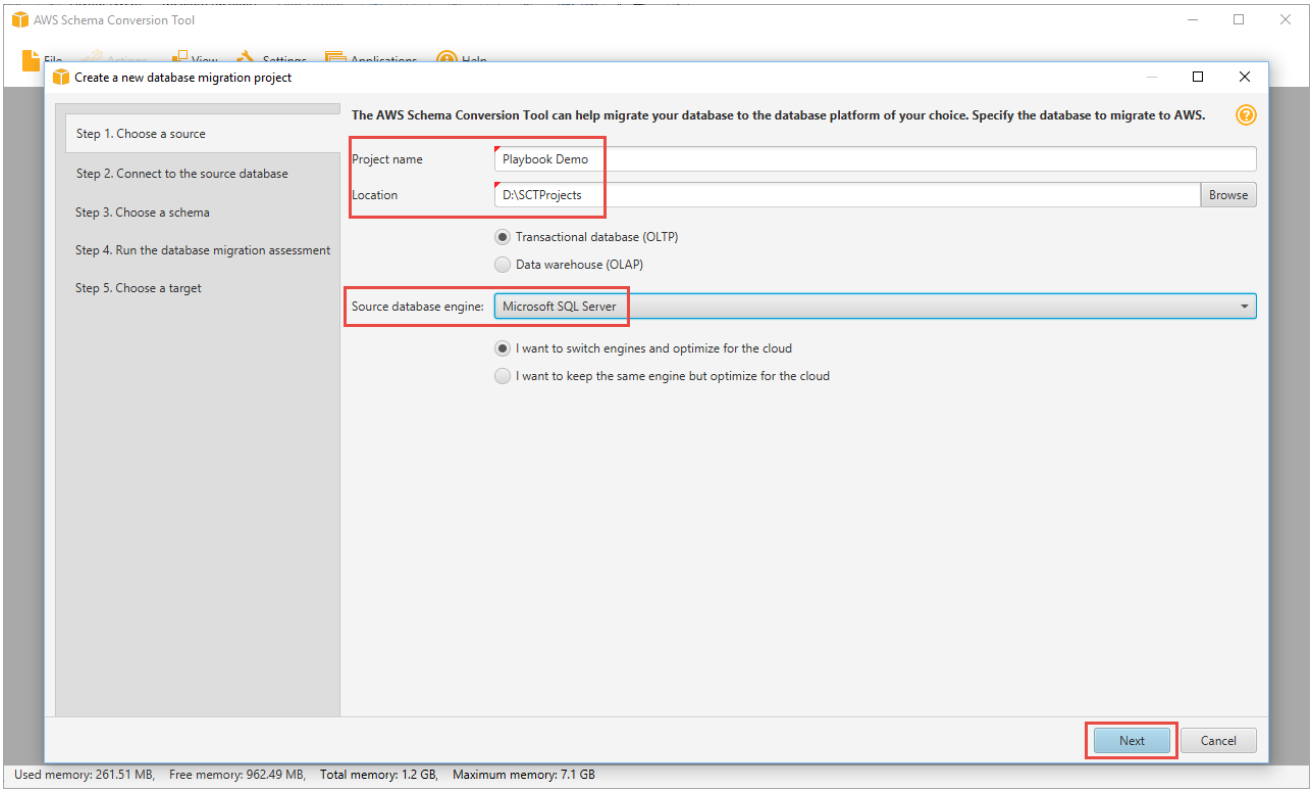


Create a New Migration Project

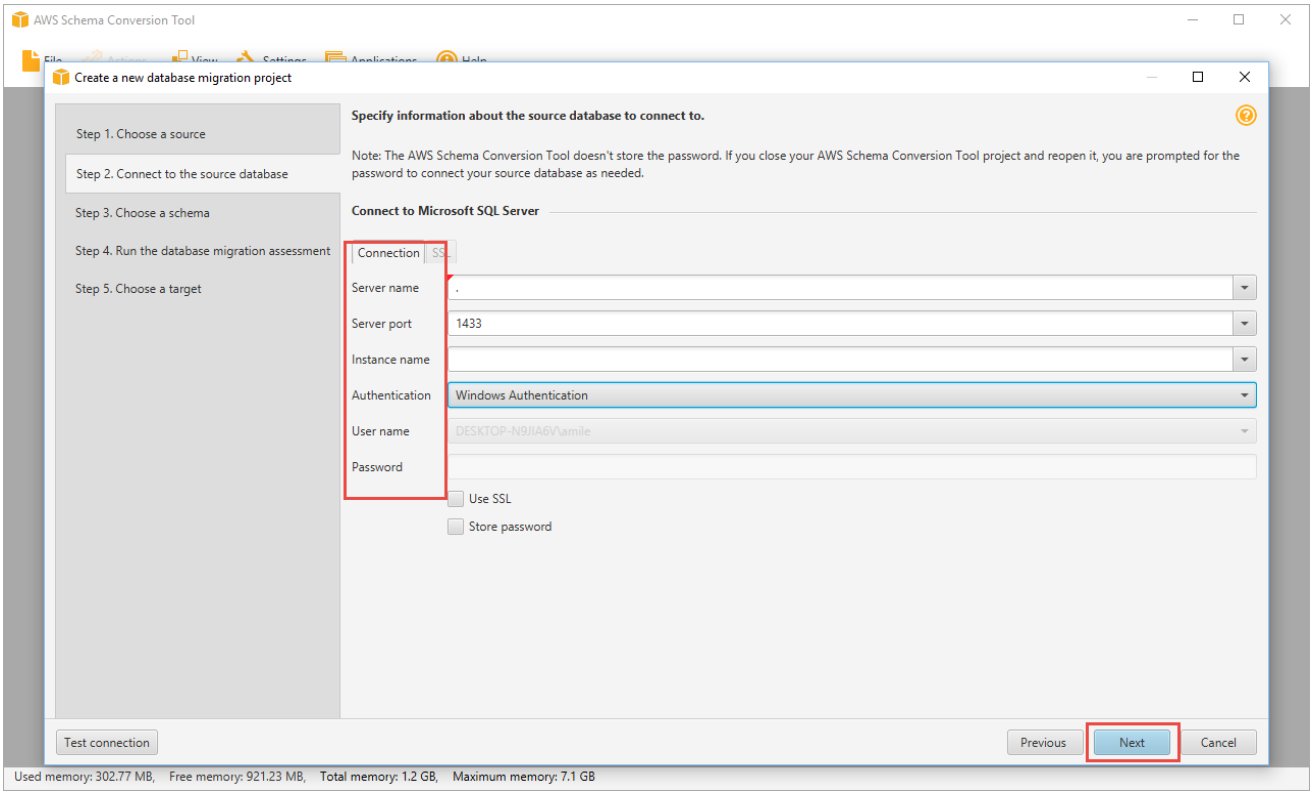
Click **File > New project wizard**. Alternatively, use the keyboard shortcut **<Ctrl+W>**.



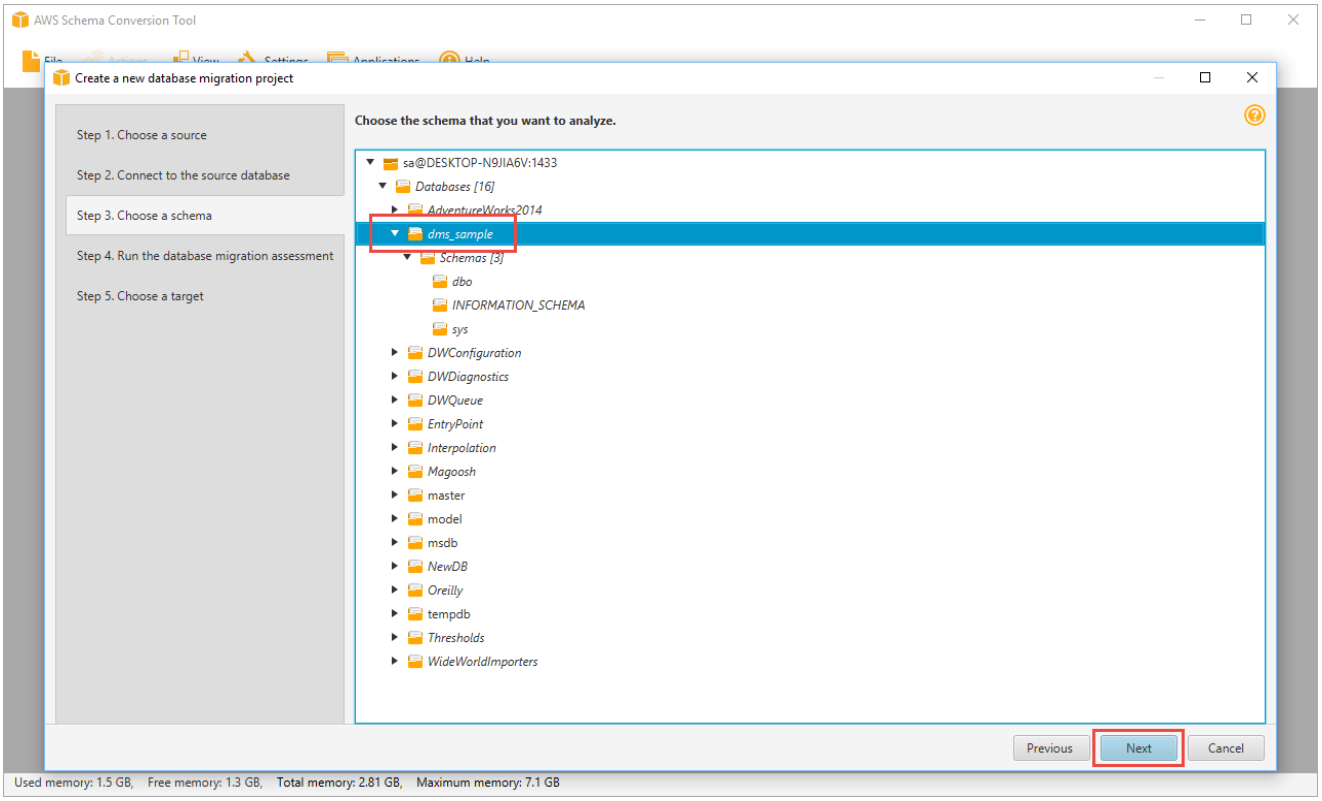
Enter a project name and select a location for the project files. Click **Next**.



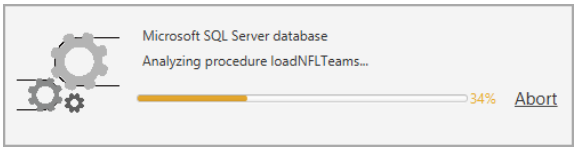
Enter connection details for the source SQL Server database and click **Test Connection** to verify. Click **Next**.



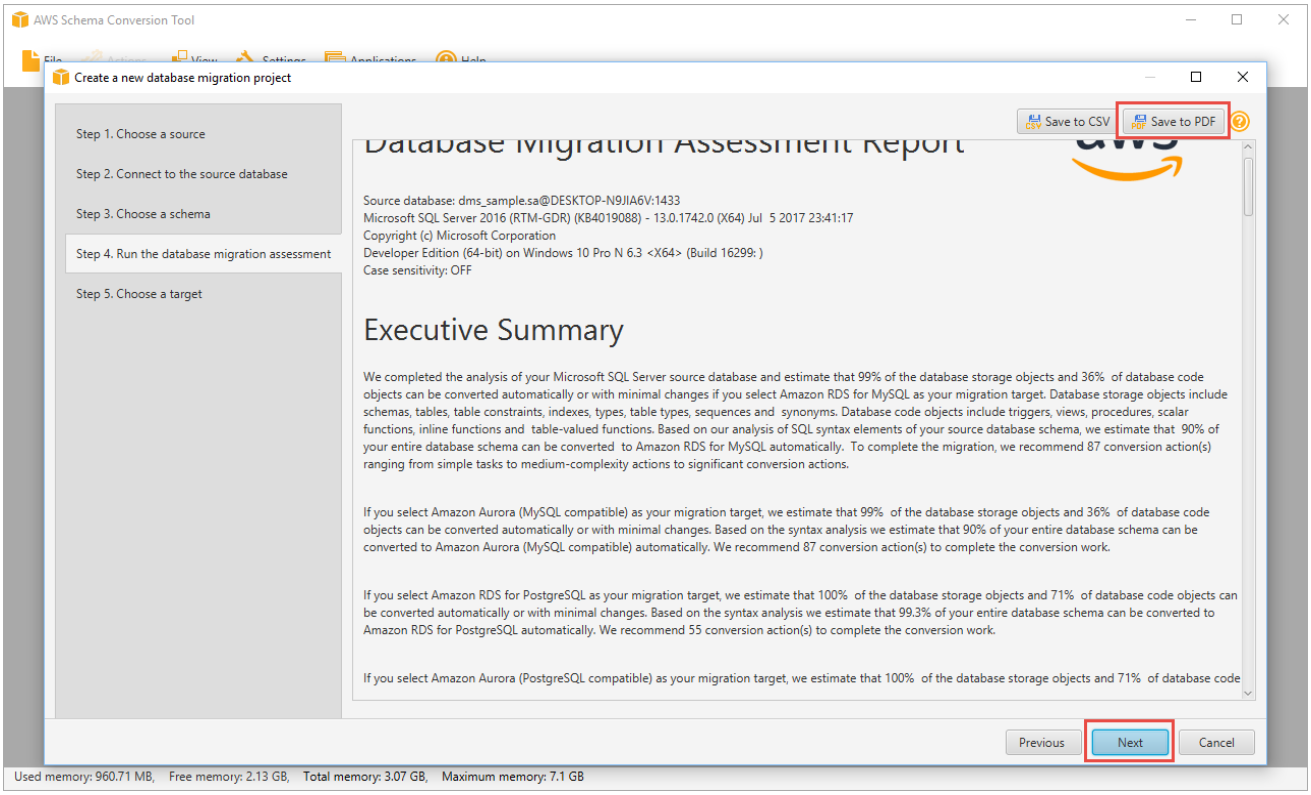
Select the schema or database to migrate and click **Next**.



The progress bar displays the objects being analyzed.



The Database Migration Assessment Report is displayed when the analysis completes. Read the Executive summary and other sections. Note that the information on the screen is only partial. To read the full report, including details of the individual issues, click **Save to PDF** and open the PDF document.



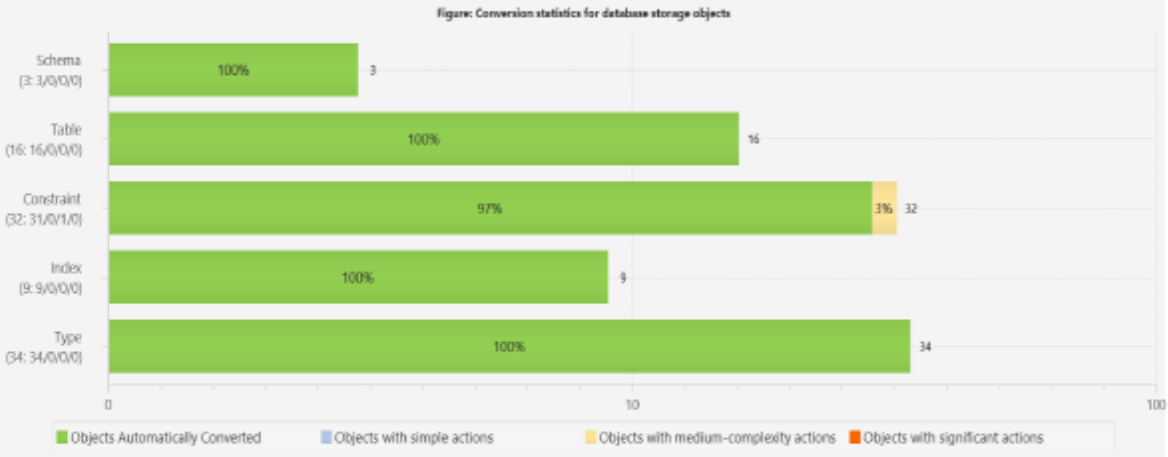
Scroll down to the section **Database objects with conversion actions for Amazon Aurora (MySQL compatible)**.

Database objects with conversion actions for Amazon Aurora (MySQL compatible)

Of the total 94 database storage object(s) and 14 database code object(s) in the source database, we identifed 93 (99%) database storage object(s) and 5 (36%) database code object(s) that can be converted to Amazon Aurora (MySQL compatible) automatically or with minimal changes.

1 (1%) database storage object(s) require 1 medium user action(s) to complete the conversion.

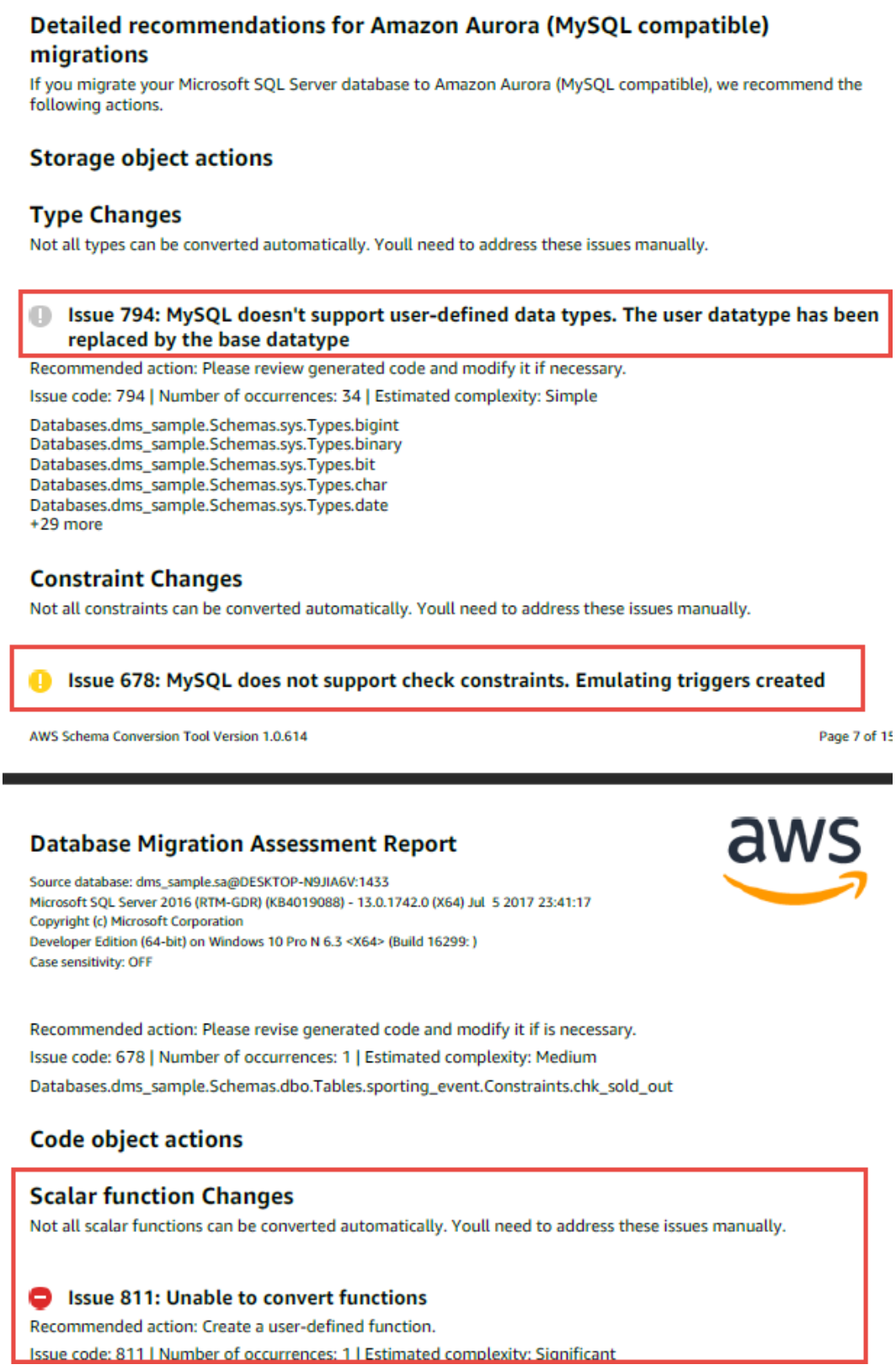
9 (64%) database code object(s) require 2 medium and 38 significant user action(s) to complete the conversion.



Scroll further down to the section **Detailed recommendations for Amazon Aurora (MySQL compatible) migrations**.

The example report below displays a few types of automatic corrections:

- The first issue, 794, has a gray exclamation mark indicating the automatic correction action has low risk. In this case, the user data type was replaced by the base data type. Note that you must verify that NULL constraints that may have been assigned to the type are preserved.
  For more information, see User Defined Types.
- The next issue, 678, has a yellow mark indicating a more significant higher risk. In this case, AWS SCT created triggers to replace the original CHECK constraints, which are not supported in Aurora MySQL.
- The last issue, 811, has a red exclamation mark indicating manual intervention is required. The scalar function could not be automatically converted and requires manual code changes.



Detailed recommendations for Amazon Aurora (MySQL compatible) migrations

If you migrate your Microsoft SQL Server database to Amazon Aurora (MySQL compatible), we recommend the following actions.

Storage object actions

Type Changes

Not all types can be converted automatically. Youll need to address these issues manually.

Issue 794: MySQL doesn't support user-defined data types. The user datatype has been replaced by the base datatype

Recommended action: Please review generated code and modify it if necessary.

Issue code: 794 | Number of occurrences: 34 | Estimated complexity: Simple

Databases.dms_sample.Schemas.sys.Types.bigint
Databases.dms_sample.Schemas.sys.Types.binary
Databases.dms_sample.Schemas.sys.Types.bit
Databases.dms_sample.Schemas.sys.Types.char
Databases.dms_sample.Schemas.sys.Types.date
+29 more

Constraint Changes

Not all constraints can be converted automatically. Youll need to address these issues manually.

Issue 678: MySQL does not support check constraints. Emulating triggers created

AWS Schema Conversion Tool Version 1.0.614 Page 7 of 15

Database Migration Assessment Report

Source database: dms_sample.sa@DESKTOP-N9JIA6V:1433
Microsoft SQL Server 2016 (RTM-GDR) (KB4019088) - 13.0.1742.0 (X64) Jul 5 2017 23:41:17
Copyright (c) Microsoft Corporation
Developer Edition (64-bit) on Windows 10 Pro N 6.3 <X64> (Build 16299:)
Case sensitivity: OFF

Recommended action: Please revise generated code and modify it if is necessary.

Issue code: 678 | Number of occurrences: 1 | Estimated complexity: Medium

Databases.dms_sample.Schemas.dbo.Tables.sporting_event.Constraints.chk_sold_out

Code object actions

Scalar function Changes

Not all scalar functions can be converted automatically. Youll need to address these issues manually.

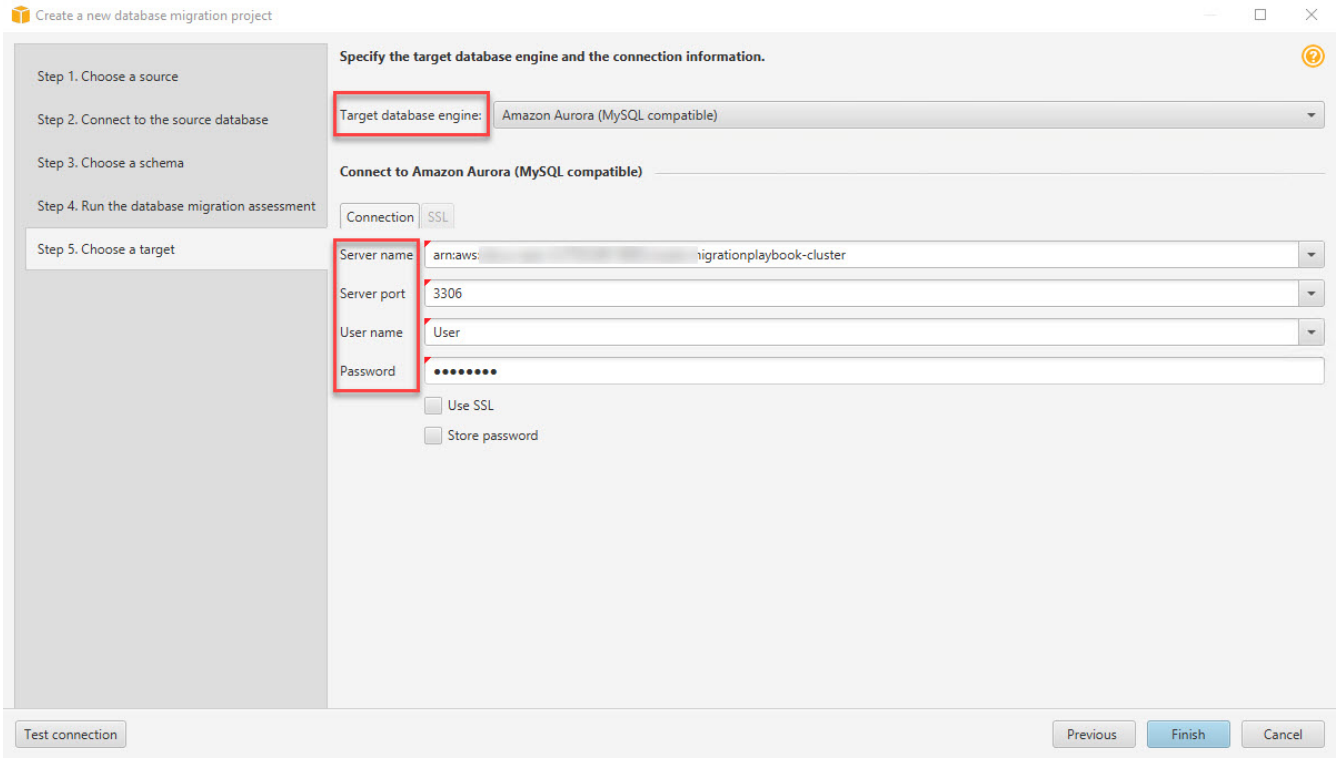
Issue 811: Unable to convert functions

Recommended action: Create a user-defined function.

Issue code: 811 | Number of occurrences: 1 | Estimated complexity: Significant

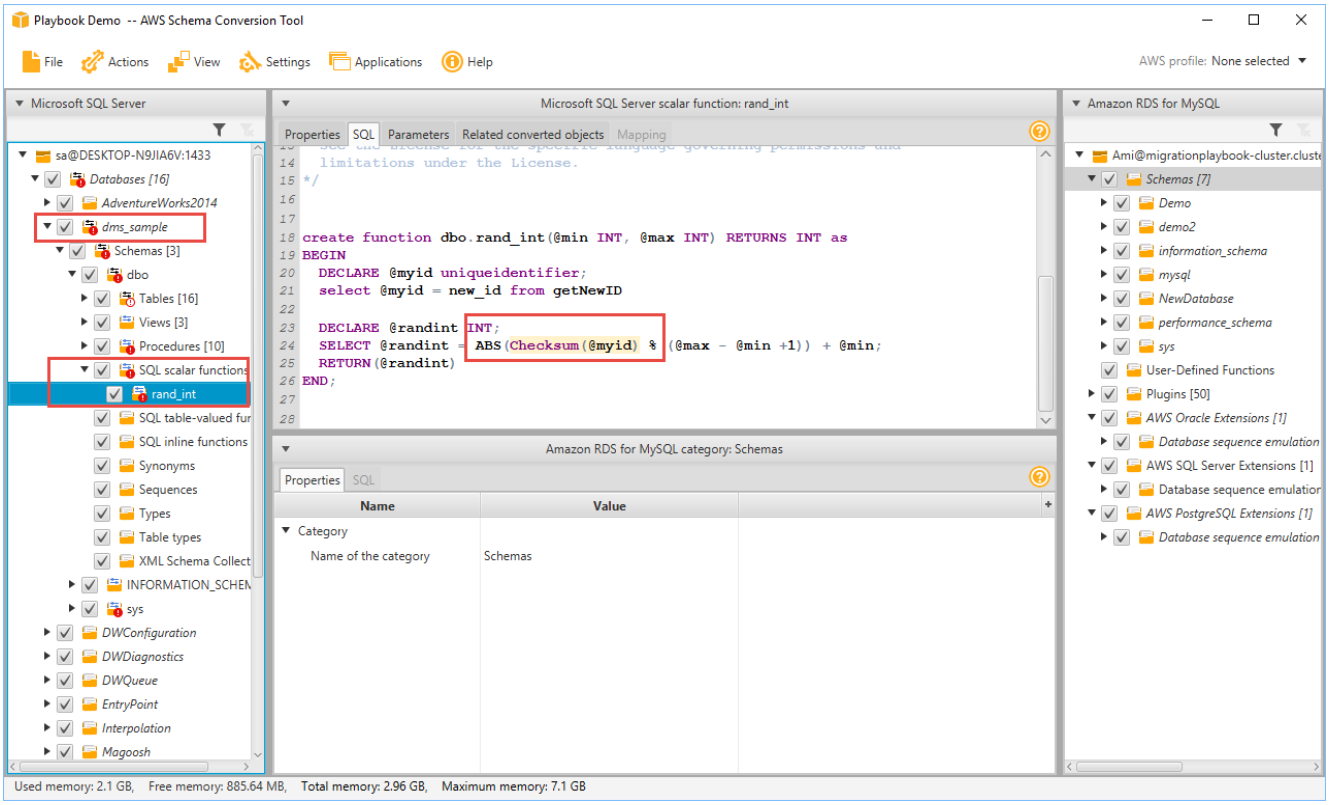
Return to AWS SCT and click **Next**. Enter the connection details for the target Aurora MySQL database and click **Finish**.

Note: The changes have not yet been saved to the target.

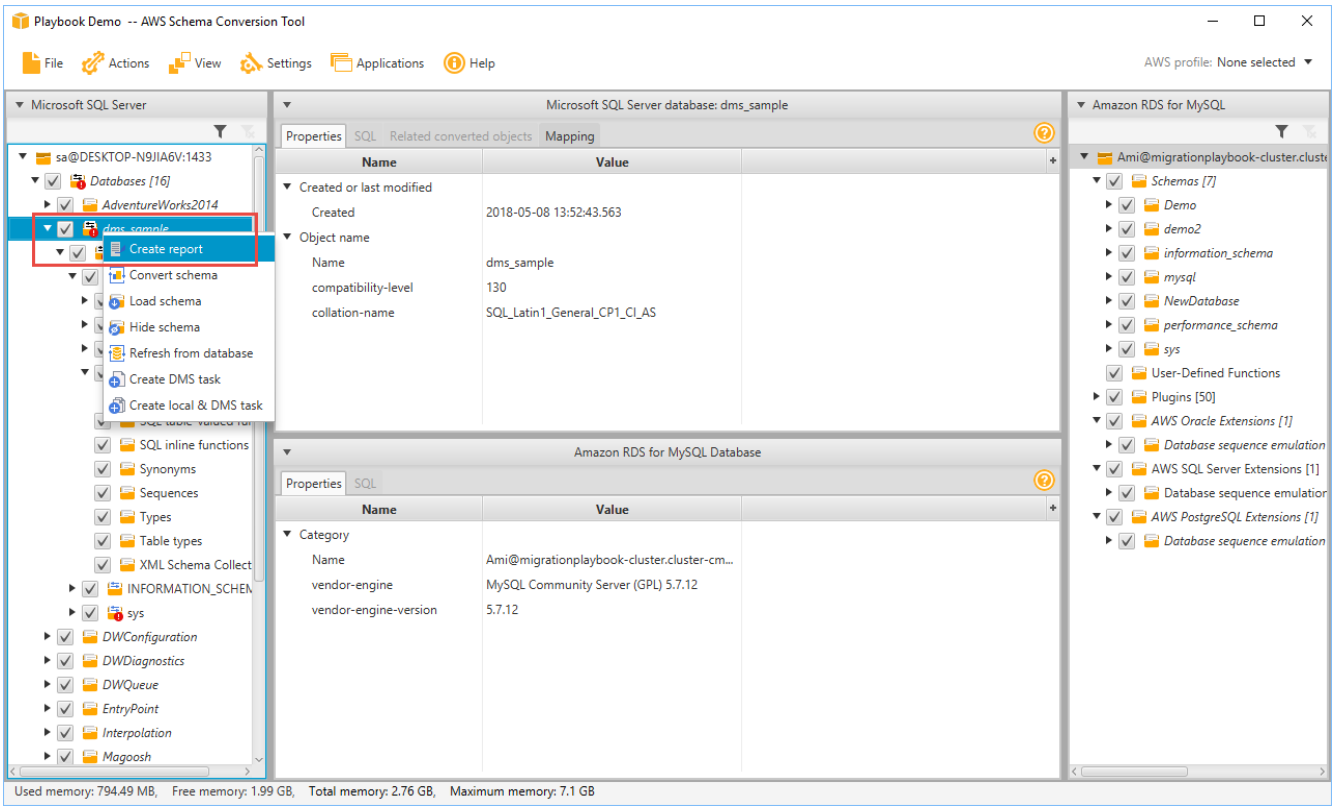


When the connection is complete, AWS SCT displays the main window. In this interface, you can explore the individual issues and recommendations discovered by AWS SCT.

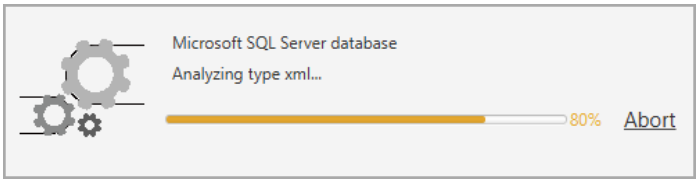
For example, expand **sample database > dbo default schema > SQL scalar functions > rand_int**. This issue has a red marker indicating it could not be automatically converted and requires a manual code change (issue 811 above). Select the object to highlight the incompatible code section.



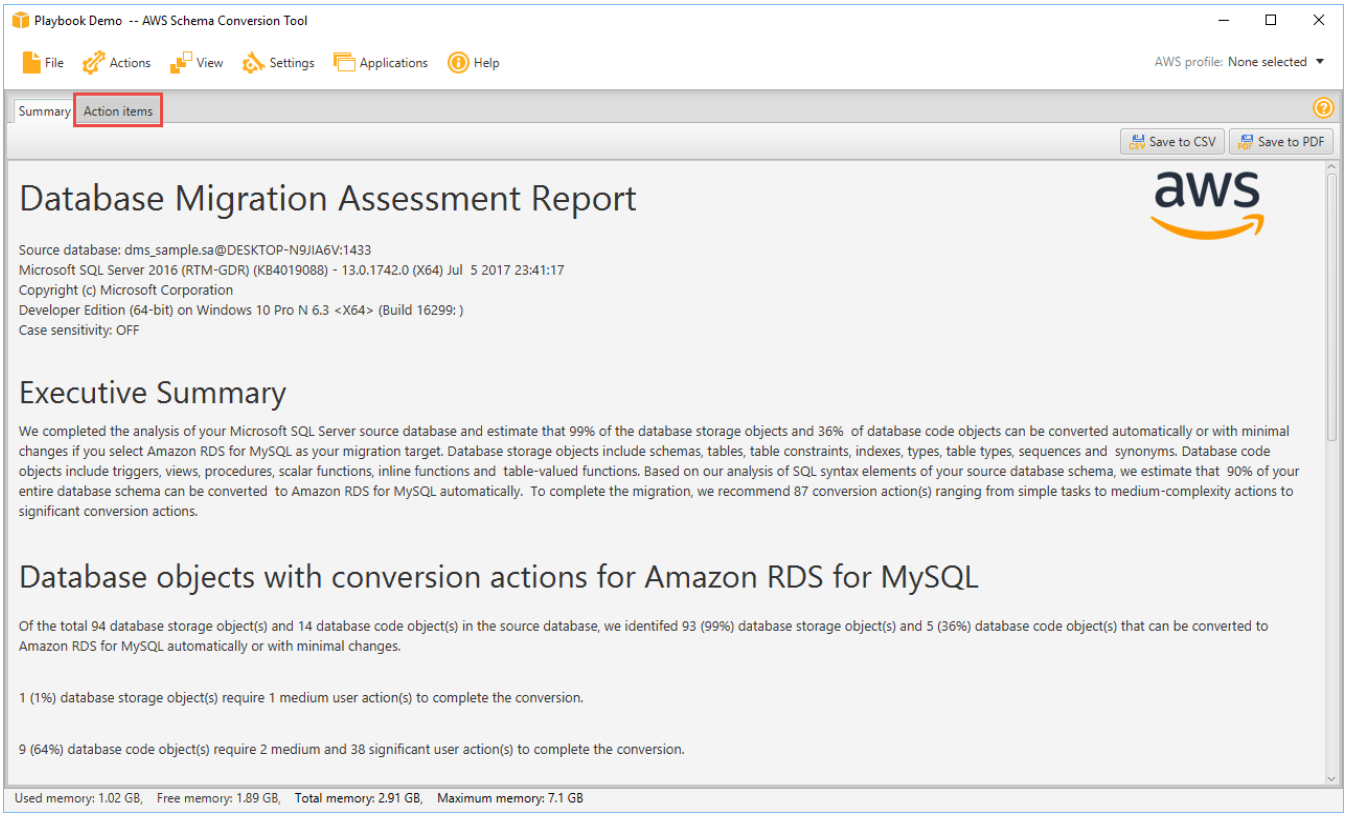
Right-click the database and then click **Create Report** to create a report tailored for the target database type. It can be viewed in AWS SCT.



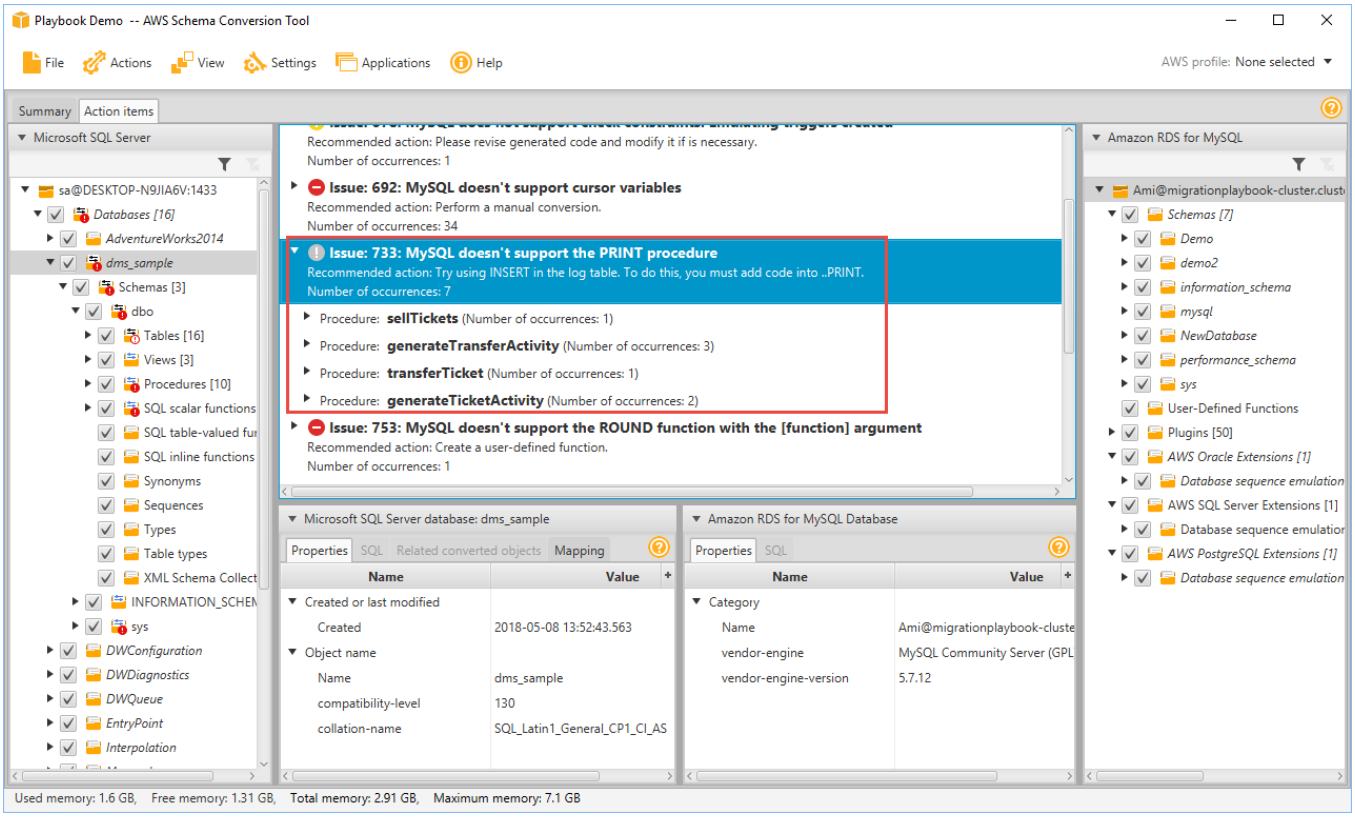
The progress bar updates while the report is generated.



The executive summary page displays. Click the **Action Items** tab.



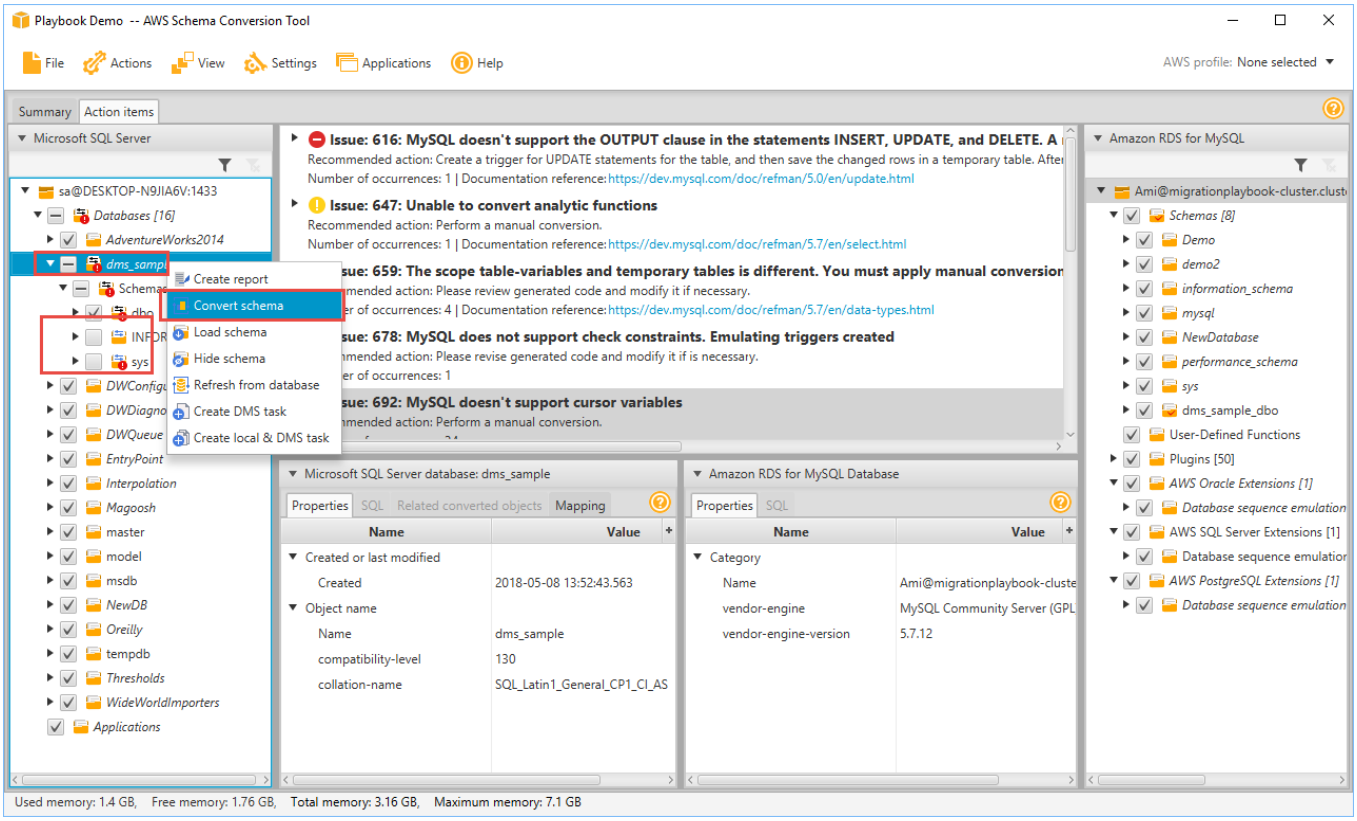
In this window, you can investigate each issue in detail and view the suggested course of action. For each issue, drill down to view all instances of that issue.



Right-click the database name and click **Convert Schema**.

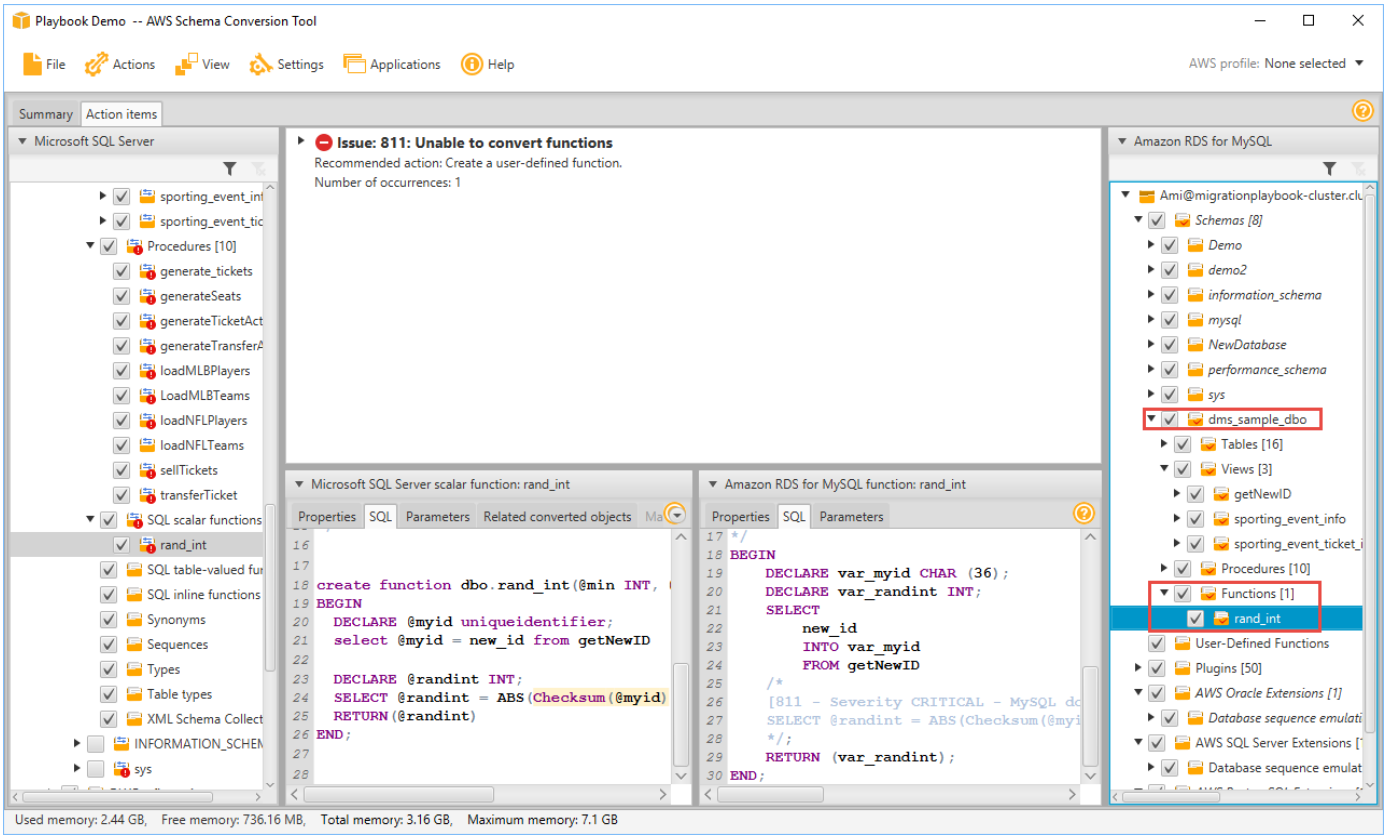
> **Note:** Be sure to uncheck the **sys** and **information_schema** system schemas. Aurora MySQL already has an **information_schema** schema.

> **Note:** This step does not make any changes to the target database.



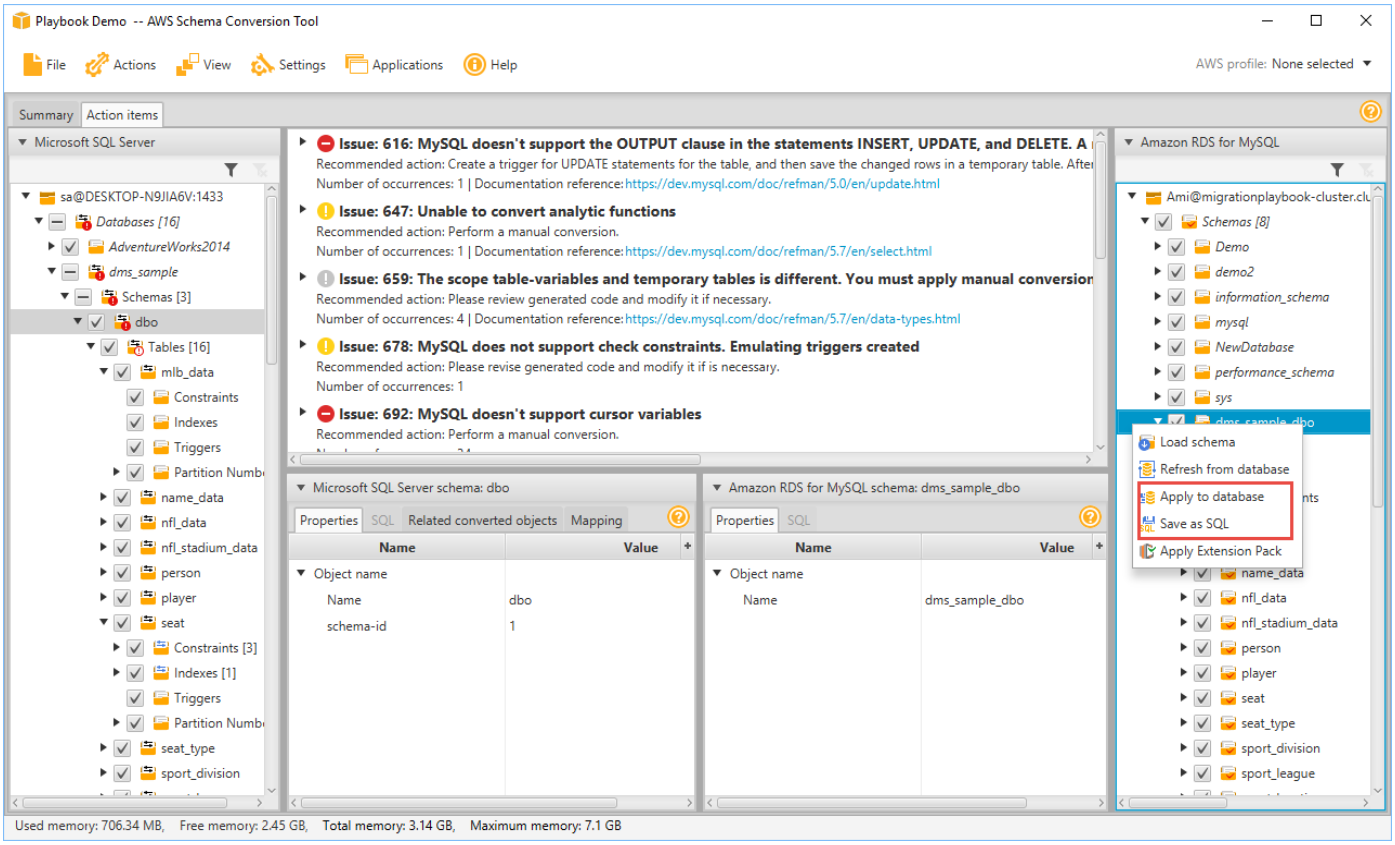
On the right pane, the new virtual schema is displayed as if it exists in the target database. Drilling down into individual objects displays the actual syntax generated by AWS SCT to migrate the objects.

For example, the rand_int scalar function was partly migrated using the appropriate Aurora MySQL variable declaration and types. The section with the CHECKSUM function was not migrated. In its place, AWS SCT inserted a comment block with the specific details about the issue.



Right-click the database on the right pane and choose either **Apply to database** to automatically execute the conversion script against the target database, or click **Save as SQL** to save to an SQL file. You can control the save file settings per SQL object type in the global settings, which may be very useful during data migrations.

Saving to an SQL file is recommended because it allows you to verify and QA the SCT code. Also, you can make the adjustments needed for objects that could not be automatically converted.



For more information, see https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Welcome.html

SCT Action Code Index

Legend

SCT Automation Level Symbol	Description
	Full Automation SCT performs fully automatic conversion, no manual conversion needed.
	High Automation: Minor, simple manual conversions may be needed.
	Medium Automation: Low-medium complexity manual conversions may be needed.
	Low Automation: Medium-high complexity manual conversions may be needed.
	Very Low Automation: High risk or complex manual conversions may be needed.
	No Automation: Not currently supported by SCT, manual conversion is required for this feature.

The following sections list the Schema Conversion Tool Action codes for topics that are covered in this playbook.

> **Note:** The links in the table point to the Microsoft SQL Server topic pages, which are immediately followed by the MySQL pages for the same topics.

Creating Tables

AWS SCT automatically converts the most commonly used constructs of the CREATE TABLE statement as both SQL Server and Aurora MySQL support the entry level ANSI compliance. These items include table names, containing security schema (or database), column names, basic column data types, column and table constraints, column default values, primary, candidate (UNIQUE), and foreign keys. Some changes may be required for computed columns and global temporary tables.

For more details, see Creating Tables.

Action Code	Action Message
659	The scope table-variables and temporary tables is different. You must apply manual conversion, if you are using recursion
679	A computed column is replaced by the trigger
680	MySQL doesn't support global temporary tables

Constraints

Most constraints are automatically converted by AWS SCT as as both SQL Server and Aurora MySQL support the entry level ANSI compliance. These items include primary keys, foreign keys, null constraints, unique constraints, and default constraints with some exceptions. Manual conversions are required for some foreign key cascading options. Check constraints are replaced with triggers by AWS SCT, and some default expressions for DateTime columns are not supported for automatic conversion. Complex expressions for other default values cannot be automatically converted by AWS SCT.

For more details, see Constraints .

Action Code	Action Message
676	MySQL Doesn't support the referential action SET DEFAULT
677	MySQL doesn't support using a function or expressions as a default value and default value for BLOB/TEXT columns. It is emulated by trigger
678	MySQL does not support check constraints. Emulating triggers created
825	The default value for a DateTime column removed
826	Check the default value for a DateTime variable
827	Unable to convert default value

Data Types

Data type syntax and rules are very similar between SQL Server and Aurora MySQL and most are converted automatically by AWS SCT. Note that date and time handling paradigms are different for SQL Server and Aurora MySQL and require manual verifications and/or conversion. Also note that due to dif-

ferences in data type behavior between SQL Server and Aurora MySQL, manual verification and strict testing are highly recommended.

For more details, see Data Types.

Action Code	Action Message
601	BLOB and TEXT columns cannot be included in a foreign key
706	Unsupported data type %s of variable/column was replaced. Check the conversion result
707	Unable convert variable reference of unsupported %s datatype
708	Unable convert complex usage of unsupported %s datatype
775	Check the data type conversion. Possible loss of accuracy
844	MySQL expands fractional seconds support for TIME, DATETIME2 and DATETIMEOFFSET values, with up to microseconds (6 digits) of precision

Collations

The collation paradigms of SQL Server and Aurora MySQL are significantly different. The AWS SCT tool can successfully migrate most common use cases including data type differences such as NCHAR and NVARCHAR in SQL Server that do not exist in Aurora MySQL. Aurora MySQL provides more options and flexibility in terms of collations. Rewrites are required for explicit collation clauses that are not supported by Aurora MySQL.

For more details, see Collations.

Action Code	Action Message
646	MySQL doesn't support the COLLATE option. Automatic conversion ignores this clause

Window Functions

Aurora MySQL version 5.7 does not support windowed functions and they cannot be automatically converted by AWS SCT.

For workarounds using traditional SQL syntax, see Window Functions.

Action Code	Action Message
647	MySQL doesn't support analytic form of function %s
648	MySQL doesn't support the RANK function
649	MySQL doesn't support the DENSE_RANK function
650	MySQL doesn't support the NTILE function
754	MySQL doesn't support the STDEV function with the DISTINCT clause
755	MySQL doesn't support the STDEVP function with the DISTINCT clause
756	MySQL doesn't support the VAR function with the DISTINCT clause
757	MySQL doesn't support the VARP function with the DISTINCT clause

PIVOT and UNPIVOT

Aurora MySQL version 5.7 does not support the PIVOT and UNPIVOT syntax and it cannot be automatically converted by AWS SCT.

For workarounds using traditional SQL syntax, see PIVOT and UNPIVOT.

Action Code	Action Message
905	MySQL doesn't support the PIVOT clause for the SELECT statement
906	MySQL doesn't support the UNPIVOT clause for the SELECT statement

TOP and FETCH

Aurora MySQL supports the non-ANSI compliant (although popular with other common RDBMS engines) LIMIT... OFFSET operator for paging of results sets. Despite the differences, AWS SCT can automatically convert most common paging queries to use the Aurora MySQL syntax. Some options such as PERCENT and WITH TIES cannot be automatically converted and require manual conversion.

For more details, see TOP and FETCH.

Action Code	Action Message
604	MySQL doesn't support the PERCENT option, this option is ignored
605	MySQL doesn't support the WITH TIES option, this option is ignored
608	MySQL doesn't support the PERCENT option. A manual conversion is required
612	MySQL doesn't support the PERCENT option. A manual conversion is required
621	MySQL doesn't support the PERCENT option. A manual conversion is required
830	MySQL does not support LIMIT with IN/ALL/ANY/SOME subquery

Common Table Expressions

Aurora MySQL version 5.7 does not support common table expressions and they cannot be automatically converted by AWS SCT.

For workarounds using traditional SQL syntax, see Common Table Expressions.

Action Code	Action Message
611	MySQL doesn't support queries using the WITH syntax defined. A manual conversion is required
619	MySQL doesn't support common table expression (CTE) query definitions. A manual conversion is required
839	Unused CTE query definition.
840	Conversion of updated CTE is not supported.

Cursors

The most commonly used cursor operations are converted automatically by AWS SCT. These operations include forward-only, read only cursors, and the DECLARE CURSOR, CLOSE CURSOR, and FETCH

NEXT operations. Modifications through cursors and non-forward-only fetches, which are not supported by Aurora MySQL, require manual conversions.

For more details, see Cursors.

Action Code	Action Message
618	MySQL doesn't support use of a CURRENT OF clause for data manipulation language (DML) queries that are in the body of a cursor loop. A manual conversion is required
624	MySQL doesn't support use of a CURRENT OF clause for data manipulation language (DML) queries that are in the body of a cursor loop. A manual conversion is required
625	MySQL doesn't support procedure arguments of the CURSOR data type
637	MySQL doesn't support the GLOBAL CURSORS option. Manual conversion is required
638	MySQL doesn't support the SCROLL option in cursors
639	MySQL doesn't support dynamic cursors
667	MySQL doesn't support the %s option in cursors
668	MySQL doesn't support the FIRST option in cursors
669	MySQL doesn't support the PRIOR option in cursors
670	MySQL doesn't support the ABSOLUTE option in cursors
671	MySQL doesn't support the RELATIVE option in cursors. Possibly required manual conversion in the code somewhere
692	MySQL doesn't support cursor variables
700	The membership and order of rows never changes for cursors in MySQL, so this option are skipped
701	Setting this option corresponds to the typical behavior of cursors in MySQL, so this option are skipped
702	Because all MySQL cursors are read-only, so this option are skipped
703	MySQL doesn't support the option SCROLL_LOCKS, so this option are skipped
704	MySQL doesn't support the option OPTIMISTIC, so this option are skipped
705	MySQL doesn't support the option TYPE_WARNING, so this option are skipped

Action Code	Action Message
842	MySQL doesn't support the %s option in cursors

Flow Control

Although the flow control syntax of SQL Server differs from Aurora MySQL, the AWS SCT can convert most constructs automatically including loops, command blocks, and delays. Aurora MySQL does not support the GOTO command nor the WAITFOR TIME command, which require manual conversion.

For more details, see Flow Control.

Action Code	Action Message
628	MySQL doesn't support the GOTO option. Automatic conversion cannot be performed
691	MySQL doesn't support WAITFOR TIME feature

Transaction Isolation

Aurora MySQL supports the four transaction isolation levels specified in the SQL:92 standard: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE, all of which are automatically converted by AWS SCT. AWS SCT also converts BEGIN / COMMIT and ROLLBACK commands that use slightly different syntax. Manual conversion is required for named, marked, and delayed durability transactions that are not supported by Aurora MySQL.

For more details, see Transaction Isolation.

Action Code	Action Message
629	MySQL doesn't support named transactions. Automatic conversion cannot be performed
630	MySQL doesn't support the WITH MARK option. Automatic conversion cannot be performed
631	MySQL doesn't support distributed transactions
632	MySQL doesn't support named transactions. Automatic conversion cannot be per-

Action Code	Action Message
	formed
633	MySQL doesn't support the DELAYED_DURABILITY option. Automatic conversion ignores this clause

Stored Procedures

Aurora MySQL Stored Procedures provide very similar functionality to SQL Server stored procedures and can be automatically converted by AWS SCT. Manual conversion is required for procedures that use RETURN values and some less common EXECUTE options such as the RECOMPILE and RESULTS SETS options.

For more details, see Stored Procedures.

Action Code	Action Message
640	The EXECUTE with RECOMPILE option is ignored
641	The EXECUTE with RESULT SETS UNDEFINED option is ignored
642	The EXECUTE with RESULT SETS NONE option is ignored
643	The EXECUTE with RESULT SETS DEFINITION option is ignored
689	MySQL does not support returning a value from a procedure using the RETURN statement
695	MySQL doesn't support the execution of a procedure as a variable

Triggers

Aurora MySQL supports BEFORE and AFTER triggers for INSERT, UPDATE, and DELETE. However, Aurora MySQL triggers differ substantially from SQL Server's triggers, but most common use cases can be migrated with minimal code changes. Although AWS SCT can automatically migrate trigger code, manual inspection and potential code modifications may be requiredbecause Aurora MySQL triggers are executed once per row, not once per statement like SQL Server's triggers.

For more details, see Triggers.

Action Code	Action Message
686	MySQL doesn't support triggers with FOR STATEMENT

GROUP BY

GROUP BY queries are automatically converted by AWS SCT, except for CUBE and GROUPING SETS. These queries can be easily worked around, but they do require manual code changes.

For more details, see GROUP BY.

Action Code	Action Message
654	MySQL doesn't support the option GROUP BY CUBE. Automatic conversion cannot be performed
655	MySQL doesn't support the option GROUP BY GROUPING SETS. Automatic conversion cannot be performed

Identity and Sequences

Although the syntax for SQL Server IDENTITY and Aurora MySQL AUTO_INCREMENT auto-enumeration columns differs significantly, it can be automatically converted by AWS SCT. Some limitations imposed by Aurora MySQL require manual conversion such as explicit SEED and INCREMENT auto-enumeration columns that are not part of the primary key and the table-independent SEQUENCE objects.

For more details, see Sequences and Identity.

Action Code	Action Message
696	MySQL doesn't support an Identity column with seed and increment
697	MySQL doesn't support an Identity column outside a primary key
732	MySQL doesn't support an Identity column in a compound primary key
815	MySQL doesn't support sequences
841	MySQL does not allow numeric (x, 0) or decimal (x, 0) data type to be used in columns with AUTO_INCREMENT option, data type is replaced with a compatible one

Error handling

Aurora MySQL and SQL Server's error handling paradigms are significantly different; the former uses condition and handler objects. The basic error handling constructs are migrated automatically by AWS SCT, but due to the paradigm differences, it is highly recommended that strict inspection and validation of the migrated code are performed. Manual conversions are required for THROW with variables and for SQL Server's built in messages.

For more details, see Error Handling.

Action Code	Action Message
729	Unable to perform automatic migration of the THROW operator with variables
730	The error code has been truncated
814	Unable to perform an automatic migration of the RAISERROR operator with messages from sys.messages
837	MySQL and MS SQL Server handle errors differently, so check result of code conversion

Date and Time Functions

The most commonly used date and time functions are automatically converted by AWS SCT despite the significant difference in syntax. Be aware of differences in data types, time zone awareness, and locale handling as well the functions themselves and inspect the expression value output carefully. Some less commonly used options such as millisecond, nanosecond, and time zone offsets require manual conversion.

For more details, see Date and Time Functions.

Action Code	Action Message
759	MySQL doesn't support the DATEADD function with the [nanosecond] datepart
760	MySQL doesn't support the DATEDIFF function with the [week] datepart
761	MySQL doesn't support the DATEDIFF function with the [millisecond] datepart

Action Code	Action Message
762	MySQL doesn't support the DATEDIFF function with the [nanosecond] datepart
763	MySQL doesn't support the DATENAME function with the [millisecond] datepart
764	MySQL doesn't support the DATENAME function with the [nanosecond] datepart
765	MySQL doesn't support the DATENAME function with the [TZoffset] datepart
767	MySQL doesn't support the DATEPART function with the [nanosecond] datepart
768	MySQL doesn't support the DATEPART function with the [TZoffset] datepart
773	Unable to perform an automated migration of arithmetic operations with several dates

User Defined Functions

Aurora MySQL supports only scalar user defined functions, which are automatically converted by AWS SCT. Table valued user defined functions, both in-line and multi-statement, require manual conversion. Workarounds using views or derived tables should be straightforward in most cases.

For more details, see User Defined Functions.

Action Code	Action Message
777	MySQL doesn't support the table-valued functions
822	MySQL doesn't support the inline functions

User Defined Types

Aurora MySQL 5.7 does not support user defined types nor user defined table valued parameters. AWS SCT can convert standard user defined types by replacing it with their base types, but manual conversion is required for user defined table types, which are used for table valued parameters for stored procedures.

For more details, see User Defined Types.

Action Code	Action Message
690	MySQL doesn't support table types

Synonyms

Aurora MySQL version 5.7 does not support synonyms and they cannot be automatically converted by AWS SCT.

For more details, see Synonyms.

Action Code	Action Message
792	MySQL doesn't support a synonyms

XML and JSON

Aurora MySQL provides minimal support for XML, but it does offer a native JSON data type and more than 25 dedicated JSON functions. Despite these differences, the most commonly used basic XML functions can be automatically migrated by AWS SCT. Some options such as EXPLICIT, used in functions or with sub queries, require manual conversion.

For more details, see JSON and XML.

Action Code	Action Message
817	Automatic migration of the FOR XML statement with EXPLICIT option is not supported
818	Automatic migration of correlated subqueries with FOR XML option is not supported
843	Automatic migration of the FOR XML statement in functions is not supported

Table Joins

The most commonly used join types are automatically converted by AWS SCT including INNER, OUTER, and CROSS joins. APPLY joins, (AKA LATERAL joins) are not supported by Aurora MySQL and require manual conversion.

For more details, see Joins.

Action Code	Action Message
831	MySQL doesn't have analogue for the operators CROSS APPLY and OUTER APPLY in case when subquery after the operator has reference to the column of attachable table

MERGE

Aurora MySQL version 5.7 does not support the MERGE statement and it cannot be automatically converted by AWS SCT. Manual conversion is straight-forward in most cases.

For more details and potential workarounds, see MERGE.

Action Code	Action Message
832	MySQL doesn't support the MERGE statement

Query hints and plan guides

Basic query hints such as index hints can be converted automatically by AWS SCT, except for DML statements. Note that specific optimizations used for SQL Server may be completely inapplicable to a new query optimizer. It is recommended to start migration testing with all hints removed. Then, selectively apply hints as a last resort if other means such as schema, index, and query optimizations have failed. Plan guides are not supported by Aurora MySQL.

For more details, see Query hints and Plan Guides.

Action Code	Action Message
610	MySQL doesn't support hints in an update statement. The conversion skips options in the format WITH (Table_Hint_Limited)
617	MySQL doesn't support hints in update statements. The conversion skips options

Action Code	Action Message
	in the format WITH (Table_Hint_Limited)
623	MySQL doesn't support hints in update statements. The conversion skips options in the format WITH (Table_Hint_Limited)
823	MySQL doesn't support table hints in DML statements

Full Text Search

Migrating Full-Text indexes from SQL Server to Aurora MySQL requires a full rewrite of the code that deals with both creating, managing, and querying of Full-Text indexes. They cannot be automatically converted by AWS SCT.

For more details, see Full Text Search.

Action Code	Action Message
687	MySQL doesn't support the CONTAINS predicate
688	MySQL doesn't support the FREETEXT predicate

Indexes

Basic non-clustered indexes, which are the most commonly used type of indexes are automatically migrated by AWS SCT. User defined clustered indexes are not supported by Aurora MySQL as they are always created for the primary key. In addition, filtered indexes, indexes with included columns, and some SQL Server specific index options can not be migrated automatically and require manual conversion.

For more details, see Indexes.

Action Code	Action Message
602	MySQL has the InnoDB internal maximum key length
681	MySQL doesn't support creating indexes with a CLUSTER option. The user can't create CLUSTER INDEX, MySQL will create it automatically

Action Code	Action Message
682	MySQL doesn't support the INCLUDE option in indexes
683	MySQL doesn't support the WHERE option in indexes
684	MySQL doesn't support the WITH option in indexes

Partitioning

Because Aurora MySQL stores each table in its own file, and since file management is performed by AWS and cannot be modified, some of the physical aspects of partitioning in SQL Server do not apply to Aurora MySQL. For example, the concept of file groups and assigning partitions to file groups. Aurora MySQL supports a much richer framework for table partitioning than SQL Server, with many additional options such as hash partitioning, and sub partitioning. Due to the vast differences between partition creation, query, and management between Aurora MySQL and SQL Server, AWS SCT does not automatically convert table and index partitions. These items require manual conversion.

For more details, see Partitioning.

Action Code	Action Message
907	Unable to complete the automatic migration of tables arranged in several partitions

Backup

Migrating from a self-managed backup policy to a Platform as a Service (PaaS) environment such as Aurora MySQL is a complete paradigm shift. You no longer need to worry about transaction logs, file groups, disks running out of space, and purging old backups. Amazon RDS provides guaranteed continuous backup with point-in-time restore up to 35 days. Therefor, AWS SCT does not automatically convert backups.

For more details, see Backup and Restore.

Action Code	Action Message
903	MySQL does not have functionality similar to SQL Server Backup

SQL Server Mail

Aurora MySQL does not provide native support sending mail from the database.

For more details and potential workarounds, see Database Mail.

Action Code	Action Message
900	MySQL does not have functionality similar to SQL Server Database Mail

SQL Server Agent

Aurora MySQL does not provide functionality similar to SQL Server Agent as an external, cross-instance scheduler. However, Aurora MySQL does provide a native, in-database scheduler. It is limited to the cluster scope and can't be used to manage multiple clusters. Therefore, AWS SCT can not automatically convert Agent jobs and alerts.

For more details, see SQL Server Agent.

Action Code	Action Message
902	MySQL does not have functionality similar to SQL Server Agent

Linked Servers

Aurora MySQL does not support remote data access from the database. Connectivity between schemas is trivial, but connectivity to other instances require a custom solution; it can not be automatically converted by AWS SCT.

For more details, see Linked Servers.

Action Code	Action Message
645	MySQL doesn't support executing a pass-through command on a linked server

AWS Database Migration Service (DMS)

Overview

The AWS Database Migration Service (DMS) helps you migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. The AWS Database Migration Service can migrate your data to and from most widely-used commercial and open-source databases.

The service supports homogenous migrations such as Oracle to Oracle as well as heterogeneous migrations between different database platforms such as Oracle to Amazon Aurora or Microsoft SQL Server to MySQL. It also allows you to stream data to Amazon Redshift, Amazon DynamoDB, and Amazon S3 from any of the supported sources, which are Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, SAP ASE, SQL Server, IBM DB2 LUW, and MongoDB, enabling consolidation and easy analysis of data in a petabyte-scale data warehouse. AWS Database Migration Service can also be used for continuous data replication with high-availability.

When migrating databases to Aurora, Redshift or DynamoDB, you can use DMS free for six months.

All supported sources for DMS: https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Source.html

All supported target for DMS: https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Target.html

Migration Tasks That AWS DMS Performs

- In a traditional solution, you need to perform capacity analysis, procure hardware and software, install and administer systems, and test and debug the installation. AWS DMS automatically manages the deployment, management, and monitoring of all hardware and software needed for your migration. Your migration can be up and running within minutes of starting the AWS DMS configuration process.
- With AWS DMS, you can scale up (or scale down) your migration resources as needed to match your actual workload. For example, if you determine that you need additional storage, you can easily increase your allocated storage and restart your migration, usually within minutes. On the other hand, if you discover that you aren't using all of the resource capacity you configured, you can easily downsize to meet your actual workload.
- AWS DMS uses a pay-as-you-go model. You only pay for AWS DMS resources while you use them as opposed to traditional licensing models with up-front purchase costs and ongoing maintenance charges.
- AWS DMS automatically manages all of the infrastructure that supports your migration server including hardware and software, software patching, and error reporting.
- AWS DMS provides automatic failover. If your primary replication server fails for any reason, a backup replication server can take over with little or no interruption of service.
- AWS DMS can help you switch to a modern, perhaps more cost-effective database engine than the one you are running now. For example, AWS DMS can help you take advantage of the managed database services provided by Amazon RDS or Amazon Aurora. Or, it can help you move to

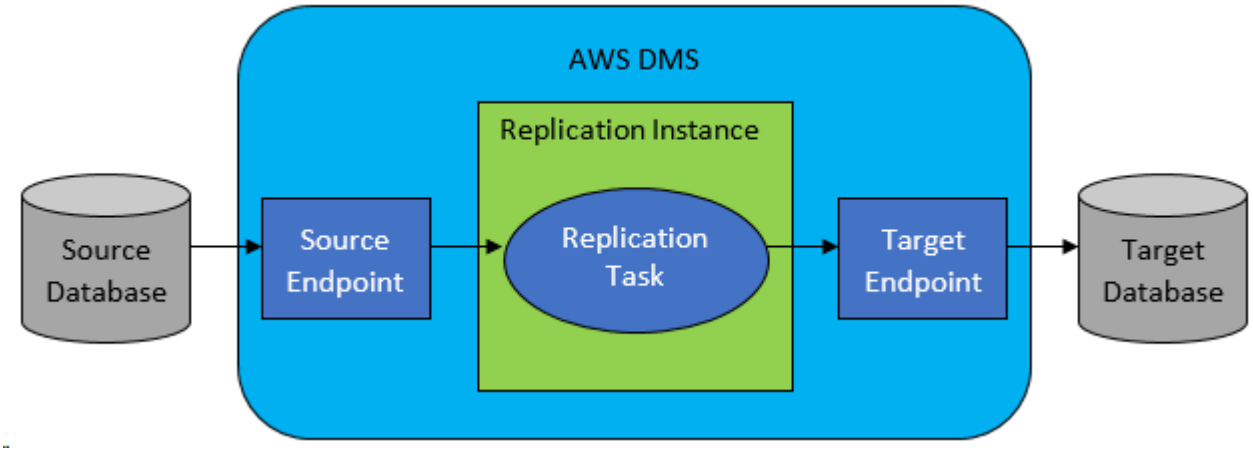
the managed data warehouse service provided by Amazon Redshift, NoSQL platforms like Amazon DynamoDB, or low-cost storage platforms like Amazon S3. Conversely, if you want to migrate away from old infrastructure but continue to use the same database engine, AWS DMS also supports that process.

- AWS DMS supports nearly all of today's most popular DBMS engines as data sources, including Oracle, Microsoft SQL Server, MySQL, MariaDB, PostgreSQL, Db2 LUW, SAP, MongoDB, and Amazon Aurora.
- AWS DMS provides a broad coverage of available target engines including Oracle, Microsoft SQL Server, PostgreSQL, MySQL, Amazon Redshift, SAP ASE, S3, and Amazon DynamoDB.
- You can migrate from any of the supported data sources to any of the supported data targets. AWS DMS supports fully heterogeneous data migrations between the supported engines.
- AWS DMS ensures that your data migration is secure. Data at rest is encrypted with AWS Key Management Service (AWS KMS) encryption. During migration, you can use Secure Socket Layers (SSL) to encrypt your in-flight data as it travels from source to target.

How AWS DMS Works

At its most basic level, AWS DMS is a server in the AWS Cloud that runs replication software. You create a source and target connection to tell AWS DMS where to extract from and load to. Then you schedule a task that runs on this server to move your data. AWS DMS creates the tables and associated primary keys if they don't exist on the target. You can pre-create the target tables manually if you prefer. Or you can use AWS SCT to create some or all of the target tables, indexes, views, triggers, and so on.

The following diagram illustrates the AWS DMS process.



For a complete guide with step-by-step walkthrough, including all the latest notes for migrating SQL Server to Aurora MySQL with DMS, see https://-docs.aws.amazon.com/dms/latest/sbs/CHAP_SQLServer2Aurora.html

For more information about DMS, see:

- *https://docs.aws.amazon.com/dms/latest/userguide/Welcome.html*
- *https://docs.aws.amazon.com/dms/latest/userguide/CHAP_BestPractices.html*

ANSI SQL

Migrate from SQL Server Constraints

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Constraints	• Unsupported CHECK • Indexing requirements for UNIQUE

Overview

Column and table constraints are defined by the SQL standard and enforce relational data consistency. There are four types of SQL constraints: Check Constraints, Unique Constraints, Primary Key Constraints, and Foreign Key Constraints.

Check Constraints

Syntax

```
CHECK (<Logical Expression>)
```

CHECK constraints enforce domain integrity by limiting the data values stored in table columns. They are logical boolean expressions that evaluate to one of three values: TRUE, FALSE, and UNKNOWN.

> **Note:** CHECK constraint expressions behave differently than predicates in other query clauses. For example, in a WHERE clause, a logical expression that evaluates to UNKNOWN is functionally equivalent to FALSE and the row is filtered out. For CHECK constraints, an expression that evaluates to UNKNOWN is functionally equivalent to TRUE because the value is permitted by the constraint.

Multiple CHECK constraints may be assigned to a column. A single CHECK constraint may apply to multiple columns (in this case, it is known as a Table-Level Check Constraint).

In ANSI SQL, CHECK constraints can not access other rows as part of the expression.
SQL Server allows using User Defined Functions in constraints to access other rows, tables, or even databases.

Unique Constraints

Syntax

```
UNIQUE [CLUSTERED | NONCLUSTERED] (<Column List>)
```

UNIQUE constraints should be used for all candidate keys. A candidate key is an attribute or a set of attributes (columns) that uniquely identify each tuple (row) in the relation (table data).

UNIQUE constraints guarantee that no rows with duplicate column values exist in a table.

A UNIQUE constraint can be simple or composite. Simple constraints are composed of a single column. Composite constraints are composed of multiple columns. A column may be a part of more than one constraint.

Although the ANSI SQL standard allows multiple rows having NULL values for UNIQUE constraints, SQL Server allows a NULL value for only one row. Use a NOT NULL constraint in addition to a UNIQUE constraint to disallow all NULL values.

To improve efficiency, SQL Server creates a unique index to support UNIQUE constraints. Otherwise, every INSERT and UPDATE would require a full table scan to verify there are no duplicates. The default index type for UNIQUE constraints is non- clustered.

Primary Key Constraints

Syntax

```
PRIMARY KEY [CLUSTERED | NONCLUSTERED] (<Column List>)
```

A PRIMARY KEY is a candidate key serving as the unique identifier of a table row. PRIMARY KEYS may consist of one or more columns. All columns that comprise a primary key must also have a NOT NULL constraint. Tables can have one primary key.

The default index type for PRIMARY KEYS is a clustered index.

Foreign Key Constraints

Syntax

```
FOREIGN KEY (<Referencing Column List>)
REFERENCES <Referenced Table>(<Referenced Column List>)
```

FOREIGN KEY constraints enforce domain referential integrity. Similar to CHECK constraints, FOREIGN KEYS limit the values stored in a column or set of columns.

FOREIGN KEYS reference columns in other tables, which must be either PRIMARY KEYS or have UNIQUE constraints. The set of values allowed for the referencing table is the set of values existing the referenced table.

Although the columns referenced in the parent table are indexed (since they must have either a PRIMARY KEY or UNIQUE constraint), no indexes are automatically created for the referencing columns in the child table. A best practice is to create appropriate indexes to support joins and constraint enforcement.

FOREIGN KEY constraints impose DML limitations for the referencing child table and for the parent table. The constraint's purpose is to guarantee that no "orphan" rows (rows with no corresponding matching values in the parent table) exist in the referencing table. The constraint limits INSERT and UPDATE to the child table and UPDATE and DELETE to the parent table. For example, you can not delete an order having associated order items.

Foreign keys support Cascading Referential Integrity (CRI). CRI can be used to enforce constraints and define action paths for DML statements that violate the constraints. There are four CRI options:

- **NO ACTION:** When the constraint is violated due to a DML operation, an error is raised and the operation is rolled back.
- **CASCADE:** Values in a child table are updated with values from the parent table when they are updated or deleted along with the parent.
- **SET NULL:** All columns that are part of the foreign key are set to NULL when the parent is deleted or updated.
- **SET DEFAULT:** All columns that are part of the foreign key are set to their DEFAULT value when the parent is deleted or updated.

These actions can be customized independently of others in the same constraint. For example, a cascading constraint may have CASCADE for UPDATE, but NO ACTION for UPDATE.

Examples

Create a composite non-clustered PRIMARY KEY.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
 PRIMARY KEY NONCLUSTERED (Col1, Col2)
);
```

Create a table-level CHECK constraint

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
 PRIMARY KEY NONCLUSTERED (Col1, Col2),
CONSTRAINT CK_MyTableCol1Col2
 CHECK (Col2 >= Col1)
);
```

Create a simple non-null UNIQUE constraint.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
 PRIMARY KEY NONCLUSTERED (Col1, Col2),
CONSTRAINT UQ_Col2Col3
```

```
 UNIQUE (Col2, Col3)
);
```

Create a FOREIGN KEY with multiple cascade actions.

```
CREATE TABLE MyParentTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PK_MyTable
 PRIMARY KEY NONCLUSTERED (Col1, Col2)
);
```

```
CREATE TABLE MyChildTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 INT NOT NULL,
Col3 INT NOT NULL,
CONSTRAINT FK_MyChildTable_MyParentTable
 FOREIGN KEY (Col2, Col3)
 REFERENCES MyParentTable (Col1, Col2)
 ON DELETE NO ACTION
 ON UPDATE CASCADE
);
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints*
- *https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints*

Migrate to Aurora MySQL Constraints

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Constraints	• Unsupported CHECK • Indexing requirements for UNIQUE

Overview

Similar to SQL Server, Aurora MySQL supports all ANSI constraint types, except CHECK.

> **Note:** You can work around some of the functionality of CHECK (<Column>) IN (<Value List>) using the SET and ENUM data types.
> For more information, see Data Types.

Unlike SQL Server, constraint names, or *symbols* in Aurora MySQL terminology, are optional. Identifiers are created automatically and are similar to SQL Server column constraints that are defined without an explicit name.

Unique Constraints

Unlike SQL Server where UNIQUE constraints are objects supported by unique indexes, Aurora MySQL only provides unique indexes. A unique index is the equivalent to a SQL Server UNIQUE constraint.

As with SQL Server, unique indexes enforce distinct values for index columns. If a new row is added or an existing row is updated with a value that matches an existing row, an error is raised and the operation is rolled back.

Unlike SQL Server, Aurora MySQL permits multiple rows with NULL values for unique indexes.

> **Note:** If a unique index consists of only one INT type column, the alias _rowid can be used to reference the index in SELECT statements.

Primary Key Constraints

Similar to SQL Server, a Primary Key constraint in Aurora MySQL is a unique index where all columns are NOT NULL. Each table can have only one PRIMARY KEY. The name of the constraint is always PRIMARY.

Primary keys in Aurora MySQL are always clustered. They cannot be configured as NON CLUSTERED like SQL Server. For more information, see Clustered and Non Clustered Indexes .

Applications can reference a PRIMARY KEY using the alias PRIMARY. If a table has no primary key (not recommended), Aurora MySQL uses the first NOT NULL and UNIQUE index.

> **Note:** Keep the primary key short to minimize storage overhead for secondary indexes. In Aurora MySQL, the primary key is clustered. Therefore, every secondary (non clustered)

index maintains a copy of the clustering key as the row pointer.
It is also recommended to create tables and declare the primary key first, followed by the unique indexes. Then create the non-unique indexes.

If a PRIMARY KEY consists of a single INTEGER column, it can be referenced using the _rowid alias in SELECT commands.

Foreign Key Constraints

> **Note:** Foreign Key constraints are not supported for partitioned tables.
> For more information, see Partitioning.

Aurora MySQL supports foreign key constraints for limiting values in a column, or a set of columns, of a child table based on their existence in a parent table.

Unlike SQL Server and contrary to the ANSI standard, Aurora MySQL allows foreign keys to reference non-unique columns in the parent table. The only requirement is that the columns are indexed as the leading columns of an index, but not necessarily a unique index.

Aurora MySQL supports cascading referential integrity actions using the ON UPDATE and ON DELETE clauses. The available referential actions are RESTRICT (the default), CASCADE, SET NULL, and NO ACTION. RESTRICT and NO ACTION are synonymous.

> **Note:** SET DEFAULT is supported by some other MySQL Server engines. Aurora MySQL uses the InnoDB engine exclusively, which does not support SET DEFAULT.

> **Note:** Some database engines support the ANSI standard for deferred checks. NO ACTION is a deferred check as opposed to RESTRICT, which is immediate. In MySQL, foreign key constraints are always validated immediately. Therefore, NO ACTION is the same as the RESTRICT action.

Aurora MySQL handles foreign keys differently than most other engines in the following ways:

- If there are multiple rows in the parent table that have the same values for the referenced foreign key, Aurora MySQL foreign key checks behave as if the other parent rows with the same key value do not exist. For example, if a RESTRICT action is defined and a child row has several parent rows, Aurora MySQL does not permit deleting them.
- If ON UPDATE CASCADE or ON UPDATE SET NULL causes a recursion and updates the same table that has been updated as part of the same cascade operation, Aurora MySQL treats it as if it was a RESTRICT action. This effectively disables self-referencing ON UPDATE CASCADE or ON UPDATE SET NULL operations to prevent potential infinite loops resulting from cascaded updates. A self-referencing ON DELETE SET NULL or ON DELETE CASCADE are allowed because there is no risk of an infinite loop.
- Cascading operations are limited to 15 levels deep.

Check Constraints

Standard ANSI CHECK clauses are parsed correctly and do not raise syntax errors. However, they are ignored and are not stored as part of the Aurora MySQL table definition.

Syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <Table Name>
(
<Column Definition>
[CONSTRAINT [<Symbol>]]
 PRIMARY KEY (<Column List>)
| [CONSTRAINT [<Symbol>]]
 UNIQUE [INDEX|KEY] [<Index Name>] [<Index Type>] (<Column List>)
| [CONSTRAINT [<Symbol>]]
 FOREIGN KEY [<Index Name>] (<Column List>)
    REFERENCES <Table Name> (<Column List>)
      [ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT]
      [ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT]

);
```

Migration Considerations

- CHECK constraints are not supported in Aurora MySQL.
  The engine parses the syntax for CHECK constraints, but they are ignored.
- Consider using triggers or stored routines to validate data values for complex expressions.
- When using check constraints for limiting to a value list such as CHECK (Col1 IN (1,2,3)), consider using the ENUM or SET data types.
- In Aurora MySQL, the constraint name (symbol) is optional, even for table constraints defined with the CONSTRAINT keyword.
  In SQL Server it is mandatory.
- Aurora MySQL requires that both the child table and the parent table in foreign key relationship are indexed.
  If the appropriate index does not exist, Aurora MySQL automatically creates one.

Examples

Create a composite primary key.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PRIMARY KEY (Col1, Col2)
);
```

Create a simple non-null unique constraint.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
```

```
CONSTRAINT PRIMARY KEY (Col1, Col2),
CONSTRAINT UNIQUE (Col2, Col3)
);
```

Create a named foreign key with multiple cascade actions.

```
CREATE TABLE MyParentTable
(
Col1 INT NOT NULL,
Col2 INT NOT NULL,
Col3 VARCHAR(20) NULL,
CONSTRAINT PRIMARY KEY (Col1, Col2)
);
```

```
CREATE TABLE MyChildTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 INT NOT NULL,
Col3 INT NOT NULL,
FOREIGN KEY (Col2, Col3)
 REFERENCES MyParentTable (Col1, Col2)
 ON DELETE NO ACTION
 ON UPDATE CASCADE
);
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feautre	SQL Server	Aurora MySQL	Comments
CHECK constraints	CHECK	Not supported	Aurora MySQL parses CHECK syntax, but ignores it.
UNIQUE constraints	UNIQUE	UNIQUE	
PRIMARY KEY constraints	PRIMARY KEY	PRIMARY KEY	
FOREIGN KEY constraints	FOREIGN KEY	FOREIGN KEY	
Cascaded referential actions	NO ACTION | CASCADE | SET NULL | SET DEFAULT	RESTRICT | CASCADE | SET NULL | NO ACTION	NO ACTION and RESTRICT are synonymous.
Indexing of referencing columns	Not required	Required	If not specified, an index is created silently to support the constraint.
Indexing of ref-	PRIMARY KEY or	Required	Aurora MySQL does not

Feautre	SQL Server	Aurora MySQL	Comments
erenced columns	UNIQUE		enforce uniqueness of referenced columns.
Cascade recursion	not allowed, discovered at CREATE time	Not allowed, discovered at run time.	

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/create-table.html*
- *https://dev.mysql.com/doc/refman/5.7/en/constraints.html*
- *https://dev.mysql.com/doc/refman/5.7/en/create-table-foreign-keys.html*

Migrate from SQL Server Creating Tables

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Creating Tables	• IDENTITY vs. AUTO_INCREMENT • Primary key always clustered • CREATE TEMPORARY TABLE syntax • Unsupported @table variables

Overview

ANSI Syntax Conformity

Tables in SQL Server are created using the CREATE TABLE statement and conform to the ANSI/ISO entry level standard. The basic features of CREATE TABLE are similar for most relational database management engines and are well defined in the ANSI/ISO standards.

In it's most basic form, the CREATE TABLE statement in SQL Server is used to define:

- Table names, the containing security schema, and database
- Column names
- Column data types
- Column and table constraints
- Column default values
- Primary, candidate (UNIQUE), and foreign keys

T-SQL Extensions

SQL Server extends the basic syntax and provides many additional options for the CREATE TABLE or ALTER TABLE statements. The most often used options are:

- Supporting index types for primary keys and unique constraints, clustered or non-clustered, and index properties such as FILLFACTOR
- Physical table data storage containers using the ON <File Group> clause
- Defining IDENTITY auto-enumerator columns
- Encryption
- Compression
- Indexes

For more information, see Data Types, Column Encryption, and Databases and Schemas.

Table Scope

SQL Server provides five scopes for tables:

- Standard tables are created on disk, globally visible, and persist through connection resets and server restarts.
- Temporary Tables are designated with the "# " prefix. Temporary tables are persisted in TempDB and are visible to the execution scope where they were created (and any sub-scope). Temporary tables are cleaned up by the server when the execution scope terminates and when the server restarts.
- Global Temporary Tables are designated by the "## " prefix. They are similar in scope to temporary tables, but are also visible to concurrent scopes.
- Table Variables are defined with the DECLARE statement, not with CREATE TABLE. They are visible only to the execution scope where they were created.
- Memory-Optimized tables are special types of tables used by the In-Memory Online Transaction Processing (OLTP) engine. They use a non standard CREATE TABLE syntax.

Creating a Table Based on an Existing Table or Query

SQL Server allows creating new tables based on SELECT queries as an alternate to the CREATE TABLE statement. A SELECT statement that returns a valid set with unique column names can be used to create a new table and populate data.

SELECT INTO is a combination of DML and DDL.
The simplified syntax for SELECT INTO is:

```
SELECT <Expression List>
INTO <Table Name>
[FROM <Table Source>]
[WHERE <Filter>]
[GROUP BY <Grouping Expressions>...];
```

When creating a new table using SELECT INTO, the only attributes created for the new table are column names, column order, and the data types of the expressions. Even a straight forward statement such as SELECT * INTO <New Table> FROM <Source Table> does not copy constraints, keys, indexes, identity property, default values, or any other related objects.

TIMESTAMP Syntax for ROWVERSION Deprecated Syntax

The TIMESTAMP syntax synonym for ROWVERSION has been deprecated as of SQL Server 2008R2 in accordance with https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms143729(v=sql.105).

Previously, you could use either the TIMESTAMP or the ROWVERSION keywords to denote a special data type that exposes an auto-enumerator. The auto-enumerator generates unique eight-byte binary numbers typically used to version-stamp table rows. Clients read the row, process it, and check the

ROWVERSION value against the current row in the table before modifying it. If they are different, the row has been modified since the client read it. The client can then apply different processing logic.

Note that when migrating to Aurora MySQL using the Amazon RDS Schema Conversion Tool (SCT), neither ROWVERSION nor TIMESTAMP are supported. SCT raises error 706 - "Unsupported data type ... of variable/column was replaced. Check the conversion result" .
You must add customer logic, potentially in the form of a trigger, to maintain this functionality.

See a full example in Aurora MySQL Creating Tables.

Syntax

Simplified syntax for CREATE TABLE:

```
CREATE TABLE [<Database Name>.<Schema Name>].<Table Name> (<Column Definitions>)
[ON{<Partition Scheme Name> (<Partition Column Name>)];
```

```
<Column Definition>:
<Column Name> <Data Type>
[CONSTRAINT <Column Constraint>
[DEFAULT <Default Value>]]
[IDENTITY [(<Seed Value>, <Increment Value>)]
[NULL | NOT NULL]
[ENCRYPTED WITH (<Encryption Specifications>)
[<Column Constraints>]
[<Column Index Specifications>]
```

```
<Column Constraint>:
[CONSTRAINT <Constraint Name>]
{{PRIMARY KEY | UNIQUE} [CLUSTERED | NONCLUSTERED]
[WITH FILLFACTOR = <Fill Factor>]
| [FOREIGN KEY]
REFERENCES <Referenced Table> (<Referenced Columns>)]
```

```
<Column Index Specifications>:
INDEX <Index Name> [CLUSTERED | NONCLUSTERED]
[WITH(<Index Options>]
```

Examples

Create a basic table.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Create a table with column constraints and an identity.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY IDENTITY (1,1),
```

```
Col2 VARCHAR(20) NOT NULL CHECK (Col2 <> ''),
Col3 VARCHAR(100) NULL
 REFERENCES MyOtherTable (Col3)
);
```

Create a table with an additional index.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
 INDEX IDX_Col2 NONCLUSTERED
);
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql

Migrate to Aurora MySQL Creating Tables

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Creating Tables	• IDENTITY vs. AUTO_INCREMENT • Primary key always clustered • CREATE TEMPORARY TABLE syntax • Unsupported @table variables

Overview

ANSI Syntax

Like SQL Server, Aurora MySQL provides ANSI/ISO syntax entry level conformity for CREATE TABLE and custom extensions to support Aurora MySQL specific functionality.

> **Note:** Unlike SQL Server that uses a single set of physical files for each database, Aurora MySQL tables are created as separate files for each table. Therefore, the SQL Server concept of File Groups does not apply to Aurora MySQL. For more information see Databases and Schemas.

In it's most basic form, and very similar to SQL Server, the CREATE TABLE statement in Aurora MySQL is used to define:

- Table name, containing security schema ,and/or database
- Column names
- Column data types
- Column and table constraints
- Column default values
- Primary, candidate (UNIQUE), and foreign keys

Aurora MySQL Extensions

Aurora MySQL Server extends the basic syntax and allows many additional options to be defined as part of the CREATE TABLE or ALTER TABLE statements. The most often used options are:

- Defining AUTO_INCREMENT properties for auto-enumerator columns
- Encryption

- Compression
- Indexes

Table Scope

Aurora MySQL provides two table scopes:

- Standard tables are created on disk, visible globally, and persist through connection resets and server restarts.
- Temporary tables are created using the CREATE TEMPORARY TABLE statement.
  A TEMPORARY table is visible only to the session that creates it and is dropped automatically when the session is closed.

Creating a table based on an existing table or query

Aurora MySQL provides two ways to create standard or temporary tables based on existing tables and queries.

CREATE TABLE <New Table> LIKE <Source Table> creates an empty table based on the definition of another table including any column attributes and indexes defined in the original table.

CREATE TABLE ... AS <Query Expression> is very similar to SQL Server's SELECT INTO and allows creating a new table and populating data in a single step. Unlike SQL Server, Aurora MySQL also allows combining standard column definitions and additional columns derived from the query. This statement does not copy supporting objects or attributes from the source table, similar to SQL Server. For example:

```
CREATE TABLE SourceTable
(
Col1 INT
);
```

```
INSERT INTO SourceTable
VALUES (1)
```

```
CREATE TABLE NewTable
(
Col1 INT
)
AS
SELECT Col1 AS Col2
FROM SourceTable;
```

```
INSERT INTO NewTable (Col1, Col2)
VALUES (2,3);
```

```
SELECT * FROM NewTable
```

```
Col1    Col2
----    ----
NULL    1
2       3
```

Converting TIMESTAMP and ROWVERSION columns

> **Note:** Aurora MySQL does have a TIMESTAMP data type, which is a temporal type not to be confused with SQL Server's TIMESTAMP. For more information, see the Data Types topic.

SQL server provides an automatic mechanism for stamping row versions for application concurrency control.

For example:

```
CREATE TABLE WorkItems
(
 WorkItemID INT IDENTITY(1,1) PRIMARY KEY,
 WorkItemDescription XML NOT NULL,
 Status VARCHAR(10) NOT NULL DEFAULT ('Pending'),
 -- other columns...
 VersionNumber ROWVERSION
);
```

The VersionNumber column automatically updates when a row is modified. The actual value is meaningless, just the fact that it changed is what indicates a row modification. The client can now read a work item row, process it, and ensure no other clients updated the row before updating the status.

```
SELECT @WorkItemDescription = WorkItemDescription,
       @Status = Status,
       @VersionNumber = VersionNumber
FROM WorkItems
WHERE WorkItemID = @WorkItemID;

EXECUTE ProcessWorkItem @WorkItemID, @WorkItemDescription, @Stauts OUTPUT;

IF (
    SELECT VersionNumber
    FROM WorkItems
    WHERE WorkItemID = @WorkItemID
   ) = @VersionNumber;
 EXECUTE UpdateWorkItems @WorkItemID, 'Completed'; -- Success
ELSE
 EXECUTE ConcurrencyExceptionWorkItem; -- Row updated while processing
```

In Aurora MySQL, you can add a trigger to maintain the updated stamp per row.

```
 CREATE TABLE WorkItems
(
 WorkItemID INT AUTO_INCREMENT PRIMARY KEY,
 WorkItemDescription JSON NOT NULL,
 Status VARCHAR(10) NOT NULL DEFAULT 'Pending',
 -- other columns...
 VersionNumber INTEGER NULL
);

CREATE TRIGGER MaintainWorkItemVersionNumber
AFTER UPDATE
ON WorkItems FOR EACH ROW
SET NEW.VersionNumber = OLD.VersionNumber + 1;
```

For more information on triggers in Aurora MySQL, see the Triggers topic.

Syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <Table Name>
(<Create Definition> ,...)[<Table Options>];
```

```
<Create Definition>:
<Column Name> <Column Definition> | [CONSTRAINT [symbol]]
[PRIMARY KEY | UNIQUE | FOREIGN KEY <Foreign Key Definition> | CHECK (<Check Predic-
ate>)]
(INDEX <Index Column Name>,...)
```

```
<Column Definition>:
<Data Type> [NOT NULL | NULL]
[DEFAULT <Default Value>]
[AUTO_INCREMENT]
[UNIQUE [KEY]] [[PRIMARY] KEY]
[COMMENT <comment>]
```

Migration Considerations

Migrating CREATE TABLE statements should be mostly compatible with the SQL Server syntax when using only ANSI standard syntax.

IDENTITY columns should be rewritten to use the Aurora MySQL syntax of AUTO_INCREMENT. Note that similar to SQL Server, there can be only one such column in a table, but in Aurora MySQL it also must be indexed.

Temporary table syntax should be modified to use the CREATE TEMPORARY TABLE statement instead of the CREATE #Table syntax of SQL Server. Global temporary tables and table variables are not supported by Aurora MySQL. For sharing data across connections, use standard tables.

SELECT INTO queries should be rewritten to use CREATE TABLE ... AS syntax. When copying tables, remember that the CREATE TABLE ... LIKE syntax also retains all supporting objects such as constraints and indexes.

Aurora MySQL does not require specifying constraint names when using the CONSTRAINT keyword. Unique constraint names are created automatically. If specifying a name, the name must be unique for the database.

Unlike SQL Server IDENTITY columns, which require EXPLICIT SET IDENTITY_INSERT ON to bypass the automatic generation, Aurora MySQL allows inserting explicit values into the column. To generate an automatic value, insert a NULL or a 0 value. To reseed the automatic value, use ALTER TABLE as opposed to SQL Server's DBCC CHECKIDENT.

Aurora MySQL also allows adding a comment to a column for documentation purposes, similar to SQL Server extended properties feature.

Note: Contrary to the SQL standard, foreign keys in Aurora MySQL are allowed to point to non-unique parent column values. In this case, the foreign key prohibits deletion of any of the parent rows. For more information, see Constraints and https://dev.mysql.-com/doc/refman/5.7/en/ansi-diff-foreign-keys.html.

Examples

Create a basic table.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Create a table with column constraints and an auto increment column.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
 CHECK (Col2 <> ''),
Col3 VARCHAR(100) NULL
 REFERENCES MyOtherTable (Col3)
);
```

Create a table with an additional index.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL,
INDEX IDX_Col2 (Col2)
);
```

Summary

Feature	SQL Server	Aurora MySQL	Comments
ANSI compliance	Entry level	Entry level	Basic syntax is compatible.
Auto generated enumerator	IDENTITY	AUTO_INCREMENT	Only one allowed for each table. In Aurora MySQL insert NULL or 0 to generate new value.
Reseed auto generated value	DBCC CHECKIDENT	ALTER TABLE	For more information, see https://dev.mysql.-com/doc/refman/5.7/en/alter-table.html

Feature	SQL Server	Aurora MySQL	Comments
Index types	CLUSTERED / NONCLUSTERED	Implicit - Primary keys use clustered indexes	See the Clustered and Non Clustered Indexes .
Physical storage location	ON <File Group>	Not supported	Physical storage is managed by AWS.
Temporary tables	#TempTable	CREATE TEMPORARY TABLE	
Global Temporary Tables	##GlobalTempTable	Not supported	Use standard tables to share data between connections.
Table Variables	DECLARE @Table	Not supported	
Create table as query	SELECT... INTO	CREATE TABLE... AS	
Copy table structure	Not supported	CREATE TABLE... LIKE	
Memory optimized tables	Supported	Not supported	For workloads that require memory resident tables, consider using AWS Elsticache for Redis. See https://aws.amazon.com/elasticache/redis/

For more information, see https://dev.mysql.com/doc/refman/5.7/en/create-table.html

Migrate from SQL Server Common Table Expressions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Common Table Expressions	• Rewrite non- recursive CTE to use views and derived tables • Redesign recursive CTE code

Overview

Common Table Expressions (CTE), which have been a part of the ANSI standard since SQL:1999, simplify queries and make them more readable by defining a temporary view, or derived table, that a subsequent query can reference. SQL Server CTEs can be the target of DML modification statements and have similar restrictions as updateable views.

SQL Server CTEs provide recursive functionality in accordance with the the ANSI 99 standard. Recursive CTEs can reference themselves and re-execute queries until the data set is exhausted, or the maximum number of iterations is exceeded.

CTE Syntax (simplified)

```
WITH <CTE NAME>
AS
(
SELECT ....
)
SELECT ...
FROM CTE
```

Recursive CTE syntax

```
WITH <CTE NAME>
AS (
    <Anchor SELECT query>
    UNION ALL
    <Recursive SELECT query with reference to <CTE NAME>>
    )
SELECT ... FROM <CTE NAME>...
```

Examples

Create and populate an `OrderItems`table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
```

```
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

Define a CTE to calculate the total quantity in every order and then join to the `OrderItems`table to obtain the relative quantity for each item.

```
WITH AggregatedOrders
AS
( SELECT OrderID, SUM(Quantity) AS TotalQty
  FROM OrderItems
  GROUP BY OrderID
)
SELECT  O.OrderID, O.Item,
        O.Quantity,
        (O.Quantity / AO.TotalQty) * 100 AS PercentOfOrder
FROM    OrderItems AS O
        INNER JOIN
        AggregatedOrders AS AO
        ON O.OrderID = AO.OrderID;
```

The example above produces the following results:

```
OrderID     Item      Quantity  PercentOfOrder
-------     ----      --------  -------------
1           M8 Bolt   100       100.0000000000
2           M8 Nut    100       100.0000000000
3           M8 Washer 100       33.3333333300
3           M6 Washer 200       66.6666666600
```

Using a Recursive CTE, create and populate the `Employees`table with the `DirectManager`for each employee.

```
CREATE TABLE Employees
(
Employee VARCHAR(5) NOT NULL PRIMARY KEY,
DirectManager VARCHAR(5) NULL
);
```

```
INSERT INTO Employees(Employee, DirectManager)
VALUES
('John', 'Dave'),
('Jose', 'Dave'),
('Fred', 'John'),
('Dave', NULL);
```

Use a resursive CTE to display the employee-management hierarchy.

```
WITH EmpHierarchyCTE AS
(
-- Anchor query retrieves the top manager
SELECT  0 AS LVL,
        Employee,
        DirectManager
FROM Employees AS E
WHERE DirectManager IS NULL
UNION ALL
-- Recursive query gets all Employees managed by the previous level
SELECT  LVL + 1 AS LVL,
        E.Employee,
        E.DirectManager
FROM    EmpHierarchyCTE AS EH
INNER JOIN
Employees AS E
ON E.DirectManager = EH.Employee
)
SELECT *
FROM EmpHierarchyCTE;
```

The example above displays the following results:

```
LVL     Employee   DirectManager
---     --------   -------------
0       Dave       NULL
1       John       Dave
1       Jose       Dave
2       Fred       John
```

For more information, see https://technet.microsoft.com/en-us/library/ms186243.aspx

Migrate to Aurora MySQL Common Table Expressions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Common Table Expressions	• Rewrite non- recursive CTE to use views and derived tables • Redesign recursive CTE code

Overview

Aurora MySQL 5.7 does not support Common Table Expressions (CTE).
However, The next version of MySQL, version 8, will support CTEs, including recursive CTEs.

Migration Considerations

As a workaround, use VIEWs or derived tables in place of non recursive CTEs.

Since non recursive CTEs are more convenient for readability and code simplification, You can convert the code to use derived tables, which are a subquery in the parent query's FROM clause. For example, replace the following CTE:

```
WITH TopCustomerOrders
(
  SELECT Customer,
         COUNT(*) AS NumOrders
  FROM Orders
  GROUP BY Customer
)
SELECT TOP 10 *
FROM TopCustomerOrders
ORDER BY NumOrders DESC;
```

With the following subquery:

```
SELECT *
FROM   ( SELECT Customer,
                COUNT(*) AS NumOrders
         FROM Orders
         GROUP BY Customer
       ) AS TopCustomerOrders
ORDER BY NumOrders DESC
LIMIT 10 OFFSET 0;
```

When using derived tables, the derived table definition must be repeated if multiple instances are required for the query.

Converting the code for recursive CTEs is not straight forward, but you can achieve similar functionality using loops.

Examples

Replacing Non-Recursive CTEs

Use a derived table to replace non-recursive CTE functionality as follows:

Create and populate an `OrderItems` table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

Define a derived table for `TotalQty` of every order and then join to the `OrderItems`to obtain the relative quantity for each item.

```
SELECT  O.OrderID,
        O.Item,
        O.Quantity,
        (O.Quantity / AO.TotalQty) * 100 AS PercentOfOrder
FROM    OrderItems AS O
        INNER JOIN
        ( SELECT OrderID,
                 SUM(Quantity) AS TotalQty
          FROM OrderItems
          GROUP BY OrderID
        ) AS AO
        ON O.OrderID = AO.OrderID;
```

The example code above displays the following results:

```
OrderID     Item      Quantity  PercentOfOrder
-------     ----      --------  -------------
1           M8 Bolt   100       100.0000000000
2           M8 Nut    100       100.0000000000
3           M8 Washer 100       33.3333333300
3           M6 Washer 200       66.6666666600
```

Replacing Recursive CTEs

Use recursive SQL code in stored procedures and SQL loops to replace a recursive CTEs.

> **Note:** Stored procedure and function recursion in Aurora MySQL is disabled by default. You can set the server system variable *max_sp_recursion_depth* to a value of *1* or higher to

enable recursion. However, this approach is not recommended because it may increase contention for the thread stack space.

Create and populate an `Employees` table.

```
CREATE TABLE Employees
(
Employee VARCHAR(5) NOT NULL PRIMARY KEY,
DirectManager VARCHAR(5) NULL
);
```

```
INSERT INTO Employees (Employee, DirectManager)
VALUES
('John', 'Dave'),
('Jose', 'Dave'),
('Fred', 'John'),
('Dave', NULL);
```

Create an `EmpHierarcy`table.

```
CREATE TABLE EmpHierarchy
(
LVL INT,
Employee VARCHAR(5),
Manager VARCHAR(5)
);
```

Create a procedure that uses a loop to traverse the employee hierarchy. For more information on Stored Procedures in Aurora MySQL, see Stored Procedures. For more information on loops in Aurora MySQL, see Flow Control.

```
CREATE PROCEDURE P()
BEGIN
DECLARE var_lvl INT;
DECLARE var_Employee VARCHAR(5);
SET var_lvl = 0;
SET var_Employee = (
                    SELECT Employee
                    FROM Employees |
                    WHERE DirectManager IS NULL
                   );
INSERT INTO EmpHierarchy
VALUES (var_lvl, var_Employee, NULL);
WHILE var_lvl <> -1
DO
INSERT INTO EmpHierarchy (LVL, Employee, Manager)
SELECT var_lvl + 1,
       Employee,
       DirectManager
FROM Employees
WHERE DirectManager IN (
                        SELECT Employee
                        FROM EmpHierarchy
                        WHERE LVL = var_lvl
                       );
```

```
IF NOT EXISTS (
               SELECT *
               FROM EmpHierarchy
               WHERE LVL = var_lvl + 1
              )
THEN SET var_lvl = -1;
ELSE SET var_lvl = var_lvl + 1;
END IF;
END WHILE;
END;
```

Execute the procedure.

```
CALL P()
```

Select all records from the `EmpHierarchy` table.

```
SELECT * FROM EmpHierarchy;
```

```
Level   Employee   Manager
-----   --------   -------
0       Dave
1       John       Dave
1       Jose       Dave
2       Fred       John
```

Summary

SQL Server	Aurora MySQL	Comments
Non recursive CTE	Derived table	For multiple instances of the same table, the derived table definition subquery must be repeated.
Recursive CTE	Loop inside a stored procedure or stored function.	

For more information, see https://dev.mysql.com/doc/refman/8.0/en/with.html

Migrate from SQL Server Data Types

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Data Types	• Minor syntax and handling differences • No special UNICODE data types

Overview

In SQL Server, each table column, variable, expression, and parameter has an associated data type.

SQL Server provides a rich set of built-in data types as summarized in the following table.

Category	Data Types
Numeric	BIT, TINYINT, SMALLINT, INT, BIGINT, NUMERIC, DECIMAL, MONEY, SMALLMONEY, FLOAT, REAL
String and Character	CHAR, VARCHAR, NCHAR, NVARCHAR
Temporal	DATE, TIME, SMALLDATETIME, DATETIME, DATETIME2, DATETIMEOFFSET
Binary	BINARY, VARBINARY
Large Object (LOB)	TEXT, NTEXT, IMAGE, VARCHAR(MAX), NVARCHAR(MAX), VARBINARY (MAX)
Cursor	CURSOR
GUID	UNIQUEIDENTIFIER
Hierarchical identifier	HIERARCHYID
Spatial	GEOMETRY, GEOGRAPHY
Sets (Table type)	TABLE
XML	XML
Other Specialty Types	ROW VERSION, SQL_VARIANT

Note: You can create custom user defined data types using T-SQL, and the .NET Framework. Custom data types are based on the built-in system data types and are used to simplify development.
For more information, see User Defined Types.

TEXT, NTEXT, and IMAGE deprecated data types

The TEXT, NTEXT, and IMAGE data types have been deprecated as of SQL Server 2008R2 in accordance with https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms143729(v=sql.105).

These data types are legacy types for storing BLOB and CLOB data. The TEXT data type was used to store ASCII text CLOBS, the NTEXT data type to store UNICODE CLOBS, and IMAGE was used as a generic data type for storing all BLOB data. In SQL Server 2005, Microsoft introduced the new and improved VARCHAR(MAX), NVARCHAR(MAX), and VARBINARY(MAX) data types as the new BLOB and CLOB standard. These new types support a wider range of functions and operations. They also provide enhanced performance over the legacy types.

If your code uses TEXT, NTEXT or IMAGE data types, SCT automatically converts them to the appropriate Aurora MySQL BLOB data type. TEXT and NTEXT are converted to LONGTEXT and image to LONGBLOB. Make sure you use the proper collations. For more details, see the Collations.

Examples

Define table columns.

```
CREATE TABLE MyTable
(
Col1 AS INTEGER NOT NULL PRIMARY KEY,
Col2 AS NVARCHAR(100) NOT NULL
);
```

Define variable types.

```
DECLARE @MyXMLType AS XML,
        @MyTemporalType AS DATETIME2
```

```
DECLARE @MyTableType
AS TABLE
(
Col1 AS BINARY(16) NOT NULL PRIMARY KEY,
Col2 AS XML NULL
);
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

Migrate to Aurora MySQL Data Types

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Data Types	• Minor syntax and handling differences • No special UNICODE data types

Overview

Aurora MySQL supports the following data types:

Category	Data Type
Numeric	BIT, INTEGER, SMALLINT, TINYINT, MEDIUMINT, BIGINT, DECIMAL, NUMERIC, FLOAT, DOUBLE
String and Character	CHAR, VARCHAR, SET
Temporal	DATE, DATETIME, TIMESTAMP, TIME, YEAR
Binary	BINARY, VARBINARY
Large Object (LOB)	BLOB, TEXT
Cursor	CURSOR
GUID	UNIQUEIDENTIFIER
Hierarchical Identifiers	HIERARCHYID
Spatial	GEOMETRY, POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION
JSON	JSON

Be aware that Aurora MySQL uses different rules than SQL Server for handling out-of-range and overflow situations. SQL Server always raises an error for out-of-range values. Aurora MySQL exhibits different behavior depending on run time settings.
For example, a value may be "clipped" to the first or last value in the range of permitted values for the data type if STRICT SQL mode is not set.
For more information, see https://dev.mysql.com/doc/refman/5.7/en/out-of-range-and-overflow.html

Converting from TEXT, NTEXT, and IMAGE SQL Server deprecated data types

As mentioned in SQL Server Data Types, the legacy SQL Server types for storing LOB data are deprecated as of SQL Server 2008R2.

When converting from these types to Aurora MySQL using the AWS Schema Conversion Tool (SCT), they are converted as follows:

SQL Server LOB Type	Converted to Aurora MySQL data type	Comments
TEXT	LONGTEXT	Make sure to choose the right collation, for more information see the Collations topic
NTEXT	LONGTEXT	Make sure to choose the right collation, for more information see the Collations topic
IMAGE	LONGBLOB	

The size cap for all of these types is compatible and is capped at 2 GB of data, which may allow less characters depending on the chosen collation.

> **Note:** Aurora MySQL supports UCS-2 collation, which is compatible with SQL Server's UNICODE types.

While it is safe to use the default conversion types, remember that, unlike SQL Server, Aurora MySQL also provides smaller BLOB and CLOB types, which may be more efficient for your data.

Even the basic VARCHAR and VARBINARY data types can store strings up to 32 KB, which is much longer that SQL Server's 8 KB limit. If the strings or binary data that you need to store do not exceed 32 KB, it may be more efficient to store these as non-LOB types in Aurora MySQL.

For more information, see the Data Types topic.

Summary

The following table summarizes the key differences and migration considerations for migrating from SQL Server data types to Aurora MySQL data types.

SQL Server Data Type	Convert to MySQL Data Type	Comments
BIT	BIT	Aurora MySQL also supports BIT(m), which can be used to store multiple bit values. SQL Server's literal bit notation uses the numerical digits 0 and 1. Aurora MySQL uses b'<value> or 0b<value> notations. For more information see https://dev.mysql.-

SQL Server Data Type	Convert to MySQL Data Type	Comments
		com/doc/refman/5.7/en/bit-type.html and https://dev.mysql.-com/doc/refman/5.7/en/bit-value-literals.html
TINYINT	TINYINT	SQL Server only supports unsigned TINYINT, which can store values between 0 and 255, Aurora MySQL supports both signed TINYINT and TINYINT UNSIGNED. The latter can be used to store values between -128 and 127. The default for integer types in Aurora MySQL is to use signed integers. For compatibility, explicitly specify TINYINT UNSIGNED. For more information see https://dev.mysql.-com/doc/refman/5.7/en/integer-types.html
SMALLINT	SMALLINT	Compatible type. SQL Server supports only signed SMALLINT. Auora MySQL also supports SMALLINT UNSIGNED, which can store values between 0 and 65535. The default for integer types in Aurora MySQL is to use signed integers. Consider using unsigned integers for storage optimization. For more information see https://dev.mysql.-com/doc/refman/5.7/en/integer-types.html
INTEGER	INTEGER	Compatible type. SQL Server supports only signed INTEGER, which can store values between -2147483648 and 2147483647. Aurora MySQL also supports INTEGER UNSIGNED, which can store values between 0 and 4294967295. The default for integer types in Aurora MySQL is to use signed integers. Consider using unsigned integers for storage optimization. Aurora MySQL also supports a MEDIUMINT type, which uses only three bytes of storage vs. four bytes for INTEGER. For large tables, consider using MEDIUMINT instead of INT if the value range is within -8388608 to -8388607 for a SIGNED type, or 0 to 16777215 for UNSIGNED type. For more information see https://dev.mysql.-com/doc/refman/5.7/en/integer-types.html
BIGINT	BIGINT	Compatible type. SQL Server supports only signed BIGINT. Aurora MySQL also supports BIGINT UNSIGNED, which can store values between 0 and 2^64-1. The default for integer types in Aurora MySQL is to use signed integers. Consider using unsigned integers for storage optimization.

SQL Server Data Type	Convert to MySQL Data Type	Comments
		For more information see https://dev.mysql.-com/doc/refman/5.7/en/integer-types.html
NUMERIC / DECIMAL	NUMERIC / DECIMAL	Compatible types. DECIMAL and NUMERIC are synonymous.
MONEY / SMALLMONEY	N/A	Aurora MySQL does not support dedicated monetary types. Use NUMERIC / DECIMAL instead. If your application uses literals with monetary signs (for example, $50.23), rewrite to remove the monetary sign.
FLOAT / REAL	FLOAT / REAL / DOUBLE	Compatible types. In SQL Server, both REAL and FLOAT(n) (where n<=24, use 4 bytes of storage) are equivalent to Aurora MySQL's FLOAT and REAL. SQL Server's FLOAT(n), where n>24, uses 8 bytes. The Aurora MySQL DOUBLE PRECISION type always uses 8 bytes. Aurora MySQL also supports the non standard FLOAT(M,D), REAL(M,D) or DOUBLE PRECISION(M,D) where (M,D) indicates values can be stored with up to M digits in total with D digits after the decimal point. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/floating-point-types.html
CHAR	CHAR / VARCHAR	Compatible types up to 255 characters only. SQL Server supports CHAR data types up to 8,000 characters. The Aurora MySQL CHAR data type is limited to a maximum of 255 characters. For strings requiring more than 255 characters, use VARCHAR. When converting from CHAR to VARCHAR, exercise caution because VARCHAR behaves differently than CHAR; trailing spaces are trimmed. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/char.html
VARCHAR	VARCHAR	Compatible types. SQL Server supports VARCHAR columns up to 8,000 characters. Aurora MySQL can store up to 65,535 characters with regard to the maximal row size limit. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/char.html

SQL Server Data Type	Convert to MySQL Data Type	Comments
NCHAR	CHAR	Aurora MySQL does not require the use of specific data types for storing UNICODE data. Use the CHAR data type and define a UNICODE collation using the CHARACTER SET or COLLATE keywords. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/charset-unicode-sets.html
NVARCHAR	VARCHAR	Aurora MySQL does not require the use of specific data types for storing UNICODE data. Use the VARCHAR data type and define a UNICODE collation using the CHARACTER SET or COLLATE keywords. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/charset-unicode-sets.html
DATE	DATE	Compatible types. The range for SQL Server's DATE data type is '0001-01-01' through '9999-12-31'. The range for Aurora MySQL is '1000-01-01' through '9999-12-31'. Aurora MySQL does not support dates before 1000 AD. For more information about handling dates, see Date and Time Functions. For more information about the DATE data type, see https://dev.mysql.com/doc/refman/5.7/en/datetime.html
TIME	TIME	Compatible types. SQL Server supports explicit fractional seconds using the format TIME(n) where n is between 0 to 7. Aurora MySQL does not allow explicit fractional setting. Aurora MySQL supports up to 6 digits for microsecond resolution of fractional seconds. SQL Server provides one more digit to support a resolution of up to 100 nanoseconds. If your application uses the TIME(n) format, rewrite to remove the (n) setting. Aurora MySQL also supports TIME values that range from '-838:59:59' to '838:59:59'. You can use the hours part to represent the time of day (where hours must be less that 24) or to represent a time interval (which can be greater than 24 hours and have negative values). For more information see https://dev.mysql.-com/doc/refman/5.7/en/time.html

SQL Server Data Type	Convert to MySQL Data Type	Comments
SMALLDATETIME	DATETIME / TIMESTAMP	Aurora MySQL does not support SMALLDATETIME. Use DATETIME instead. Use SMALLDATETIME for storage space optimization where lower ranges and resolutions are required. For more information about handling temporal data, see Date and Time.
DATETIME	DATETIME	SQL Server's DATETIME data type supports a value range between '1753-01-01' and '9999-12-31' with a resolution of up to 3.33ms. Aurora MySQL DATETIME supports a wider value range between '1000-01-01 00:00:00' and '9999-12-31 23:59:59' with a higher resolution of microseconds and optional six fractional second digits. For more information about handling temporal data, see Date and Time Functions. For more information about DATETIME, see https://dev.mysql.com/doc/refman/5.7/en/datetime.html
DATETIME2	DATETIME	SQL Server's DATETIME2 data type supports a value range between '0001-01-01' and '9999-12-31' with a resolution of up to 100 nanoseconds using seven fractional second digits. Aurora MySQL DATETIME supports a narrower value range between '1000-01-01 00:00:00' and '9999-12-31 23:59:59' with a lower resolution of microseconds and optional six fractional second digits. For more information about handling temporal data, see Date and Time Functions. For more information about DATETIME, see https://dev.mysql.com/doc/refman/5.7/en/datetime.html
DATETIMEOFFSET	TIMESTAMP	Aurora MySQL does not support full time zone awareness and management functions. Use the time_zone system variable in conjunction with TIMESTAMP columns to achieve partial time zone awareness. For more information about system variables, see Server Options. Aurora MySQL's TIMESTAMP is not the same as SQL Server's TIMESTAMP data type, the latter being a synonym for ROW_VERSION. Aurora MySQL TIMESTAMP is equivalent to the

SQL Server Data Type	Convert to MySQL Data Type	Comments
		DATETIME type with a smaller range. Aurora MySQL DATETIME allows for values between '1000-01-01 00:00:00' and '9999-12-31 23:59:59'. TIMESTAMP is limited to values between '1970-01-01 00:00:01' and '2038-01-19 03:14:07'. Aurora MySQL converts TIMESTAMP values from the current time zone to UTC for storage and back from UTC to the current time zone for retrieval. For more information, see https://dev.mysql.com/doc/refman/5.7/en/time-zone-support.html
BINARY	BINARY / VARBINARY	In Aurora MySQL, the BINARY data type is considered to be a string data type and is similar to CHAR. BINARY contains byte strings rather than character strings and uses the binary character set and collation. Comparison and sorting are based on the numeric values of the bytes in the values. SQL Server supports up to 8,000 bytes for a BINARY data types. Aurora MySQL BINARY is limited to 255 characters, similar to CHAR. If larger values are needed, use VARBINARY. Literal assignment for Aurora MySQL BINARY types use string literals, unlike SQL Server's explicit binary 0x notation. For more information, see https://dev.mysql.com/doc/refman/5.7/en/binary-varbinary.html and https://dev.mysql.com/doc/refman/5.7/en/charset-binary-collations.html
VARBINARY	VARBINARY	In Aurora MySQL, the VARBINARY data type is considered a string data type, similar to VARCHAR. VARBINARY contains byte strings rather than character strings and has a binary character set. Collation, comparison, and sorting are based on the numeric values of the bytes in the values. Aurora MySQL VARBINARY supports up to 65,535 characters, significantly larger than SQL Server's 8,000 byte limit. Literal assignment for Aurora MySQL BINARY types use string literals, unlike SQL Server's explicit binary 0x notation. For more information, see https://dev.mysql.com/doc/refman/5.7/en/binary-varbinary.html and

<table>
<tr><th>SQL Server Data Type</th><th>Convert to MySQL Data Type</th><th>Comments</th></tr>
<tr><td></td><td></td><td>https://dev.mysql.com/doc/refman/5.7/en/charset-binary-collations.html</td></tr>
<tr><td>TEXT / VARCHAR (MAX)</td><td>VARCHAR / TEXT / MEDIUMTEXT / LONGTEXT</td><td>In SQL Server, a TEXT data type is a variable-length ASCII string data type with a maximum string length of 2^31-1 (2 GB).

Use the following table to determine the optimal Aurora MySQL data type:

<table><tr><th>Maximal string length</th><th>Use</th></tr><tr><td>2^16-1 bytes</td><td>VARCHAR or TEXT</td></tr><tr><td>2^24-1 bytes</td><td>MEDIUMTEXT</td></tr><tr><td>2^32-1 bytes</td><td>LONGTEXT</td></tr></table>
For more information, see https://dev.mysql.com/doc/refman/5.7/en/blob.html and https://dev.mysql.com/doc/refman/5.7/en/storage-requirements.html#data-types-storage-reqs-strings</td></tr>
<tr><td>NTEXT / NVARCHAR (MAX)</td><td>VARCHAR / TEXT / MEDIUMTEXT / LONGTEXT</td><td>Aurora MySQL does not require the use of specific data types for storing UNICODE data. Use the TEXT compatible data types listed above and define a UNICODE collation using the CHARACTER SET or COLLATE keywords.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/charset-unicode-sets.html</td></tr>
<tr><td>IMAGE / VARBINARY (MAX)</td><td>VARBINARY / BLOB / MEDIUMBLOB / LONGBLOB</td><td>In SQL Server, an IMAGE data type is a variable-length binary data type with a range of 0 through 2^31-1 (2 GB).

Similar to the BINARY and VARBINARY data types, The BLOB data types are considered string data types. BLOB data types contain byte strings rather than character strings and use a binary character set. Collation, comparison, and sorting are based on the numeric values of the bytes in the values.

Use the following table to determine the optimal Aurora MySQL data type:</td></tr>
</table>

<table>
<tr><th>SQL Server Data Type</th><th>Convert to MySQL Data Type</th><th>Comments</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><th>Maximal binary bytes</th><th>Use</th></tr>
<tr><td>2^16-1 bytes</td><td>VARBINARY or BLOB</td></tr>
<tr><td>2^24-1 bytes</td><td>MEDIUMBLOB</td></tr>
<tr><td>2^32-1 bytes</td><td>LONGBLOB</td></tr>
</table>
For more information, see https://dev.mysql.-com/doc/refman/5.7/en/blob.html, https://dev.mysql.-com/doc/refman/5.7/en/storage-requirements.html#data-types-storage-reqs-strings, and https://dev.mysql.-com/doc/refman/5.7/en/charset-binary-collations.html
</td></tr>
<tr><td>CURSOR</td><td>CURSOR</td><td> Types are compatible, although in Aurora MySQL a Cursor is not really considered to be a type. For more information, see Cursors. </td></tr>
<tr><td>UNIQUEIDENTIFIER</td><td>N/A</td><td> Aurora MySQL does not support a native unique identifier type. Use BINARY(16), which is the same base type used for SQL Server's UNIQUEIDENTIFIER type. It generates values using the UUID() function, which is the equivalent of SQL Server's NEW_ID function. UUID returns a Universal Unique Identifier generated according to RFC 4122, A Universally Unique IDentifier (UUID) URN Namespace (http://www.ietf.org/rfc/rfc4122.txt). For more information, see https://dev.mysql.-com/doc/refman/5.7/en/miscellaneous-functions.html#function_uuid </td></tr>
<tr><td>HIERARCHYID</td><td>N/A</td><td> Aurora MySQL does not support native hierarchy representation. Rewrite functionality with custom application code using one the common SQL hierarchical data representation approaches: Adjacency list
Nested set
Closure table
Materialized path For more information about potential implementations, see https://en.wikipedia.org/wiki/Adjacency_list, and https://en.wikipedia.org/wiki/Nested_set_model </td></tr>
</table>

SQL Server Data Type	Convert to MySQL Data Type	Comments
GEOMETRY	GEOMETRY	SQL Server's GEOMETRY type represents data in a Euclidean (flat) coordinate system. SQL Server supports a set of methods for this type, which include methods defined by the Open Geospatial Consortium (OGC) standard, and a set of additional extensions. Aurora MySQL supports GEOMETRY spatial data, although the syntax and functionality may differ significantly from SQL Server. A rewrite of the code is required. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/spatial-types.html
TABLE	N/A	Aurora MySQL does not support a TABLE data type. For more information, and a discussion of alternative solutions, see User Defined Types.
XML	N/A	Aurora MySQL does not support a native XML data type . However, it does provide full support for JSON data types, which SQL Server does not. Since XML and JSON are text documents, consider migrating the XML formatted documents to JSON or use string BLOBs and custom code to parse and query documents. For more information, see https://dev.mysql.-com/doc/refman/5.7/en/json.html
ROW_VERSION	N/A	Aurora MySQL does not support a row version. Use triggers to update a dedicated column from a master sequence value table.
SQL_VARIANT	N/A	Aurora MySQL does not support a hybrid, all-purpose data type similar to SQL Server's SQL_VARIANT. Rewrite applications to use explicit types.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/data-types.html

Migrate from SQL Server GROUP BY

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - GROUP BY	• Basic syntax compatible • Advanced options like ALL, CUBE, GROUPING SETS will require rewrites to use multiple queries with UNION

Overview

GROUP BY is an ANSI SQL query clause used to group individual rows that have passed the WHERE filter clause into groups to be passed on to the HAVING filter and then to the SELECT list. This grouping supports the use of aggregate functions such as SUM, MAX, AVG and others.

Syntax

ANSI compliant GROUP BY Syntax:

```
GROUP BY
[ROLLUP | CUBE]
<Column Expression> ...n
[GROUPING SETS (<Grouping Set>)...n
```

Backward compatibility syntax:

```
GROUP BY
    [ ALL ] <Column Expression> ...n
    [ WITH  CUBE | ROLLUP ]
```

The basic ANSI syntax for GROUP BY supports multiple grouping expressions, the CUBE and ROLLUP keywords, and the GROUPING SETS clause; all used to add super-aggregate rows to the output.

Up to SQL Server 2008 R2, the database engine supported a legacy, proprietary syntax (not ANSI Compliant) using the WITH CUBE and WITH ROLLUP clauses. These clauses added super-aggregates to the output.

Also, up to SQL Server 2008 R2, SQL Server supported the GROUP BY ALL syntax, which was used to create an empty group for rows that failed the WHERE clause.

SQL Server supports the following aggregate functions:

```
AVG, CHECKSUM_AGG, COUNT, COUNT_BIG, GROUPING, GROUPING_ID, STDEV, STDEVP, STRING_AGG,
SUM, MIN, MAX, VAR, VARP
```

Examples

Legacy CUBE and ROLLUP Syntax

```
CREATE TABLE Orders
(
       OrderID INT IDENTITY(1,1) NOT NULL
       PRIMARY KEY,
       Customer VARCHAR(20) NOT NULL,
       OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders(Customer, OrderDate)
VALUES ('John', '20180501'),  ('John', '20180502'), ('John', '20180503'),
       ('Jim', '20180501'), ('Jim', '20180503'), ('Jim', '20180504')
```

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer, OrderDate
WITH ROLLUP
```

```
Customer             OrderDate  NumOrders
-------------------- ---------- -----------
Jim                  2018-05-01 1
Jim                  2018-05-03 1
Jim                  2018-05-04 1
Jim                  NULL       3
John                 2018-05-01 1
John                 2018-05-02 1
John                 2018-05-03 1
John                 NULL       3
NULL                 NULL       6
```

The highlighted rows were added as a result of the WITH ROLLUP clause and contain super aggregates for the following:

- All orders for Jim and John regardless of OrderDate (Orange).
- A super aggregated for all customers and all dates (Red).

Using CUBE instead of ROLLUP adds super aggregates in all possible combinations, not only in group by expression order.

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer, OrderDate
WITH CUBE
```

```
Customer             OrderDate  NumOrders
-------------------- ---------- -----------
```

```
Jim                  2018-05-01 1
John                 2018-05-01 1
NULL                 2018-05-01 2
John                 2018-05-02 1
NULL                 2018-05-02 1
Jim                  2018-05-03 1
John                 2018-05-03 1
NULL                 2018-05-03 2
Jim                  2018-05-04 1
NULL                 2018-05-04 1
NULL                 NULL       6
Jim                  NULL       3
John                 NULL       3
```

Note the additional four green highlighted rows, which were added by the CUBE. They provide super aggregates for every date for all customers that were not part of the ROLLUP results above.

Legacy GROUP BY ALL

Use the Orders table from the previous example.

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM   Orders AS O
WHERE  OrderDate <= '20180503'
GROUP BY ALL Customer, OrderDate
```

```
Customer             OrderDate  NumOrders
-------------------- ---------- -----------
Jim                  2018-05-01 1
John                 2018-05-01 1
John                 2018-05-02 1
Jim                  2018-05-03 1
John                 2018-05-03 1
Jim                  2018-05-04 0
Warning: Null value is eliminated by an aggregate or other SET operation.
```

The row highlighted in orange for 2018-05-04 failed the WHERE clause and was returned as an empty group as indicated by the warning for the empty COUNT(*) = 0.

Use GROUPING SETS

The following query uses the ANSI compliant GROUPING SETS syntax to provide all possible aggregate combinations for the Orders table, similar to the result of the CUBE syntax. This syntax requires specifying each dimension that needs to be aggregated.

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM   Orders AS O
GROUP BY GROUPING SETS (
                        (Customer, OrderDate),
                        (Customer),
                        (OrderDate),
```

```
                          ()
                        )
```

```
Customer             OrderDate  NumOrders
-------------------- ---------- -----------
Jim                  2018-05-01 1
John                 2018-05-01 1
NULL                 2018-05-01 2
John                 2018-05-02 1
NULL                 2018-05-02 1
Jim                  2018-05-03 1
John                 2018-05-03 1
NULL                 2018-05-03 2
Jim                  2018-05-04 1
NULL                 2018-05-04 1
NULL                 NULL       6
Jim                  NULL       3
John                 NULL       3
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/queries/select-group-by-transact-sql*

Migrate to Aurora MySQL GROUP BY

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - GROUP BY	• Basic syntax compatible • Advanced options like ALL, CUBE, GROUPING SETS will require rewrites to use multiple queries with UNION

Overview

Aurora MySQL supports only the basic ANSI syntax for GROUP BY and does not support GROUPING SETS or the standard GROUP BY CUBE and GROUP BY ROLLUP. It does support the WITH ROLLUP non-ANSI syntax like SQL Server, but not the CUBE option.

Aurora MySQL Supports a wider range of aggregate functions than SQL Server and include:

```
AVG, BIT_AND, BIT_OR, BIT_XOR, COUNT, GROUP_CONCAT, JSON_ARRAYAGG, JSON_OBJECTAGG,
MAX, MIN, STD, STDDEV, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP, VARIANCE
```

The bitwise aggregates and the JSON aggregates not available in SQL Server may prove to be very useful in many scenarios. For more information, see https://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html.

Unlike SQL Server, Aurora MySQL does not allow using ROLLUP and ORDER BY clauses in the same query. As a workaround, encapsulate the ROLLUP query as a derived table and add the ORDER BY clause to the outer query.

```
SELECT *
FROM (
       SELECT   Customer,
                OrderDate,
                COUNT(*) AS NumOrders
       FROM Orders AS O
       GROUP BY Customer, OrderDate
       WITH ROLLUP
      )
ORDER BY OrderDate, Customer;
```

Additionally, rows produced by ROLLUP cannot be referenced in a WHERE clause or in a FROM clause as a join condition because the super aggregates are added late in the processing phase.

Even more problematic is the lack of a function equivalent to the GROUPING_ID function in SQL Server, which can be used to distinguish super aggregate rows from the base groups. Unfortunately, it is currently not possible to distinguish rows that have NULLs due to being super aggregates from rows where the NULL is from the base set.

Until SQL92, column expressions not appearing in the GROUP BY list were not allowed in the HAVING, SELECT, and ORDER BY clauses. This limitation still applies in SQL Server today. For example, the following query is not legal in SQL Serve since a customer group may contain multiple order dates.

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer
```

However, in some cases, when the columns that do not appear in the GROUP BY clause are functionally dependent on the GROUP BY columns, it does make sense to allow it and ANSI SQL optional feature T301 does allow it. Aurora MySQL can detect such functional dependencies and allows such queries to execute.

> **Note:** To allow non-aggregate columns in the HAVING, SELECT, and ORDER BY clauses, ONLY_FULL_GROUP_BY SQL mode must be disabled.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/group-by-functional-dependence.html.

Syntax

```
SELECT <Select List>
FROM <Table Source>
WHERE <Row Filter>
GROUP BY  <Column Name> | <Expression> | <Position>
          [ASC | DESC], ...
          [WITH ROLLUP]]
```

Migration Considerations

For most aggregate queries that use only grouping expressions without modifiers, the migration should be straightforward. Even the WITH ROLLUP syntax is supported as-is in Aurora MySQL. For more complicated aggregates such as CUBE and GROUPING SETS, a rewrite to include all sub-aggregate queries as UNION ALL sets is required.

Since Aurora MySQL supports a wider range of aggregate functions, the migration should not present major challenges. Some minor syntax changes, for example replacing STDEVP with STDDEV_POP), can be performed automatically by the AWS Schema Conversion Tool (SCT). Some may need manual intervention such as rewriting the STRING_AGG syntax to GROUP_CONCAT. Also consider using Aurora MySQL additional aggregate functions for query optimizations.

If you plan to keep using the WITH ROLLUP groupings, you must consider how to handle NULLS since Aurora MySQL does not support an equivalent function to SQL Server's GROUPING_ID.

Examples

Rewrite SQL Server WITH CUBE modifier for migration. Also see the example in SQL Server GROUP BY.

```
CREATE TABLE Orders
(
       OrderID INT NOT NULL AUTO_INCREMENT
       PRIMARY KEY,
       Customer VARCHAR(20) NOT NULL,
       OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders(Customer, OrderDate)
VALUES ('John', '20180501'),  ('John', '20180502'), ('John', '20180503'),
       ('Jim', '20180501'), ('Jim', '20180503'), ('Jim', '20180504')
```

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY Customer, OrderDate
WITH ROLLUP
UNION ALL -- Add the super aggregate rows per OrderDate
SELECT  NULL,
        OrderDate,
        COUNT(*) AS NumOrders
FROM Orders AS O
GROUP BY OrderDate
```

```
Customer OrderDate NumOrders
-------- --------- ---------
Jim      2018-05-01    1
Jim      2018-05-03    1
Jim      2018-05-04    1
Jim      NULL          3
John     2018-05-01    1
John     2018-05-02    1
John     2018-05-03    1
John     NULL          3
NULL     NULL          6
NULL     2018-05-01    2
NULL     2018-05-02    1
NULL     2018-05-03    2
NULL     2018-05-04    1
```

Rewrite SQL Server GROUP BY ALL for migration. Also see the example in SQL Server GROUP BY.

```
SELECT  Customer,
        OrderDate,
        COUNT(*) AS NumOrders
FROM   Orders AS O
WHERE  OrderDate <= '20180503'
GROUP BY Customer, OrderDate
UNION ALL -- Add the empty groups
SELECT DISTINCT Customer,
       OrderDate,
       NULL
```

```
FROM   Orders AS O
WHERE  OrderDate > '20180503';
```

```
Customer OrderDate NumOrders
-------- --------- ---------
Jim      2018-05-01    1
Jim      2018-05-03    1
John     2018-05-01    1
John     2018-05-02    1
John     2018-05-03    1
Jim      2018-05-04    NULL
```

Summary

Table of similarities, differences, and key migration considerations.

SQL Server feature	Aurora MySQL feature	Comments
MAX, MIN, AVG, COUNT, COUNT_BIG	MAX, MIN, AVG, COUNT	In Aurora MySQL, COUNT returns a BIGINT and is compatible with SQL Server's COUNT and COUNT_BIG.
CHECKSUM_AGG	N/A	Use a loop to calculate checksums.
GROUPING, GROUPING_ID	N/A	Reconsider query logic to avoid having NULL groups that are ambiguous with the super aggregates.
STDEV, STDEVP, VAR, VARP	STDDEV, STDDEV_POP, VARIANCE, VAR_POP	Rewrite keywords only.
STRING_AGG	GROUP_CONCAT	Rewrite syntax. See https://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html#function_group-concat
WITH ROLLUP	WITH ROLLUP	Compatible
WITH CUBE	N/A	See the example for rewrite using UNION ALL.
ANSI CUBE / ROLLUP	N/A	Rewrite using WITH ROLLUP and using UNION ALL queries.
GROUPING SETS	N/A	Rewrite using UNION ALL queries.
N/A	NON AGGREGATE COLUMNS IN HAVING, SELECT, ORDER BY	Requires ONLY_FULL_GROUP_BY SQL Mode to be disabled, functional dependencies are evaluated by the engine.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/group-by-functions-and-modifiers.html

Migrate from SQL Server Table JOIN

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Table Joins	• Basic syntax compatible • FULL OUTER, APPLY, and ANSI SQL 89 outer joins will need to be rewritten

Overview

ANSI JOIN

SQL Server supports the standard ANSI join types:

- **<Set A> CROSS JOIN <Set B>:** Results in a Cartesian product of the two sets. Every JOIN starts as a Cartesian product.
- **<Set A> INNER JOIN <Set B> ON <Join Condition>:** Filters the cartesian product to only the rows where the join predicate evaluates to TRUE.
- **<Set A> LEFT OUTER JOIN <Set B> ON <Join Condition>:** Adds to the INNER JOIN all the rows from the reserved left set with NULL for all the columns that come from the right set.
- **<Set A> RIGHT OUTER JOIN <Set B> ON <Join Condition>:** Adds to the INNER JOIN all the rows from the reserved right set with NULL for all the columns that come from the left set.
- **<Set A> FULL OUTER JOIN <Set B> ON <Join Condition>:** Designates both sets as reserved and adds non matching rows from both, similar to a LEFT OUTER JOIN and a RIGHT OUTER JOIN.

APPLY

SQL Server also supports the APPLY operator, which is somewhat similar to a join. However, APPLY operators enable the creation of a correlation between <Set A> and <Set B> such as that <Set B> may consist of a sub query, a VALUES row value constructor, or a table valued function that is evaluated per row of <Set A> where the <Set B> query can reference columns from the current row in <Set A>. This functionality is not possible with any type of standard JOIN operator.

There are two APPLY types:

- **<Set A> CROSS APPLY <Set B>:** Similar to an CROSS JOIN in the sense that every row from <Set A> is matched with every row from <Set B>.
- **<Set A> OUTER APPLY <Set B>:** Similar to a LEFT OUTER JOIN in the sense that rows from <Set A> are returned even if the sub query for <Set B> produces an empty set. In that case, NULL is assigned to all columns of <Set B>.

ANSI SQL 89 JOIN Syntax

Up until SQL Server version 2008R2, SQL Server also supported the "old style" JOIN syntax including LEFT and RIGHT OUTER JOIN.

The ANSI syntax for a CROSS JOIN operator was to list the sets in the FROM clause using commas as separators. For example:

```
SELECT *
FROM Table1,
     Table2,
     Table3...
```

To perform an INNER JOIN, you only needed to add the JOIN predicate as part of the WHERE clause. For example:

```
SELECT *
FROM Table1,
     Table2
WHERE Table1.Column1 = Table2.Column1
```

Although the ANSI standard didn't specify outer joins at the time, most RDBMS supported them in one way or another. T-SQL supported outer joins by adding an asterisk to the left or the right of equality sign of the join predicate to designate the reserved table. For example:

```
SELECT *
FROM Table1,
     Table2
WHERE Table1.Column1 *= Table2.Column1
```

To perform a FULL OUTER JOIN, asterisks were placed on both sides of the equality sign of the join predicate.

As of SQL Server 2008R2, outer joins using this syntax have been depracated in accordance with https://technet.microsoft.com/it-it/library/ms143729(v=sql.105).aspx.

> **Note:** Even though INNER JOINs using the ANSI SQL 89 syntax are still supported, they are highly discouraged due to being notorious for introducing hard to catch programming bugs.

Syntax

CROSS JOIN

```
FROM  <Table Source 1>
      CROSS JOIN
      <Table Source 2>
```

```
-- ANSI 89
FROM  <Table Source 1>,
      <Table Source 2>
```

INNER / OUTER JOIN

```
FROM <Table Source 1>
     [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } }] JOIN
     <Table Source 2>
     ON <JOIN Predicate>
```

```
-- ANSI 89
FROM  <Table Source 1>,
      <Table Source 2>
WHERE <Join Predicate>
<Join Predicate>:: <Table Source 1 Expression> | = | *= | =* | *=* <Table Source 2
Expression>
```

APPLY

```
FROM <Table Source 1>
     { CROSS | OUTER } APPLY
     <Table Source 2>
<Table Source 2>:: <SELECT sub-query> | <Table Valued UDF> | <VALUES clause>
```

Examples

Create the Orders and Items tables.

```
CREATE TABLE Items
(
Item VARCHAR(20) NOT NULL
  PRIMARY KEY
Category VARCHAR(20) NOT NULL,
Material VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Items (Item, Category, Material)
VALUES
('M8 Bolt', 'Metric Bolts', 'Stainless Steel'),
('M8 Nut', 'Metric Nuts', 'Stainless Steel'),
('M8 Washer', 'Metric Washers', 'Stainless Steel'),
('3/8" Bolt', 'Imperial Bolts', 'Brass')
```

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL
REFERENCES Items(Item),
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200)
```

INNER JOIN

```
SELECT *
FROM Items AS I
     INNER JOIN
     OrderItems AS OI
```

```
     ON I.Item = OI.Item;

-- ANSI SQL 89
SELECT *
FROM Items AS I,
     OrderItems AS OI
WHERE I.Item = OI.Item;
```

LEFT OUTER JOIN

Find Items that were never ordered.

```
SELECT Item
FROM Items AS I
     LEFT OUTER JOIN
     OrderItems AS OI
     ON I.Item = OI.Item
WHERE OI.OrderID IS NULL;

-- ANSI SQL 89
SELECT Item
FROM
(
 SELECT I.Item, O.OrderID
 FROM Items AS I,
     OrderItems AS OI
 WHERE I.Item *= OI.Item
) AS LeftJoined
WHERE LeftJoined.OrderID IS NULL;
```

FULL OUTER JOIN

```
CREATE TABLE T1(Col1 INT, COl2 CHAR(2));
CREATE TABLE T2(Col1 INT, COl2 CHAR(2));

INSERT INTO T1 (Col1, Col2)
VALUES (1, 'A'), (2,'B');

INSERT INTO T2 (Col1, Col2)
VALUES (2,'BB'), (3,'CC');

SELECT *
FROM   T1
       FULL OUTER JOIN
       T2
       ON T1.Col1 = T2.Col1;
```

```
Result:
Col1    COl2    Col1    COl2
----    ----    ----    ----
1       A       NULL    NULL
2       B       2       BB
NULL    NULL    3       CC
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/queries/from-transact-sql

Migrate to Aurora MySQL Table JOIN

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Table Joins	• Basic syntax compatible • FULL OUTER, APPLY, and ANSI SQL 89 outer joins will need to be rewritten

Overview

Aurora MySQL supports the following types of joins in the same way as SQL Server, except for FULL OUTER JOIN:

- **<Set A> CROSS JOIN <Set B>:** Results in a Cartesian product of the two sets. Every JOIN starts as a Cartesian product.
- **<Set A> INNER JOIN <Set B> ON <Join Condition>:** Filters the Cartesian product to only the rows where the join predicate evaluates to TRUE.
- **<Set A> LEFT OUTER JOIN <Set B> ON <Join Condition>:** Adds to the INNER JOIN all the rows from the reserved left set with NULL for all the columns that come from the right set.
- **<Set A> RIGHT OUTER JOIN <Set B> ON <Join Condition>:** Adds to the INNER JOIN all the rows from the reserved right set with NULL for all the columns that come from the left set.

In addition, Aurora MySQL supports the following join types not supported by SQL Server:

- **<Set A> NATURAL [INNER | LEFT OUTER | RIGHT OUTER] JOIN <Set B>:** Implicitly assumes that the join predicate consists of all columns with the same name from <Set A> and <Set B>.
- **<Set A> STRAIGHT_JOIN <Set B>:** Forces <Set A> to be read before <Set B> and is used as an optimizer hint.

Aurora MySQL also supports the USING clause as an alternative to the ON clause. The USING clause consists of a list of comma separated columns that must appear in both tables. The Join predicate is the equivalent of an AND logical operator for equality predicates of each column. For example, the following two JOINs are equivalent:

```
FROM Table1
     INNER JOIN
     Table2
     ON Table1.Column1 = Table2.column1;
```

```
FROM Table1
     INNER JOIN
     Table2
     USING (Column1);
```

If Column1 is the only column with a common name between Table1 and Table2, the following statement is also equivalent:

```
FROM Table1
     NATURAL JOIN
     Table2
```

Note: Aurora MySQL supports the ANSI SQL 89 syntax for joins using commas in the FROM clause, but only for INNER JOINs.

Note: Aurora MySQL supports neither APPLY nor the equivalent LATERAL JOIN used by some other RDBMS.

Syntax

```
FROM
    <Table Source 1> CROSS JOIN <Table Source 2>
  | <Table Source 1> INNER JOIN <Table Source 2>
     ON <Join Predicate> | USING (Equality Comparison Column List)
  | <Table Source 1> {LEFT|RIGHT} [OUTER] JOIN <Table Source 2>
     ON <Join Predicate> | USING (Equality Comparison Column List)
  | <Table Source 1> NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN <Table Source 2>
  | <Table Source 1> STRAIGHT_JOIN <Table Source 2>
  | <Table Source 1> STRAIGHT_JOIN <Table Source 2>
     ON <Join Predicate>
```

Migration Considerations

For most JOINs, the syntax should be equivalent and no rewrites should be needed.

- CROSS JOIN using either ANSI SQL 89 or ANSI SQL 92 syntax
- INNER JOIN using either ANSI SQL 89 or ANSI SQL 92 syntax
- OUTER JOIN using the ANSI SQL 92 syntax only

FULL OUTER JOIN and OUTER JOIN using the pre-ANSI SQL 92 syntax are not supported, but they can be easily worked around (see the examples below).

CROSS APPLY and OUTER APPLY are not supported and need to be rewritten.

Examples

Create the Orders and Items tables.

```
CREATE TABLE Items
(
Item VARCHAR(20) NOT NULL
  PRIMARY KEY
Category VARCHAR(20) NOT NULL,
Material VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Items (Item, Category, Material)
VALUES
('M8 Bolt', 'Metric Bolts', 'Stainless Steel'),
('M8 Nut', 'Metric Nuts', 'Stainless Steel'),
```

```
('M8 Washer', 'Metric Washers', 'Stainless Steel'),
('3/8" Bolt', 'Imperial Bolts', 'Brass')
```

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL
REFERENCES Items(Item),
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200)
```

INNER JOIN and OUTER JOIN

```
SELECT *
FROM Items AS I
     INNER JOIN
     OrderItems AS OI
     ON I.Item = OI.Item;

-- ANSI SQL 89
SELECT *
FROM Items AS I,
     Orders AS O
WHERE I.Item = OI.Item;
```

LEFT OUTER JOIN

```
SELECT Item
FROM Items AS I
     LEFT OUTER JOIN
     OrderItems AS OI
     ON I.Item = OI.Item
WHERE OI.OrderID IS NULL;
```

Rewrite for FULL OUTER JOIN

```
CREATE TABLE T1(Col1 INT, COl2 CHAR(2));
CREATE TABLE T2(Col1 INT, COl2 CHAR(2));

INSERT INTO T1 (Col1, Col2)
VALUES (1, 'A'), (2,'B');

INSERT INTO T2 (Col1, Col2)
VALUES (2,'BB'), (3,'CC');

SELECT  *
FROM   T1
```

```
       LEFT OUTER JOIN
       T2
       ON T1.Col1 = T2.Col1
UNION ALL
SELECT NULL, NULL, Col1, Col2
FROM   T2
WHERE  Col1 NOT IN (SELECT Col1 FROM T1);
```

```
Result:
Col1    Col2    Col1    Col2
----    ----    ----    ----
1       A       NULL    NULL
2       B       2       BB
NULL    NULL    3       CC
```

Summary

Table of similarities, differences, and key migration considerations.

SQL Server	Aurora MySQL	Comments
INNER JOIN with ON clause or commas	Supported	
OUTER JOIN with ON cluase	Supported	
OUTER JOIN with commas	Not supported	Requires T-SQL rewrite post SQL Server 2008R2.
CROSS JOIN or using commas	Supported	
CROSS APPLY and OUTER APPLY	Not Supported	Rewrite required.
Not Supported	NATURAL JOIN	Not recommended, may cause unexpected issues if table structure changes.
Not Supported	STRAIGHT_JOIN	
Not Supported	USING clause	

For more information, see https://dev.mysql.com/doc/refman/5.7/en/join.html

Migrate from SQL Server Views

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Minor syntax and handling differences • Indexes, Triggers, and temporary views not supported

Overview

Views are schema objects that provide stored definitions for virtual tables. Similar to tables, views are data sets with uniquely named columns and rows. With the exception of indexed views, view objects do not store data. They consist only of a query definition and are reevaluated for each invocation.

Views are used as abstraction layers and security filters for the underlying tables. They can JOIN and UNION data from multiple source tables and use aggregates, window functions, and other SQL features as long as the result is a semi-proper set with uniquely identifiable columns and no order to the rows. You can use Distributed Views to query other databases and data sources using linked servers.

As an abstraction layer, a view can decouple application code from the database schema. The underlying tables can be changed without the need to modify the application code, as long as the expected results of the view do not change. You can use this approach to provide backward compatible views of data.

As a security mechanism, a view can screen and filter source table data. You can perform permission management at the view level without explicit permissions to the base objects, provided the ownership chain is maintained.
For more information on ownership chains in SQL Server, see https://docs.microsoft.com/en-us/-dotnet/framework/data/adonet/sql/overview-of-sql-server-security.

View definitions are evaluated when they are created and are not affected by subsequent changes to the underlying tables.
For example, a view that uses SELECT * does not display columns that were added later to the base table. Similarly, if a column was dropped from the base table, invoking the view results in an error. Use the SCHEMABINDING option to prevent changes to base objects.

Modifying Data Through Views

Updatable Views can both SELECT and modify data. For a view to be updatable, the following conditions must be met:

- The DML targets only one base table.
- Columns being modified must be directly referenced from the underlying base tables. Computed columns, set operators, functions, aggregates, or any other expressions are not permitted.
- If a view is created with the CHECK OPTION, rows being updated can not be filtered out of the view definition as the result of the update.

Special View Types

SQL Server also provides three types of "special" views:

- **Indexed Views** (also known as materialized views or persisted views) are standard views that have been evaluated and persisted in a unique clustered index, much like a normal clustered primary key table. Each time the source data changes, SQL Server re-evaluates the indexed views automatically and updates the indexed view. Indexed views are typically used as a means to optimize performance by pre-processing operators such as aggregations, joins, and others. Queries needing this pre-processing don't have to wait for it to be reevaluated on every query execution.
- **Partitioned Views** are views that rejoin horizontally partitioned data sets from multiple underlying tables, each containing only a subset of the data. The view uses a UNION ALL query where the underlying tables can reside locally or in other databases (or even other servers). These types of views are called Distributed Partitioned Views (DPV).
- **System Views** are used to access server and object meta data. SQL Server also supports a set of standard INFORMATION_SCHEMA views for accessing object meta data.

Syntax

```
CREATE [OR ALTER] VIEW [<Schema Name>.] <View Name> [(<Column Aliases> ])]
[WITH [ENCRYPTION][SCHEMABINDING][VIEW_METADATA]]
AS <SELECT Query>
[WITH CHECK OPTION][;]
```

Examples

Create a view that aggregates items for each customer.

```
CREATE TABLE Orders
(
OrderID INT NOT NULL PRIMARY KEY,
OrderDate DATETIME NOT NULL
 DEFAULT GETDATE()
);
```

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL
 REFERENCES Orders(OrderID),
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
CREATE VIEW SalesView
AS
SELECT O.Customer,
       OI.Product,
       SUM(CAST(OI.Quantity AS BIGINT)) AS TotalItemsBought
```

```
FROM Orders AS O
     INNER JOIN
     OrderItems AS OI
      ON O.OrderID = OI.OrderID;
```

Create an indexed view that pre-aggregates items for each customer.

```
CREATE VIEW SalesViewIndexed
AS
SELECT O.Customer,
       OI.Product,
       SUM_BIG(OI.Quantity) AS TotalItemsBought
FROM Orders AS O
     INNER JOIN
     OrderItems AS OI
      ON O.OrderID = OI.OrderID;
```

```
CREATE UNIQUE CLUSTERED INDEX IDX_SalesView
ON SalesViewIndexed (Customer, Product);
```

Create a Partitioned View.

```
CREATE VIEW dbo.PartitioneView
WITH SCHEMABINDING
AS
SELECT *
FROM Table1
UNION ALL
SELECT *
FROM Table2
UNION ALL
SELECT *
FROM Table3
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/relational-databases/views/views*
- *https://docs.microsoft.com/en-us/sql/relational-databases/views/modify-data-through-a-view*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql*

Migrate to Aurora MySQL Views

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Minor syntax and handling differences • Indexes, Triggers, and temporary views not supported

Overview

Similar to SQL Server, Aurora MySQL views consist of a SELECT statement that can references base tables and other views.

Aurora MySQL views are created using the CREATE VIEW statement. The SELECT statement comprising the definition of the view is evaluated only when the view is created and is not affected by subsequent changes to the underlying base tables.

Aurora MySQL Views have the following restrictions:

- A view cannot reference system variables or user-defined variables.
- When used within a stored procedure or function, the SELECT statement cannot reference parameters or local variables.
- A view cannot reference prepared statement parameters.
- All objects referenced by a view must exist when the view is created. If an underlying table or view is later dropped, invoking the view results in an error.
- Views cannot reference TEMPORARY tables.
- TEMPORARY views are not supported.
- Views do not support triggers.
- Aliases are limited to a maximum length of 64 characters (not the typical 256 maximum alias length).

Aurora MySQL provides additional properties not available in SQL Server:

- The ALGORITHM clause is a fixed hint that affects they way the MySQL query processor handles the view physical evaluation operator.
  The MERGE algorithm uses a dynamic approach where the definition of the view is merged to the outer query.
  The TEMPTABLE algorithm materializes the view data internally.
  For more information, see https://dev.mysql.com/doc/refman/5.7/en/view-algorithms.html.
- The DEFINER and SQL SECURITY clauses can be used to specify a specific security context for checking view permissions at run time.

Similar to SQL Server, Aurora MySQL supports updatable views and the ANSI standard CHECK OPTION to limit inserts and updates to rows referenced by the view.

The LOCAL and CASCADED keywords are used to determine the scope of violation checks. When using the LOCAL keyword, the CHECK OPTION is evaluated only for the view being created. CASCADED causes evaluation of referenced views. The default is CASCADED.

In general, only views having a one-to-one relationship between the source rows and the exposed rows are updatable.
Adding the following constructs prevents modification of data:

- Aggregate functions
- DISTINCT
- GROUP BY
- HAVING
- UNION or UNION ALL
- Subquery in the select list
- Certain joins
- Reference to a non-updatable view
- Subquery in the WHERE clause that refers to a table in the FROM clause
- ALGORITHM = TEMPTABLE
- Multiple references to any column of a base table

A view must have unique column names. Column aliases are derived from the base tables or explicitly specified in the SELECT statement of column definition list. ORDER BY is permitted in Aurora MySQL, but ignored if the outer query has an ORDER BY clause.

Aurora MySQL assesses data access privileges as follows:

- The user creating a view must have all required privileges to use the top-level objects referenced by the view.
  For example, for a view referencing table columns, the user must have privilege for each column in any clause of the view definition.
- If the view definition references a stored function, only the privileges needed to invoke the function are checked. The privileges required at run time can be checked only at run time because different invocations may use different execution paths within the function code.
- The user referencing a view must have appropriate SELECT, INSERT, UPDATE, or DELETE privileges , as with a normal table.
- When a view is referenced, privileges for all objects accessed by the view are evaluated using the privileges for the view DEFINER account, or the invoker, depending on whether SQL SECURITY is set to DEFINER or INVOKER.
- When a view invocation triggers the execution of a stored function, privileges are checked for statements executed within the function based on the function's SQL SECURITY setting. For

functions where the security is set to DEFINER, the function executes with the privileges of the DEFINER account. For functions where it is set to INVOKER, the function executes with the privileges determined by the view's SQL SECURITY setting as described above.

Syntax

```
CREATE [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { <User> | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW <View Name> [(<Column List>)]
    AS <SELECT Statement>
    [WITH [CASCADED | LOCAL] CHECK OPTION];
```

Migration Considerations

The basic syntax for views is very similar to SQL Server and is ANSI compliant. Code migration should be straightforward.

Aurora MySQL does not support triggers on views. In SQL Server, INSTEAD OF triggers are supported. For more information, see Triggers.

In Aurora MySQL, ORDER BY is permitted in a view definition. It is ignored if the outer SELECT has its own ORDER BY. This behavior is different than SQL Server where ORDER BY is allowed only for TOP filtering. The actual order of the rows is not guaranteed.

Security context is explicit in Aurora MySQL, which is not supported in SQL Server. Use security contexts to work around the lack of ownership-chain permission paths.

Unlike SQL Server, a view in Aurora MySQL can invoke functions, which in turn may introduce a change to the database.
For more information, see User Defined Functions.

The WITH CHECK option in Aurora MySQL can be scoped to LOCAL or CASCADED. The CASCADED causes the CHECK option to be evaluated for nested views referenced in the parent.

Indexed views are not supported in Aurora MySQL. Consider using application maintained tables instead. Change application code to reference those tables instead of the base table.

Examples

Create and populate an `Invoices` table.

```
CREATE TABLE Invoices(
InvoiceID       INT NOT NULL PRIMARY KEY,
Customer        VARCHAR(20) NOT NULL,
TotalAmount     DECIMAL(9,2) NOT NULL);

INSERT INTO Invoices (InvoiceID,Customer,TotalAmount)
VALUES
(1, 'John', 1400.23),
```

```
(2, 'Jeff', 245.00),
(3, 'James', 677.22);
```

Create the `TotalSales` view.

```
CREATE VIEW TotalSales
AS
SELECT Customer,
       SUM(TotalAmount) AS CustomerTotalAmount
GROUP BY Customer;
```

Invoke the view.

```
SELECT * FROM TotalSales
ORDER BY CustomerTotalAmount DESC;

Customer        CustomerTotalAmount
--------        -----------------
John            1400.23
James           677.22
Jeff            245.00
```

Summary

Feature	SQL Server	Aurora MySQL	Comments
Indexed Views	Supported	N/A	
Partitioned Views	Supprted	N/A	While you can create partitioned views in the same way as SQL Server, they won't benefit from the internal optimizations such as partition elimination.
Updateable Views	Supported	Supported	
Prevent schema conflicts	SCHEMABINDING option		
Triggers on views	INSTEAD OF	N/A	See Triggers.
Temporary Views	CREATE VIEW #View...	N/A	
Refresh view definition	sp_refreshview / ALTER VIEW	ALTER VIEW	

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/create-view.html*
- *https://dev.mysql.com/doc/refman/5.7/en/view-restrictions.html*
- *https://dev.mysql.com/doc/refman/5.7/en/view-updatability.html*

Migrate from SQL Server Window Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Window Functions	• Rewrite window functions to use alternative SQL syntax

Overview

Windowed functions use an OVER clause to define the window and frame for a data set to be processed. They are part of the ANSI standard and are typically compatible among various SQL dialects. However, most RDBMS do not yet support the full ANSI specification.

Windowed functions are a relatively new, advanced, and efficient T-SQL programming tool. They are highly utilized by developers to solve numerous programming challenges.

SQL Server currently supports the following windowed functions:

- **Ranking functions:** ROW_NUMBER, RANK, DENSE_RANK, and NTILE
- **Aggregate functions:** AVG, MIN, MAX, SUM, COUNT, COUNT_BIG, VAR, STDEV, STDEVP, STRING_AGG, GROUPING, GROUPING_ID, VAR, VARP, and CHECKSUM_AGG
- **Analytic functions:** LAG, LEAD, FIRST_Value, LAST_VALUE, PERCENT_RANK, PERCENTILE_CONT, PERCENTILE_DISC, and CUME_DIST
- **Other functions:** NEXT_VALUE_FOR (See the *Identity and Sequences* section)

Syntax

```
<Function()>
OVER
(
[ <PARTITION BY clause> ]
[ <ORDER BY clause> ]
[ <ROW or RANGE clause> ]
)
```

Examples

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

Use a windowed ranking function to rank items based on the ordered quantity.

```
SELECT  Item,
        Quantity,
        RANK() OVER(ORDER BY Quantity) AS QtyRank
FROM    OrderItems;
```

```
Item           Quantity  QtyRank
----           --------  -------
M8 Bolt        100       1
M8 Nut         100       1
M8 Washer      200       3
M6 Locking Nut 300       4
```

Use a partitioned windowed aggregate function to calculate the total quantity per order (without using a GROUP BY clause).

```
SELECT  Item,
        Quantity,
        OrderID,
        SUM(Quantity)
         OVER (PARTITION BY OrderID) AS TotalOrderQty
FROM    OrderItems;
```

```
Item           Quantity  OrderID  TotalOrderQty
----           --------  -------  -------------
M8 Bolt        100       1        100
M8 Nut         100       2        100
M6 Locking Nut 300       3        500
M8 Washer      200       3        500
```

Use an analytic LEAD function to get the next largest quantity for the order.

```
SELECT  Item,
       Quantity,
       OrderID,
       LEAD(Quantity)
        OVER (PARTITION BY OrderID ORDER BY Quantity) AS NextQtyOrder
FROM   OrderItems;
```

```
Item           Quantity OrderID NextQtyOrder
----           -------- ------- ------------
M8 Bolt        100     1        NULL
M8 Nut         100     2        NULL
M8 Washer      200     3        300
M6 Locking Nut 300     3        NULL
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql

Migrate to Aurora MySQL Window Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Window Functions	• Rewrite window functions to use alternative SQL syntax

Overview

Aurora MySQL version 5.7 does not support windowed functions. However, the next version of MySQL, version 8, will include partial support.

Migration Considerations

As a temporary workaround, rewrite the code to remove the use of windowed functions, and revert to using more traditional SQL code solutions.

In most cases, you can find an equivalent SQL query, although it may be less optimal in terms of performance, simplicity, and readability.
See the examples below for migrating windowed functions to code that uses correlated subqueries.

> **Note:** You may want to archive the original code and then reuse it in the future when Aurora MySQL is upgraded to version 8. The documentation for version 8 indicates the windowed function syntax is ANSI compliant and will be compatible with SQL Server's T-SQL syntax.

For more information about the upcoming version 8 support for window functions, see https://dev.mysql.com/doc/refman/8.0/en/window-functions.html

Examples

The following examples demonstrate ANSI SQL compliant subquery solutions as replacements for the two example queries from the previous SQL Server section:

Create and populate an `OrderItems` table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
```

```
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

Rank items based on ordered quantity (workaround for the windowed ranking function).

```
SELECT  Item,
        Quantity,
       (
         SELECT COUNT(*)
         FROM OrderItems AS OI2
         WHERE OI.Quantity > OI2.Quantity) + 1
        AS QtyRank
FROM    OrderItems AS OI;
```

```
Item           Quantity  QtyRank
----           --------  -------
M8 Bolt        100       1
M8 Nut         100       1
M6 Locking Nut  300       4
M8 Washer      200       3
```

Calculate the grand total (workaround for a partitioned windowed aggregate function) .

```
SELECT  Item,
        Quantity,
        OrderID,
        (
         SELECT SUM(Quantity)
         FROM OrderItems AS OI2
         WHERE OI2.OrderID = OI.OrderID)
         AS TotalOrderQty
FROM    OrderItems AS OI;
```

```
Item           Quantity  OrderID  TotalOrderQty
----           --------  -------  -------------
M8 Bolt        100       1        100
M8 Nut         100       2        100
M6 Locking Nut 300       3        500
M8 Washer      200       3        500
```

```
Get the next largest quantity for the order (workaround for LEAD analytical func-
tion)SELECT     Item,
               Quantity,
               OrderID,
               (
                SELECT Quantity
                FROM   OrderItems AS OI2
                WHERE  OI.OrderID = OI2.OrderID
                               AND
                               OI2.Quantity > OI.Quantity
                ORDER BY Quantity
                LIMIT 1
               )
```

```
               AS NextQtyOrder
FROM OrderItems AS OI
```

```
Item           Quantity OrderID NextQtyOrder
----           -------- ------- ------------
M8 Bolt        100      1       [NULL]
M8 Nut         100      2       [NULL]
M6 Locking Nut  300      3       [NULL]
M8 Washer      200      3       300
```

Summary

SQL Server	Aurora MySQL	Comments
Window Functions and OVER clause	Not supported yet	Convert code to use traditional SQL techniques such as correlated sub queries.

For more information, see https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html

T-SQL

Migrate from SQL Server Collations

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Collations	• UNICODE uses CHARACTER SET property instead of NCHAR/NVARCHAR data types

Overview

SQL Server collations define the rules for string management and storage in terms of sorting, case sensitivity, accent sensitivity, and code page mapping. SQL Server supports both ASCII and UCS-2 UNICODE data.

UCS-2 UNICODE data uses a dedicated set of UNICODE data types denoted by the prefix "N": Nchar and Nvarchar. Their ASCII counterparts are CHAR and VARCHAR.

Choosing a collation and a character set has significant implications on data storage, logical predicate evaluations, query results, and query performance.

> **Note:** To view all collations supported by SQL Server, use the fn_helpcollations function: `SELECT * FROM sys.fn_helpcollations()`.

Collations define the actual bitwise binary representation of all string characters and the associated sorting rules. SQL Server supports multiple collations down to the column level. A table may have multiple string columns that use different collations. Collations for non-UNICODE character sets determine the code page number representing the string characters.

> **Note:** UNICODE and non-UNICODE data types in SQL Server are not compatible. A predicate or data modification that introduces a type conflict is resolved using predefined collation precedence rules.
> For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/collation-precedence-transact-sql

Collations define sorting and matching sensitivity for the following string characteristics:

- Case
- Accent
- Kana
- Width
- Variation selector

SQL Server uses a suffix naming convention that appends the option name to the collation name. For example, the collation Azeri_Cyrillic_100_CS_AS_KS_WS_SC, is an Azeri-Cyrillic-100 collation that is case-sensitive, accent-sensitive, kana type-sensitive, width-sensitive, and has supplementary characters.

SQL Server supports three types of collation sets:

- **Windows Collations** use the rules defined for collations by the operating system locale where UNICODE and non-UNICODE data use the same comparison algorithms.
- **Binary Collations** use the binary bit-wise code for comparison. Therefore, the locale does not affect sorting.
- **SQL Server Collations** provide backward compatibility with previous SQL Server versions. They are not compatible with the windows collation rules for non-UNICODE data.

Collations can be defined at various levels:

- **Server Level Collations** determine the collations used for all system databases and is the default for future user databases. While the system databases collation can not be changed, an alternative collation can be specified as part of the CREATE DATABASE statement
- **Database Level Collations** inherit the server default unless the CREATE DATABASE statement explicitly sets a different collation. This collation is used as a default for all CREATE TABLE and ALTER TABLE statements
- **Column Level Collations** can be specified as part of the CREATE TABLE or ALTER TABLE statements to override the database's default collation setting.
- **Expression Level Collations** can be set for individual string expressions using the COLLATE function. For example, `SELECT * FROM MyTable ORDER BY StringColumn COLLATE Latin1_General_CS_AS`.

  Note: SQL Server supports UCS-2 UNICODE only.

Syntax

```
CREATE DATABASE <Database Name>
[ ON   <File Specifications> ]
  COLLATE <Collation>
[ WITH  <Database Option List> ];
```

```
CREATE TABLE <Table Name>
(
<Column Name> <String Data Type>
COLLATE <Collation> [ <Column Constraints> ]...
);
```

Examples

Create a database with a default Bengali_100_CS_AI collation.

```
CREATE DATABASE MyBengaliDatabase
ON
( NAME = MyBengaliDatabase_Datafile,
    FILENAME = 'C:\Program Files\Microsoft SQL Server-
\MSSQL13.MSSQLSERVER\MSSQL\DATA\MyBengaliDatabase.mdf',
    SIZE = 100)
LOG ON
( NAME = MyBengaliDatabase_Logfile,
```

```
    FILENAME = 'C:\Program Files\Microsoft SQL Server-
\MSSQL13.MSSQLSERVER\MSSQL\DATA\MyBengaliDblog.ldf',
    SIZE = 25)
COLLATE Bengali_100_CS_AI;
```

Create a table with two different collations.

```
CREATE TABLE MyTable
(
Col1 CHAR(10) COLLATE Hungarian_100_CI_AI_SC NOT NULL PRIMARY KEY,
COL2 VARCHAR(100) COLLATE Sami_Sweden_Finland_100_CS_AS_KS NOT NULL
);
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support

Migrate to Aurora MySQL Collations

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Collations	• UNICODE uses CHARACTER SET property instead of NCHAR/NVARCHAR data types

Overview

Aurora MySQL supports multiple character sets and a variety of collations that can be used for comparison. Similar to SQL Server, collations can be defined at the server, database, and column level. They can also be defined at the table level.

The paradigm of collations in Aurora MySQL is different than in SQL Server and consists of separate character set and collation objects. Aurora MySQL Supports 41 different character sets and 222 collations. Seven different UNICODE character sets are supported including UCS-2, UTF-8 and UTF-32.

> **Note:** Use UCS-2 which is compatible with SQL Server UNICODE types

Each character set can have one or more associated collations with a single default collation.

Collation names have prefixes consisting of the name of their associated character set followed by suffixes that indicate additional characteristics.

To see all character sets supported by Aurora MySQL, use the INFORMATION_SCHEMA.CHARACTER_SETS table or the SHOW CHARACTER SET statement.
To see all collations for a character set, use the INFORMATION_SCHEMA.COLLATIONS table or the SHOW COLLATION statement.

> **Note:** Character set and collation settings also affect client-to -server communications. You can set explicit collations for sessions using the SET command.
> For example, `SET NAMES 'utf8';` causes Aurora MySQL to treat incoming object names as UTF-8 encoded.

The default character set and collations can be set at the server level using custom cluster parameter groups.
For more information, see the example below and Server Options.

At the database level, a default character set and collation can be set with the CREATE DATABASE and ALTER DATABASE statements.
For example:

```
CREATE DATABASE MyDatabase
CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

To view the default character set and collation for an Aurora MySQL databases, use the following statement:

```
SELECT DEFAULT_CHARACTER_SET_NAME,
       DEFAULT_COLLATION_NAME
FROM INFORMATION_SCHEMA.SCHEMATA
WHERE SCHEMA_NAME = '<Database Name>';
```

> **Note:** In Aurora MySQL, a "database" is equivalent to an SQL Server schema.
> For more information, see Databases and Schemas.

Every string column in Aurora MySQL has a character set and an associated collation. If not explicitly specified, it will inherit the table default. To specify a non-default character set and collation, use the CHARACTER SET and COLLATE clauses of the CREATE TABLE statement.

```
CREATE TABLE MyTable
(
StringColumn VARCHAR(5) NOT NULL
CHARACTER SET latin1
COLLATE latin1_german1_ci
);
```

At the expression level, similar to SQL Server, the COLLATE function can be used to explicitly declare a string's collation. In addition, a prefix to the string can be used to denote a specific character set. For example:

```
SELECT _latin1'Latin non-UNICODE String',
       _utf8'UNICODE String' COLLATE utf8_danish_ci;
```

> **Note:** The Aurora MySQL term for this "prefix" or "string header" is "introducer". It doesn't change the value of the string; only the character set.

At the session level, the server's setting determines the default character set and collation used to evaluate non-qualified strings.

Although the server's character set and collation default settings can be modified using the cluster parameter groups, it is recommended that client applications do not assume a specific setting and explicitly set the required character set and collation using the `SET NAMES` and `SET CHARACTER SET` statements.
For more details about these SET options, see https://dev.mysql.com/doc/refman/5.7/en/charset-connection.html.

Syntax

Database level collation:

```
CREATE DATABASE <Database Name>
[DEFAULT] CHARACTER SET <Character Set>
[[DEFAULT] COLLATE <Collation>];
```

Table level collation:

```
CREATE TABLE <Table Name>
(Column Specifications)
```

```
[DEFAULT] CHARACTER SET <Character Set>
[COLLATE <Collation>];
```

Column collation:

```
CREATE TABLE <Table Name>
(
<Column Name> {CHAR | VARCHAR | TEXT} (<Length>)
CHARACTER SET CHARACTER SET <Character Set>
[COLLATE <Collation>];
```

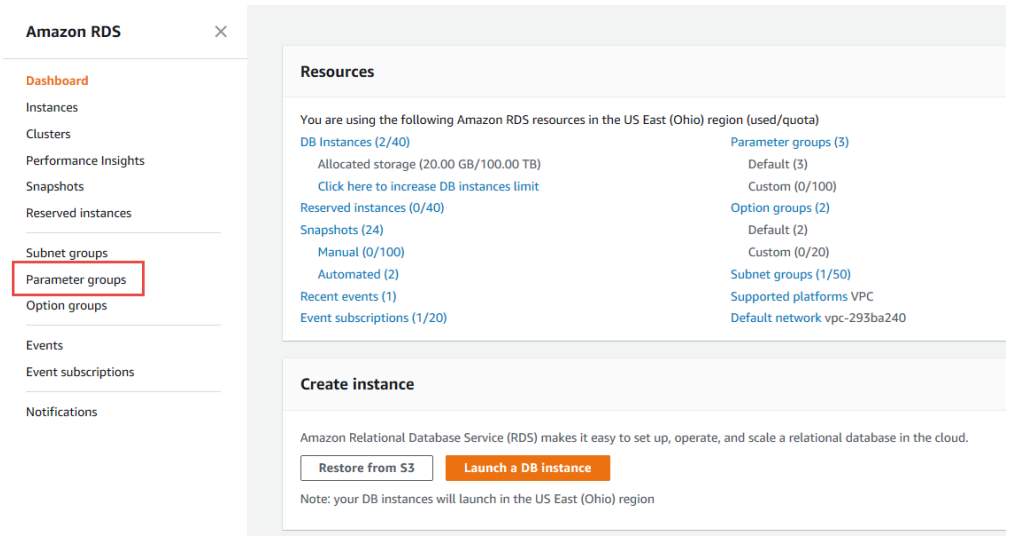
Expression collation:

```
_<Character Set>'<String>' COLLATE <Collation>
```

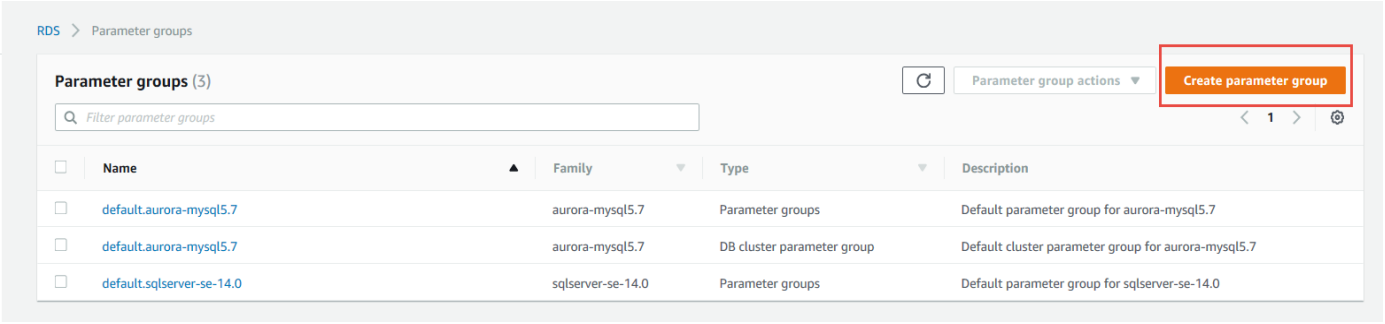
Examples

The following wlkthrough describes how to change the Cluster Character Set and Collation:

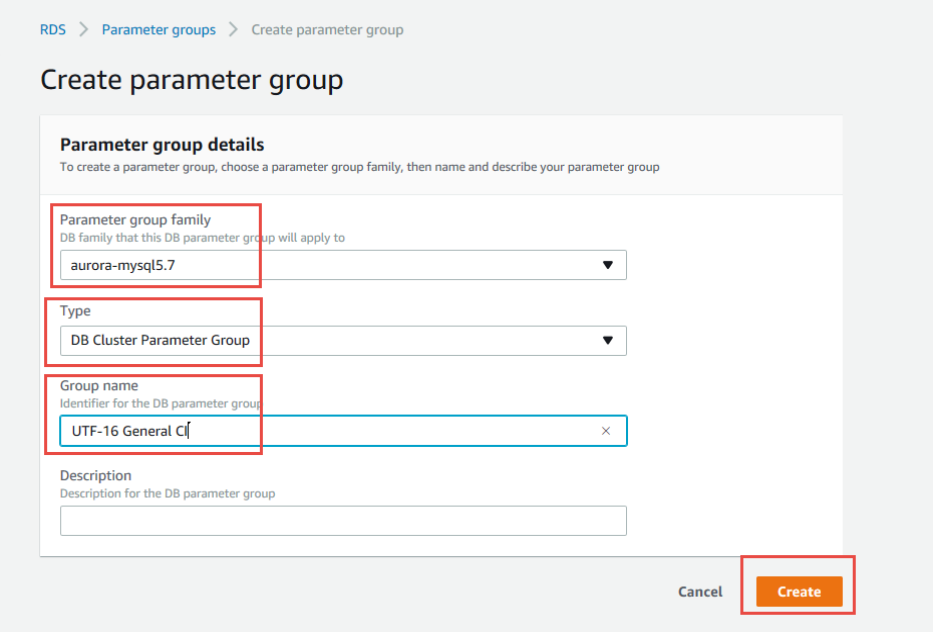
Log in to the AWS RDS Console and click **Parameter Groups**.



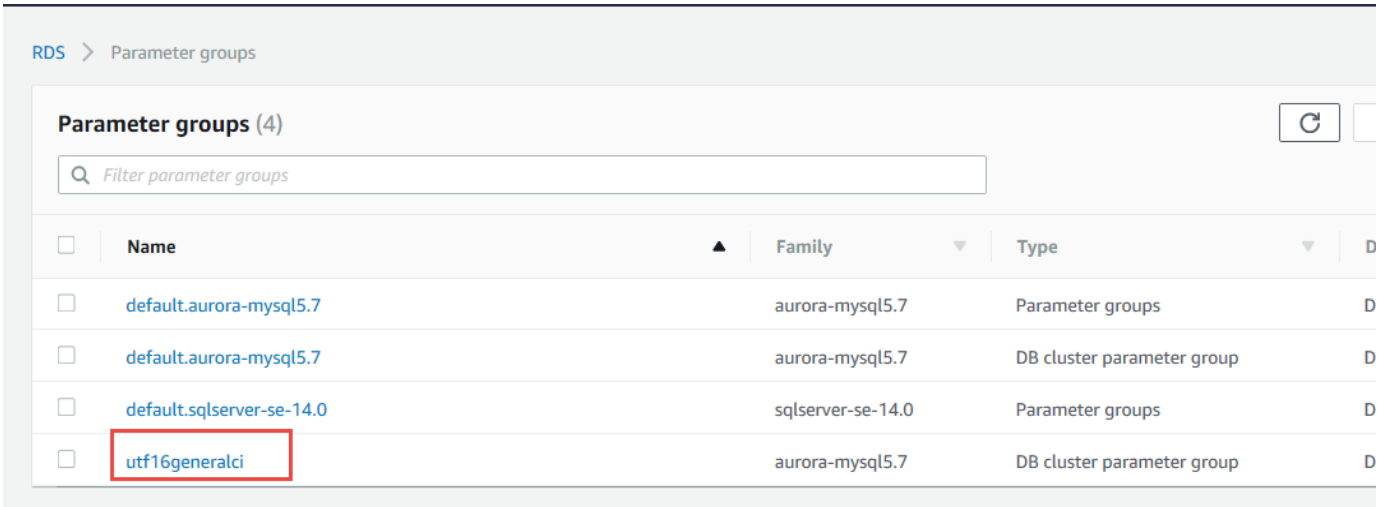
Click the **Create Parameter Group** button on the top right.



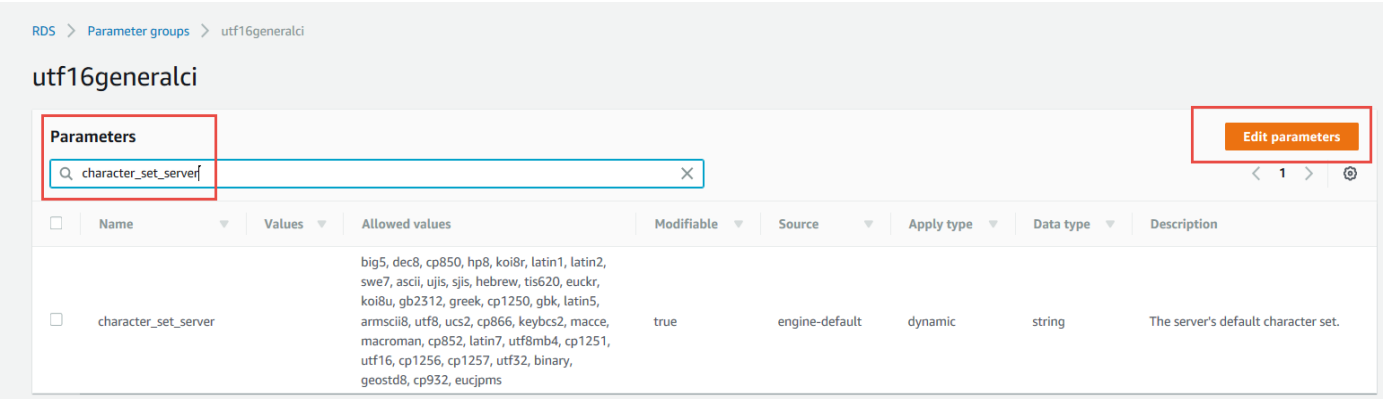
Set **Parameter Group Family** to **aurora-mysql5.7** and **Type** to **DB Cluster Parameter Group**. Enter a value for **Group name**. Click **Create**.



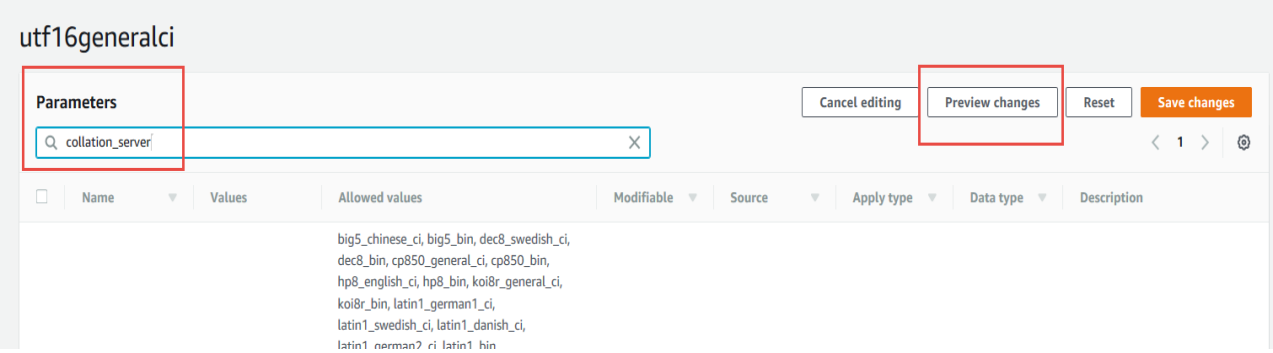
Select the newly created group from the **Parameter groups** list and click the link.



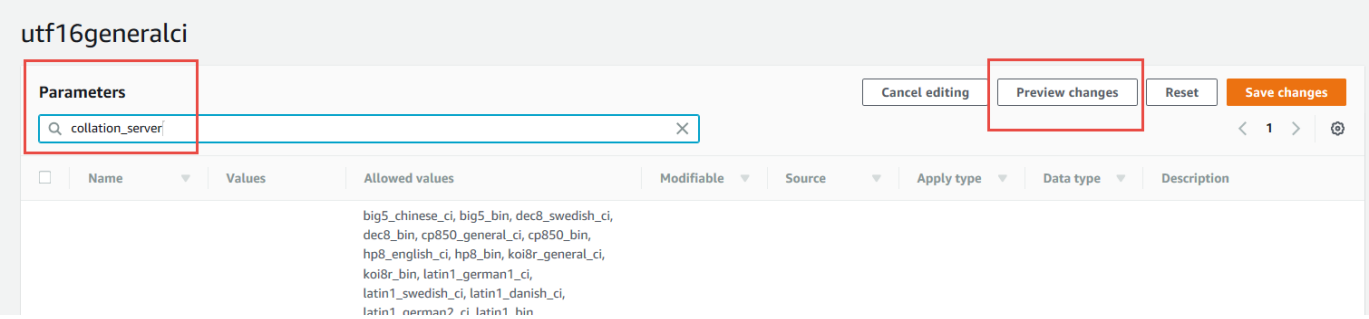
In the **Parameter Groups** window, enter **character_set_server** in the **Parameters** search box and click **Edit Parameters**.



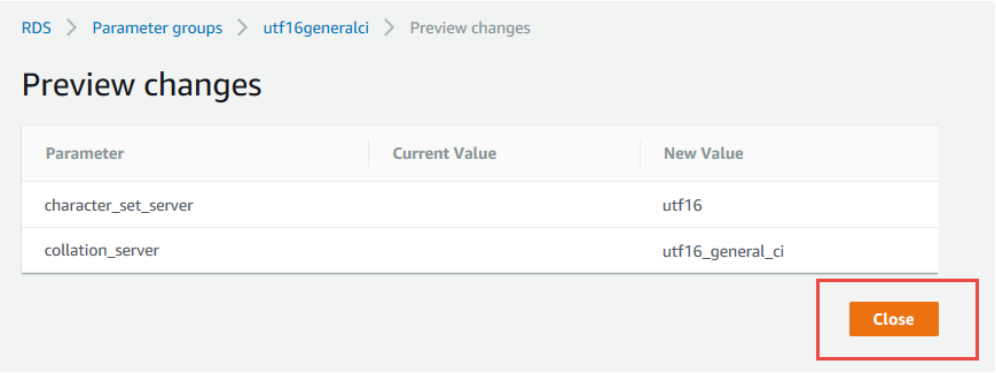
Select the server default character set from the the drop-down menu.



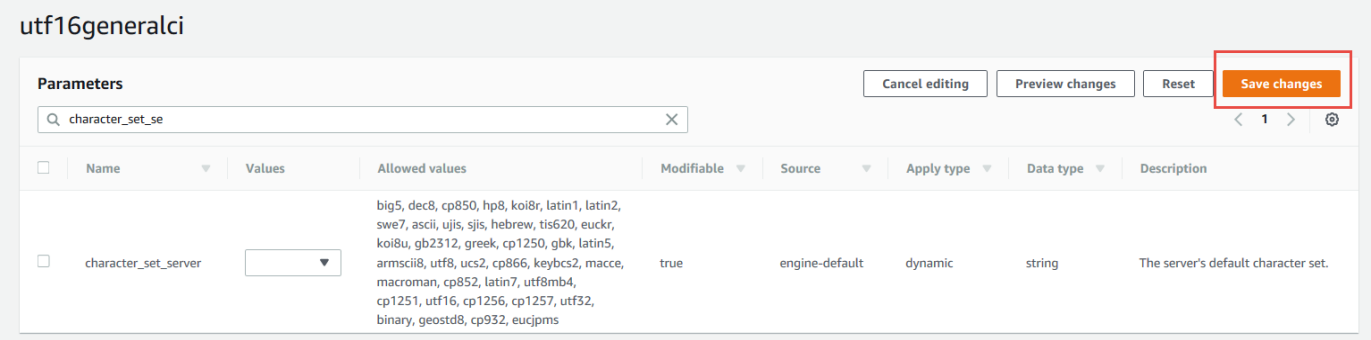
Delete the search term and enter **collation**. Use the pull down menu to select the desired default server collation from the drop-down menu. Click **Preview Changes**.



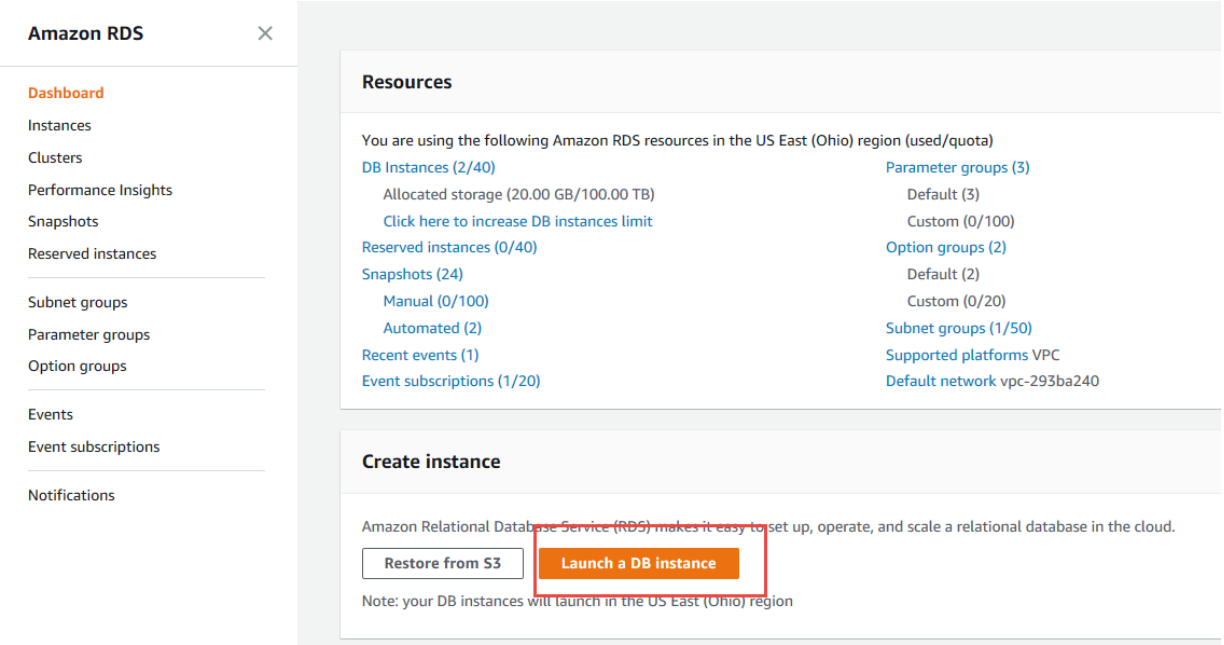
Check the values and click **Close**.



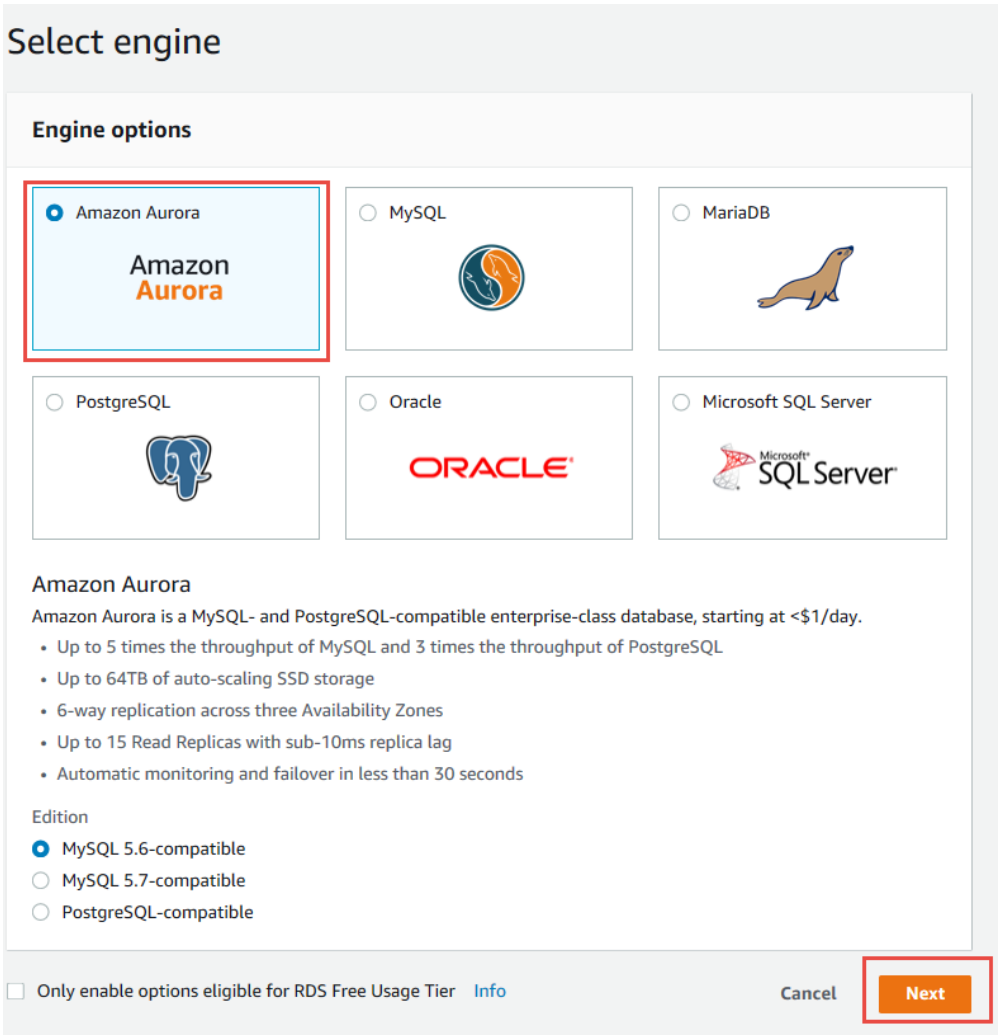
Click **Save Changes**.



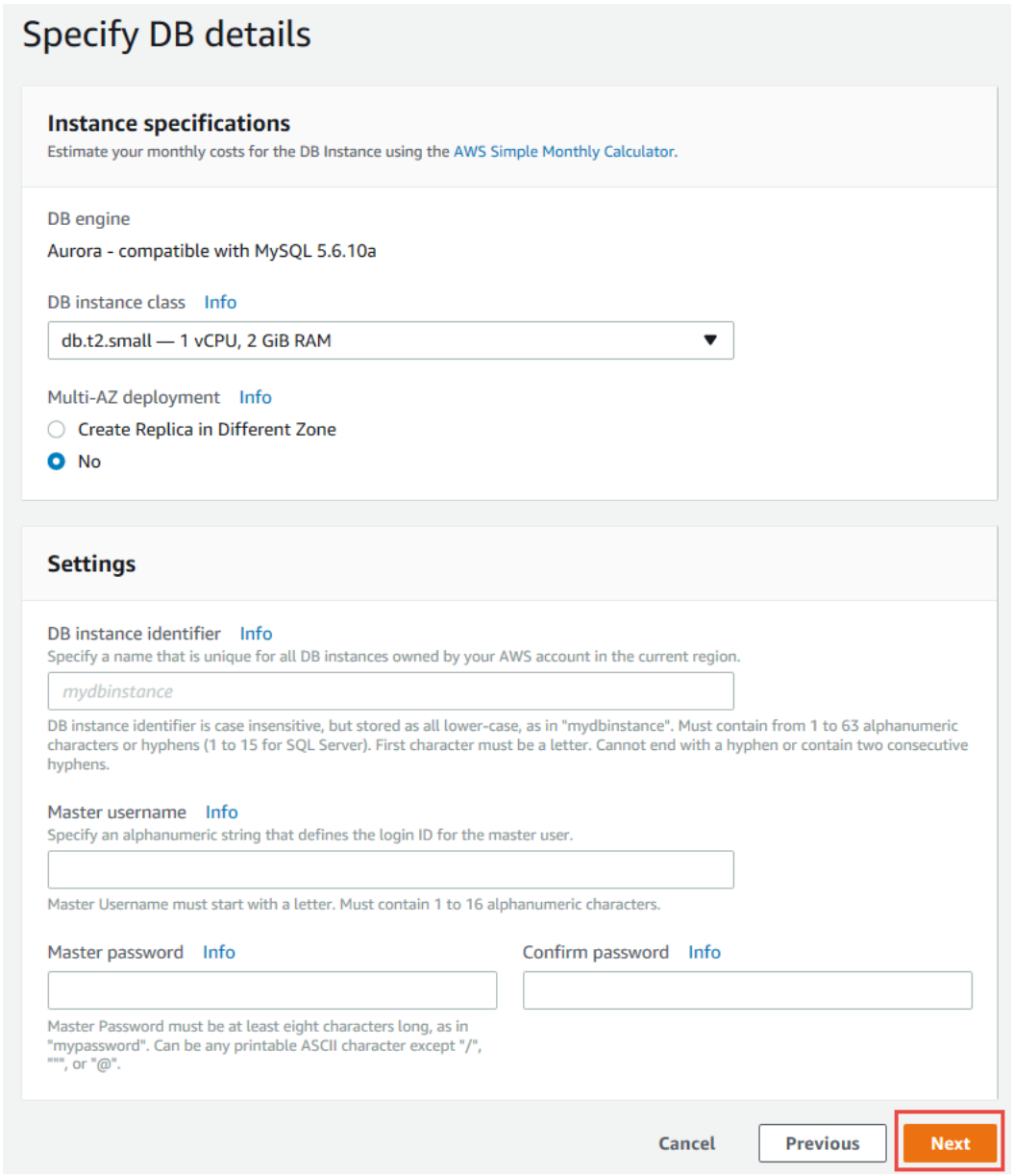
Return to the AWS RDS Console dashboard and click **Launch a DB Instance**.



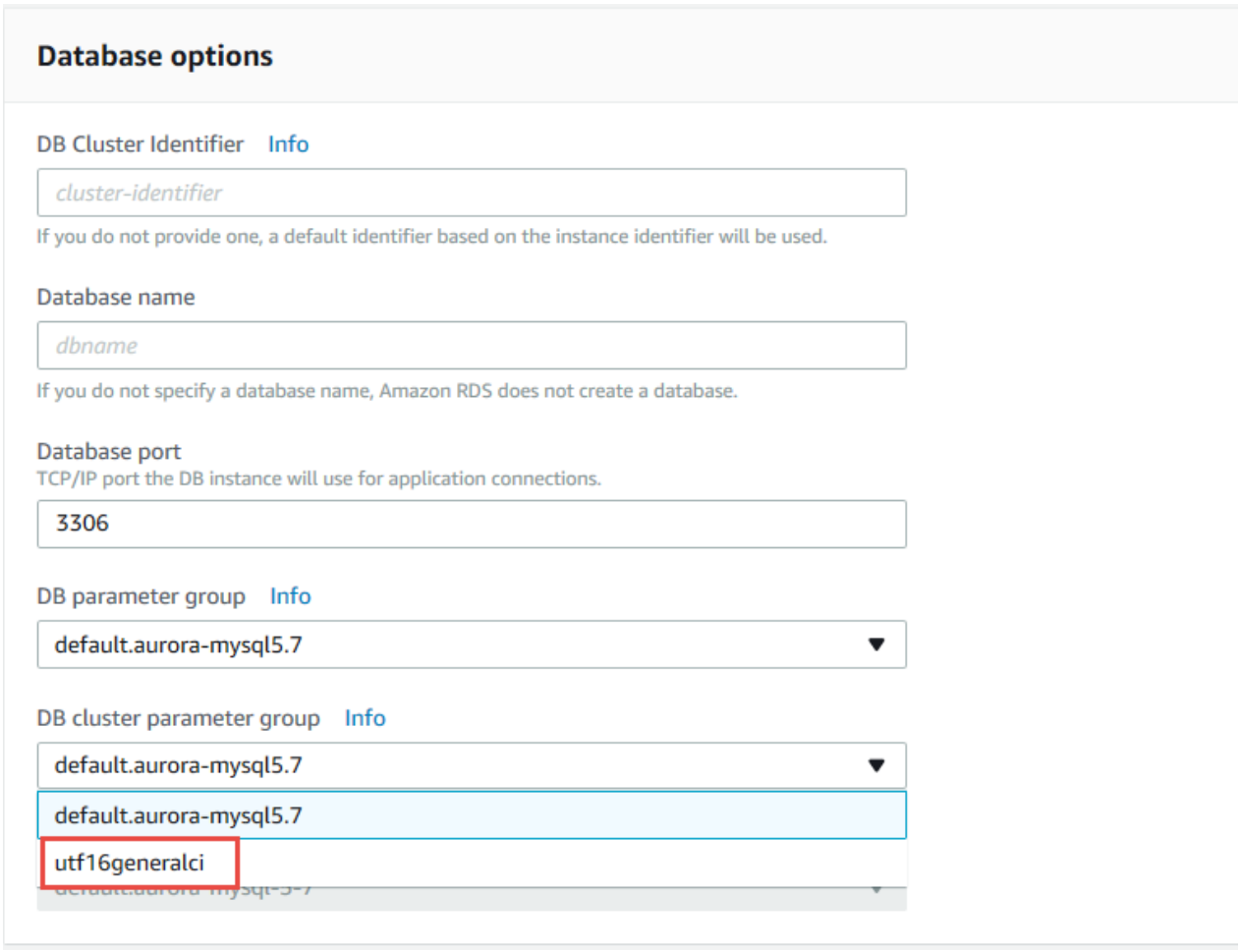
Select **Amazon Aurora** and click **Next**.



Specify the instance parameters and click **Next**.



In the advanced setting window, scroll down to the **Database options** section and select the newly created parameter group for the new cluster.



Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
Unicode support	UTF 16 via NCHAR and NVARCHAR data types	8 UNICODE character sets, via the CHARACTER SET option	
Collations levels	Server, Database, Column, Expression	Server, Database, Table, Column, Expression	
View collation metadata	fn_helpcollation system view	INFORMATION_ SCHEMA.SCHEMATA, SHOW COLLATION, SHOW CHARACTER SET	

For more information, see https://dev.mysql.com/doc/refman/5.7/en/charset.html

Migrate from SQL Server Cursors

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Cursors	• Only static, forward only, read-only cursors are supported in Aurora MySQL

Overview

A *set* is a fundamental concept of the relation data model, from which SQL is derived. SQL is a declarative language that operates on whole sets, unlike most procedural languages that operate on individual data elements. A single invocations of an SQL statements can return a whole set or modify millions of rows.

Many developers are accustom to using procedural or imperative approaches to develop solutions that are difficult to implement using set-based querying techniques. Also, operating on row data sequentially may be a more appropriate approach is certain situations.

Cursors provide an alternative mechanism for operating on result sets. Instead of receiving a table object containing rows of data, applications can use cursors to access the data sequentially, row-by-row. Cursors provide the following capabilities:

- Positioning the cursor at specific rows of the result set using absolute or relative offsets.
- Retrieving a row, or a block of rows, from the current cursor position.
- Modifying data at the current cursor position.
- Isolating data modifications by concurrent transactions that affect the cursor's result.
- T-SQL statements can use cursors in scripts, stored procedures, and triggers.

Syntax

```
DECLARE <Cursor Name>
CURSOR [LOCAL | GLOBAL]
     [FORWARD_ONLY | SCROLL]
     [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
     [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
     [TYPE_WARNING]
     FOR <SELECT statement>
     [ FOR UPDATE [ OF <Column List>]][;]
```

```
FETCH [NEXT | PRIOR | FIRST | LAST | ABSOLUTE <Value> | RELATIVE <Value>]
FROM  <Cursor Name> INTO <Variable List>;
```

Examples

Process data in a cursor.

```
DECLARE MyCursor CURSOR FOR
   SELECT *
   FROM Table1 AS T1
        INNER JOIN
        Table2 AS T2
        ON T1.Col1 = T2.Col1;
   OPEN MyCursor;
   DECLARE @VarCursor1 VARCHAR(20);
   FETCH NEXT
    FROM MyCursor INTO @VarCursor1;

   WHILE @@FETCH_STATUS = 0
   BEGIN
       EXEC MyPRocessingProcedure
            @InputParameter = @VarCursor1;
       FETCH NEXT
        FROM product_cursor INTO @VarCursor1;
   END

   CLOSE MyCursor;
   DEALLOCATE MyCursor ;
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/cursors*
- *https://docs.microsoft.com/en-us/sql/t-sql/language-elements/cursors-transact-sql*

Migrate to Aurora MySQL Cursors

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Cursors	• Only static, forward only, read-only cursors are supported in Aurora MySQL

Overview

Aurora MySQL supports cursors only within stored routines, functions and stored procedures.

Unlike SQL Server, which offers an array of cursor types, Aurora MySQL Cursors have the following characteristics:

- **Asensitive:** The server can choose to either make a copy of its result table or to access the source data as the cursor progresses.
- **Read only:** Cursors are not updatable.
- **Nonscrollable:** Cursors can only be traversed in one direction and cannot skip rows. The only supported cursor advance operation is FETCH NEXT.

Cursor declarations must appear before handler declarations and after variable and condition declarations.

Similar to SQL Server, cursors are declared with the DECLARE CURSOR, opened with OPEN, fetched with FETCH, and closed with CLOSE.

> **Note:** Aurora MySQL does not have a DEALLOCATE statement because it is not needed.

DECLARE cursor

Syntax

```
DECLARE <Cursor Name> CURSOR
FOR <Cursor SELECT Statement>
```

The DECLARE CURSOR statement instantiates a cursor object and associates it with a SELECT statement. This SELECT is then used to retrieve the cursor rows.

To fetch the rows, use the FETCH statement. As mentioned above, only FETCH NEXT is supported. The number of output variables specified in the FETCH statement must match the number of columns retrieved by the cursor.

Aurora MySQL cursors have additional characteristics:

- SELECT INTO is not allowed in a cursor.
- Stored routing can have multiple cursor declarations, but every cursor declared in a given code

block must have a unique name.

- Cursors can be nested.

OPEN cursor

Syntax

```
OPEN <Cursor Name>;
```

The OPEN command populates the cursor with the data, either dynamically or in a temporary table, and readies the first row for consumption by the FETCH statement.

FETCH cursor

Syntax

```
FETCH [[NEXT] FROM] <Cursor Name>
INTO <Variable 1> [,<Variable n>]
```

The FETCH statement retrieves the current pointer row, assigns the column values to the variables listed in the FETCH statement, and advances the cursor pointer by one row. If the row is not available, meaning the cursor has been exhausted, a No Data condition is raised with an SQLSTATE value of '0200000'.

To catch this condition, or the alternative NOT FOUND condition, you must create a condition handler. For more information, see Error Handling.

> **Note:** Carefully plan your error handling flow. The same condition might be raised by other SELECT statements or other cursors than the one you intended. Place operations within BEGIN-END blocks to associate each cursor with its own handler.

CLOSE cursor

Syntax

```
CLOSE <Cursor Name>;
```

The CLOSE statement closes an open cursor. If the cursor with the specified name does not exist, an error is raised. If a cursor is not explicitly closed, Aurora MySQL closes it automatically at the end of the BEGIN ... END block in which it was declared.

Migration Considerations

The Aurora MySQL Cursors framework is much simpler than SQL Server and provides only the basic types. If your code relies on advanced cursor features, these will need to be rewritten.

However, most applications use forward only, read only cursors, and those will be easy to migrate.

If your application uses cursors in ad-hoc batches, move the code to a stored procedure or a function.

Examples

The following example uses a cursor to iterate over source rows and merges into an `OrderItems` table.

Create an `OrderItems`table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create and populate the `SourceTable`.

```
CREATE TABLE SourceTable
(
OrderID INT,
Item VARCHAR(20),
Quantity SMALLINT,
PRIMARY KEY (OrderID, Item)
);
```

```
INSERT INTO SourceTable (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Create a procedure to loop through `SourceTable`and insert rows.

> **Note:** There are syntax differences between T-SQL for the CREATE PROCEDURE and the CURSOR declaration. For more details, see Stored Procedures and Cursors.

```
CREATE PROCEDURE LoopItems()
BEGIN
 DECLARE done INT DEFAULT FALSE;
 DECLARE var_OrderID INT;
 DECLARE var_Item VARCHAR(20);
 DECLARE var_Quantity SMALLINT;
 DECLARE ItemCursor CURSOR
 FOR
  SELECT OrderID,
         Item,
         Quantity
  FROM SourceTable;
 DECLARE CONTINUE HANDLER
  FOR NOT FOUND
  SET done = TRUE;
 OPEN ItemCursor;
 CursorStart: LOOP
 FETCH NEXT
  FROM ItemCursor
```

```
  INTO var_OrderID,
       var_Item,
       var_Quantity;
 IF Done
  THEN LEAVE CursorStart;
 END IF;
   INSERT INTO OrderItems (OrderID, Item, Quantity)
   VALUES (var_OrderID, var_Item, var_Quantity);
 END LOOP;
 CLOSE ItemCursor;
END;
```

Execute the stored procedure.

```
CALL LoopItems();
```

Select all rows from the `OrderItems`table.

```
SELECT * FROM OrderItems;

OrderID     Item        Quantity
-------     ----        --------
1           M8 Bolt     100
2           M8 Nut      100
3           M8 Washer   200
```

Summary

Feature	SQL Server	Aurora MySQL	Comments
Cursor options	[FORWARD_ONLY | SCROLL] [STATIC | KEYSET | DYNAMIC | FAST_FORWARD] [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]		
Updateable cursors	DECLARE CURSOR... FOR UPDATE	Not supported	
Cursor declaration	DECLARE CURSOR	DECLARE CURSOR	No options for DECLARE CURSOR in Aurora MySQL.
Cursor open	OPEN	OPEN	
Cursor fetch	FETCH NEXT | PRIOR | FIRST | LAST | ABSOLUTE | RELATIVE	FETCH NEXT	
Cursor close	CLOSE	CLOSE	
Cursor Deallocate	DEALLOCATE	N/A	Not required, CLOSE also deallocates

Feature	SQL Server	Aurora MySQL	Comments
Cursor end condition	@@FETCH_STATUS system variable	Event Handler	Event handlers are not specific to a cursor, see Error Handling.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/cursors.html

Migrate from SQL Server Date and Time Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Date and Time Functions	• Timezone handling • Syntax differneces

Overview

Date and Time Functions are scalar functions that perform operations on temporal or numeric input and return temporal or numeric values.

System date and time values are derived from the operating system of the server where SQL Server is running.

> **Note:** This section does not address timezone considerations and timezone aware functions.
> For more information about time zone handling, see Data Types.

Syntax and Examples

The following table lists the most commonly used Date and Time Functions.

Function	Purpose	Example	Result	Comments
GETDATE and GETUTCDATE	Return a datetime value that contains the current local or UTC date and time	SELECT GETDATE()	2018-04-05 15:53:01.380	
DATEPART, DAY, MONTH, and YEAR	Return an integer value representing the specified datepart of a specified date	SELECT MONTH (GETDATE()), YEAR (GETDATE())	4, 2018	
DATEDIFF	Returns an integer value of datepart boundaries that are crossed between two dates	SELECT DATEDIFF (DAY, GETDATE(), EOMONTH (GETDATE()))	25	How many days left until end of the month
DATEADD	Returns a datetime value that is cal-	SELECT DATEADD (DAY, 25, GETDATE	2018-04-30 15:55:52.147	

Function	Purpose	Example	Result	Comments
	culated with an off-set interval to the specified datepart of a date.	())		
CAST and CONVERT	Converts datetime values to and from string literals and to and from other datetime formats	SELECT CAST (GETDATE() AS DATE) SELECT CONVERT (VARCHAR(20), GETDATE(), 112)	2018-04-05 20180405	Default date format Style 112 (ISO) with no seprartors

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql#DateandTimeFunctions

Migrate to Aurora MySQL Date and Time Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Date and Time Functions	• Timezone handling • Syntax difernences

Overview

Aurora MySQL provides a very rich set of scalar date and time functions; more than SQL Server.

> **Note:** While some of the functions such as DATEDIFF seem to be similar to those in SQL Server, the functionality can be significantly different. Take extra care when migrating temporal logic to Aurora MySQL paradigms.

Syntax and Examples

Function	Purpose	Example	Result	Comments
NOW | LOCALTIME |CURRENT_TIMESTAMP, and SYSDATE	Returns a datetime value that contains the current local date and time	SELECT NOW()	2018-04-06 18:57:54	SYSDATE returns the time at which it executes, vs. NOW which returns a constant time when the statement started executing. Also, SET TIMESTAMP does not affect SYSDATE.
UTC_TIMESTAMP	Returns a datetime value that contains the current UTC date and time	SELECT UTC_TIMESTAMP()	2018-04-07 04:57:54	
SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, and YEAR	Returns an integer value representing the specified date part of a specified date function	SELECT MONTH (NOW()), YEAR (NOW())	4, 2018	
DATEDIFF	Returns an integer value of	SELECT DATEDIFF (NOW(), '2018-05-	-25	DATEDIFF in Aurora MySQL is only for cal-

Function	Purpose	Example	Result	Comments
	the difference in days between two dates	01')		culating difference in Days. Use TIMESTAMPDIFF instead
TIMESTAMPDIFF	Returns an integer value of the difference in datepart between two dates	SELECT TIMESTAMPDIFF (DAY, NOW(), '2018-05-01')	24	
DATE_ADD, DATE_SUB	Returns a dat-etime value that is calculated with an offset interval to the specified date-part of a date	SELECT DATE_ ADD(NOW(), INTERVAL 1 DAY);	2018-04-07 19:35:32	
CAST and CONVERT	Converts dat-etime values to and from string literals and to and from other datetime formats	SELECT CAST (GETDATE() AS DATE) SELECT CONVERT (VARCHAR(20), GETDATE(), 112)	2018-04-05 20180405	Default date format Style 112 (ISO) with no seprartors

Migration Considerations

The Date and Time handling paradigm in Aurora MySQL differs from SQL Server.
Be aware of differences in data types, time zone awareness, and locale handling. For more inform-ation, see Data Types.

Summary

The following table identifies similarities, differences, and key migration considerations.

SQL Server Function	Aurora MySQL Func-tion	Comments
`GETDATE, CURRENT_ TIMESTAMP`	`NOW | LOCALTIME | CURRENT_ TIMESTAMP, and SYSDATE`	CURRENT_TIMESTAMP is the ANSI standard and it is compatible. SYSDATE returns the time at which it executes, unlike NOW which returns a constant time when the statement started executing. Also, SET TIMESTAMP does

SQL Server Function	Aurora MySQL Function	Comments
		not affect SYSDATE.
`GETUTCDATE`	`UTC_TIMESTAMP`	
`DAY, MONTH, and YEAR`	`DAY, MONTH, YEAR`	Compatible syntax.
`DATEPART`	`EXTRACT, or one of: MICROSECOND, SECOND, MINUTE, HOUR, DAY, DAYNAME, DAYOFWEEK, DAYOFYEAR, WEEK, MONTH, MONTHNAME, QUARTER, YEAR`	Aurora MySQL supports EXTRACT as a generic DATEPART function. For example. `EXTRACT (YEAR FROM NOW())`. It also supports individual functions for each day part.
`DATEDIFF`	`TIMESTAMPDIFF`	DATEDIFF in Aurora MySQL only calculates differences in Days.
`DATEADD`	`DATE_ADD, DATE_SUB, TIMESTAMPADD`	DATEADD in Aurora MySQL only adds full days to a datetime value. Aurora MySQL also supports DATE_SUB for subtracting date parts from a date time expression. The argument order and syntax is also different and requires a rewrite.
`CAST and CONVERT`	`DATE_FORMAT, TIME_FORMAT`	Although Aurora MySQL supports both CAST and CONVERT, they are not used for style conversion like in SQL Server. Use DATE_FORMAT and TIME_FORMAT.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/date-and-time-functions.html

Migrate from SQL Server String Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• UNICODE paradigm (See Collations) • Syntax and option differences

Overview

String Functions are typically scalar functions that perform an operation on string input and return a string or a numeric value.

Syntax and Examples

The following table lists the most commonly used string functions.

Function	Purpose	Example	Result	Comments
`ASCII and UNICODE`	Convert an ASCII or UNICODE character to its ASCII or UNICODE code	`SELECT ASCII ('A')`	65	Returns a numeric integer value
`CHAR and NCHAR`	Convert between ASCII or UNICODE code to a string character	`SELECT CHAR(65)`	'A'	Numeric integer value as input
`CHARINDEX and PATINDEX`	Find the starting position of one string expression (or string pattern) within another string expression	`SELECT CHARINDEX('ab', 'xabcdy')`	2	Returns a numeric integer value
`CONCAT and CONCAT_WS`	Combine multiple string input expressions into a single string with, or without, a separator character (WS)	`SELECT CONCAT ('a','b'), CONCAT_WS (',','a','b')`	'ab', 'a,b'	
`LEFT, RIGHT, and SUBSTRING`	Return a partial	`SELECT LEFT`	'ab', 'bc'	

Function	Purpose	Example	Result	Comments
	string from another string expression based on position and length	`('abs',2), SUBSTRING ('abcd',2,2)`		
`LOWER and UPPER`	Return a string with all characters in lower or upper case. Use for presentation or to handle case insensitive expres-sions	`SELECT LOWER ('ABcd')`	'abcd'	
`LTRIM, RTRIM and TRIM`	Remove leading and trailing spaces	`SELECT LTRIM (' abc d ')`	'abc d '	
`STR`	Convert a numeric value to a string	`SELECT STR (3.1415927,5,3)`	3.142	Numeric expressions as input
`REVERSE`	Return a string in reverse order	`SELECT REVERSE ('abcd')`	'dcba'	
`REPLICATE`	Return a string that consists of zero or more concatenated copies of another string expression	`SELECT REPLICATE ('abc', 3)`	'abcabcabc'	
`REPLACE`	Replace all occur-rences of a string expression with another	`SELECT REPLACE ('abcd', 'bc', 'xy')`	'axyd'	
`STRING_SPLIT`	Parse a list of values with a separator and return a set of all individual elements	`SELECT * FROM STRING_ SPLIT('1,2', ',') AS X(C)`	1 2	STRING_ SPLIT is a table valued function
`STRING_AGG`	Return a string that consists of con-catenated string val-ues in row groups	`SELECT STRING_ AGG(C, ',') FROM VALUES(1, 'a'), (1, 'b'), (2,'c') AS X (ID,C) GROUP BY I`	1 'ab' 2 'c'	STRING_AGG is an aggreg-ate function

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql

Migrate to Aurora MySQL String Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• UNICODE paradigm (See Collations) • Syntax and option differences

Overview

Aurora MySQL supports a large set of string functions; far more than SQL Server. See the link at the end of this section for the full list. Some of the functions, such as regular expressions (REGEXP), do not exist in SQL Server and may be useful for your application.

Syntax and Examples

The following table lists the most commonly used string functions.

Function	Purpose	Example	Result	Comments
`ASCII and ORD`	Convert an ASCII or multi-byte code to its string character	`SELECT ASCII ('A')`	65	Returns a numeric integer value
`CHAR`	Convert between a character and its UNICODE code	`SELECT CHAR (65)`	'A'	Numeric integer value as input
`LOCATE`	Find the starting position of one string expression (or string pattern) within another string expression	`SELECT LOCATE ('ab', 'xabcdy')`	2	Returns a numeric integer value
`CONCAT and CONCAT_WS`	Combine multiple string input expressions into a single string with or without a separator char-	`SELECT CONCAT ('a','b'), CONCAT_WS (',','a','b')`	'ab', 'a,b'	

Function	Purpose	Example	Result	Comments
	acter (WS)			
`LEFT, RIGHT, and SUBSTRING`	Return a partial string from another string expression based on position and length	`SELECT LEFT ('abs',2), SUBSTRING ('abcd',2,2)`	'ab', 'bc'	
`LOWER and UPPER`	Return a string with all characters in lower or upper case. Use for presentation or to handle case insensitive expressions	`SELECT LOWER ('ABcd')`	'abcd'	These have no effect when applied to binary collation strings. Convert the string to a non binary string collation to convert letter case
`LTRIM, RTRIM and TRIM`	Remove leading and trailing spaces	`SELECT LTRIM (' abc d ') SELECT TRIM (LEADING 'x' FROM 'xxxabcxxx')`	'abc d ' 'abcxxx'	TRIM in Aurora MySQL is not limited to spaces. `TRIM ([{BOTH | LEADING | TRAILING} [<Remove String>] FROM] <String>)`
`FORMAT`	Convert a numeric value to a string	`SELECT FORMAT (3.1415927,5)`	3.14159	Numeric expressions as input
`REVERSE`	Return a string in reverse order	`SELECT REVERSE ('abcd')`	'dcba'	
`REPEAT`	Return a string that consists of zero or more concatenated copies of another string expression	`SELECT REPEAT ('abc', 3)`	'abcabcabc'	
`REPLACE`	Replace all occurrence of a string expression with another	`SELECT REPLACE ('abcd', 'bc', 'xy')`	'axyd'	

Migration Considerations

Aurora MySQL does not handle ASCII and UNICODE types separately. Any string can be either UNICODE or ASCII, depending on its collation property. See Data Types.

Many of the Aurora MySQL string functions that are compatible with SQL Server also support additional functionality. For example, the TRIM and CHAR functions. Aurora MySQL also supports many functions that SQL Server does not. For example, functions that deal with a delimited list set of values. Be sure to explore all options.

Aurora MySQL also supports regular expressions. See the REGEXP and RLIKE functions to get started.

Summary

The following table identifies similarities, differences, and key migration considerations.

SQL Server function	Aurora MySQL function	Comments
`ASCII and UNICODE`	`ASCII and ORD`	Compatible, See Data Types for more information about UNICODE handling.
`CHAR and NCHAR`	`CHAR`	See Data Types for more information about UNICODE handling. Unlike SQL Server, CHAR in Aurora MySQL accepts a list of values and constructs a concatenated string.
`CHARINDEX and PATINDEX`	`LOCATE | POSITION`	LOCATE and POSITION are synonymous but do not support wild cards like PATINDEX. Use the FIND_IN_SET function to extract an element position in a comma separated value string.
`CONCAT and CONCAT_WS`	`CONCAT and CONCAT_WS`	Compatible syntax.
`LEFT, RIGHT, and SUBSTRING`	`LEFT, RIGHT, and SUBSTRING`	Compatible syntax. Aurora MySQL supports MID and SUBSTR, which are synonymous with SUBSTRING. Use the SUBSTRING_INDEX function to extract an element from a delimited list.
`LOWER and UPPER`	`LOWER AND UPPER`	Compatible syntax. LOWER and UPPER have no effect when applied to binary collation strings.

SQL Server function	Aurora MySQL function	Comments
`LTRIM, RTRIM and TRIM`	`LTRIM, RTRIM and TRIM`	Compatible syntax. TRIM in Aurora MySQL is not limited to both ends and spaces. It can by used to trim either leading or trailing characters. The syntax is> `TRIM ([{BOTH | LEADING | TRAILING} [<Remove String>] FROM] <String>)`
`STR`	`FORMAT`	FORMAT does not support full precision and scale definition, but does support locale formatting.
`REVERSE`	`REVERSE`	Compatible syntax.
`REPLICATE`	`REPEAT`	Compatible arguments.
`REPLACE`	`REPLACE`	Compatible syntax.
`STRING_SPLIT`	`Not supported`	Requires iterative code to extract elements with scalar string functions.
`STRING_AGG`	`Not supported`	Requires iterative code to build a list with scalar string functions.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/string-functions.html

Migrate from SQL Server Databases and Schemas

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• SCHEMA and DATABASE are synonymous

Overview

Databases and Schemas are logical containers for security and access control. Administrators can grant permissions collectively at both the databases and the schema levels. SQL Server instances provide security at three levels: Individual Objects, Schemas (collections of objects), Databases (collections of schemas).
For more information, see Data Control Language.

> **Note:** In previous versions of SQL server, the term *user* was interchangeable with the term *schema*. For backward compatibility, each database has several built-in security schemas including guest, dbo, db_datareaded, sys, INFORMATION_SCHEMA, and others. You most likely will not need to migrate these schemas.

Each SQL Server instance can host and manage a collection of databases, which consist of SQL Server processes and the Master, Model, TempDB, and MSDB system databases.

The most common SQL Server administrator tasks at the database level are:

- Managing Physical Files: Add, remove, change file growth settings, and re-size files.
- Managing Filegroups: Partition schemes, object distribution, and read-only protection of tables.
- Managing default options.
- Creating database snapshots.

Unique object identifiers within an instance use three-part identifiers: <Database name>.<Schema name>.<Object name>.

The recommended way to view database objects' meta data, including schemas, is to use the ANSI standard Information Schema views. In most cases, these views are compatible with other ANSI compliant RDBMS.

To view a list of all databases on the server, use the sys.databases table.

Syntax

Simplified syntax for CREATE DATABASE:

```
CREATE DATABASE <database name>
[ ON [ PRIMARY ] <file specifications>[,<filegroup>]
[ LOG ON <file specifications>
[ WITH  <options specification> ] ;
```

Simplified syntax for CREATE SCHEMA:

```
CREATE SCHEMA <schema name> | AUTHORIZATION <owner name>;
```

Examples

Add a file to a database and create a table using the new file.

```
USE master;
```

```
ALTER DATABASE NewDB
ADD FILEGROUP NewGroup;
```

```
ALTER DATABASE NewDB
ADD FILE (
          NAME = 'NewFile',
          FILENAME = 'D:\NewFile.ndf',
          SIZE = 2 MB
         )
TO FILEGROUP NewGroup;
```

```
USE NewDB;
```

```
CREATE TABLE NewTable
(
Col1 INT PRIMARY KEY
)
ON NewGroup;
```

```
SELECT Name
FROM sys.databases
WHERE database_id > 4;
```

Create a table within a new schema and database.

```
USE master
```

```
CREATE DATABASE NewDB;
```

```
USE NewDB;
```

```
CREATE SCHEMA NewSchema;
```

```
CREATE TABLE NewSchema.NewTable
(
NewColumn VARCHAR(20) NOT NULL PRIMARY KEY
);
```

Note: This example uses default settings for the new database and schema.

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-databases-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-schema-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-database-sql-server-transact-sql*

Migrate to Aurora MySQL Databases and Schemas

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• SCHEMA and DATABASE are synonymous

Overview

Aurora MySQL supports both the CREATE SCHEMA and CREATE DATABASE statements. However, in Aurora MySQL, these statements are synonymous.

Unlike SQL Server, Aurora MySQL does not have the concept of an instance hosting multiple databases, which in turn contain multiple schemas. Objects in Aurora MySQL are referenced as a two part name: <schema>.<object>. You can use the term *database* in place of schema, but it is conceptually the same thing.

> **Note:** This terminology conflict can lead to confusion for SQL Server database administrators unfamiliar with the Aurora MySQL concept of a database.

> **Note:** Each database/schema in Aurora MySQL is managed as a separate set of physical files similar to an SQL Server database.

Aurora MySQL does not have the concept of a schema owner. Permissions must be granted explicitly. However, Aurora MySQL supports a custom default collation at the schema level, whereas SQL Server supports it at the database level only. For more details, see Collations.

Syntax

Syntax for CREATE DATABASE:

```
CREATE {DATABASE | SCHEMA} <database name>
[DEFAULT] CHARACTER SET [=] <character set>|
[DEFAULT] COLLATE [=] <collation>
```

Migration Considerations

Similar to SQL Server, Aurora MySQL supports the USE command to specify the default database (schema) for missing object qualifiers.

The syntax is identical to SQL Server:

```
USE <database name>;
```

After the USE command is executed, the default database for the calling scope is changed to the specified database.

There is a relatively straightforward migration path for a class of common application architectures that use multiple databases but have all objects in a single schema (typically the default dbo schema) and require cross database queries. For these types of applications, create an Aurora MySQL Instance and then create multiple databases as you would in SQL Server (using the CREATE DATABASE command).

Reference all objects using a two-part name instead of a three-part name by omitting the default schema identifier. For application code using the USE command instead of a three-part identifier, no rewrite is needed other than replacing the double dot with a single dot.

```
SELECT * FROM MyDB..MyTable -> SELECT * FROM MyDB.MyTable
```

For applications using a single database and multiple schemas, the migration path is the same and requires fewer rewrites because two-part names are already being used.

Applications that use multiple schemas and multiple databases will need to use multiple instances.

Use the SHOW DATABASES command to View databases (schemas) in Aurora MySQL.

```
SHOW DATABASES;
```

```
database
---------------
information_schema
Demo
mysql
performance_schema
sys
```

Aurora MySQL also supports a CREATE DATABASE syntax reminder command.

```
SHOW CREATE DATABASE Demo;
```

```
Database      Create Database
--------      ---------------
Demo          CREATE DATABASE `Demo` /*!40100 DEFAULT CHARACTER SET latin1 */
```

Examples

Create a new table in a new database.

```
CREATE DATABASE NewDatabase;
```

```
USE NewDatabase;
```

```
CREATE TABLE NewTable
(
NewColumn VARCHAR(20) NOT NULL PRIMARY KEY
);
```

```
INSERT INTO NewTable VALUES('NewValue');
```

```
SELECT * FROM NewTable;
```

Summary

The following table summarizes the migration path for each architecture.

Current Object Architecture	Migrate to Aurora MySQL	Rewrites
Single database, all objects in dbo schema	Single instance, single database/schema	If the code already uses two-part object notation such as dbo.<object>, consider creating a dbo schema in Aurora MySQL to minimize code changes.
Single database, objects in multiple schemas	Single instance, multiple databases/schemas	No identifier hierarchy rewrites needed. Code should be compatible with respect to the object hierarchy.
Multiple databases, all objects in dbo schema	Single instance, multiple databases/schemas	Identifier rewrite is required to remove the SQL Server schema name or the default dot. Change SELECT * FROM MyDB..MyTable to SELECT * FROM MyDB.MyTable.
Multiple databases, objects in multiple schemas	Multiple instances	Connectivity between the instances will need to be implemented at the application level

For more information, see https://dev.mysql.com/doc/refman/5.7/en/create-database.html

Migrate from SQL Server Transactions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Transactions	• Default isolation level REPEATABLE READ • Default mechanism CONSISTENT SNAPSHOT is similar to SQL Server's READ COMMITTED SNAPSHOT isolation • Syntax and option differences

Overview

A Transaction is a unit of work performed against a database and typically represents a change in the database. Transactions serve the following purposes:

- Provide units of work that enable recovery from logical or physical system failures while keeping the database in a consistent state.
- Provide units of work that enable recovery from failures while keeping a database in a consistent state when a logical or physical system failure occurs.
- Provide isolation between users and programs accessing a database concurrently.

Transactions are an "all-or-nothing" unit of work. Each transactional unit of work must either complete, or it must rollback all data changes. Also, transacions must be isolated from other transactions. The results of the "view of data" for each transaction must conform to the defined database isolation level.

Database transactions must comply with ACID properties:

- **Atomic:** Transactions are "all or nothing". If any part of the transaction fails, the entire transaction fails and the database remains unchanged.

  Note: There are exceptions to this rule. For example, some constraint violations, per ANSI definitions, should not cause a transaction rollback.

- **Consistent:** All transactions must bring the database from one valid state to another valid state. Data must be valid according to all defined rules, constraints, triggers, etc.
- **Isolation:** Concurrent execution of transactions must result in a system state that would occur if transactions were executed sequentially.

  Note: There are several exceptions to this rule based on the lenience of the required isolation level.

- **Durable:** After a transaction commits successfully and is acknowledged to the client, the engine must guarantee that its changes are persisted even in the event of power loss, system crashes, or any other errors.

> **Note:** By default, SQL Server uses the "auto commit" (also known as "implicit transactions") mode set to ON. Every statement is treated as a transaction on its own unless a transaction was explicitly defined. This behavior different than other engines like Oracle where, by default, every DML requires an explicit COMMIT statement to be persisted.

Syntax

Simplified syntax for the commands defining transaction boundaries:

Define the beginning of a transaction.

```
BEGIN  TRAN | TRANSACTION  [<transaction name>]
```

Committing work and the end of a transaction.

```
COMMIT WORK | [ TRAN | TRANSACTION [<transaction name>]]
```

Rollback work at the end of a transaction.

```
ROLLBACK WORK | [ TRAN | TRANSACTION [<transaction name>]]
```

SQL Server supports the standard ANSI isolation levels defined by the ANSI/ISO SQL standard (SQL92):

> **Note:** Each level provides a different approach for managing the concurrent execution of transactions. The main purpose of a transaction isolation levels is to manage the visibility of changed data as seen by other running transactions. Additionally, when concurrent transactions access the same data, the level of transaction isolation affects the way they interact with each other.

- **Read Uncommitted:** A current transaction can see uncommitted data from other transactions. If a transaction performs rollback, all data is restored to its previous state.
- **Read committed:** A transaction only sees data changes that were committed. Therefore, dirty reads are not possible. However, after issuing a commit, it would be visible to the current transaction (while it's still in a running state).
- **Repeatable read:** A transaction sees data changes made by the other transactions only after both transactions issue a commit or are rolled back.
- **Serializable:** This isolation level is the strictest because it does not permit transaction overwrites of another transactions' actions. Concurrent execution of a set of serializable transactions is guaranteed to produce the same effect as running them sequentially in the same order.

The main difference between isolation levels is the phenomena they prevent from appearing. The three preventable phenomena are:

- **Dirty Reads:** A transaction can read data written by another transaction but not yet committed.
- **Non-Repeatable (fuzzy) Reads:** When reading the same data several times, a transaction can find the data has been modified by another transaction that has just committed. The same query executed twice can return different values for the same rows.

- **Phantom (ghost) Reads:** Similar to a non-repeatable read, but it is related to new data created by another transaction. The same query executed twice can return different numbers of records.

The following table summarizes the four ANSI/ISO SQL standard (SQL92) isolation levels and indicates which phenomena are allowed (√) or disallowed (X).

Transaction Isolation Level	Dirty Reads	Non Repeatable Reads	Phantom Reads
Read Uncommitted	√	√	√
Read Committed	X	√	√
Repeatable Read	X	X	√
Serializable	X	X	X

There are two common implementations for transaction isolation:

- **Pessimistic Isolation (Locking):** Resources accessed by a transaction are locked for the duration of the transaction. Depending on the operation, resource, and transaction isolation level, other transactions can "see" changes made by the locking transaction, or they must wait for it to complete. With this mechanism, there is only one copy of the data for all transactions, which minimizes memory and disk resource consumption at the expense of transaction lock waits.
- **Optimistic Isolation (MVCC):** Every transaction owns a set of the versions of the resources (typically rows) that it accessed. In this mode, transactions don't have to wait for one another at the expense of increased memory and disk utilization. In this isolation mechanism, there is a chance that conflicts will arise when transactions attempt to commit. In case of a conflict, the application needs to be able to handle the rollback, and attempt a retry.

SQL Server implements both mechanisms and they can be used concurrently.

For Optimistic Isolation, SQL Server introduced two additional isolation levels: Read Committed Snapshot and Snapshot. For more details see the links at end of this section.

Set the transaction isolation level using SET command. It affects the current execution scope only.

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ
| SNAPSHOT | SERIALIZABLE }
```

Examples

Execute two DML statements within a serializable transaction.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
INSERT INTO Table1
VALUES (1, 'A');
UPDATE Table2
 SET Column1 = 'Done'
```

```
WHERE KeyColumn = 1;
COMMIT TRANSACTION;
```

For more information, see https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels and https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql

Migrate to Aurora MySQL Transactions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Transactions	• Default isolation level REPEATABLE READ • Default mechanism CONSISTENT SNAPSHOT is similar to SQL Server's READ COMMITTED SNAPSHOT isolation • Syntax and option differences

Overview

Aurora MySQL supports the four transaction isolation levels specified in the SQL:92 standard: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE.

The default isolation level for Aurora MySQL is REPEATABLE READ.

Syntax

Simplified syntax for setting transaction boundaries:

```
SET [SESSION] TRANSACTION ISOLATION LEVEL [READ WRITE | READ ONLY] | REPEATABLE READ |
READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE]
```

> **Note:** Setting a GLOBAL isolation level is not supported in Aurora MySQL. Only session scope can be changed; similar to SQL Server SET scope. The default behavior of transactions is to use REPEATABLE READ and consistent reads. Applications designed to run with READ COMMITTED may need to be modified. Alternatively, they can explicitly change the default to READ COMMITTED.

In Aurora MySQL, a Transaction Intent can be optionally specified. Setting a transaction to READ ONLY disables the transaction's ability to modify or lock both transactional and non-transactional tables visible to other transactions, but the transaction can still modify or lock temporary tables. It also enables internal optimization to improve performance and concurrency. The default is READ WRITE.

Simplified syntax for the commands defining transaction boundaries:

```
START TRANSACTION WITH CONSISTENT SNAPSHOT | READ WRITE | READ ONLY
```

Or

```
BEGIN [WORK]
```

The WITH CONSISTENT SNAPSHOT option starts a consistent read transaction. The effect is the same as issuing a START TRANSACTION followed by a SELECT from any table. WITH CONSISTENT SNAPSHOT does not change the transaction isolation level.

A consistent read uses snapshot information to make query results available based on a point in time regardless of modifications performed by concurrent transactions. If queried data has been changed by another transaction, the original data is reconstructed using the undo log. Consistent reads avoid locking issues that may reduce concurrency. With the REPEATABLE READ isolation level, the snapshot is based on the time the first read operation is performed. With the READ COMMITTED isolation level, the snapshot is reset to the time of each consistent read operation.

End a transaction and commit the work.

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

End a transaction and rollback the work.

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

The AND CHAIN clause causes a new transaction to begin as soon as the current one ends using the same isolation level and access mode as the just-terminated transaction. The RELEASE clause causes the server to disconnect the current session after terminating the current transaction. Including the NO keyword suppresses both CHAIN and RELEASE completion. AND CHAIN can be useful if the completion_type system variable is set to cause chaining or release completion.

Aurora MySQL supports both auto commit and explicit commit modes. You can change mode using the "autocommit" system variable.

```
SET autocommit = {0 | 1}
```

Examples

Execute two DML statements within a serializable transaction.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
INSERT INTO Table1
VALUES (1, 'A');
UPDATE Table2
 SET Column1 = 'Done'
WHERE KeyColumn = 1;
COMMIT;
```

Summary

The following table summarizes the key differences in transaction support and syntax when migrating from SQL Server to Aurora MySQL.

Transaction Property	SQL Server	Aurora MySQL	Comments
Default isolation level	READ COMMITTED	REPEATABLE READ	The Aurora MySQL default isolation level is stricter than SQL

Transaction Property	SQL Server	Aurora MySQL	Comments
			Server. Evaluate application needs and set appropriately.
initialize transaction syntax	BEGIN TRAN|TRANSACTION	START TRANSACTION	Code rewrite is required from BEGIN to START. If using the shorthand TRAN, rewrite to TRANSACTION.
Default isolation mechanism	Pessimistic lock based	Lock based for writes, consistent read for SELECTs	The Aurora MySQL default mode is similar to SQL Server's READ COMMITTED SNAPSHOT isolation.
Commit transaction	COMMIT [WORK|TRAN|TRANSACTION]	COMMIT [WORK]	If using only COMMIT or COMMIT WORK, no change is needed. Otherwise, rewrite TRAN and TRANSACTION to WORK.
Rollback transaction	ROLLBACK [WORK |[TRAN | TRANSACTION]	ROLLBACK [WORK]	If using only ROLLBACK or ROLLBACK WORK, no change is needed. Otherwise, rewrite TRAN and TRANSACTION to WORK.
Set autocommit off/on	SET IMPLICIT_TRANSACTIONS OFF | ON	SET autocommit = 0 | 1	See Session Options.
ANSI Isolation	REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE	REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE	Compatible syntax.

Transaction Property	SQL Server	Aurora MySQL	Comments
MVCC	SNAPSHOT and READ COMMITTED SNAPSHOT	WITH CONSISTENT SNAPSHOT	Aurora MySQL consistent read in READ COMMITTED isolation is similar to SQL Server READ COMMITTED SNAPSHOT.
Nested transactions	Supported, view level with @@trancount	Not Supported	Starting a new transaction in Aurora MySQL while another transaction is active causes a COMMIT of the previous transaction.
Transaction Chaining	Not Supported	Causes a new transaction to open immediately upon transaction completion	
Transaction Release	Not supported	Causes the client session to disconnect upon transaction completion	

For more information, see https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

Migrate from SQL Server DELETE and UPDATE FROM

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Rewrite to use sub-queries

Overview

SQL Server supports an extension to the ANSI standard that allows using an additional FROM clause in UPDATE and DELETE statements.

This additional FROM clause can be used to limit the number of modified rows by joining the table being updated, or deleted from, to one or more other tables. This functionality is similar to using a WHERE clause with a derived table subquery. For UPDATE, you can use this syntax to set multiple column values simultaneously without repeating the subquery for every column.

However, these statements can introduce logical inconsistencies if a row in an updated table is matched to more than one row in a joined table. The current implementation chooses an arbitrary value from the set of potential values and is non deterministic.

Syntax

```
UPDATE <Table Name>
SET <Column Name> = <Expression> ,...
FROM <Table Source>
WHERE <Filter Predicate>;
```

```
DELETE FROM <Table Name>
FROM <Table Source>
WHERE <Filter Predicate>;
```

Examples

Delete customers with no orders.

```
CREATE TABLE Customers
(
Customer VARCHAR(20) PRIMARY KEY
);
```

```
INSERT INTO Customers
VALUES
('John'),
('Jim'),
('Jack')
```

```
CREATE TABLE Orders
(
OrderID INT NOT NULL PRIMARY KEY,
Customer VARCHAR(20) NOT NULL,
OrderDate DATE NOT NULL
);
```

```
INSERT INTO Orders (OrderID, Customer, OrderDate)
VALUES
(1, 'Jim', '20180401'),
(2, 'Jack', '20180402');
```

```
DELETE FROM Customers
FROM   Customers AS C
       LEFT OUTER JOIN
       Orders AS O
       ON O.Customer = C.Customer
WHERE  O.OrderID IS NULL;
```

```
SELECT *
FROM Customers;
```

```
Customer
--------
Jim
Jack
```

Update multiple columns in `Orders` based on the values in `OrderCorrections`.

```
CREATE TABLE OrderCorrections
(
OrderID INT NOT NULL PRIMARY KEY,
Customer VARCHAR(20) NOT NULL,
OrderDate DATE NOT NULL
);
```

```
INSERT INTO OrderCorrections
VALUES (1, 'Jack', '20180324');
```

```
UPDATE O
SET Customer = OC.Customer,
       OrderDate = OC.OrderDate
FROM   Orders AS O
       INNER JOIN
       OrderCorrections AS OC
       ON O.OrderID = OD.OrderID;
```

```
SELECT *
FROM Orders;
```

```
Customer   OrderDate
---------  ---------
Jack       2018-03-24
Jack       2018-04-02
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/t-sql/queries/update-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/delete-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/queries/from-transact-sql*

Migrate to Aurora MySQL DELETE and UPDATE FROM

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Rewrite to use sub-queries

Overview

Aurora MySQL does not support DELETE and UPDATE FROM syntax.

Migration Considerations

You can easily rewrite the DELETE and UPDATE FROM statements as subqueries.

For DELETE, place the subqueries in the WHERE clause.
For UPDATE, place the subqueries either in the WHERE or SET clause.

> **Note:** When rewriting UPDATE FROM queries, include a WHERE clause to limit which rows are updated even if the SQL Server version (where the rows were limited by the join condition) did not have one .

For DELETE statements, the workaround is simple and, in most cases, easier to read and understand.

For UPDATE statements, the workaround involves repeating the correlated subquery for each column being set.
Although this approach makes the code longer and harder to read, it does solve the logical challenges associated with updates having multiple matched rows in the joined tables.

In the current implementation, the SQL Server engine silently chooses an arbitrary value if more than one value exists for the same row.
When you rewrite the statement to use a correlated sub query, like the example below, if more than one value is returned from the sub query, a SQL error will be raised:

```
SQL Error [1242] [21000]: Subquery returns more than 1 row
```

Consult the documentation for the Aurora MySQL UPDATE statement as there are significant processing differences from SQL Server.
For example:

- In Aurora MySQL, you can update multiple tables in a single UPDATE statement.
- UPDATE expressions are evaluated in order from left to right. This behavior differs from SQL Server and the ANSI standard, which require an "all at once" evaluation.

For example, in the statement `UPDATE Table SET Col1 = Col1 + 1, Col2 = Col1`, `Col2` is set to the new value of `Col1`. The end result is `Col1 = Col2`.

Examples

Delete customers with no orders.

```
CREATE TABLE Customers
(
Customer VARCHAR(20) PRIMARY KEY
);

INSERT INTO Customers
VALUES
('John'),
('Jim'),
('Jack')

CREATE TABLE Orders
(
OrderID INT NOT NULL PRIMARY KEY,
Customer VARCHAR(20) NOT NULL,
OrderDate DATE NOT NULL
);

INSERT INTO Orders (OrderID, Customer, OrderDate)
VALUES
(1, 'Jim', '20180401'),
(2, 'Jack', '20180402');

DELETE FROM Customers
WHERE Customer NOT IN (
                       SELECT Customer
                       FROM Orders
                      );
```

```
SELECT *
FROM Customers;
```

```
Customer
--------
Jim
Jack
```

Update multiple columns in `Orders` based on the values in `OrderCorrections`.

```
CREATE TABLE OrderCorrections
(
OrderID INT NOT NULL PRIMARY KEY,
Customer VARCHAR(20) NOT NULL,
OrderDate DATE NOT NULL
);
```

```
INSERT INTO OrderCorrections
VALUES (1, 'Jack', '20180324');
```

```
UPDATE Orders
SET Customer  = (
```

```
                  SELECT Customer
                  FROM OrderCorrections AS OC
                  WHERE Orders.OrderID = OC.OrderID
                 ),
    OrderDate = (
                  SELECT OrderDate
                  FROM OrderCorrections AS OC
                  WHERE Orders.OrderID = OC.OrderID
                 )
WHERE OrderID IN (
                   SELECT OrderID
                   FROM OrderCorrections
);
```

```
SELECT *
FROM Orders;
```

```
Customer   OrderDate
---------  ---------
Jack       2018-03-24
Jack       2018-04-02
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
Join as part of DELETE	DELETE FROM ... FROM	N/A	Rewrite to use WHERE clause with a subquery.
Join as part of UPDATE	UPDATE ... FROM	N/A	Rewrite to use correlated subquery in the SET clause and add WHERE clause to limit updates set.

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/update.html*
- *https://dev.mysql.com/doc/refman/5.7/en/delete.html*

Migrate from SQL Server Stored Procedures

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Stored Procedures	• No support for Table Valued Parameters • Syntax and option differences

Stored Procedures are encapsulated, persisted code modules you can execute using the EXECUTE T-SQL statement. They may have multiple input (IN) and output (OUT) parameters. Table valued user defined types can be used as input parameters. IN is the default direction for parameters, but OUT must be explicitly specified. You can specify parameters as both IN and OUT.

SQL Server allows you to run stored procedures in any security context using the EXECUTE AS option. They can be explicitly recompiled for every execution using the RECOMPILE option and can be encrypted in the database using the ENCRYPTION option to prevent unauthorized access to the source code.

SQL Server provides a unique feature that allows you to use a stored procedure as an input to an INSERT statement. When using this feature, only the first row in the data set returned by the stored procedure is evaluated.

As part of the stored procedure syntax, SQL Server supports a default output integer parameter that can be specified along with the RETURN command, i.e RETURN -1.
It is typically used to signal status or error to the calling scope, which can use the syntax EXEC @Parameter = <Stored Procedure Name> to retrieve the RETURN value, without explicitly stating it as part of the parameter list.

Syntax

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } <Procedure Name>
[<Parameter List>
[ WITH [ ENCRYPTION ]|[ RECOMPILE ]|[ EXECUTE AS ...]]
AS {
[ BEGIN ]
 <SQL Code Body>
 [RETURN [<Integer Value>]]
[ END ] }[;]
```

Examples

Creating and Executing a Stored Procedure

Create a simple parameterized Stored Procedure to validate the basic format of an Email.

```
CREATE PROCEDURE ValidateEmail
@Email VARCHAR(128), @IsValid BIT = 0 OUT
AS
```

```
BEGIN
IF @Email LIKE N'%@%'
 SET @IsValid = 1
ELSE
 SET @IsValid = 0
RETURN
END;
```

Execute the procedure.

```
DECLARE @IsValid BIT
EXECUTE [ValidateEmail]
 @Email = 'X@y.com', @IsValid = @IsValid OUT;
SELECT @IsValid;

-- Returns 1
```

```
EXECUTE [ValidateEmail]
 @Email = 'Xy.com', @IsValid = @IsValid OUT;
SELECT @IsValid;

-- Returns 0
```

Create a stored procedure that uses RETURN to pass the application an error value.

```
CREATE PROCEDURE ProcessImportBatch
@BatchID INT
AS
BEGIN
 BEGIN TRY
  EXECUTE Step1 @BatchID
  EXECUTE Step2 @BatchID
  EXECUTE Step3 @BatchID
 END TRY
 BEGIN CATCH
  IF ERROR_NUMBER() = 235
   RETURN -1 -- indicate special condition
  ELSE
   THROW -- handle error normally
 END CATCH
END
```

Using a Table-Valued Input Parameter

Create and populate an `OrderItems` table.

```
CREATE TABLE OrderItems(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
```

```
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Washer', 100);
```

Create a tabled valued type for the `OrderItem` table valued parameter.

```
CREATE TYPE OrderItems
AS TABLE
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create a procedure to process order items.

```
CREATE PROCEDURE InsertOrderItems
@OrderItems AS OrderItems READONLY
AS
BEGIN
    INSERT INTO OrderItems(OrderID, Item, Quantity)
    SELECT OrderID,
           Item,
           Quantity
    FROM @OrderItems
END;
```

Instantiate and populate the table valued variable and pass the data set to the stored procedure.

```
DECLARE @OrderItems AS OrderItems;

INSERT INTO @OrderItems ([OrderID], [Item], [Quantity])
VALUES
(1, 'M8 Bolt', 100),
(1, 'M8 Nut', 100),
(1, M8 Washer, 200);

EXECUTE [InsertOrderItems]
 @OrderItems = @OrderItems;

(3 rows affected)
   Item                Quantity
   ------------        ------------
 1      M8 Bolt          100
 2      M8 Nut           100
 3      M8 Washer        200
```

INSERT... EXEC Syntax

```
INSERT INTO <MyTable>
EXECUTE <MyStoredProcedure>;
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql

Migrate to Aurora MySQL Stored Procedures

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Stored Procedures	• No support for Table Valued Parameters • Syntax and option differences

Overview

Aurora MySQL Stored Procedures provide similar functionality to SQL Server stored procedures.

As with SQL Server, Aurora MySQL supports security execution context. It also supports input, output, and bi-directional parameters.

Stored procedures are typically used for:

- **Code reuse:** Stored procedures offer a convenient code encapsulation and reuse mechanism for multiple applications, potentially written in various languages, requiring the same database operations.
- **Security management:** By allowing access to base tables only through stored procedures, administrators can manage auditing and access permissions. This approach minimizes dependencies between application code and database code. Administrators can use stored procedures to process business rules and to perform auditing and logging.
- **Performance improvements:** Full SQL query text does not need to be transferred from the client to the database.

Stored procedures, triggers, and user defined functions in Aurora MySQL are collectively referred to as *Stored Routines*. When binary logging is enabled, MySQL SUPER privilege is required to run stored routines. However, you can run stored routines with binary logging enabled without SUPER privilege by setting the*log_bin_trust_function_creators* parameter to *true* for the DB parameter group for your MySQL instance.

Aurora MySQL permits stored routines to contain control flow, DML, DDL, and transaction management statements including START TRANSACTION, COMMIT and ROLLBACK.

Syntax

```
CREATE [DEFINER = { user | CURRENT_USER }] PROCEDURE sp_name
([ IN | OUT | INOUT ] <Parameter> <Parameter Data Type> ... )
COMMENT 'string' |
LANGUAGE SQL |
[NOT] DETERMINISTIC |
{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } |
SQL SECURITY { DEFINER | INVOKER }
```

```
<Stored Procedure Code Body>
```

Examples

Replace RETURN value parameter with standard OUTPUT parameters.

```
CREATE PROCEDURE ProcessImportBatch()
IN @BatchID INT, OUT @ErrorNumber INT
BEGIN
  CALL Step1 (@BatchID)
  CALL Step2 (@BatchID)
  CALL Step3 (@BatchID)
IF error_count > 1
   SET @ErrorNumber = -1 -- indicate special condition
END
```

Use a LOOP Cursor with a source table to replace table valued parameters:

Create an `OrderItems` table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create and populate `SourceTable` as a temporary data store for incoming rows.

```
CREATE TABLE SourceTable
(
OrderID INT,
Item VARCHAR(20),
Quantity SMALLINT,
PRIMARY KEY (OrderID, Item)
);
```

```
INSERT INTO SourceTable (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Create a procedure to loop through all rows in `SourceTable` and insert them into the `OrderItems` table.

```
CREATE PROCEDURE LoopItems()
BEGIN
 DECLARE done INT DEFAULT FALSE;
 DECLARE var_OrderID INT;
 DECLARE var_Item VARCHAR(20);
 DECLARE var_Quantity SMALLINT;
 DECLARE ItemCursor CURSOR
```

```
  FOR SELECT OrderID,
             Item,
             Quantity
      FROM SourceTable;
 DECLARE CONTINUE HANDLER
  FOR NOT FOUND SET done = TRUE;
 OPEN ItemCursor;
 CursorStart: LOOP
 FETCH NEXT FROM ItemCursor
  INTO var_OrderID, var_Item, var_Quantity;
 IF Done THEN LEAVE CursorStart;
 END IF;
   INSERT INTO OrderItems (OrderID, Item, Quantity)
   VALUES (var_OrderID, var_Item, var_Quantity);
 END LOOP;
 CLOSE ItemCursor;
END;
```

Call the stored procedure.

```
CALL LoopItems();
```

Select all rows from the `OrderItems` table.

```
SELECT * FROM OrderItems;
```

```
OrderID     Item        Quantity
-------     ----        --------
1           M8 Bolt     100
2           M8 Nut      100
3           M8 Washer   200
```

Summary

The following table summarizes the differences between MySQL Stored Procedures and SQL Server Stored Procedures.

	SQL Server	Aurora MySQL	Workaround
General CREATE Syntax differences	CREATE PROC|PROCEDURE <Procedure Name> @Parameter1 <Type>, ...n AS <Body>	CREATE PROCEDURE <Procedure Name> (Parameter1 <Type>,...n) <Body>	Rewrite stored procedure creation scripts to use PROCEDURE instead of PROC. Rewrite stored procedure creation scripts to omit the AS keyword. Rewrite stored procedure parameters to not use the @ symbol in parameter names. Add parentheses around the parameter declaration.

	SQL Server	Aurora MySQL	Workaround
			Rewrite stored procedure parameter direction OUTPUT to OUT or INOUT for bidirectional parameters. IN is the for both MySQL and SQL Server.
Security Context	{ EXEC | EXECUTE } AS { CALLER | SELF | OWNER | 'user_name' }	DEFINER = 'user' | CURRENT_USER in conjunction with SQL SECURITY { DEFINER | INVOKER }	For Stored procedures that use an explicit user name, rewrite the code from EXECUTE AS 'user' to DEFINER = 'user' and SQL SECURITY DEFINER. For Stored Procedures that use the CALLER option, rewrite the code to include SQL SECURITY INVOKER. For Stored procedures that use the SELF option, rewrite the code to DEFINER = CURRENT_USER and SQL SECURITY DEFINER. Unlike SQL Server, OWNERs can not be specified and must be explicitly named.
Encryption	Use WITH ENCRYPTION option	Not supported in Aurora MySQL	
Parameter direction	IN and OUT|OUTPUT, by default OUT can be used as IN as well.	IN, OUT, and INOUT	Although the functionality of these parameters is the same for SQL Server and MySQL, you must rewrite the code for syntax compliance: Use OUT instead of OUTPUT USE INOUT instead of OUT for bidirectional parameters
Recompile	Use WITH RECOMPILE option	Not supported in Aurora MySQL	
Table Valued Parameters	Use declared table type user defined parameters	Not supported in Aurora MySQL	See the example above for a workaround.
INSERT...	Use the output of the	Not supported in Aur-	Use tables to hold the data or

	SQL Server	Aurora MySQL	Workaround
EXEC	stored procedure as input to an INSERT statement	ora MySQL	pass string parameters formatted as CSV, XML, JSON (or any other convenient format) and then parse the parameters before the INSERT statement.
Additional restrictions	Use BULK INSERT to load data from text file	The LOAD DATA statement is not allowed in stored procedures	
RETURN Value	RETURN <Integer Value>	Not supported	Use a standard OUTPUT parameter instead.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/faqs-stored-procs.html and https://dev.mysql.com/doc/refman/5.7/en/create-procedure.html

Migrate from SQL Server Error Handling

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Error Handling	• Different paradigm and syntax requires rewrite of error handling code

Overview

SQL Server error handling capabilities have significantly improved throughout the years. However, previous features are retained for backward compatibility.
Before SQL Server 2008, only very basic error handling features were available. RAISERROR was the primary statement used for error handling.

Since SQL 2008, SQL Server has added extensive ".Net like" error handling capabilities including TRY/CATCH blocks, THROW statements, the FORMATMESSAGE function, and a set of system functions that return metadata for the current error condition.

TRY/CATCH Blocks

TRY/CATCH blocks implement error handling similar to Microsoft Visual C# and Microsoft Visual C++. TRY ... END TRY statement blocks can contain T-SQL statements .

If an error is raised by any of the statements within the TRY ... END TRY block, execution stops and is moved to the nearest set of statements that are bounded by a CATCH ... END CATCH block.

Syntax

```
BEGIN TRY
<Set of SQL Statements>
END TRY
BEGIN CATCH
<Set of SQL Error Handling Statements>
END CATCH
```

Examples

(See the examples in Error Handling.)

THROW

The THROW statement raises an exception and transfers execution of the TRY ... END TRY block of statements to the associated CATCH ... END CATCH block of statements.

Throw accepts either constant literals or variables for all parameters.

Syntax

```
THROW [Error Number>, <Error Message>, < Error State>] [;]
```

Examples

Use TRY/CATCH error blocks to handle key violations.

```
CREATE TABLE ErrorTest (Col1 INT NOT NULL PRIMARY KEY);
```

```
BEGIN TRY
 BEGIN TRANSACTION
   INSERT INTO ErrorTest(Col1) VALUES(1);
   INSERT INTO ErrorTest(Col1) VALUES(2);
   INSERT INTO ErrorTest(Col1) VALUES(1);
 COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  THROW; -- Throw with no parameters = RETHROW
END CATCH;
```

```
(1 row affected)
(1 row affected)
(0 rows affected)
Msg 2627, Level 14, State 1, Line 7
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE54D8676973'.
Cannot insert duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

> **Note:** Contrary to what many SQL developers believe, the values 1 and 2 are indeed inserted into ErrorTestTable in the above example. This behavior is in accordance with ANSI specifications stating that a constraint violation should not roll back an entire transaction.

Use THROW with variables

```
BEGIN TRY
BEGIN TRANSACTION
INSERT INTO ErrorTest(Col1) VALUES(1);
INSERT INTO ErrorTest(Col1) VALUES(2);
INSERT INTO ErrorTest(Col1) VALUES(1);
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
DECLARE @CustomMessage VARCHAR(1000),
               @CustomError INT,
               @CustomState INT;
SET @CustomMessage = 'My Custom Text ' + ERROR_MESSAGE();
SET @CustomError = 54321;
SET @CustomState = 1;
THROW @CustomError, @CustomMessage, @CustomState;
END CATCH;
```

```
(0 rows affected)
Msg 54321, Level 16, State 1, Line 19
```

```
My Custom Text Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE545CBDBB9A'.
Cannot insert duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

RAISERROR

The RAISERROR statement is used to explicitly raise an error message, similar to THROW. It causes an error state for the executing session and forwards execution to either the calling scope or, if the error occurred within a TRY ... END TRY block, to the associated CATCH ... END CATCH block. RAISERROR can reference a user-defined message stored in the sys.messages system table or can be used with dynamic message text.

The key differences between THROW and RAISERROR are:

- Message IDs passed to RAISERROR must exist in the sys.messages system table. The error number parameter passed to THROW does not.
- RAISERROR message text may contain printf formatting styles. The message text of THROW may not.
- RAISERROR uses the severity parameter for the error returned. For THROW, severity is always 16.

Syntax

```
RAISERROR (<Message ID>|<Message Text> ,<Message Severity> ,<Message State>
[WITH option [<Option List>]])
```

Examples

Raise a custom error.

```
RAISERROR (N'This is a custom error message with severity 10 and state 1.', 10, 1)
```

FORMATMESSAGE

FORMATMESSAGE returns a sting message consisting of an existing error message in the sys.messages system table, or from a text string, using the optional parameter list replacements. The FORMATMESSAGE statement is similar to the RAISERROR statement.

Syntax

```
FORMATMESSAGE (<Message Number> | <Message String>, <Parameter List>)
```

Error State Functions

SQL Server provides the following error state functions:

- ERROR_LINE
- ERROR_MESSAGE
- ERROR_NUMBER
- ERROR_PROCEDURE

- ERROR_SEVERITY
- ERROR_STATE
- @@ERROR

Examples

Use Error State Functions within a CATCH block.

```
CREATE TABLE ErrorTest (Col1 INT NOT NULL PRIMARY KEY);
```

```
BEGIN TRY;
 BEGIN TRANSACTION;
   INSERT INTO ErrorTest(Col1) VALUES(1);
   INSERT INTO ErrorTest(Col1) VALUES(2);
   INSERT INTO ErrorTest(Col1) VALUES(1);
 COMMIT TRANSACTION;
END TRY
BEGIN CATCH
  SELECT        ERROR_LINE(),
                ERROR_MESSAGE(),
                ERROR_NUMBER(),
                ERROR_PROCEDURE(),
                ERROR_SEVERITY(),
                ERROR_STATE(),
                @@Error;
THROW;
END CATCH;
```

```
6
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE543C8912D8'. Cannot insert
duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
2627
NULL
14
1
2627
```

```
(1 row affected)
(1 row affected)
(0 rows affected)
(1 row affected)
Msg 2627, Level 14, State 1, Line 25
Violation of PRIMARY KEY constraint 'PK__ErrorTes__A259EE543C8912D8'. Cannot insert
duplicate key in object 'dbo.ErrorTest'. The duplicate key value is (1).
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/t-sql/language-elements/raiserror-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/language-elements/try-catch-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/language-elements/throw-transact-sql*

Migrate to Aurora MySQL Error Handling

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Error Handling	• Different paradigm and syntax requires rewrite of error handling code

Overview

Aurora MySQL offers a rich error handling framework with a different paradigm than SQL Server. The Aurora MySQL terminology is:

- **CONDITION:** The equivalent of an ERROR in SQL Server.
- **HANDLER:** An object that can handle conditions and perform actions.
- **DIAGNOSTICS:** The meta data about the CONDITION.
- **SIGNAL and RESIGNAL:** Statements similar to THROW and RAISERROR in SQL Server.

Errors in Aurora MySQL are identified by the follow items:

- A numeric error code specific to MySQL and, therefore, is not compatible with other database systems.
- A five character SQLSTATE value that uses the ANSI SQL and ODBC standard error conditions.

  Note: Not every MySQL error number has a corresponding SQLSTATE value. For errors that don't have a corresponding SQLSTATE, the general 'HY000' error is used.
- A textual message string that describes the nature of the error.

DECLARE ... CONDITION

The DECLARE ... CONDITION statement declares a named error condition and associates the name with a condition that requires handling. This declared name can then be referenced in subsequent DECLARE ... HANDLER statements.

Syntax

```
DECLARE <Condition Name> CONDITION
FOR <Condition Value>
```

```
<Condition Value> = <MySQL Error Code> | <SQLSTATE [VALUE] <SQLState Value>
```

Examples

Declare a condition for MySQL error 1051 (Unknown table error).

```
DECLARE TableDoesNotExist CONDITION FOR 1051;
```

Declare a condition for SQL State 42S02 (Base table or view not found) .

> **Note:** This SQLState error corresponds to the MySQL Error 1051.

```
DECLARE TableDoesNotExist CONDITION FOR SQLSTATE VALUE '42S02';
```

DECLARE ... HANDLER

A HANDLER object defines the actions or statements to be executed when a CONDITION arises. The handler object may be used to CONTINUE or EXIT execution.

The condition may be a previously defined condition using the DECLARE ... CONDITION statement or an explicit condition for one of the following items:

- An Explicit Aurora MySQL error code. For example 1051, which represents "Unknown Table Error".
- An Explicit SQLSTATE value. For example '42S02'.
- Any SQLWARNING event representing any SQLSTATE with a '01' prefix.
- Any NOTFOUND event representing any SQLSTATE with a '02' prefix. This condition is relevant for cursors. For more information, see Cursors.
- Any SQLEXCEPTION event, representing any SQLSTATE without a '00', '01', or '02' prefix. These conditions are considered exception errors.

  > **Note:** SQLSTATE events with a '00' prefix are not errors; they are used to represent successful execution of statements.

Syntax

```
DECLARE {CONTINUE | EXIT | UNDO}
HANDLER FOR
 <MySQL Error Code> |
 <SQLSTATE [VALUE] <SQLState Value> |
 <Condition Name> |
 SQLWARNING |
 NOT FOUND |
 SQLEXCEPTION
<Statement Block>
```

Examples

Declare a handler to ignore warning messages and continue execution by assigning an empty statement block.

```
DECLARE CONTINUE HANDLER
FOR SQLWARNING BEGIN END
```

Declare a handler to EXIT upon duplicate key violation and log a message to a table.

```
DECLARE EXIT HANDLER
FOR SQLSTATE '23000'
BEGIN
 INSERT INTO MyErrorLogTable
```

```
  VALUES(NOW(), CURRENT_USER(), 'Error 23000')
END
```

GET DIAGNOSTICS

Each execution of an SQL statement produces diagnostic information that is stored in the diagnostics area. The GET DIAGNOSTICS statement enables users to retrieve and inspect this information.

> **Note:** Aurora MySQL also supports the SHOW WARNINGS and SHOW ERRORS statements to retrieve conditions and errors.

The GET DIAGNOSTICS statement is typically used in the handler code within a stored routine. GET CURRENT DIAGNOSTICS is permitted outside the context of a handler to check the execution result of an SQL statement.

The keyword CURRENT causes retrieval of the current diagnostics area. The keyword STACKED causes retrieval of the information from the second diagnostics area. The second diagnostic area is only available if the current context is within a code block of a condition handler. The default is CURRENT.

Syntax

```
GET [CURRENT | STACKED] DIAGNOSTICS
<@Parameter = NUMBER | ROW_COUNT>
|
CONDITION <Condition Number> <@Parameter = CLASS_ORIGIN | SUBCLASS_ORIGIN | RETURNED_
SQLSTATE | MESSAGE_TEXT | MYSQL_ERRNO | CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA |
CONSTRAINT_NAME | CATALOG_NAME | SCHEMA_NAME | TABLE_NAME | COLUMN_NAME | CURSOR_NAME>
```

Examples

Retrieve SQLSTATE and MESSAGE_TEXT from the diagnostic area for the last statement executed.

```
GET DIAGNOSTICS CONDITION 1 @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT
```

SIGNAL/RESIGNAL

The SIGNAL statement is used to raise an explicit condition or error. It can be used to provide full error information to a handle, to an outer scope of execution, or to the SQL client. The SIGNAL statement enables explicitly defining the error's properties such as error number, SQLSTATE value, message, etc.

The difference between SIGNAL and RESIGNAL is that RESIGNAL is used to pass on the error condition information available during execution of a condition handler within a compound statement inside a stored routine or an event. RESIGNAL can be used to change none, some, or all the related condition information before passing it for processing in the next calling scope of the stack.

> **Note:** It is not possible to issue SIGNAL statements using variables.

Syntax

```
SIGNAL | RESIGNAL <SQLSTATE [VALUE] sqlstate_value | <Condition Name>
[SET <Condition Information Item Name> = <Value> [,...n]]
```

```
<Condition Information Item Name> = CLASS_ORIGIN | SUBCLASS_ORIGIN | RETURNED_SQLSTATE
| MESSAGE_TEXT | MYSQL_ERRNO | CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA | CONSTRAINT_
NAME | CATALOG_NAME | SCHEMA_NAME | TABLE_NAME | COLUMN_NAME | CURSOR_NAME
```

Examples

Raise an explicit error with SQLSTATE '55555'.

```
SIGNAL SQLSTATE '55555'
```

Re-raise an error with an explicit MySQL error number.

```
RESIGNAL SET MYSQL_ERRNO = 5
```

Migration Considerations

> **Note:** Error handling is a critical aspect of any software solution. Code migrated from one paradigm to another should be carefully evaluated and tested.

The basic operations of raising, processing, responding, and obtaining metadata is similar in nature for most relational database management systems. The technical aspects of rewriting the code to use different types of objects is not difficult.

In SQL Server, there can only be one "handler", or CATCH code block, that handles exceptions for a given statement. In Aurora MySQL, multiple handler objects can be declared. A condition may trigger more than one handler. Be sure the correct handlers are executed as expected, especially when there are multiple handlers. The following sections provides rules to help establish your requirements.

Handler Scope

A handler can be specific or general. Specific handlers are handlers defined for a specific MySQL error code, SQLSTATE, or a condition name. Therefore, only one type of event will trigger a specific handler. General handlers are handlers defined for conditions in the SQLWARNING, SQLEXCEPTION, or NOT FOUND classes. More than one event may trigger the handler.

A handler is in scope for the block in which it is declared. It cannot be triggered by conditions occurring outside the block boundaries.

A handler declared in a BEGIN ... END block is in scope for the SQL statements that follow the handler declaration.

One or more handlers may be declared in different or the same scopes using different specifications. For example, a specific MySQL error code handler may be defined in an outer code block while a more general SQLWARNING handler is defined within an inner code block. Specific MySQL error code handlers and a general SQLWARNING class handler may exist within the same code block.

Handler Choice

Only one handler is triggered for a single event. Aurora MySQL decides which handler should be triggered. The decision regarding which handler should be triggered as a response to a condition

depends on the handler's scope and value. It also depends on whether or not other handlers are present that may be more appropriate to handle the event.

When a condition occurs in a stored routine, the server searches for valid handlers in the current BEGIN ... END block scope. If none are found, the engine searches for handlers in each successive containing BEGIN ... END code block scope. When the server finds one or more applicable handlers at any given scope, the choice of which one to trigger is based on the following condition precedence:

- A MySQL error code handler takes precedence over a SQLSTATE value handler.
- An SQLSTATE value handler takes precedence over general SQLWARNING, SQLEXCEPTION, or NOT FOUND handlers.
- An SQLEXCEPTION handler takes precedence over an SQLWARNING handler.

Multiple applicable handlers with the same precedence may exist for a condition. For example, a statement could generate several warnings having different error codes. There may exist a specific MySQL Error handler for each. In such cases, the choice is non-deterministic. Different handlers may be triggered at different times depending on the circumstances.

Summary

The following identifies similarities, differences, and key migration considerations.

SQL Server Error Handling Feature	Migrate to Aurora MySQL	Comments
TRY ... END TRY and CATCH ... END CATCH blocks	Nested BEGIN ... END code blocks with per-scope handlers	DECLARE specific event handlers for each BEGIN-END code block. Note that unlike CATCH blocks, the handlers must be defined first, not later. Review the handler scope and handler choice sections above.
THROW and RAISERROR	SIGNAL and RESIGNAL	Review the handler scope and handler choice sections above.
THROW with varibles	Not supported	
FORMATMESSAGE	N/A	
Error state functions	GET DIAGNOSTIC	
Proprietary error messages in sys.messages system table	Proprietary MySQL error codes and SQLSTATE ANSI and ODBC standard	When rewriting error handling code, consider switching to the more standard SQLSTATE error codes.
Deterministic rules regarding	May be non-deterministic if	Review the handler choice section above.

SQL Server Error Handling Feature	Migrate to Aurora MySQL	Comments
condition handler execution - always the next code block in statement order	multiple handlers have the same precedence and scope	

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/error-messages-server.html*
- *https://dev.mysql.com/doc/refman/5.7/en/diagnostics-area.html*
- *https://dev.mysql.com/doc/refman/5.7/en/condition-handling.html*
- *https://dev.mysql.com/doc/refman/5.7/en/handler-scope.html*

Migrate from SQL Server Flow Control

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Flow Control	• Syntax and option differences, similar functionality

Overview

Although SQL is a mostly declarative language, it does support flow control commands, which provide run time dynamic changes in script execution paths.

> **Note:** Before SQL/PSM was introduced in SQL:1999, the ANSI standard did not include flow control constructs. Therefore, there are significant syntax differences among RDBMS engines.

SQL Server provides the following flow control keywords.

- **BEGIN... END:** Define boundaries for a block of commands that are executed together.
- **RETURN:** Exit a server code module (stored procedure, function, etc.) and return control to the calling scope. RETURN <value> can be used to return an INT value to the calling scope.
- **BREAK:** Exit WHILE loop execution.
- **THROW:** Raise errors and potentially return control to the calling stack.
- **CONTINUE:** Restart a WHILE loop.
- **TRY... CATCH:** Error handling (see Error Handling).
- **GOTO Label:** Moves the execution point to the location of the specified label.
- **WAITFOR:** Delay.
- **IF... ELSE:** Conditional flow control.
- **WHILE <condition>:** Continue looping while <condition> returns TRUE.

  > **Note:** WHILE loops are commonly used with cursors and use the system variable @@FETCH_STATUS to determine when to exit (see the Cursors section for more details).

For more information about TRY-CATCH and THROW, see Error Handling.

Examples

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
```

```
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

WAITFOR

Use WAITFOR to introduce a one minute delay between background batches purging old data.

```
SET ROWCOUNT 1000;
WHILE @@ROWCOUNT > 0;
BEGIN;
 DELETE FROM OrderItems
 WHERE OrderDate < '19900101';
 WAITFOR DELAY '00:01:00';
END;
```

GOTO

Use GOTO to skip a code section based on an input parameter in a stored procedure.

```
CREATE PROCEDURE ProcessOrderItems
@OrderID INT, @Item VARCHAR(20), @Quantity INT, @UpdateInventory BIT
AS
BEGIN
  INSERT INTO OrderItems (OrderID, Item, Quantity)
  SELECT @OrderID, @item, @Quantity
 IF @UpdateInventory = 0
  GOTO Finish
 UPDATE Inventory
 SET Stock = Stock - @Quantity
 WHERE Item = @Item
 /* Additional Inventory Processing */
finish:
/* Generate Results Log*/
END
```

Dynamic Procedure Execution Path

The following example demonstrates a solution for executing different processes based on the number of items in an order.

Declare a cursor for looping through all OrderItems and calculating the total quantity per order.

```
DECLARE  OrderItemCursor CURSOR FAST_FORWARD
FOR
SELECT  OrderID,
        SUM(Quantity) AS NumItems
FROM    OrderItems
GROUP BY OrderID
ORDER BY OrderID;
```

```
DECLARE @OrderID INT, @NumItems INT;

-- Instantiate the cursor and loop through all orders.
OPEN OrderItemCursor;

FETCH NEXT FROM OrderItemCursor
INTO @OrderID, @NumItems

WHILE @@Fetch_Status = 0
BEGIN;

 IF     @NumItems > 100
  PRINT 'EXECUTING LogLargeOrder - '
  + CAST(@OrderID AS VARCHAR(5))
  + ' ' + CAST(@NumItems AS VARCHAR(5));
 ELSE
  PRINT 'EXECUTING LogSmallOrder - '
  + CAST(@OrderID AS VARCHAR(5))
  + ' ' + CAST(@NumItems AS VARCHAR(5));

FETCH NEXT FROM OrderItemCursor
INTO @OrderID, @NumItems;
END;

-- Close and deallocate the cursor.
CLOSE OrderItemCursor;
DEALLOCATE OrderItemCursor;
```

The above code displays the following results:

```
EXECUTING LogSmallOrder - 1 100
EXECUTING LogSmallOrder - 2 100
EXECUTING LogLargeOrder - 3 200
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow

Migrate to Aurora MySQL Flow Control

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Flow Control	• Syntax and option differences, similar functionality

Overview

Aurora MySQL provides the following flow control constructs:

- **BEGIN... END:** Define boundaries for a block of commands that are executed together.
- **CASE:** Execute a set of commands based on a predicate (not to be confused with CASE expressions).
- **IF... ELSE:** Conditional flow control.
- **ITERATE:** Restart a LOOP, REPEAT, and WHILE statement.
- **LEAVE:** Exit a server code module (stored procedure, function etc.) and return control to the calling scope.
- **LOOP:** Loop indefinitely.
- **REPEAT... UNTIL:** Loop until the predicate is true.
- **RETURN:** Terminate execution of the current scope and return to the calling scope.
- **WHILE:** Continue looping while the condition returns TRUE.
- **SLEEP:** Pause execution for a specified number of seconds.

Examples

Create and populate an OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Rewrite of SQL Server WAITFOR Delay Using SLEEP

```
CREATE PROCEDURE P()
BEGIN
       DECLARE RR INT;
       SET RR =        (
                       SELECT COUNT(*)
                       FROM OrderItems
                       WHERE OrderDate < '19900101'
                       );
       WHILE RR > 0 DO
       DELETE FROM OrderItems
       WHERE OrderDate < '19900101';
        DO SLEEP (60);
       SET RR =        (
                       SELECT COUNT(*)
                       FROM OrderItems
                       WHERE OrderDate < '19900101'
                       );
       END WHILE;
END;
```

Rewrite of SQL Server GOTO Using Nested Blocks

```
CREATE PROCEDURE ProcessOrderItems
(Var_OrderID INT, Var_Item VARCHAR(20), Var_Quantity INT, UpdateInventory BIT)
BEGIN
  INSERT INTO OrderItems (OrderID, Item, Quantity)
  VALUES(Var_OrderID, Var_Item, Var_Quantity)
 IF @UpdateInventory = 1
 BEGIN
  UPDATE Inventory
  SET Stock = Stock - @Quantity
  WHERE Item = @Item
  /* Additional Inventory Processing...*/
 END
/* Generate Results Log */
END
```

Dynamic Procedure Execution Path

The following example demonstrates a solution for executing different logic based on the number of items in an order.

This example provides the same functionality as the example for SQL Server flow control. However, unlike the SQL Server example executed as a batch script, Aurora MySQL variables can only be used in stored routines (procedures and functions).

Create a procedure to declare a cursor and loop through the order items.

```
CREATE PROCEDURE P()
BEGIN
 DECLARE done INT DEFAULT FALSE;
 DECLARE var_OrderID INT;
 DECLARE var_NumItems INT;
```

```
 DECLARE OrderItemCursor CURSOR FOR
 SELECT OrderID,
        SUM(Quantity) AS NumItems
 FROM   OrderItems
 GROUP BY OrderID
 ORDER BY OrderID;

 DECLARE CONTINUE HANDLER
 FOR NOT FOUND SET done = TRUE;

 OPEN OrderItemCursor;

 CursorStart: LOOP
 FETCH NEXT FROM OrderItemCursor
  INTO var_OrderID, var_NumItems;
 IF done
    THEN LEAVE CursorStart;
 END IF;
 IF var_NumItems > 100
    THEN SELECT CONCAT('EXECUTING LogLargeOrder - ', CAST(var_OrderID AS VARCHAR(5)),
' Num Items: ', CAST(var_ NumItems AS VARCHAR(5)))
    ELSE SELECT CONCAT('EXECUTING LogSmallOrder - ', CAST(var_OrderID AS VARCHAR(5)),
' Num Items: ', CAST(var_NumItems AS VARCHAR(5)))
 END IF;

END LOOP;

CLOSE OrderItemCursor;

END;
```

Summary

While there are some syntax differences between SQL Server and Aurora MySQL flow control statements, most rewrites should be straightforward. The following table summarizes the differences and identifies how to modify T-SQL code to support similar functionality in Aurora MySQL.

	SQL Server	Aurora MySQL	Workaround
BEGIN… END	Define command block boundaries	Define command block boundaries	Compatible
RETURN	Exit the current scope and return to caller Supported for both scripts and stored code (pro-	Exit a stored function and return to caller	For Aurora MySQL, RETURN is valid only in stored (user defined) functions. It is not used in stored procedures, triggers, or events. Rewrite the T-SQL code using the LEAVE keyword.

	SQL Server	Aurora MySQL	Workaround
	cedures and functions).		The RETURN statement can return a value in both products. However, LEAVE does not support return parameters. Rewrite the code to use output parameters. You cannot use RETURN in Aurora MySQL for scripts that are not part of a stored routine.
BREAK	Exit WHILE loop execution	Not supported	Rewrite the logic to explicitly set a value that will render the WHILE condition FALSE. For example, WHILE a<100 AND control = 1. Explicitly SET control = 0, and use ITERATE to return to the begining of the loop.
THROW	Raise errors and potentially return control to the calling stack	Errors are handled by HANDLER objects.	See Error Handling for more details.
TRY - CATCH	Error handling	Errors are handled by HANDLER objects.	See Error Handling for more details.
GOTO	Move execution to specified label	Not supported	Consider rewriting the flow logic using either CASE statements or nested stored procedures. You can use nested stored procedures to circumvent this limitation by separating code sections and encapsulating them in sub-procedures. Use IF <condition> CALL <stored procedure> in place of GOTO.
WAITFOR	Delay	Not supported	Replace WAITFOR with Aurora MySQL SLEEP. SLEEP is less flexible than WAITFOR and only supports delays specified in seconds. Rewrite the code using SLEEP to replace WAITFOR DELAY and convert the units to seconds. WAITFOR TIME is not supported in

	SQL Server	Aurora MySQL	Workaround
			Aurora MySQL. You can calculate the difference in seconds between the desired time and current time using Date and Time functions and use the result to dynamically generate the SLEEP statement. Alternatively, consider using CREATE EVENT with a predefined schedule.
IF... ELSE	Conditional flow control	Conditional flow control	The functionality is compatible, but the syntax differs. SQL Server uses IF <condition> <statement> ELSE <statement>. Aurora MySQL uses IF <condition> THEN <statement> ELSE <statement> ENDIF. Rewrite T-SQL code to add the mandatory THEN and ENDIF keywords.
WHILE	Continue execution while condition is TRUE	Continue execution while condition is TRUE	The Functionality is compatible, but the syntax differs. SQL Server uses WHILE <condition> BEGIN...END, Aurora MySQL uses WHILE <condition> DO... END WHILE. Autora MySQL does not require A BEGIN...END block. Rewrite T-SQL code to use the Aurora MySQL keywords.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/flow-control-statements.html

Migrate from SQL Server Full-Text Search

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Full Text	• Syntax and option differences, less comprehensive but simpler • Most common basic functionality is similar • Requires rewrite of administration logic and queries

Overview

SQL Server supports an optional framework for executing Full-Text search queries against character-based data in SQL Server tables using an integrated, in-process Full-Text engine, and a filter daemon host process (fdhost.exe).

To run Full-Text queries, a Full-Text catalog must first be created, which in turn may contain one or more Full-Text indexes. A Full-Text index is comprised of one or more textual columns of a table.

Full-text queries perform smart linguistic searches against Full-Text indexes by identifying words and phrases based on specific language rules. The searches can be for simple words, complex phrases, or multiple forms of a word or a phrase. They can return ranking scores for matches (also known as "hits").

Full-Text Indexes

A Full-Text index can be created on one of more columns of a table or view for any of the following data types:

- **CHAR:** Fixed size ASCII string column data type
- **VARCHAR:** Variable size ASCII string column data type
- **NCHAR:** Fixed size UNICODE string column data type
- **NVARCHAR:** Variable size UNICODE string column data type
- **TEXT:** ASCII BLOB string column data type (deprecated)
- **NTEXT:** UNICODE BLOB string column data type (deprecated)
- **IMAGE:** Binary BLOB data type (deprecated)
- XML: XML structured BLOB data type
- **VARBINARY(MAX):** Binary BLOB data type
- **FILESTREAM:** File based storage data type

  Note: For more information about data types, see Data Types.

Full-text indexes are created using the CREATE FULLTEXT INDEX statement. A Full-Text index may contain up to 1024 columns from a single table or view.

When creating Full-Text indexes on BINARY type columns, documents such as Microsoft Word can be stored as a binary stream and parsed correctly by the Full-Text engine.

Full-Text catalogs

Full-text indexes are contained within Full-Text catalog objects. A Full-Text catalog is a logical container for one or more Full-Text indexes and can be used to collectively administer them as a group for tasks such as back-up, restore, refresh content, etc.

Full-text catalogs are creates using the `CREATE FULLTEXT CATALOG` statement. A Full-Text catalog may contain zero or more Full-Text indexes and is limited in scope to a single database.

Full-text queries

After a Full-Text catalog and index have been create and populated, users can perform Full-Text queries against these indexes to query for:

- Simple term match for one or more words or phrases
- Prefix term match for words that begin with a set of characters
- Generational term match for inflectional forms of a word
- Proximity term match for words or phrases which are close to another word or phrase
- Thesaurus search for synonymous forms of a word
- Weighted term match for finding words or phrases with weighted proximity values

Full-text queries are integrated into T-SQL, and use the following predicates and functions:

- CONTAINS predicate
- FREETEXT predicate
- CONTAINSTABLE table valued function
- FREETEXTTABLE table valued function

> **Note:** Do not confuse Full-Text functionality with the LIKE predicate, which is used for pattern matching only.

Updating Full-Text Indexes

By default, Full-Text indexes are automatically updated when the underlying data is modified, similar to a normal B-Tree or Columnstore index. However, large changes to the underlying data may inflict a performance impact for the Full-Text indexes update because it is a resource intensive operation. In these cases, you can disable the automatic update of the catalog and update it manually, or on a schedule, to keep the catalog up to date with the underlying tables.

> **Note:** You can monitor the status of Full-Text catalog by using the FULLTEXTCATALOGPROPERTY(<Full-text Catalog Name>, 'Populatestatus') function.

Examples

Create a ProductReviews table.

```
CREATE TABLE ProductReviews
(
ReviewID INT NOT NULL
 IDENTITY(1,1),
CONSTRAINT PK_ProductReviews PRIMARY KEY(ReviewID),
ProductID INT NOT NULL
 /*REFERENCES Products(ProductID)*/,
ReviewText VARCHAR(4000) NOT NULL,
ReviewDate DATE NOT NULL,
UserID INT NOT NULL
/*REFERENCES Users(UserID)*/
);
```

```
INSERT INTO ProductReviews
( ProductID, ReviewText, ReviewDate, UserID)
VALUES
(1, 'This is a review for product 1, it is excellent and works as expected',
'20180701', 2),
(1, 'This is a review for product 1, it is not that great and failed after two days',
'20180702', 2),
(2, 'This is a review for product 3, it has exceeded my expectations. A+++',
'20180710', 2);
```

Create a Full-Text catalog for product reviews.

```
CREATE FULLTEXT CATALOG ProductFTCatalog;
```

Create a Full-Text index for ProductReviews.

```
CREATE FULLTEXT INDEX
ON ProductReviews (ReviewText)
KEY INDEX PK_ProductReviews
ON ProductFTCatalog;
```

Query the Full-Text index for reviews containing the word 'excellent'.

```
SELECT *
FROM ProductReviews
WHERE CONTAINS(ReviewText, 'excellent');
```

```
ReviewID  ProductID     ReviewText
                                                    ReviewDate      UserID
--------  ---------     ----------
                                                    ----------      ------
1         1             This is a review for product 1, it is excellent and works as expected
2018-07-01      2
```

For more information, see https://docs.microsoft.com/en-us/sql/2014/relational-databases/search/Full-Text-search

Migrate to Aurora MySQL Full-Text Search

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Full Text	• Syntax and option differences, less comprehensive but simpler • Most common basic functionality is similar • Requires rewrite of administration logic and queries

Overview

Aurora MySQL supports all the native full-text capabilities of MySQL InnoDB Full-Text Indexes. Full-text indexes are used to speed up textual searches performed against textual data by using the Full-Text MATCH ... AGAINST predicate.

Full-text indexes can be created on any textual column of the following types:

- **CHAR:** Fixed length string data type
- **VARCHAR:** Variable length string data type
- **TEXT:** String BLOB data type

Full-text indexes can be created as part of the CREATE TABLE, ALTER TABLE, and CREATE INDEX statements.

Full-text indexes in Aurora MySQL use an inverted index design where a list of individual words is stored alongside a list of documents where the words were found. Proximity search is also supported by storing a byte offset position for each word.

Creating a full-text index in Aurora MySQL creates a set of index system tables that can be viewed using the INFORMATION_SCHEMA.INNODB_SYS_TABLES view. These tables include the auxiliary index tables representing the inverted index and a set of management tables that help facilitate management of the indexes such as deletes and sync with the underlying data, caching, configuration, and syncing processes.

Full-Text Index Cache

The index cache temporarily caches index entries for recent rows to minimize the contention associated with inserting documents. These inserts, even small ones, typically result in many singleton insertions to the auxiliary tables, which may prove to be challenging in terms of concurrency. Caching and batch flushing help minimize these frequent updates. In addition, batching also helps alleviate the overhead involved with multiple auxiliary table insertions for words and minimizes duplicate entries as insertions are merged and written to disk as a single entry.

Full-Text Index Document ID and FTS_DOC_ID Column

Aurora MySQL assigns a document identifier that maps words in the index to the document rows where those words are found. This warrants a schema change to the source table, namely adding an indicator column to point to the associated document. This column, known as FTS_DOC_ID must exist in the table where the Full-Text index is created. If the column is not present, Aurora MySQL adds it when the Full-Text index is created.

> **Note:** Adding a column to a table in Aurora MySQL triggers a full rebuild of the table that may be resource intensive for larger tables (a warning is issued).

Executing a SHOW WARNINGS statement after creating a Full-Text index on a table that does not have this column generates a warning. For example:

```
CREATE TABLE TestFT
(
KeyColumn INT AUTO_INCREMENT NOT NULL
 PRIMARY KEY,
TextColumn TEXT(200)
);
```

```
CREATE FULLTEXT INDEX FTIndex1
ON TestFT(TextColumn);
```

```
SHOW WARNINGS;
```

```
Level   Code  Message
-----   ----  -------
Warning   124  InnoDB rebuilding table to add column FTS_DOC_ID
```

If the Full-Text index is created as part of the CREATE TABLE statement, the FTS_DOC_ID column is added silently and no warning is issued. It is recommended to create the FTS_DOC_ID column for tables where full-text indexes will be created as part of the CREATE TABLE statement to avoid an expensive rebuild of a table that is already loaded with large amounts of data. Creating the FTS_DOC_ID column as an AUTO_INCREMENT column may improve performance of data loading.

> **Note:** Dropping a Full-Text index from a table does not drop the FTS_DOC_ID column.

Full-Text Index Deletes

Similar to the insert issue described earlier, deleting rows from a table with a Full-Text index may also result in concurrency challenges due to multiple singleton deletions from the auxiliary tables.

To minimize the impact of this issue, Aurora MySQL logs the deletion of a document ID (DOC_ID) in a dedicated internal system table named FTS_*_DELETED instead of actually deleting it from the auxiliary tables. The existence of a DOC_ID in the DELETED table is a type of soft-delete. The engine consults it to determine if a row that had a match in the auxiliary tables should be discarded, or if it should be returned to the client. This approach makes deletes much faster at the expense of somewhat larger index size.

Note: Soft deleted documents are not automatically managed. You must issue an OPTIMIZE TABLE statement and the innodb_optimize_fulltext_only=ON option to rebuild the Full-Text index.

Transaction Control

Due to the caching and batch processing properties of the Full-Text indexes, UPDATE and INSERT to a Full-Text index are committed when a transaction commits. Full-text search can only access committed data.

Full-Text Search Functions

To query Full-Text indexes, use the MATCH... AGAINST predicate. The MATCH clause accepts a list of column names, separated by commas, that define the column names of the columns that have a Full-Text index defined and need to be searched. In the AGAINST clause, define the string you want searched. It also accepts an optional modifier that indicates the type of search to perform.

MATCH... AGAINST Syntax

```
MATCH (<Column List>)
AGAINST (
<String Expression>
[ IN NATURAL LANGUAGE MODE
  | IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION
  | IN BOOLEAN MODE
  | WITH QUERY EXPANSION]
)
```

Note: The search expression must be constant for all rows searched. Therefore a table column is not permitted.

The three types of full-text searches are Natural Language, Boolean, and Query Expansion.

Natural Language Search

If no modifier is provided, or the `IN NATURAL LANGUAGE MODE` modifier is explicitly provided, the search string is interpreted as natural human language phrase. For this type of search, the stop-word list is considered and stop words are excluded. For each row, the search returns a "relevance" value, which denotes the similarity of the search string to the text, for the row, in all the columns listed in the MATCH column list. For more information regarding stop-words, see https://dev.mysql.-com/doc/refman/5.7/en/fulltext-stopwords.html.

Boolean Search

The IN BOOLEAN MODE modifier specifies a Boolean search. When using Boolean search, some characters imply special meaning either at the beginning or the end of the words that make up the search string. The + and - operators are used to indicate that a word must be present (+) or absent (-) for the match to resolve to TRUE.

For example, the following statement returns rows for which the ReviewText column contains the word 'Excellent', but not the work 'England'.

```
SELECT *
FROM ProductReviews
WHERE MATCH (ReviewText) AGAINST ('+Excellent -England' IN BOOLEAN MODE);
```

Additional Boolean operators include:

- The @distance operator tests if two or more words start within a specified distance, or the number of words between them.
- The < and > operators change a word's contribution to the relevance value assigned for a specific row match.
- Parentheses () are used to group words into sub-expressions and may be nested.
- The Tilde ~ is used as negative operator, resulting in the word's contribution to be deducted from the total relevance value. Use this operator to mark "noise" words that are rated lower, but not excluded, as with the - operator
- The asterisk * operator is used as a wildcard operator and is appended to the word.
- Double quotes " are used for exact, literal phrase matching.

For more information on Boolean searches, see https://dev.mysql.com/doc/refman/5.7/en/fulltext-boolean.html

Query Expansion

The WITH QUERY EXPANSION or IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION is useful when a search phrase is too short, which may indicate that the user is looking for "implied knowledge" that the Full-Text engine doesn't have.
For example, a user that searches for 'Car' may need to match specific car brands such as 'Ford', 'Toyota', 'Mercedes-Benz', and others.

Blind query expansions, also known as "Automatic Relevance Feedback", performs the searches twice. On the first pass, the engine looks for the most relevant documents. It then performs a second pass using the original search phrase concatenated with the results of the first pass. For example, if the search was looking for 'Cars' and the most relevant documents included the word 'Ford', the seconds search would find the documents that also mention 'Ford'.

For more information on query expansion, see https://dev.mysql.com/doc/refman/5.7/en/fulltext-query-expansion.html

Migration Considerations

Migrating Full-Text indexes from SQL Server to Aurora MySQL requires a full rewrite of the code that deals with both creating, management, and querying of Full-Text searches.

Although the Aurora MySQL full-text engine is significantly less comprehensive than SQL Server, it is also much simpler to create and manage and is sufficiently powerful for most common, basic full-text requirements.

For more complex full-text workloads, Amazon RDS offers CloudSearch, a managed service in the AWS Cloud that makes it simple and cost-effective to set up, manage, and scale an enterprise grade search solution. Amazon CloudSearch supports 34 languages and advanced search features such as high-lighting, autocomplete, and geospatial search.

Currently, there is no direct tooling integration with Aurora MySQL and, therefore, you must create a custom application to synchronize the data between RDS instances and the CloudSearch Service.

For more information on CloudSearch, see https://aws.amazon.com/cloudsearch/

Examples

```
CREATE TABLE ProductReviews
(
ReviewID INT
 AUTO_INCREMENT NOT NULL
 PRIMARY KEY,
ProductID INT NOT NULL
 /*REFERENCES Products(ProductID)*/,
ReviewText TEXT(4000) NOT NULL,
ReviewDate DATE NOT NULL,
UserID INT NOT NULL
/*REFERENCES Users(UserID)*/
);
```

```
INSERT INTO ProductReviews
(ProductID, ReviewText, ReviewDate, UserID)
VALUES
(1, 'This is a review for product 1, it is excellent and works as expected',
'20180701', 2),
(1, 'This is a review for product 1, it is not that great and failed after two days',
'20180702', 2),
(2, 'This is a review for product 3, it has exceeded my expectations. A+++',
'20180710', 2);
```

Query the full-text index for reviews containing the word 'excellent'.

```
SELECT *
FROM ProductReviews
WHERE MATCH (ReviewText) AGAINST ('Excellent' IN NATURAL LANGUAGE MODE);
```

For more information, see https://dev.mysql.com/doc/refman/5.7/en/innodb-fulltext-index.html

Migrate from SQL Server JSON and XML

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - JSON and XML	• Minimal XML support, extensive JSON support • No XQUERY support, optionally convert to JSON

Overview

Java Script Object Notation (JSON) and eXtensible Markup Language (XML) are the two most common types of semi-structured data documents used by a variety of data interfaces and NoSQL databases. Most REST web service APIs support JSON as their native data transfer format. XML is an older, more mature framework still widely used. It also provides many extensions such as XQuery, name spaces, schemas, and more.

The following example is a JSON document:

```
[{
    "name": "Robert",
    "age": "28"
}, {
    "name": "James",
    "age": "71"
    "lastname": "Drapers"
}]
```

It's XML counterpart is:

```
<?xml version="1.0" encoding="UTF-16" ?>
<root>
        <Person>
                <name>Robert</name>
                <age>28</age>
       </Person>
       <Person>
                <name>James</name>
                <age>71</age>
                <lastname>Drapers</lastname>
       </Person>
</root>
```

SQL Server provides native support for both XML and JSON in the database using the familiar and convenient T-SQL interface.

XML Data

SQL Server provides extensive native support for working with XML data including XML Data Types, XML Columns, XML Indexes, and XQuery.

XML Data Types and Columns

XML data can be stored using the following data types:

- The **Native XML Data Type** uses a BLOB structure but preserves the XML Infoset, which consists of the containment hierarchy, document order, and element/attribute values. An XML typed document may differ from the original text; white space is removed and the order of objects may change. XML Data stored as a native XML data type has the additional benefit of schema validation.
- An **Annotated Schema** (AXSD) can be used to distribute XML documents to one or more tables. Hierarchical structure is maintained, but element order is not.
- **CLOB or BLOB** such as VARCHAR(MAX) and VARBINARY(MAX) can be used to store the original XML document.

XML Indexes

SQL Server allows creation of PRIMARY and SECONDARY XML indexes on columns with a native XML data type. Secondary indexes can be created for PATH, VALUE, or PROPERTY, which are helpful for various types of workload queries.

XQuery

SQL Server supports a sub set of the W3C XQUERY language specification. It allows executing queries directly against XML data and using them as expressions or sets in standard T-SQL statements.

For example:

```
DECLARE @XMLVar XML = '<Root><Data>My XML Data</Data></Root>';
SELECT @XMLVar.query('/Root/Data');
```

```
Result: <Data>My XML Data</Data>
```

JSON Data

SQL Server does not support a dedicated JSON data type. However, you can store JSON documents in an NVARCHAR column. For more information about BLOBS, see Data Types.

SQL Server provides a set of JSON functions that can be used for the following tasks:

- Retrieve and modify values in JSON documents.
- Convert JSON objects to a set (table) format.
- Use standard T-SQL queries with converted JSON objects.
- Convert tabular results of T-SQL queries to JSON format.

The functions are:

- **ISJSON**tests whether a string contains a valid JSON string. Use in WHERE clause to avoid errors.
- **JSON_VALUE** retrieves a scalar value from a JSON document.
- **JSON_QUERY** retrieves a whole object or array from a JSON document.
- **JSON_MODIFY** modifies values in a JSON document.
- **OPENJSON** converts a JSON document to a SET that can be used in the FROM clause of a T-SQL query.

The FOR JSON clause of SELECT queries can be used to convert a tabular set to a JSON document.

Examples

Create a table with a native typed XML column.

```
CREATE TABLE MyTable
(
XMLIdentifier INT NOT NULL PRIMARY KEY,
XMLDocument XML NULL
);
```

Query a JSON document.

```
DECLARE @JSONVar NVARCHAR(MAX);
SET @JSONVar = '{"Data":{"Person":[{"Name":"John"},{"Name":"Jane"},
{"Name":"Maria"}]}}';
SELECT JSON_QUERY(@JSONVar, '$.Data');
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server*
- *https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server*

Migrate to Aurora MySQL JSON and XML

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - JSON and XML	• Minimal XML support, extensive JSON support • No XQUERY support, optionally convert to JSON

Overview

Aurora MySQL support for unstructured data is the opposite of SQL server.

There is minimal support for XML, but a native JSON data type and more than 25 dedicated JSON functions.

XML Support

Aurora MySQL supports two XML functions: ExtractValue and UpdateXML.

ExtractValue accepts an XML document, or fragment, and an XPATH expression. The function returns the character data of the child (or element) matched by the XPATH expression. If there is more than one match, the function returns the content of child nodes as a space delimited character string. ExtractValue returns only CDATA (it does not return tags) and does not return sub-tags contained within a matching tag or its content.

For example:

```
SELECT ExtractValue('<Root><Person>John</Person><Person>Jim</Person></Root>',
'/Root/Person');
```

```
Results: John Jim
```

UpdateXML is used to replace an XML fragment with another fragment using XPATH expressions similar to ExtractValue. If a match is found, it returns the new, updated XML. If there are no matches, or multiple matches, the original XML is returned.

For example:

```
SELECT UpdateXML('<Root><Person>John</Person><Person>Jim</Person></Root>', '/Root',
'<Person>Jack</Person>')
```

```
Results: <Person>Jack</Person>
```

> **Note:** Aurora MySQL does not support MySQL LOAD XML syntax .
> For more information about loading data into Aurora MySQL directly from Amazon S3, see https://-

doc-
s.aws.amazon.-
com/AmazonRDS/latest/UserGuide/AuroraMySQL.Integrating.LoadFromS3.html

JSON Data Type

Aurura MySQL 5.7 supports a native JSON data type for storing JSON documents, which provides several benefits over storing the same document as a generic string. The first major benefit is that all JSON documents stored as a JSON data type are validated for correctness. If the document is not valid JSON, it is rejected and an error condition is raised.

In addition, more efficient storage algorithms enable optimized read access to elements within the document. The optimized internal binary representation of the document enables much faster operation on the data without requiring expensive re-parsing.

For example:

```
CREATE TABLE JSONTable (DocumentIdentifier INT NOT NULL PRIMARY KEY, JSONDocument
JSON);
```

JSON Functions

Aurora MySQL supports a rich set of more than 25 targeted functions for working with JSON data. These functions enable adding, modifying, and searching JSON data. Additionally, spatial JSON functions can be used for GeoJSON documents.
For more information, see https://dev.mysql.com/doc/refman/5.7/en/spatial-geojson-functions.html.

The JSON_ARRAY, JSON_OBJECT, and JSON_QUOTE functions all return a JSON document from a list of values, a list of key-value pairs, or a JSON value respectively.
For example:

```
SELECT JSON_OBJECT('Person', 'John', 'Country', 'USA');
```

```
{"Person": "John", "Country": "USA"}
```

The JSON_CONTAINS, JSON_CONTAINS_PATH, JSON_EXTRACT, JSON_KEYS, and JSON_SEARCH functions are used to query and search the content of a JSON document.
The CONTAINS functions are Boolean functions that return 1 or 0 (TRUE or FALSE). JSON_EXTRACT returns a subset of the document based on the XPATH expression.
JSON_KEYS returns a JSON array consisting of the top-level key (or path top level) values of a JSON document. The JSON_SEARCH function returns the path to one or all of the instances of the search string.

For example:

```
SELECT JSON_EXTRACT('["Mary", "Paul", ["Jim", "Ryan"]]', '$[1]');
```

```
"Paul"
```

```
SELECT JSON_SEARCH('["Mary", "Paul", ["Jim", "Ryan"]]', 'one', 'Paul');
```

```
"$[1]"
```

Aurora MySQL supports the following functions for adding, deleting, and modifying JSON data, :

JSON_INSERT, JSON_REMOVE, JSON_REPLACE, and their ARRAY counterparts, which are used to create, delete, and replace existing data elements. For example:

```
SELECT JSON_ARRAY_INSERT('["Mary", "Paul", "Jim"]', '$[1]', 'Jack');
```

```
["Mary", "Jack", "Paul", "Jim"]
```

JSON_SEARCH is used to find the location of an element value within a JSON document. For example:

```
SELECT JSON_SEARCH('["Mary", "Paul", ["Jim", "Ryan"]]', 'one', 'Paul');
```

```
"$[1]"
```

JSON Indexes

JSON columns are effectively a BINARY family type, which can not be indexed.

To index JSON data, use CREATE TABLE or ALTER TABLE to add generated columns that represent some value from the JSON document and create an index on this generated column.
For more information, see Clustered and Non Clustered Indexes.

> **Note:** If indexes on generated columns exist for JSON documents, the query optimizer can use them to match JSON expressions and optimize data access.

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
XML and JSON native data types	XML with schema collections	JSON	
JSON functions	IS_JSON, JSON_VALUE, JSON_QUERY, JSON_MODFIY, OPEN_JSON, FOR JSON	A set of more than 25 dedicated JSON functions. See https://dev.mysql.com/doc/refman/5.7/en/json-function-reference.html	
XML functions	XQUERY and XPATH, OPEN_XML, FOR XML	ExtractValue and UpdateXML.	
XML and JSON Indexes	Primary and Secondary PATH, VALUE and PROPERTY	Requires adding always generated (computed and persisted) columns with JSON expressions and indexing them explicitly. The optimizer can make use of JSON expressions only.	

Feature	SQL Server	Aurora MySQL	Comments
	indexes		

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/xml-functions.html*
- *https://dev.mysql.com/doc/refman/5.7/en/json.html*
- *https://dev.mysql.com/doc/refman/5.7/en/json-functions.html*

Migrate from SQL Server MERGE

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - MERGE	• Rewrite to use REPLACE and ON DUPLICATE KEY, or individual constituent DML statements

Overview

MERGE is a complex , hybrid DML/DQL statement for performing INSERT, UPDATE, or DELETE operations on a target table based on the results of a logical join of the target table and a source data set.

MERGE can also return row sets similar to SELECT using the OUTPUT clause, which gives the calling scope access to the actual data modifications of the MERGE statement.

The MERGE statement is most efficient for non-trivial conditional DML.
For example, inserting data if a row key value does not exist and updating the existing row if the key value already exists.

You can easily manage additional logic such as deleting rows from the target that don't appear in the source. For simple, straightforward updates of data in one table based on data in another, it is typically more efficient to use simple INSERT, DELETE, and UPDATE statements. All MERGE functionality can be replicated using INSERT, DELETE, and UPDATE statements, but not necessarily less efficiently.

The SQL Server MERGE statement offers a wide range of functionality and flexibility and is compatible with ANSI standard SQL:2008. SQL Server has many extensions to MERGE that provide efficient T-SQL solutions for synchronizing data.

Syntax

```
MERGE [INTO] <Target Table> [AS] <Table Alias>]
USING <Source Table>
ON <Merge Predicate>
[WHEN MATCHED [AND <Predicate>]
THEN UPDATE SET <Column Assignments...> | DELETE]
[WHEN NOT MATCHED [BY TARGET] [AND <Predicate>]
THEN INSERT [(<Column List>)]
VALUES (<Values List>) | DEFAULT VALUES]
[WHEN NOT MATCHED BY SOURCE [AND <Predicate>]
THEN UPDATE SET <Column Assignments...> | DELETE]
OUTPUT [<Output Clause>]
```

Examples

Perform a simple one-way synchronization of two tables.

```
CREATE TABLE SourceTable
(
```

```
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE TargetTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

```
INSERT INTO SourceTable (Col1, Col2)
VALUES
(2, 'Source2'),
(3, 'Source3'),
(4, 'Source4');
```

```
INSERT INTO TargetTable (Col1, Col2)
VALUES
(1, 'Target1'),
(2, 'Target2'),
(3, 'Target3');
```

```
MERGE INTO TargetTable AS TGT
USING SourceTable AS SRC ON TGT.Col1 = SRC.Col1
WHEN MATCHED
 THEN UPDATE SET TGT.Col2 = SRC.Col2
WHEN NOT MATCHED
 THEN INSERT (Col1, Col2)
 VALUES (SRC.Col1, SRC.Col2);
```

```
SELECT * FROM TargetTable;
```

```
Col1    Col2
-----   -----
1       Target1
2       Source2
3       Source3
4       Source4
```

Perform a conditional two-way synchronization using NULL for "no change" and DELETE from the target when the data is not found in the source.

```
TRUNCATE TABLE SourceTable;
INSERT INTO SourceTable (Col1, Col2) VALUES (3, NULL), (4, 'NewSource4'), (5,
'Source5');
```

```
MERGE INTO TargetTable AS TGT
USING SourceTable AS SRC ON TGT.Col1 = SRC.Col1
WHEN MATCHED AND SRC.Col2 IS NOT NULL
 THEN UPDATE SET TGT.Col2 = SRC.Col2
WHEN NOT MATCHED
 THEN INSERT (Col1, Col2)
 VALUES (SRC.Col1, SRC.Col2)
```

```
WHEN NOT MATCHED BY SOURCE
 THEN DELETE;
```

```
SELECT *
FROM TargetTable;
```

```
Col1    Col2
-----   -----
3       Source3
4       NewSource4
5       Source5
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/merge-transact-sql

Migrate to Aurora MySQL MERGE

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - MERGE	• Rewrite to use REPLACE and ON DUPLICATE KEY, or individual constituent DML statements

Overview

Aurora MySQL does not support the MERGE statement. However, it provides two other statements for merging data: REPLACE, and INSERT... ON DUPLICATE KEY UPDATE.

REPLACE deletes a row and inserts a new row if a duplicate key conflict occurs. INSERT... ON DUPLICATE KEY UPDATE performs an in-place update. Both REPLACE and ON DUPLICATE KEY UPDATE rely on an existing primary key and unique constraints. It is not possible to define custom MATCH conditions as with SQL Server's MERGE statement.

REPLACE

REPLACE provides a function similar to INSERT. The difference is that REPLACE first deletes an existing row if a duplicate key violation for a PRIMARY KEY or UNIQUE constraint occurs.

REPLACE is a MySQL extension that is not ANSI compliant. It either performs only an INSERT when no duplicate key violations occur, or it performs a DELETE and then an INSERT if violations occur.

Syntax

```
REPLACE [INTO] <Table Name> (<Column List>)
VALUES (<Values List>)
```

```
REPLACE [INTO] <Table Name>
SET <Assignment List: ColumnName = VALUE...>
```

```
REPLACE [INTO] <Table Name> (<Column List>)
SELECT ...
```

INSERT ... ON DUPLICATE KEY UPDATE

The ON DUPLICATE KEY UPDATE clause of the INSERT statement acts as a dual DML hybrid. Similar to REPLACE, it executes the assignments in the SET clause instead of raising a duplicate key error. ON DUPLICATE KEY UPDATE is a MySQL extension that in not ANSI compliant.

Syntax

```
INSERT [INTO] <Table Name> [<Column List>]
VALUES (<Value List>
ON DUPLICATE KEY <Assignment List: ColumnName = Value...>
```

```
INSERT [INTO] <Table Name>
SET <Assignment List: ColumnName = Value...>
ON DUPLICATE KEY
 UPDATE <Assignment List: ColumnName = Value...>
```

```
INSERT [INTO] <Table Name> [<Column List>]
SELECT ...
ON DUPLICATE KEY
 UPDATE <Assignment List: ColumnName = Value...>
```

Migration Considerations

Neither REPLACE nor INSERT ... ON DUPLICATE KEY UPDATE provide a full functional replacement for SQL Server's MERGE. The key differences are:

- Key violation conditions are mandated by the primary key or unique constraints that exist on the target table. They can not be defined using an explicit predicate.
- There is no alternative for the WHEN NOT MATCHED BY SOURCE clause.
- There is no alternative for the OUTPUT clause.

The key difference between REPLACE and INSERT ON DUPLICATE KEY UPDATE is that with REPLACE, the violating row is deleted or attempted to be deleted. If foreign keys are in place, the DELETE operation may fail, which may fail the entire transaction.

For INSERT ... ON DUPLICATE KEY UPDATE, the update is performed on the existing row in place without attempting to delete it.

It should be straightforward to replace most MERGE statements with either REPLACE or INSERT... ON DUPLICATE KEY UPDATE.
Alternatively, break down the operations into their constituent INSERT, UPDATE, and DELETE statements.

Examples

Use REPLACE to create a simple one-way, two-table sync.

```
CREATE TABLE SourceTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE TargetTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

```
INSERT INTO SourceTable (Col1, Col2)
VALUES
(2, 'Source2'),
```

```
(3, 'Source3'),
(4, 'Source4');
```

```
INSERT INTO TargetTable (Col1, Col2)
VALUES
(1, 'Target1'),
(2, 'Target2'),
(3, 'Target3');
```

```
REPLACE INTO TargetTable(Col1, Col2)
SELECT Col1,
       Col2
FROM SourceTable;
```

```
SELECT *
FROM TargetTable;
```

```
Col1    Col2
-----   -----
1       Target1
2       Source2
3       Source3
4       Source4
```

Create a conditional two-way sync using NULL for "no change" and DELETE from target when not found in source.

```
TRUNCATE TABLE SourceTable;
```

```
INSERT INTO SourceTable(Col1, Col2)
VALUES
(3, NULL),
(4, 'NewSource4'),
(5, 'Source5');
```

```
DELETE FROM TargetTable
WHERE Col1 NOT IN (SELECT Col1 FROM SourceTable);
```

```
INSERT INTO TargetTable (Col1, Col2)
SELECT Col1,
       Col2
FROM SourceTable AS SRC
WHERE SRC.Col1 NOT IN (
                       SELECT Col1
                       FROM TargetTable
                      );
```

```
UPDATE TargetTable AS TGT
SET Col2 = (
            SELECT COALESCE(SRC.Col2, TGT.Col2)
            FROM SourceTable AS SRC
            WHERE SRC.Col1 = TGT.Col1
           )
WHERE TGT.Col1 IN (
```

```
                    SELECT Col1
                    FROM SourceTable
                  );
```

```
SELECT *
FROM TargetTable;
```

```
Col1     Col2
-----    -----
3        Source3
4        NewSource4
5        Source5
```

Summary

The following table describes similarities, differences, and key migration considerations.

SQL Server MERGE feature	Migrate to Aurora MySQL	Comments
Define source set in USING clause	Define source set in a SELECT query or in a table.	
Define logical duplicate key condition with an ON predicate	Duplicate key condition mandated by primary key and unique constraints on target table.	
WHEN MATCHED THEN UPDATE	REPLACE or INSERT... ON DUPLICATE KEY UPDATE	When using REPLACE, the violating row will be deleted, or attempted to be deleted. If there are foreign keys in place, the DELETE operation may fail, which may fail the entire transaction. With INSERT ... ON DUPLICATE KEY UPDATE, the updated is performed on the existing row in place, without attempting to delete it.
WHEN MATCHED THEN DELETE	DELETE FROM Target WHERE Key IN (SELECT Key FROM Source)	
WHEN NOT MATCHED THEN INSERT	REPLACE or INSERT... ON DUPLICATE KEY UPDATE	See above comment.
WHEN NOT MATCHED BY SOURCE UPDATE	UPDATE Target SET <assignments> WHERE	

SQL Server MERGE feature	Migrate to Aurora MySQL	Comments
	Key NOT IN (SELECT Key FROM Source)	
WHEN NOT MATCHED BY SOURCE DELETE	DELETE FROM Target WHERE KEY NOT IN (SELECT Key FROM Source)	
OUTPUT clause	N/A	

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/replace.html*
- *https://dev.mysql.com/doc/refman/5.7/en/insert-on-duplicate.html*

Migrate from SQL Server PIVOT and UNPIVOT

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - PIVOT and UNPIVOT	• Straight forward rewrite to use traditional SQL syntax

Overview

PIVOT and UNPIVOT are relational operations used to transform a set by rotating rows into columns and columns into rows.

PIVOT

The PIVOT operator consists of several clauses and implied expressions.

The "Anchor" column is the column that is not be pivoted and results in a single row per unique value, similar to GROUP BY.

The pivoted columns are derived from the PIVOT clause and are the row values transformed into columns. The values for these columns are derived from the source column defined in the PIVOT clause.

Syntax

```
SELECT <Anchor column>,
       [Pivoted Column 1] AS <Alias>,
       [Pivoted column 2] AS <Alias>
       ...n
FROM
    (<SELECT Statement of Set to be Pivoted>)
    AS <Set Alias>
PIVOT
(
    <Aggregate Function>(<Aggregated Column>)
FOR
[<Column With the Values for the Pivoted Columns Names>]
    IN ( [Pivoted Column 1], [Pivoted column 2] ...)
) AS <Pivot Table Alias>;
```

PIVOT Examples

Create and populate the Orders Table.

```
CREATE TABLE Orders
(
       OrderID INT NOT NULL
       IDENTITY(1,1) PRIMARY KEY,
       OrderDate DATE NOT NULL,
```

```
       Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Orders (OrderDate, Customer)
VALUES
('20180101', 'John'),
('20180201', 'Mitch'),
('20180102', 'John'),
('20180104', 'Kevin'),
('20180104', 'Larry'),
('20180104', 'Kevin'),
('20180104', 'Kevin');
```

Create a simple PIVOT for the number of orders per day (days of month 5-31 omitted for example simplicity).

```
SELECT  'Number of Orders Per Day' AS DayOfMonth,
               [1], [2], [3], [4]  /*...[31]*/
FROM  (
       SELECT  OrderID,
               DAY(OrderDate) AS OrderDay
       FROM    Orders
      ) AS SourceSet
PIVOT
(
 COUNT(OrderID)
 FOR OrderDay IN ([1], [2], [3], [4] /*...[31]*/)
) AS PivotSet;
```

```
DayOfMonth                    1        2       3       4     /*...[31]*/
------                        -        -       -       -
Number of Orders Per Day      2        1       0       4
```

> **Note:** The result set is now oriented in rows vs. columns. The first column is the description of the columns to follow.

PIVOT for number of order per day per customer.

```
SELECT  Customer,
       [1], [2], [3], [4] /*...[31]*/
FROM  (
       SELECT  OrderID,
               Customer,
               DAY(OrderDate) AS OrderDay
       FROM    Orders
       ) AS SourceSet
PIVOT
(
 COUNT(OrderID)
 FOR OrderDay IN ([1], [2], [3], [4] /*...[31]*/)
) AS PivotSet;
```

```
Customer        1       2       3       4
--------        -       -       -       -
```

```
John            1       1       0       0
Kevin           0       0       0       3
Larry           0       0       0       1
Mitch           1       0       0       0
```

UNPIVOT

UNPIVOT is similar to PIVOT in reverse, but spreads existing column values into rows.

The source set is similar to the result of the PIVOT with values pertaining to particular entities listed in columns. Since the result set has more rows than the source, aggregations aren't required.

It is less commonly used than PIVOT because most data in relational databases have attributes in columns; not the other way around.

UNPIVOT Examples

Create an populate the "pivot like" EmployeeSales table (in a actual scenario, this is most likely a view or a set from an external source).

```
CREATE TABLE EmployeeSales
(
SaleDate DATE NOT NULL PRIMARY KEY,
John   INT,
Kevin  INT,
Mary   INT
);
```

```
INSERT INTO EmployeeSales
VALUES
('20180101', 150, 0, 300),
('20180102', 0, 0, 0),
('20180103', 250, 50, 0),
('20180104', 500, 400, 100);
```

Unpivot employee sales per date into individual rows per employee.

```
SELECT SaleDate,
       Employee,
       SaleAmount
FROM
   (
     SELECT SaleDate, John, Kevin, Mary
     FROM EmployeeSales
   ) AS SourceSet
UNPIVOT (
         SaleAmount
         FOR Employee IN (John, Kevin, Mary)
        )AS UnpivotSet;
```

```
SaleDate        Employee        SaleAmount
--------        --------        ----------
2018-01-01      John            150
2018-01-01      Kevin           0
```

```
2018-01-01      Mary            300
2018-01-02      John            0
2018-01-02      Kevin           0
2018-01-02      Mary            0
2018-01-03      John            250
2018-01-03      Kevin           50
2018-01-03      Mary            0
2018-01-04      John            500
2018-01-04      Kevin           400
2018-01-04      Mary            100
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/queries/from-using-pivot-and-unpivot

Migrate to Aurora MySQL PIVOT and UNPIVOT

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - PIVOT and UNPIVOT	• Straight forward rewrite to use traditional SQL syntax

Overview

Aurora MySQL does not support the PIVOT and UNPIVOT relational operators.

Functionality of both operators can be rewritten to use standard SQL syntax, as shown in the examples below.

Examples

PIVOT

Create and populate the Orders Table

```
CREATE TABLE Orders
(
       OrderID INT
       AUTO_INCREMENT NOT NULL PRIMARY KEY,
       OrderDate DATE NOT NULL,
       Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO Orders (OrderDate, Customer)
VALUES
('20180101', 'John'),
('20180201', 'Mitch'),
('20180102', 'John'),
('20180104', 'Kevin'),
('20180104', 'Larry'),
('20180104', 'Kevin'),
('20180104', 'Kevin');
```

Simple PIVOT for number of orders per day (days of month 5-31 omitted for example simplicity)

```
SELECT 'Number of Orders Per Day' AS DayOfMonth,
       COUNT(CASE WHEN DAY(OrderDate) = 1 THEN 'OrderDate' ELSE NULL END) AS '1',
       COUNT(CASE WHEN DAY(OrderDate) = 2 THEN 'OrderDate' ELSE NULL END) AS '2',
       COUNT(CASE WHEN DAY(OrderDate) = 3 THEN 'OrderDate' ELSE NULL END) AS '3',
       COUNT(CASE WHEN DAY(OrderDate) = 4 THEN 'OrderDate' ELSE NULL END) AS '4' /*...[31]*/
FROM   Orders AS O;
```

```
DayOfMonth                  1       2       3       4     /*...[31]*/
------                      -       -       -       -
Number of Orders Per Day    2       1       0       4
```

PIVOT for number of order per day, per customer

```
SELECT  Customer,
       COUNT(CASE WHEN DAY(OrderDate) = 1 THEN 'OrderDate' ELSE NULL END) AS '1',
       COUNT(CASE WHEN DAY(OrderDate) = 2 THEN 'OrderDate' ELSE NULL END) AS '2',
       COUNT(CASE WHEN DAY(OrderDate) = 3 THEN 'OrderDate' ELSE NULL END) AS '3',
       COUNT(CASE WHEN DAY(OrderDate) = 4 THEN 'OrderDate' ELSE NULL END) AS '4' /*...[31]*/
FROM   Orders AS O
GROUP BY Customer;
```

```
Customer        1       2       3       4
--------        -       -       -       -
John            1       1       0       0
Kevin           0       0       0       3
Larry           0       0       0       1
Mitch           1       0       0       0
```

UNPIVOT

Create an populate the 'pivot like' EmployeeSales table.

> **Note:** in real life this will most likely be a view, or a set from an external source.

```
CREATE TABLE EmployeeSales
(
SaleDate DATE NOT NULL PRIMARY KEY,
John   INT,
Kevin  INT,
Mary   INT
);
```

```
INSERT INTO EmployeeSales
VALUES
('20180101', 150, 0, 300),
('20180102', 0, 0, 0),
('20180103', 250, 50, 0),
('20180104', 500, 400, 100);
```

Unpivot employee sales per date into individual rows per employee

```
SELECT SaleDate,
       Employee,
       SaleAmount
FROM
(
       SELECT  SaleDate,
               Employee,
               CASE
                       WHEN Employee = 'John'  THEN John
                       WHEN Employee = 'Kevin' THEN Kevin
                       WHEN Employee = 'Mary'  THEN Mary
```

```
                END AS SaleAmount
        FROM EmployeeSales
        CROSS JOIN
        (
                SELECT 'John' AS Employee
                UNION ALL
                SELECT 'Kevin'
                UNION ALL
                SELECT 'Mary'
        ) AS Employees
) AS UnpivotedSet;
```

```
SaleDate        Employee        SaleAmount
--------        --------        ----------
2018-01-01      John            150
2018-01-01      Kevin           0
2018-01-01      Mary            300
2018-01-02      John            0
2018-01-02      Kevin           0
2018-01-02      Mary            0
2018-01-03      John            250
2018-01-03      Kevin           50
2018-01-03      Mary            0
2018-01-04      John            500
2018-01-04      Kevin           400
2018-01-04      Mary            100
```

For more information, see https://en.wikibooks.org/wiki/MySQL/Pivot_table

Migrate from SQL Server Synonyms

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Synonyms	• Use stored procedures and functions to abstract instance-wide objects

Overview

Synonyms are database objects that server as alternative identifiers for other database objects.The referenced database object is called the 'base object' and may reside in the same database, another database on the same instance, or a remote server.

Synonyms provide an abstraction layer to isolate client application code from changes to the name or location of the base object.

In SQL Server, Synonyms are often used to simplify the use of four-part identifiers when accessing remote instances.

For Example, table A resides on Server A, and the client application accesses it directly. For scale out reasons, Table A needs to be moved to server B to offload resource consumption on Server A. Without synonyms, the client application code must be rewritten to access Server B. Instead, you can create a synonym called Table A and it will transparently redirect the calling application to Server B without any code changes.

Synonyms can be created for the following objects:

- Assembly (CLR) stored procedures, table-valued functions, scalar functions, and aggregate functions
- Replication-filter-procedures
- Extended stored procedures
- SQL scalar functions, table-valued functions, inline-tabled-valued functions, views, and stored procedures
- User defined tables including local and global temporary tables

Syntax

```
CREATE SYNONYM [ <Synonym Schema> ] . <Synonym Name>
FOR [ <Server Name> ] . [ <Database Name> ] . [ Schema Name> ] . <Object Name>
```

Examples

Create a synonym for a local object in a separate database.

```
CREATE TABLE DB1.Schema1.MyTable
(
KeyColumn INT IDENTITY PRIMARY KEY,
DataColumn VARCHAR(20) NOT NULL
);

USE DB2;
CREATE SYNONYM Schema2.MyTable
FOR DB1.Schema1.MyTable
```

Create a synonym for a remote object.

```
-- On ServerA
CREATE TABLE DB1.Schema1.MyTable
(
 KeyColumn INT IDENTITY PRIMARY KEY,
DataColumn VARCHAR(20) NOT NULL
);

-- On Server B
USE DB2;
CREATE SYNONYM Schema2.MyTable
FOR ServerA.DB1.Schema1.MyTable;
```

> **Note:** This example assumes a linked server named ServerA exists on Server B that points to Server A.

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/create-synonym-transact-sql

Migrate to Aurora MySQL Synonyms

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Synonyms	• Use stored procedures and functions to abstract instance-wide objects

Overview

Synonyms are not supported in Aurora MySQL and there is no known generic workaround.

For accessing tables (or views), a partial workaround is to use encapsulating views as an abstraction layer. Similarly, you can use functions or stored procedures that call other functions or stored procedures.

> **Note:** Synonyms are often used in conjunction with Linked Servers, which are not supported by Aurora MySQL.

For more information, see Linked Servers, Views , Functions , and Stored Procedures.

Migrate from SQL Server TOP and FETCH

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - TOP and FETCH	• Syntax rewrite, very similar functionality • Convert PERCENT and TIES to sub-queries

Overview

SQL Server supports two options for limiting and paging result sets returned to the client. TOP is a legacy, proprietary T-SQL keyword that is still supported due to its wide usage. The ANSI compliant syntax of FETCH and OFFSET were introduced in SQL Server 2012 and are recommended for paginating results sets.

TOP

The TOP (n) operator is used in the SELECT list and limits the number of rows returned to the client based on the ORDER BY clause.

> **Note:** When TOP is used with no ORDER BY clause, the query is non-deterministic and may return any rows up to the number specified by the TOP operator.

TOP (n) can be used with two modifier options:

- **TOP (n) PERCENT** is used to designate a percentage of the rows to be returned instead of a fixed maximal row number limit (n). When using PERCENT, n can be any value from 1-100.
- **TOP (n) WITH TIES** is used to allow overriding the n maximal number (or percentage) of rows specified in case there are additional rows with the same ordering values as the last row.

  > **Note:** If TOP (n) is used without WITH TIES and there are additional rows that have the same ordering value as the last row in the group of n rows, the query is also non-deterministic because the last row may be any of the rows that share the same ordering value.

Syntax

```
SELECT TOP (<Limit Expression>) [PERCENT] [ WITH TIES ] <Select Expressions List>
FROM...
```

OFFSET... FETCH

OFFSET... FETCH as part of the ORDER BY clause is the ANSI compatible syntax for limiting and paginating result sets. It allows specification of the starting position and limits the number of rows returned, which enables easy pagination of result sets.

Similar to TOP, OFFSET... FETCH relies on the presentation order defined by the ORDER BY clause. Unlike TOP, it is part of the ORDER BY clause and can't be used without it.

> **Note:** Queries using FETCH... OFFSET can still be non-deterministic if there is more than one row that has the same ordering value as the last row.

Syntax

```
ORDER BY <Ordering Expression> [ ASC | DESC ] [ ,...n ]
OFFSET  <Offset Expression> { ROW | ROWS }
[FETCH { FIRST | NEXT } <Page Size Expression> { ROW | ROWS } ONLY ]
```

Examples

Create the OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

Retrieve the 3 most ordered items by quantity.

```
-- Using TOP
SELECT TOP (3) *
FROM OrderItems
ORDER BY Quantity DESC;

-- USING FETCH
SELECT *
FROM OrderItems
ORDER BY Quantity DESC
OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY;
```

```
OrderID Item            Quantity
------- ----            --------
3       M6 Locking Nut  300
3       M8 Washer       200
2       M8 Nut          100
```

Include rows with ties.

```
SELECT TOP (3) WITH TIES *
FROM OrderItems
ORDER BY Quantity DESC;
```

```
OrderID Item            Quantity
------- ----            --------
3       M6 Locking Nut  300
3       M8 Washer       200
2       M8 Nut          100
1       M8 Bolt         100
```

Retrieve half the rows based on quantity.

```
SELECT TOP (50) PERCENT *
FROM OrderItems
ORDER BY Quantity DESC;
```

```
OrderID Item            Quantity
------- ----            --------
3       M6 Locking Nut  300
3       M8 Washer       200
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/t-sql/queries/select-order-by-clause-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/queries/top-transact-sql*

Migrate to Aurora MySQL LIMIT (TOP and FETCH Equivalent)

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - TOP and FETCH	• Syntax rewrite, very similar functionality • Convert PERCENT and TIES to sub-queries

Overview

Aurora MySQL supports the non-ANSI compliant (but popular with other engines) LIMIT... OFFSET operator for paging results sets.

The LIMIT clause limits the number of rows returned and does not require an ORDER BY clause, although that would make the query non-deterministic.

The OFFSET clause is zero-based, similar to SQL Server and used for pagination.

Syntax

```
syntax specification
```

Migration Considerations

LIMIT... OFFSET syntax can be used to replace the functionality of both TOP(n) and FETCH... OFFSET in SQL Server. It is automatically converted by the Schema Conversion Tool (SCT) except for the WITH TIES and PERCENT modifiers.

To replace the PERCENT option, you must first calculate how many rows the query returns and then calculate the fixed number of rows to be returned based on that number (see the example below).

> **Note:** Since this technique involves added complexity and accessing the table twice, consider changing the logic to use a fixed number instead of percentage.

To replace the WITH TIES option, you must rewrite the logic to add another query that checks for the existence of additional rows that have the same ordering value as the last row returned from the LIMIT clause.

> **Note:** Since this technique introduces significant added complexity and three accesses to the source table, consider changing the logic to introduce a tie-breaker into the ORDER BY clause (see the example below).

Examples

Create the OrderItems table.

```
CREATE TABLE OrderItems
(
 OrderID INT NOT NULL,
 Item VARCHAR(20) NOT NULL,
 Quantity SMALLINT NOT NULL,
 PRIMARY KEY(OrderID, Item)
);
```

```
INSERT INTO OrderItems (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200),
(3, 'M6 Locking Nut', 300);
```

Retrieve the three most ordered items by quantity.

```
SELECT *
FROM OrderItems
ORDER BY Quantity DESC
LIMIT 3 OFFSET 0;
```

```
OrderID Item            Quantity
------- ----            --------
3       M6 Locking Nut  300
3       M8 Washer       200
2       M8 Nut          100
```

Include rows with ties.

```
SELECT *
FROM
(
 SELECT *
 FROM OrderItems
 ORDER BY Quantity DESC
 LIMIT 3 OFFSET 0
) AS X
UNION
SELECT *
FROM OrderItems
WHERE Quantity = (
                  SELECT Quantity
                  FROM OrderItems
                  ORDER BY Quantity DESC
                  LIMIT 1 OFFSET 2
                 )
ORDER BY Quantity DESC
```

```
OrderID Item            Quantity
------- ----            --------
3       M6 Locking Nut  300
3       M8 Washer       200
2       M8 Nut          100
1       M8 Bolt         100
```

Retrieve half the rows based on quantity.

```
CREATE PROCEDURE P(Percent INT)
BEGIN
DECLARE N INT;
SELECT COUNT(*) * Percent / 100 FROM OrderItems INTO N;
SELECT *
FROM OrderItems
ORDER BY Quantity DESC
LIMIT N OFFSET 0;
END
```

```
CALL P(50);
```

```
OrderID Item            Quantity
------- ----            --------
3        M6 Locking Nut  300
3       M8 Washer       200
```

Summary

SQL Server	Aurora MySQL	Comments
TOP (n)	LIMIT n	
TOP (n) WITH TIES	Not supported	See examples for work-around
TOP (n) PERCENT	Not supported	See examples for work-around
OFFSET... FETCH	LIMIT... OFFSET	

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/select.html*
- *https://dev.mysql.com/doc/refman/5.7/en/limit-optimization.html*

Migrate from SQL Server Triggers

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Triggers	• Only FOR EACH ROW processing • No DDL or EVENT triggers • BEFORE triggers replace INSTEAD OF triggers

Overview

Triggers are special type of stored procedure that execute automatically in response to events and are most commonly used for Data Manipulation Language (DML).

SQL Server supports AFTER/FOR and INSTEAD OF triggers, which can be created on tables and views (AFTER and FOR are synonymous). SQL Server also provides an event trigger framework at the server and database levels that includes Data Definition Language (DDL), Data Control Language (DCL), and general system events such as login.

> **Note:** SQL Sever does not support FOR EACH ROW triggers in which the trigger code is executed once for each row of modified data.

Trigger Execution

- AFTER triggers execute after DML statements complete execution.
- INSTEAD OF triggers execute code in place of the original DML statement.

AFTER triggers can be created on table only. INSTEAD OF Triggers can be created on tables and Views.

Only a single INSTEAD OF trigger can be created for any given object and event. When multiple AFTER triggers exist for the same event and object, you can partially set the trigger order by using the *sp_settriggerorder* system stored procedure. It allows setting the first and last triggers to be executed, but not the order of others.

Trigger Scope

SQL Server supports only statement level triggers. The trigger code is executed only once per statement. The data modified by the DML statement is available to the trigger scope and is saved in two virtual tables: INSERTED and DELETED. These tables contain the entire set of changes performed by the DML statement that caused trigger execution.

SQL triggers are always execute within the transaction of the statement that triggered the execution. If the trigger code issues an explicit ROLLBACK, or causes an exception that mandates a rollback, the DML statement is also rolled back (for INSTEAD OF triggers, the DML statement is not executed and, therefore, does not require a rollback).

Examples

Use a DML Trigger to Audit Invoice Deletions

The following example demonstrates how to use a trigger to log rows deleted from a table.

Create and populate an `Invoices` table.

```
CREATE TABLE Invoices
(
InvoiceID       INT NOT NULL PRIMARY KEY,
Customer        VARCHAR(20) NOT NULL,
TotalAmount     DECIMAL(9,2) NOT NULL
);

INSERT INTO Invoices (InvoiceID,Customer,TotalAmount)
VALUES
(1, 'John', 1400.23),
(2, 'Jeff', 245.00),
(3, 'James', 677.22);
```

Create an `InvoiceAuditLog` table.

```
CREATE TABLE InvoiceAuditLog
(
InvoiceID       INT NOT NULL PRIMARY KEY,
Customer        VARCHAR(20) NOT NULL,
TotalAmount     DECIMAL(9,2) NOT NULL,
DeleteDate      DATETIME NOT NULL DEFAULT (GETDATE()),
DeletedBy       VARCHAR(128) NOT NULL DEFAULT (CURRENT_USER)
);
```

Create an AFTER DELETE trigger to log deletions from the `Invoices` table to the audit log.

```
CREATE TRIGGER LogInvoiceDeletes
ON Invoices
AFTER DELETE
AS
BEGIN
INSERT INTO InvoiceAuditLog (InvoiceID, Customer, TotalAmount)
SELECT  InvoiceID,
        Customer,
        TotalAmount
FROM    Deleted
END;
```

Delete an invoice.

```
DELETE FROM Invoices
WHERE InvoiceID = 3;
```

Query the content of both tables.

```
SELECT *
FROM Invoices AS I
```

```
     FULL OUTER JOIN
     InvoiceAuditLog AS IAG
     ON I.InvoiceID = IAG.InvoiceID;
```

The code above displays the following results.

```
InvoiceID   Customer   TotalAmount    InvoiceID   Customer
TotalAmount     DeleteDate       DeletedBy
---------   --------   -----------    ---------   --------     -----------     -------
---       ----------
1           John       1400.23        NULL        NULL         NULL            NULL
         NULL
2           Jeff       245.00         NULL        NULL         NULL            NULL
         NULL
NULL        NULL       NULL           3           James
677.22          20180224 13:02   Domain/JohnCortney
```

Create a DDL Trigger

Create a trigger to protect all tables in the database from accidental deletion.

```
CREATE TRIGGER PreventTableDrop
ON DATABASE FOR DROP_TABLE
AS
BEGIN
    RAISERROR ('Tables Can''t be dropped in this database', 16, 1)
    ROLLBACK TRANSACTION
END;
```

Test the trigger by attempting to drop a table.

```
DROP TABLE [Invoices];
   GO
```

The system displays the follow message indicating the `Invoices` table cannot be dropped.

```
Msg 50000, Level 16, State 1, Procedure PreventTableDrop, Line 5 [Batch Start Line 56]
Tables Can't be dropped in this database
Msg 3609, Level 16, State 2, Line 57
The transaction ended in the trigger. The batch has been aborted.
```

For more information, see

- https://docs.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers
- https://docs.microsoft.com/en-us/sql/relational-databases/triggers/ddl-triggers

Migrate to Aurora MySQL Triggers

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Triggers	• Only FOR EACH ROW processing • No DDL or EVENT triggers • BEFORE triggers replace INSTEAD OF triggers

Overview

Aurora MySQL provides Data manipulation Language (DML) triggers only.

It supports BEFORE and AFTER triggers for INSERT, UPDATE, and DELETE with full control over trigger execution order.

MySQL triggers differ substantially from SQL Server. However, you can migrate most common use cases with minimal code changes. The following list identifies the major differences between the SQL Server and Aurora MySQL triggers:

- Aurora MySQL triggers are executed once per row, not once per statement as with SQL Server.
- Aurora MySQL does not support DDL or system event triggers.
- Aurora MySQL supports BEFORE triggers; SQL does not.
- Aurora MySQL supports full execution order control for multiple triggers.

  Note: Stored procedures, triggers, and user defined functions in Aurora MySQL are collectively referred to as *Stored Routines*. When binary logging is enabled, MySQL SUPER privilege is required to run stored routines. However, you can run stored routines with binary logging enabled without SUPER privilege by setting the*log_bin_trust_function_creators* parameter to *true* for the DB parameter group for your MySQL instance.

Syntax

```
CREATE [DEFINER = { user | CURRENT_USER }] TRIGGER <Trigger Name>
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON <Table Name>
FOR EACH ROW
[{ FOLLOWS | PRECEDES } <Other Trigger Name>]
<Trigger Code Body>
```

Examples

Use a DML Trigger to Audit Invoice Deletions

The following example demonstrates how to use a trigger to log rows deleted from a table.

Create and populate an `Invoices` table.

```
CREATE TABLE Invoices
(
InvoiceID       INT NOT NULL PRIMARY KEY,
Customer        VARCHAR(20) NOT NULL,
TotalAmount     DECIMAL(9,2) NOT NULL
);

INSERT INTO Invoices (InvoiceID, Customer, TotalAmount)
VALUES
(1, 'John', 1400.23),
(2, 'Jeff', 245.00),
(3, 'James', 677.22);
```

Create an `InvoiceAuditLog` table.

```
CREATE TABLE InvoiceAuditLog
(
InvoiceID       INT NOT NULL
 PRIMARY KEY,
Customer        VARCHAR(20) NOT NULL,
TotalAmount     DECIMAL(9,2) NOT NULL,
DeleteDate      DATETIME NOT NULL
 DEFAULT (GETDATE()),
DeletedBy       VARCHAR(128) NOT NULL
 DEFAULT (CURRENT_USER)
);
```

Create a trigger to log deleted rows.

```
CREATE OR REPLACE TRIGGER LogInvoiceDeletes
ON Invoices
FOR EACH ROW
AFTER DELETE
AS
 BEGIN
 INSERT INTO InvoiceAuditLog (InvoiceID, Customer, TotalAmount, DeleteDate, DeletedBy)
 SELECT InvoiceID,
        Customer,
        TotalAmount,
        NOW(),
        CURRENT_USER()
 FROM   OLD
END;
```

Test the trigger by deleting an invoice.

```
DELETE FROM Invoices
WHERE InvoiceID = 3;
```

Select all rows from the `InvoiceAuditLog` table.

```
SELECT * FROM InvoiceAuditLog;
```

The above statement displays the following results.

```
InvoiceID    Customer     TotalAmount     DeleteDate       DeletedBy
---------    --------     -----------     ----------       ---------
        3    James        677.22          20180224 13:02   GeorgeZ
```

Note: Additional code changes were required for this example because the GETDATE() function is not supported by MySQL.
See Date and Time Functions for workarounds.

Summary

	SQL Server	Aurora MySQL	Workaround
DML Triggers Scope	Statement level only	FOR EACH ROW only	Most trigger code, such as the SQL Server example in the previous section, will work without significant code changes. Even though SQL Server triggers process a set of rows at once, typically no changes are needed to process one row at a time. A set of one row, is a valid set and should be processed correctly either way. The main drawback of FOR EACH ROW triggers, is that you can't access **other rows** that were modified in the same operation. The NEW and OLD virtual tables can only reference the current row. Therefore, for example, tasks such as logging aggregate data for the entire DML statement set, may require more significant code changes. If your SQL Server trigger code uses loops and cursors to process one row at a time, the loop and cursor sections can be safely removed.
Access to change set	INSERTED and DELETED Virtual multi-row tables	OLD and NEW virtual one-row tables	You must modify the trigger code to use NEW instead of INSERTED, and OLD instead of DELETED.
System event triggers	DDL, DCL and other event types	Not supported	
Trigger exe-	AFTER and INSTEAD	AFTER and BEFORE	For INSTEAD OF triggers, you must

	SQL Server	Aurora MySQL	Workaround
cution phase	OF		modify the trigger code to remove the explicit execution of the calling DML, which is not needed in a BEFORE trigger. In Aurora MySQL, the OLD and NEW tables are updateable. If your trigger code needs to modify the change set, update the OLD and NEW tables with the changes. The updated data is applied to the table data when the trigger execution completes.
Multi-trigger execution order	Can only set first and last using *sp_settriggerorder*	Can set any execution order using PRECEDS and FOLLOWS	Update the trigger code to reflect the desired execution order
Drop a trigger	DROP TRIGGER <trigger name>;	DROP TRIGGER <trigger name>;	Compatible syntax
Modify trigger code	Use the ALTER TRIGGER statement	Not supported	
Enable/Disable a trigger	Use the ALTER TRIGGER <trigger name> ENABLE; and ALTER TRIGGER <trigger name> DISABLE;	Not supported	A common workaround is to use a database table with flags indicating which trigger to execute. Modify the trigger code using conditional flow control (IF) to query the table and determine whether or not the trigger should execute additional code or exit without performing any modifications to the database.
Triggers on views	INSTEAD OF TRIGGERS only	Not supported	

For more information, see https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html and https://dev.mysql.com/doc/refman/5.7/en/create-trigger.html

Migrate from SQL Server User Defined Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - User Defined Functions	• Scalar functions only, rewrite inline TVF as views or derived tables, and multi-statememt TVF as stored procedures

Overview

User Defined Functions (UDF) are code objects that accept input parameters and return either a scalar value or a set consisting of rows and columns.
SQL Server UDFs can be implemented using T-SQL or Common Language Runtime (CLR) code.

> **Note:** This section does not cover CLR code objects.

Function invocations can not have any lasting impact on the database. They must be contained and can only modify objects and data local to their scope (for example, data in local variables). Functions are not allowed to modify data or the structure of a database.

Functions may be deterministic or non-deterministic. Deterministic functions always return the same result when executed with the same data. Non-deterministic functions may return different results each time they execute. For example, a function that returns the current date or time.

SQL Server supports three types of T-SQL UDFs: Scalar Functions, Inline Table-Valued Functions, and Multi-Statement Table-Valued Functions.

Scalar User Defined Functions

Scalar UDFs accept zero or more parameters and return a scalar value. They can be used in T-SQL expressions.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS <Return Data Type>
[AS]
BEGIN
<Function Body Code>
RETURN <Scalar Expression>
END[;]
```

Examples

Create a scalar function to change the first character of a string to upper case.

```
CREATE FUNCTION dbo.UpperCaseFirstChar (@String VARCHAR(20))
RETURNS VARCHAR(20)
```

```
AS
BEGIN
RETURN UPPER(LEFT(@String, 1)) + LOWER(SUBSTRING(@String, 2, 19))
END;
```

```
SELECT dbo.UpperCaseFirstChar ('mIxEdCasE');

---------
Mixedcase
```

Inline User Defined Table-Valued Functions

Inline table-valued UDFs are similar to views or a Common Table Expressions (CTE) with the added benefit of parameters. They can be used in FROM clauses as subqueries and can be joined to other source table rows using the APPLY and OUTER APPLY operators. In-line table valued UDFs have many associated internal optimizer optimizations due to their simple, view-like characteristics.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS TABLE
[AS]
RETURN (<SELECT Query>)[;]
```

Examples

Create a table valued function to aggregate employee orders

```
CREATE TABLE Orders
(
OrderID INT NOT NULL PRIMARY KEY,
EmployeeID INT NOT NULL,
OrderDate DATETIME NOT NULL
);
```

```
INSERT INTO Orders (OrderID, EmployeeID, OrderDate)
VALUES
(1, 1, '20180101 13:00:05'),
(2, 1, '20180201 11:33:12'),
(3, 2, '20180112 10:22:35');
```

```
CREATE FUNCTION dbo.EmployeeMonthlyOrders
(@EmployeeID INT)
RETURNS TABLE AS
RETURN
(
SELECT  EmployeeID,
        YEAR(OrderDate) AS OrderYear,
        MONTH(OrderDate) AS OrderMonth,
        COUNT(*) AS NumOrders
FROM    Orders AS O
WHERE   EmployeeID = @EmployeeID
GROUP BY EmployeeID,
```

```
         YEAR(OrderDate),
         MONTH(OrderDate)
);
```

```
SELECT  *
FROM    dbo.EmployeeMonthlyOrders (1)

EmployeeID      OrderYear       OrderMonth      NumOrders
----------      ---------       ----------      ---------
1               2018            1               1
1               2018            2               1
```

Multi-Statment User Defined Table-Valued Functions

Multi-statement table valued UDFs , like In-line UDFs, are also similar to views or CTEs, with the added benefit of allowing parameters. They can be used in FROM clauses as sub queries and can be joined to other source table rows using the APPLY and OUTER APPLY operators.

The difference between multi-statement UDFs and the inline UDFs is that multi-statement UDFs are not restricted to a single SELECT statement. They can consist of multiple statements including logic implemented with flow control, complex data processing, security checks, etc.

The downside of using multi-statement UDFs is that there are far less optimizations possible and performance may suffer.

Syntax

```
CREATE FUNCTION <Function Name> ([{<Parameter Name> [AS] <Data Type> [= <Default
Value>] [READONLY]} [,...n]])
RETURNS <@Return Variable> TABLE <Table Definition>
[AS]
BEGIN
<Function Body Code>
RETURN
END[;]
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql

Migrate to Aurora MySQL User Defined Functions

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - User Defined Functions	• Scalar functions only, rewrite inline TVF as views or derived tables, and multi-statememt TVF as stored procedures

Overview

Aurora MySQL supports the creation of User Defined scalar functions only. There is no support for table-valued functions.

Unlike SQL Server, Aurora MySQL allows routines to read and write data using INSERT, UPDATE and DELETE. It also allows DDL statements such as CREATE and DROP. Aurora MySQL does not permit stored functions to contain explicit SQL transaction statements such as COMMIT and ROLLBACK.

In Aurora MySQL, you can explicity specify several options with the CREATE FUNCTION statement. These characteristics are saved with the function definition and are viewable with the SHOW CREATE FUNCTION statement.

- The DETERMINISTIC option must be explicitly stated. Otherwise, the engine assumes it is not deterministic.

  Note: The MySQL engine does not check the validity of the deterministic property declaration. If you wrongly specify a function as DETERMINISTIC when in fact it is not, unexpected results and errors may occur.

- CONTAINS SQL indicates the function code does not contain statements that read or modify data.
- READS SQL DATA indicates the function code contains statements that read data (for example, SELECT) but not statements that modify data (for example, INSERT, DELETE, or UPDATE).
- MODIFIES SQL DATA indicates the function code contains statements that may modify data.

  Note: The above options are advisory only. The server does not constrain the function code based on the declaration. This feature is useful in assisting code management.

Syntax

```
CREATE FUNCTION <Function Name> ([<Function Parameter>[,...]])
RETURNS <Returned Data Type> [characteristic ...]
<Function Code Body>
```

```
characteristic:
COMMENT '<Comment>' | LANGUAGE SQL | [NOT] DETERMINISTIC
```

```
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
```

Migration Considerations

For scalar functions, migration should be straight forward as far as the function syntax is concerned. Note that rules in Aurora MySQL regarding functions are much more lenient than SQL Server.

A function in Aurora MySQL may modify data and schema. Function determinism must be explicitly stated, unlike SQL Server that infers it from the code. Additional properties can be stated for a function, but most are advisory only and have no functional impact.

Also note that the AS keyword, which is mandatory in SQL Server before the function's code body, is not valid Aurora MySQL syntax and must be removed.

Table valued functions will be harder to migrate. For most in-line table valued functions, a simple path may consist of migrating to using views, and letting the calling code handle parameters.
Complex multi-statement table valued functions will require rewrite as a stored procedure, which may in turn write the data to a temporary or standard table for further processing.

Examples

Create a scalar function to change the first character of string to upper case.

```
CREATE FUNCTION UpperCaseFirstChar (String VARCHAR(20))
RETURNS VARCHAR(20)
BEGIN
RETURN CONCAT(UPPER(LEFT(String, 1)) , LOWER(SUBSTRING(String, 2, 19)));
END
```

```
SELECT UpperCaseFirstChar ('mIxEdCasE');
```

```
---------
Mixedcase
```

Summary

The following table identifies similarities, differences, and key migration considerations.

SQL Server User Defined Function Feature	Migrate to Aurora MySQL	Comment
Scalar UDF	Scalar UDF	Use CREATE FUNCTION with similar syntax, remove the AS keyword.
Inline table valued UDF	N/A	Use views and replace parameters with WHERE filter predicates.
Multistatement table valued UDF	N/A	Use stored procedures to populate tables and read from the table directly.

SQL Server User Defined Function Feature	Migrate to Aurora MySQL	Comment
UDF determinism implicit	Explicit declaration	Use the DETERMINISTIC characteristic explicitly to denote a deterministic function, which enables engine optimizations.
UDF boundaries local only	Can change data and schema	UDF rules are more lenient, avoid unexpected changes from function invocation.

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/create-procedure.html*
- *https://dev.mysql.com/doc/refman/5.7/en/create-function-udf.html*

Migrate from SQL Server User Defined Types

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - User Defined Types	• Replace scalar UDT with base types • Rewrite Stored Procedures that use table-type input parameters to use strings with CSV, XML, or JSON, or to process row-by-row

Overview

SQL Server User defined Types provide a mechanism for encapsulating custom data types and for adding NULL constraints.

SQL Server also supports table-valued user defined types, which you can use to pass a set of values to a stored procedure.

User defined types can also be associated to CLR code assemblies. Beginning with SQL Server 2014, memory optimized types support memory optimized tables and code.

> **Note:** If your code uses custom rules bound to data types, Microsoft recommends discontinuing use of this deprecated feature.

All user defined types are based on an existing system data types. They allow developers to reuse the definition, making the code and schema more readable.

Syntax

The simplified syntax for the CREATE TYPE statement is specified below.

```
CREATE TYPE <type name> {
FROM <base type> [ NULL | NOT NULL ] | AS TABLE (<Table Definition>)}
```

Examples

User Defined Types

Create a `ZipCode`Scalar User Defined Type.

```
CREATE TYPE ZipCode
FROM CHAR(5)
NOT NULL
```

Use the `ZipCode`type in a table.

```
CREATE TABLE UserLocations
(UserID INT NOT NULL PRIMARY KEY, ZipCode ZipCode);
```

```
INSERT INTO [UserLocations] ([UserID],[ZipCode]) VALUES (1, '94324');
INSERT INTO [UserLocations] ([UserID],[ZipCode]) VALUES (2, NULL);
```

The above code displays the following error message indicating NULL values for `ZipCode`are not allowed.

```
Msg 515, Level 16, State 2, Line 78
Cannot insert the value NULL into column 'ZipCode', table 'tempdb.dbo.UserLocations';
column does not allow nulls. INSERT fails.
The statement has been terminated.
```

Table-Valued types

The following example demonstrates how to create and use a table valued types to pass a set of values to a stored procedure:

Create the OrderItems table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create a table valued type for the `OrderItems`table.

```
CREATE TYPE OrderItems
AS TABLE
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create the `InsertOrderItems`procedure. Note that the entire set of rows from the table valued parameter is handled with one statement.

```
CREATE PROCEDURE InsertOrderItems
@OrderItems AS OrderItems READONLY
AS
BEGIN
    INSERT INTO OrderItems(OrderID, Item, Quantity)
    SELECT OrderID,
           Item,
           Quantity
    FROM @OrderItems;
END
```

Instantiate the `OrderItems`type, insert the values, and pass it to a stored procedure.

```
DECLARE @OrderItems AS OrderItems;
```

```
INSERT INTO @OrderItems ([OrderID], [Item], [Quantity])
VALUES
(1, 'M8 Bolt', 100),
(1, 'M8 Nut', 100),
(1, M8 Washer, 200);

EXECUTE [InsertOrderItems] @OrderItems = @OrderItems;

(3 rows affected)
```

Select all rows from the `OrderItems`table.

```
SELECT * FROM OrderItems;

OrderID    Item         Quantity
-------    ----         --------
1          M8 Bolt      100
1          M8 Nut       100
1          M8 Washer    200
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/create-type-transact-sql

Migrate to Aurora MySQL User Defined Types

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - User Defined Types	• Replace scalar UDT with base types • Rewrite Stored Procedures that use table-type input parameters to use strings with CSV, XML, or JSON, or to process row-by-row

Overview

Aurora MySQL 5.7 does not support user defined types and user defined table valued parameters.

The current documentation does not indicate these features will be supported in Aurora MySQL version 8.

Migration Considerations

For scalar User Defined Types, replace the type name with base type and optional NULL constraints.

For table valued user defined types used as stored procedure parameters, the workaround is more complicated.

Common solutions include using either temporary tables to hold the data or passing large string parameters containing the data in CSV, XML, JSON (or any other convenient format) and then writing code to parse these values in a stored procedure. Alternatively, if the logic does not require access to the entire set of changes, and for small data sets, it is easier to call the stored procedure in a loop and pass the columns as standard parameters, row by row.

Memory Optimized Engines are not yet supported in Aurura MySQL. You must convert memory optimized tables to disk based tables (see the *In-Memory OLTP* section).

Examples

Replacing a User Defined Type

Replace a `ZipCode`user defined type with a base type.

```
CREATE TABLE UserLocations
(
UserID INT NOT NULL
 PRIMARY KEY,
/*ZipCode*/ CHAR(5) NOT NULL
);
```

Replacing a Tabled Valued Stored Procedure Parameter

The following steps describe how to replace a table valued parameter with a source table and a LOOP Cursor.

Create an `OrderItems`table.

```
CREATE TABLE OrderItems
(
OrderID INT NOT NULL,
Item VARCHAR(20) NOT NULL,
Quantity SMALLINT NOT NULL,
PRIMARY KEY(OrderID, Item)
);
```

Create and populate the `SourceTable`.

```
CREATE TABLE SourceTable
(
OrderID INT,
Item VARCHAR(20),
Quantity SMALLINT,
PRIMARY KEY (OrderID, Item)
);
```

```
INSERT INTO SourceTable (OrderID, Item, Quantity)
VALUES
(1, 'M8 Bolt', 100),
(2, 'M8 Nut', 100),
(3, 'M8 Washer', 200);
```

Create a procedure to loop through the `SourceTable`and insert rows.

> **Note:** There are syntax differences from T-SQL for both the CREATE PROCEDURE and the CURSOR declaration and use.
> For more details, see the Stored Procedures and Cursors topics.

```
CREATE PROCEDURE LoopItems()
BEGIN
 DECLARE done INT DEFAULT FALSE;
 DECLARE var_OrderID INT;
 DECLARE var_Item VARCHAR(20);
 DECLARE var_Quantity SMALLINT;
 DECLARE ItemCursor CURSOR
  FOR SELECT OrderID,
             Item,
             Quantity
      FROM SourceTable;
 DECLARE CONTINUE HANDLER
  FOR NOT FOUND
  SET done = TRUE;
 OPEN ItemCursor;
 CursorStart: LOOP
 FETCH NEXT FROM ItemCursor
  INTO var_OrderID, var_Item, var_Quantity;
```

```
 IF Done
  THEN LEAVE CursorStart;
 END IF;
   INSERT INTO OrderItems (OrderID, Item, Quantity)
   VALUES (var_OrderID, var_Item, var_Quantity);
 END LOOP;
 CLOSE ItemCursor;
END;
```

Execute the stored procedure.

```
CALL LoopItems();
```

Select all rows from the `OrderItems`table.

```
SELECT * FROM OrderItems;

OrderID     Item        Quantity
-------     ----        --------
1           M8 Bolt     100
2           M8 Nut      100
3           M8 Washer   200
```

Summary

SQL Server	Aurora MySQL	Comments
Table Valued Parameters	Not supported	Use either temporary tables, or CSV, XML, JSON string parameters and parse the data. Alternatively, rewrite the stored procedure to accept the data one row at a time and process the data in a loop.
Memory Optimized Table valued User Defined Types	Not supported	Not supported

For more information, see https://dev.mysql.com/doc/refman/5.7/en/cursors.html

Migrate from SQL Server Sequences and Identity

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Sequences and Identity	• SEQUENCE objects not supported • Rewrite IDENTITY to AUTO_INCREMENT • Last value evaluated as MAX(Existing Value) + 1 on every restart!

Overview

Automatic enumeration functions and columns are common with relational database management systems and are often used for generating surrogate keys.

SQL Server provides several features that support automatic generation of monotonously increasing value generators:

- The IDENTITY property of a table column.
- The SEQUENCE objects framework.
- The numeric functions such as IDENTITY and NEWSEQUENTIALID.

Identity

The IDENTITY property is probably the most widely used means of generating surrogate primary keys in SQL Server applications. Each table may have a single numeric column assigned as an IDENTITY using the CREATE TABLE or ALTER TABLE DDL statements. You can explicitly specify a starting value and increment.

> **Note:** The identity property does not enforce uniqueness of column values, indexing, or any other property. Additional constraints such as Primary or Unique keys, explicit index specifications, or other properties must be specified in addition to the IDENTITY property.

The IDENTITY value is generated as part of the transaction that inserts table rows. Applications can obtain IDENTITY values using the @@IDENTITY, SCOPE_IDENTITY, and IDENT_CURRENT functions.

IDENTITY columns may be used as primary keys by themselves, as part of a compound key, or as non-key columns.

You can manage IDENTITY columns using the DBCC CHECKIDENT command, which provides functionality for reseeding and altering properties.

Syntax

```
IDENTITY [(<Seed Value>, <Increment Value>)]
```

View the original seed value of an IDENTITY column with the IDENT_SEED system function.

```
SELECT IDENT_SEED (<Table>)
```

Reseed an IDENTITY column.

```
DBCC CHECKIDENT (<Table>, RESEED, <Seed Value>)
```

Examples

Create a table with an IDENTITY primary key column.

```
CREATE TABLE MyTABLE
(
Col1 INT NOT NULL
 PRIMARY KEY NONCLUSTERED IDENTITY(1,1),
Col2 VARCHAR(20) NOT NULL
);
```

Insert a row and retrieve the generated IDENTITY value .

```
DECLARE @LastIdent INT;
INSERT INTO MyTable(Col2)
VALUES('SomeString');
SET @LastIdent = SCOPE_IDENTITY()
```

Create a table with a non-key IDENTITY column and an increment of 10.

```
CREATE TABLE MyTABLE
(
Col1 VARCHAR(20) NOT NULL
 PRIMARY KEY,
Col2 INT NOT NULL
 IDENTITY(1,10),
);
```

Create a table with a compound PK including an IDENTITY column.

```
CREATE TABLE MyTABLE
(
Col1 VARCHAR(20) NOT NULL,
Col2 INT NOT NULL
 IDENTITY(1,10),
PRIMARY KEY (Col1, Col2)
);
```

SEQUENCE

Sequences are objects that are independent of a particular table or column and are defined using the CREATE SEQUENCE DDL statement. You can manage sequences using the ALTER SEQUENCE statement. Multiple tables and multiple columns from the same table may use the values from one or more SEQUENCE objects.

You can retrieve a value from a SEQUENCE object using the NEXT VALUE FOR function. For example, a SEQUENCE value can be used as a default value for a surrogate key column.

SEQUENCE objects provide several advantages over IDENTITY columns:

- Can be used to obtain a value before the actual INSERT takes place.
- Value series can be shared among columns and tables.
- Easier management, restart, and modification of sequence properties.
- Allow assignment of value ranges using sp_sequence_get_range and not just per-row values.

Syntax

```
CREATE SEQUENCE <Sequence Name> [AS <Integer Data Type> ]
START WITH <Seed Value>
INCREMENT BY <Increment Value>;
```

```
ALTER SEQUENCE <Sequence Name>
RESTART [WITH <Reseed Value>]
INCREMENT BY <New Increment Value>;
```

Examples

Create a sequence for use as a primary key default.

```
CREATE SEQUENCE MySequence AS INT START WITH 1 INCREMENT BY 1;
CREATE TABLE MyTable
(
Col1 INT NOT NULL
 PRIMARY KEY NONCLUSTERED DEFAULT (NEXT VALUE FOR MySequence),
Col2 VARCHAR(20) NULL
);
```

```
INSERT MyTable (Col1, Col2) VALUES (DEFAULT, 'cde'), (DEFAULT, 'xyz');
```

```
SELECT * FROM MyTable;
```

```
Col1    Col2
----    ----
1       cde
2       xyz
```

Sequential Enumeration Functions

SQL Server provides two sequential generation functions: IDENTITY and NEWSEQUENTIALID.

> **Note:** The IDENTITY function should not be confused with the IDENTITY property of a column.

The IDENTITY function can be used only in a SELECT ... INTO statement to insert IDENTITY column values into a new table.

The NEWSEQUNTIALID function generates a hexadecimal GUID, which is an integer. While the NEWID function generates a random GUID, the NEWSEQUENTIALID function guarantees that every GUID cre-

ated is greater (in numeric value) than any other GUID previously generated by the same function on the same server since the operating system restart.

> **Note:** NEWSEQUENTIALID can be used only with DEFAULT constraints associated with columns having a UNIQUEIDENTIFIER data type.

Syntax

```
IDENTITY (<Data Type> [, <Seed Value>, <Increment Value>]) [AS <Alias>]
```

```
NEWSEQUENTIALID()
```

Examples

Use the IDENTITY function as surrogate key for a new table based on an existing table.

```
CREATE TABLE MySourceTable
(
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(10) NOT NULL,
Col3 VARCHAR(10) NOT NULL
);
```

```
INSERT INTO MySourceTable
VALUES
(12, 'String12', 'String12'),
(25, 'String25', 'String25'),
(95, 'String95', 'String95');
```

```
SELECT  IDENTITY(INT, 100, 1) AS SurrogateKey,
        Col1,
        Col2,
        Col3
INTO MyNewTable
FROM MySourceTable
ORDER BY Col1 DESC;
```

```
SELECT *
FROM MyNewTable;
```

```
SurrogateKey   Col1    Col2       Col3
------------   ----    ----       ----
100            95      String95   String95
101            25      String25   String25
102            12      String12   String12
```

Use NEWSEQUENTIALID as a surrogate key for a new table.

```
CREATE TABLE MyTable
(
Col1 UNIQUEIDENTIFIER NOT NULL
 PRIMARY KEY NONCLUSTERED DEFAULT NEWSEQUENTIALID()
);
```

```
INSERT INTO MyTable
DEFAULT VALUES;
```

```
SELECT *
FROM MyTable;
```

```
Col1
----
9CC01320-C5AA-E811-8440-305B3A017068
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/sequence-numbers/sequence-numbers*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql-identity-property*

Migrate to Aurora MySQL Sequences and Identity

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Sequences and Identity	• SEQUENCE objects not supported • Rewrite IDENTITY to AUTO_INCREMENT • Last value evaluated as MAX(Existing Value) + 1 on every restart!

Overview

Aurora MySQL supports automatic sequence generation using the AUTO_INCREMENT column property, similar to SQL Server's IDENTITY column property.

Aurora MySQL does not support table-independent sequence objects.

Any numeric column may be assigned the AUTO_INCREMENT property. To make the system generate the next sequence value, the application must insert a value of NULL 0 into an indexed AUTO_INCREMENT column. In most cases, the seed value is 1 and the increment is 1.

Client applications use the LAST_INSERT_ID function to obtain the last generated value.

Each table can have only one AUTO_INCREMENT column. The column must be explicitly indexed or be a primary key (which is indexed by default).

The AUTO_INCREMENT mechanism is designed to be used with positive numbers only. Do not use negative values because they will be misinterpreted as a complementary positive value. This limitation is due to precision issues with sequences crossing a zero boundary.

There are two server parameters used to alter the default values for new AUTO_INCREMENT columns:

- **auto_increment_increment:** Controls the sequence interval.
- **auto_increment_offset:** Determines the starting point for the sequence.

To reseed the AUTO_INCREMENT value, use ALTER TABLE <Table Name> AUTO_INCREMENT = <New Seed Value>.

Syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <Table Name>
(<Column Name> <Data Type> [NOT NULL | NULL]
AUTO_INCREMENT [UNIQUE [KEY]] [[PRIMARY] KEY]...
```

Migration Considerations

Since Aurora MySQL does not support table-independent SEQUENCE objects, applications that rely on its properties must use a custom solution to meet their requirements.

Aurora MySQL AUTO_INCREMENT can be used instead of SQL Server's IDENTITY for most cases. For AUTO_INCREMENT columns, the application must explicitly INSERT a NULL or a 0.

> **Note:** Omitting the AUTO_INCREMENT column from the INSERT column list has the same effect as inserting a NULL value.

AUTO_INCREMENT columns must be indexed (the following section explains why) and can not have default constraints assigned to the same column. There is a critical difference between IDENTITY and AUTO_INCREMENT in the way the sequence values are maintained upon service restart. Application developers must be aware of this difference.

Sequence Value Initialization

SQL Server stores the IDENTITY metadata in system tables on disk. Although some values may be cached and lost when the service is restarted, the next time the server restarts, the sequence value continues after the last block of values that was assigned to cache. If you run out of values, you can explicitly set the sequence value to start the cycle over. As long as there are no key conflicts, it can be reused after the range has been exhausted.

In Aurora MySQL, an AUTO_INCREMENT column for a table uses a special counter called the auto-increment counter to assign new values for the column. This counter is stored in cache memory only and is not persisted to disk. After a service restart, and when Aurora MySQL encounters an INSERT to a table containing an AUTO_INCREMENT column, it issues an equivalent of the following statement:

```
SELECT MAX(<Auto Increment Column>) FROM <Table Name> FOR UPDATE;
```

> **Note:** The FOR UPDATE CLAUSE is required to maintain locks on the column until the read completes.

Aurora MySQL then increments the value retrieved by the statement above and assigns it to the in-memory auto-increment counter for the table. By default, the value is incremented by one. You can change the default using the auto_increment_increment configuration setting. If the table has no values, Aurora MySQL uses the value 1. You can change the default using the auto_increment_offset configuration setting.

Every server restart effectively cancels any AUTO_INCREMENT = <Value> table option in CREATE TABLE and ALTER TABLE statements.

Unlike SQL Server's IDENTITY columns, which by default do not allow inserting explicit values, Aurora MySQL allows explicit values to be set. If a row has an explicitly specified AUTO_INCREMENT column value and the value is greater than the current counter value, the counter is set to the specified column value.

Examples

Create a table with an AUTO_INCREMENT column.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL
 AUTO_INCREMENT PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Insert AUTO_INCREMENT Values.

```
INSERT INTO MyTable (Col2)
VALUES ('AI column omitted');
```

```
INSERT INTO MyTable (Col1, Col2)
VALUES (NULL, 'Explicit NULL');
```

```
INSERT INTO MyTable (Col1, Col2)
VALUES (10, 'Explicit value');
```

```
INSERT INTO MyTable (Col2)
VALUES ('Post explicit value');
```

```
SELECT *
FROM MyTable;
```

```
Col1    Col2
----    ------------
1       AI column omitted
2       Explicit NULL
10      Explicit value
11      Post explicit value
```

Reseed AUTO_INCREMENT.

```
ALTER TABLE MyTable AUTO_INCREMENT = 30;
```

```
INSERT INTO MyTable (Col2)
VALUES ('Post ALTER TABLE');
```

```
SELECT *
FROM MyTable;
```

```
Col1    Col2
----    ------------
1       AI column omitted
2       Explicit NULL
10      Explicit value
11      Post explicit value
30      Post ALTER TABLE
```

Change the increment value to 10.

> **Note:** Changing the @@auto_increment_increment value to 10 impacts all AUTO_INCREMENT enumerators in the database.

```
SET @@auto_increment_increment=10;
```

Verify variable change.

```
SHOW VARIABLES LIKE 'auto_inc%';
```

```
Variable_name              Value
--------------             -----
auto_increment_increment   10
auto_increment_offset      1
```

Insert several rows and then read.

```
INSERT INTO MyTable (Col1, Col2)
VALUES (NULL, 'Row1'), (NULL, 'Row2'), (NULL, 'Row3'), (NULL, 'Row4');
```

```
SELECT Col1, Col2
FROM MyTable;
```

```
Col1   Col2
-----   -----
1       AI column omitted
2       Explicit NULL
10      Explicit value
11      Post explicit value
30      Post ALTER TABLE
40      Row1
50      Row2
60      Row3
70      Row4
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
independent SEQUENCE object	CREATE SEQUENCE	Not supported	
Automatic enumerator column property	IDENTITY	AUTO_INCREMENT	
Reseed sequence value	DBCC CHECKIDENT	ALTER TABLE <Table Name> AUTO_ INCREMENT = <New Seed Value>	
Column restrictions	Numeric	Numeric, indexed, and no DEFAULT	
Controlling seed and	CREATE/ALTER TABLE	auto_increment_incre-	Aurora MySQL settings

Feature	SQL Server	Aurora MySQL	Comments
interval values		ment auto_increment_offset	are global and cannot be customized for each column as in SQL Server.
Sequence setting initialization	Maintained through service restarts	Re-initialized every service restart	See the **Sequence Value Initialization** section above.
Explicit values to column	not allowed by default, SET IDENTITY_INSERT ON required	Supported	Aurora MySQL requires explicit NULL or 0 to trigger sequence value assignment. Inserting an explicit value larger than all others will reinitialize the sequence.
Non PK auto enumerator column	Supported	Not Supported	Implement an application enumerator
Compound PK with auto enumerator column	Supported	Not Supported	Implement an application enumerator

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/example-auto-increment.html*
- *https://dev.mysql.com/doc/refman/5.7/en/create-table.html*
- *https://dev.mysql.com/doc/refman/5.7/en/innodb-auto-increment-handling.html#innodb-auto-increment-initialization*

Migrate from SQL Server Managing Statistics

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Statistics contain only density information, and only for index key columns

Overview

Statistics objects in SQL Server are designed to support SQL Server's cost-based query optimizer. It uses statistics to evaluate the various plan options and choose an optimal plan for optimal query performance.

Statistics are stored as BLOBs in system tables and contain histograms and other statistical information about the distribution of values in one or more columns. A histogram is created for the first column only and samples the occurrence frequency of distinct values. Statistics and histograms are collected by either scanning the entire table or by sampling only a percentage of the rows.

You can view Statistics manually using the DBCC SHOW_STATISTICS statement or the more recent sys.dm_db_stats_properties and sys.dm_db_stats_histogram system views.

SQL Server provides the capability to create filtered statistics containing a WHERE predicate. Filtered statistics are useful for optimizing histogram granularity by eliminating rows whose values are of less interest, for example NULLs.

SQL Server can manage the collection and refresh of statistics automatically, which is the default. Use the AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS database options to change the defaults.

When a query is submitted with AUTO_CREATE_STATISTICS on, and the query optimizer may benefit from a statistics that does not yet exist, SQL Server creates the statistics automatically. You can use the AUTO_UPDATE_STATISTICS_ASYNC database property to set new statistics creation to occur immediately (causing queries to wait) or to run asynchronously. When run asynchronously, the triggering execution cannot benefit from optimizations the optimizer may derive from it.

After creation of a new statistics object, either automatically or explicitly using the CREATE STATISTICS statement, the refresh of the statistics is controlled by the AUTO_UPDATE_STATISTICS database option. When set to ON, statistics are recalculated when they are stale, which happens when significant data modifications have occurred since the last refresh.

Syntax

```
CREATE STATISTICS <Statistics Name>
ON <Table Name> (<Column> [,...])
[WHERE <Filter Predicate>]
[WITH <Statistics Options>;
```

Examples

Create new statistics on multiple columns. Set to use a full scan and to not refresh.

```
CREATE STATISTICS MyStatistics
ON MyTable (Col1, Col2)
WITH FULLSCAN, NORECOMPUTE;
```

Update statistics with a 50% sampling rate.

```
UPDATE STATISTICS MyTable(MyStatistics)
WITH SAMPLE 50 PERCENT;
```

View the statistics histogram and data.

```
DBCC SHOW_STATISTICS ('MyTable','MyStatistics');
```

Turn off automatic statistics creation for a database.

```
ALTER DATABASE MyDB SET AUTO_CREATE_STATS OFF;
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-statistics-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-show-statistics-transact-sql*

Migrate to Aurora MySQL Managing Statistics

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Statistics contain only density information, and only for index key columns

Overview

Aurora MySQL supports two modes of statistics management: Persistent Optimizer Statistics and Non-Persistent Optimizer Statistics. As the name suggests, persistent statistics are written to disk and survive service restart. Non-persistent statistics are kept in memory only and need to be recreated after service restart. It is recommended to use persistent optimizer statistics (the default for Aurora MySQL) for improved plan stability.

Statistics in Aurora MySQL are created for indexes only. Aurora MySQL does not support independent statistics objects on columns that are not part of an index.

Typically, administrators change the statistics management mode by setting the global parameter innodb_stats_persistent = ON. This option is not supported for Aurora MySQL because it requires server SUPER privileges. Therefore, control the statistics management mode by changing the behavior for individual tables using the table option STATS_PERSISTENT = 1. There are no column-level or statistics-level options for setting parameter values.

To view statistics metadata, use the INFORMATION_SCHEMA.STATISTICS standard view. To view detailed persistent optimizer statistics, use the innodb_table_stats and innodb_index_stats views.

Automatic refresh of statistics is controlled by the global parameter innodb_stats_auto_recalc, which is set to ON in Aurora MySQL. You can set it individually for each table using the STATS_AUTO_RECALC=1 option.

To explicitly force refresh of table statistics, use the ANALYZE TABLE statement. It is not possible to refresh individual statistics or columns.

Use the NO_WRITE_TO_BINLOG or its clearer alias LOCAL to avoid replication to replication slaves.

Use ALTER TABLE ... ANALYZE PARTITION to analyze one or more individual partitions.
For more information, see Partitioning.

> **Note:** Execute ANALYZE TABLE after every CREATE INDEX when the persistent statistics mode is enabled to ensure statistics are collected.

Syntax

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE <Table Name> [,...];
```

```
CREATE TABLE ( <Table Definition> ) | ALTER TABLE <Table Name>
STATS_PERSISTENT = <1|0>,
```

```
STATS_AUTO_RECALC = <1|0>,
STATS_SAMPLE_PAGES = <Statistics Sampling Size>;
```

Migration Considerations

Unlike SQL Server, Aurora MySQL collects only density information. It does not collect detailed key distribution histograms. This difference is critical for understanding execution plans and troubleshooting performance issues, which are not affected by individual values used by query parameters.

Statistics collection is managed at the table level. You cannot manage individual statistics objects or individual columns. In most cases, that should not pose a challenge for successful migration.

Examples

Create a table with explicitly set statistics options.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL AUTO_INCREMENT,
Col2 VARCHAR(255),
DateCol DATETIME,
PRIMARY KEY  (Col1),
INDEX IDX_DATE (DateCol)
) ENGINE=InnoDB,
STATS_PERSISTENT=1,
STATS_AUTO_RECALC=1,
STATS_SAMPLE_PAGES=25;
```

Refresh all statistics for MyTable1 and MyTable2.

```
ANALYZE TABLE MyTable1, MyTable2;
```

Change MyTable to use non persistent statistics.

```
ALTER TABLE MyTable STATS_PERSISTENT=0;
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
Column statistics	CREATE STATISTICS	N/A	
Index statistics	Implicit with every index	Implicit with every index	Statistics are maintained automatically for every table index.
Refresh / update statistics	UPDATE STATISTICS EXECUTE sp_updatestats	ANALYZE TABLE	Minimal scope in Aurora MySQL is the entire table. No control over individual statistics.

Feature	SQL Server	Aurora MySQL	Comments
Auto create statistics	AUTO_CREATE_ STATISTICS data-base option	N/A	
Auto update statistics	AUTO_UPDATE_ STATISTICS data-base option	STATS_AUTO_ RECALC table option	
Statistics sampling	Use the SAMPLE option of CREATE and UPDATE STATISTICS	STATS_SAMPLE_ PAGES table option	Can only use page number, not percentage for STATS_SAMPLE_ PAGES.
Full scan refresh	Use the FULLSCAN option of CREATE and UPDATE STATISTICS	N/A	Using a very large STATS_ SAMPLE_PAGES may server the same purpose.
Non-persistent stat-istics	N/A	Use STATS_ PERSISTENT=0 table option	

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/statistics-table.html*
- *https://dev.mysql.com/doc/refman/5.7/en/innodb-persistent-stats.html*
- *https://dev.mysql.com/doc/refman/5.7/en/innodb-statistics-estimation.html*
- *https://dev.mysql.com/doc/refman/5.7/en/innodb-performance-optimizer-statistics.html*

Configuration

Migrate from SQL Server Session Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• SET options are significantly different, except for transaction isolation control

Overview

Session Options in SQL Server is a collection of run-time settings that control certain aspects of how the server handles data for individual sessions. A session is the period between a login event and a disconnect event (or an exec sp_reset_connection command for connection pooling).

Each session may have multiple execution scopes, which are all the statements before the GO keyword used in SQL Server management Studio scripts, or any set of commands sent as a single execution batch by a client application. Each execution scope may contain additional sub-scopes. For example, scripts calling stored procedures or functions.

You can set the global session options, which all execution scopes use by default, using the SET T-SQL command. Server code modules such as stored procedures and functions may have their own execution context settings, which are saved along with the code to guarantee the validity of results.

Developers can explicitly use SET commands to change the default settings for any session or for an execution scope within the session. Typically, client applications send explicit SET commands upon connection initiation.

You can view the metadata for current sessions using the sp_who_system stored procedure and the sysprocesses system table.

> **Note:** To change the default setting for SQL Server Management Studio, click **Tools >Options > Query Execution > SQL Server > Advanced**.

Syntax

Syntax for the SET command:

```
SET
Category        Setting
-------------   ----------
Date and time   DATEFIRST | DATEFORMAT
Locking         DEADLOCK_PRIORITY | SET LOCK_TIMEOUT
Miscellaneous   CONCAT_NULL_YIELDS_NULL | CURSOR_CLOSE_ON_COMMIT | FIPS_FLAGGER | SET
IDENTITY_INSERT
                LANGUAGE | OFFSETS | QUOTED_IDENTIFIER
Query Execution ARITHABORT | ARITHIGNORE | FMTONLY | NOCOUNT | NOEXEC | NUMERIC_
ROUNDABORT | PARSEONLY
                QUERY_GOVERNOR_COST_LIMIT | ROWCOUNT | TEXTSIZE
ANSI            ANSI_DEFAULTS | ANSI_NULL_DFLT_OFF | ANSI_NULL_DFLT_ON | ANSI_NULLS |
```

```
ANSI_PADDING
                ANSI_WARNINGS
Execution Stats FORCEPLAN | SHOWPLAN_ALL | SHOWPLAN_TEXT | SHOWPLAN_XML | STATISTICS
IO | STATISTICS XML
                STATISTICS PROFILE | STATISTICS TIME
Transactions    IMPLICIT_TRANSACTIONS | REMOTE_PROC_TRANSACTIONS | TRANSACTION ISOLATION
LEVEL | XACT_ABORT
```

Note: For more details about individual settings, see the link at the end of this section.

SET ROWCOUNT for DML Deprecated Setting

The SET ROWCOUNT for DML statements has been deprecated as of SQL Server 2008 in accordance with https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms143729(v=sql.105).

Up to and including SQL Server 2008 R2, you could limit the amount of rows affected by INSERT, UPDATE, and DELETE operations using SET ROWCOUNT. For example, it is a common practice in SQL Server to batch large DELETE or UPDATE operations to avoid transaction logging issues. The following example loops and deletes rows having 'ForDelete' set to 1, but only 5000 rows at a time in separate transactions (assuming the loop is not within an explicit transaction).

```
SET ROWCOUNT 5000;
WHILE @@ROWCOUNT > 0
BEGIN
 DELETE FROM MyTable
 WHERE ForDelete = 1;
END
```

Begining with SQL Server 2012, SET ROWCOUNT is ignored for INSERT, UPDATE and DELETE statements. The same functionality can be achieved by using TOP, which can be converted to Aurora MySQL's LIMIT.

For example, the previous code could be rewritten as:

```
WHILE @@ROWCOUNT > 0
BEGIN
 DELETE TOP (5000)
 FROM MyTable
 WHERE ForDelete = 1;
END
```

The latter syntax can be converted automatically by SCT to Aurora MySQL. See the code example in Aurora MySQL Session Options.

Examples

Use SET within a stored procedure.

```
CREATE PROCEDURE <ProcedureName>
AS
BEGIN
  <Some non critical transaction code>
  SET TRANSACTION_ISOLATION_LEVEL SERIALIZABLE;
  SET XACT_ABORT ON;
```

```
  <Some critical transaction code>
END
```

Note: Explicit SET commands affect their execution scope and sub scopes.
After the scope terminates and the procedure code exits, the calling scope resumes its original settings used before the calling the stored procedure.

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/set-statements-transact-sql

Migrate to Aurora MySQL Session Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• SET options are significantly different, except for transaction isolation control

Overview

Aurora MySQL supports hundreds of Server System Variables to control server behavior and the global and session levels.

Use the SHOW VARIABLES command to view a list of all variables.

```
SHOW SESSION VARIABLES;
-- 532 rows returned
```

> **Note:** Aurora MySQL 5.7 provides additional variables that do not exist in MySQL 5.7 standalone installations. These variables are prefixed with *aurora* or *aws* .

You can view Aurora MySQL variables using the MySQL command line utility, Aurora database cluster parameters, Aurora database instance parameters, or SQL interface system variables.

To view all sessions, use the SHOW PROCESSLIST command or the information_schema PROCESSLIST view, which displays information such as session current status, default database, host name, and application name.

> **Note:** Unlike standalone installations of MySQL, Amazon Aurora does not provide access to the configuration file containing system variable defaults. Cluster-level parameters are managed in database cluster parameter groups and instance-level parameters are managed in database parameter groups. In Aurora MySQL, some parameters from the full base set of standalone MySQL installations can not be modified and others were removed. See Server Options for a walkthrough of creating a custom parameter group.

Converting from SQL Server 2008 SET ROWCOUNT for DML operations

As mentioned in SQL Server Sessions Options, the use of SET ROWCOUNT for DML operations is deprecated as of SQL Server 2008 R2. Code that uses the SET ROWCOUNT syntax can not be converted automatically. Either rewrite to use TOP before running SCT, or manually change it afterward.

The example used to batch DELETE operations in SQL Server using TOP:

```
WHILE @@ROWCOUNT > 0
BEGIN
 DELETE TOP (5000)
 FROM MyTable
 WHERE ForDelete = 1;
END
```

can be easily rewritten to use Aurora MySQL LIMIT clause :

```
WHILE row_count() > 0
DO
 DELETE
 FROM MyTable
 WHERE ForDelete = 1
 LIMIT 5000;
END WHILE;
```

Examples

View the metadata for all processes.

```
SELECT *
FROM information_schema.PROCESSLIST;
```

```
SHOW PROCESSLIST;
```

Use the SET command to change session isolation level and SQL mode.

```
SET sql_mode = 'ANSI_QUOTES';
SET SESSION TRANSACTION ISOLATION LEVEL 'READ-COMMITTED';
```

Set isolation level using a system variable.

```
SET SESSION tx_isolation = 'READ-COMMITTED'
```

The SET SESSION command is the equivalent to the SET command in T-SQL.
However, there are far more configurable parameters in Aurora MySQL than in SQL Server.

Summary

The following table summarizes commonly used SQL Server session options and their corresponding Aurora MySQL system variables.

Category	SQL Server	Aurora MySQL	Comments
Date and time	DATEFIRST DATEFORMAT	default_week_format date_format (deprecated)	default_week_format opertates different than DATEFIRST; it allows only Sunday and Monday as start weeks. It also controls what is considered week one of the year and whether returned WEEK value is zero- based, or one-based. There is no alternative to the deprecated date_format variable (see Date and Time Functions.

Category	SQL Server	Aurora MySQL	Comments
Locking	LOCK_TIMEOUT	lock_wait_timeout	Set in database parameter groups.
ANSI	ANSI_NULLS ANSI_PADDING	N/A PAD_CHAR_TO_FULL_LENGTH	Set with the sql_mode system variable.
Transactions	IMPLICIT_TRANSACTIONS TRANSACTION ISOLATION LEVEL	autocommit SET SESSION TRANSACTION ISOLATION LEVEL	The default for Aurora MySQL, as in SQL server, is to commit automatically. Syntax is compatible except the addition of the SESSION keyword.
Query execution	IDENTITY_INSERT LANGUAGE QUOTED_IDENTIFIER NOCOUNT	See Identity and sequences lc_time_names ANSI_QUOTES N/A and not needed	lc_time_names are set in a database parameter group. lc_messages is not supported in Aurora MySQL. ANSI_QUOTES is a value for the sql_mode parameter. Aurora MySQL does not add row count information to the errors collection.
Execution stats	SHOWPLAN_ALL, TEXT, and XML STATISTICS IO, XML, PROFILE, and TIME	See Execution Plans	
Miscellaneous	CONCAT_NULL_YIELDS_NULL ROWCOUNT	N/A sql_select_limit	Aurora MySQL always returns NULL for any NULL concat operation. sql_select_limit only affects SELECT statements unlike ROWCOUNT, which also affects all DML.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

Migrate from SQL Server Database Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• SQL Server's database options are inapplicable to Aurora MySQL

Overview

SQL Server provides database level options that can be set using the ALTER DATABASE ... SET command. These settings enable you to:

- Set default session options. For more information, see Session Options.
- Enable or disable database features such as SNAPSHOT_ISOLATION, CHANGE_TRANCKING, and ENABLE_BROKER.
- Configure High availability and disaster recovery options such as always on availability groups
- Configure security access control such as restricting access to a single user, setting the database offline, or setting the database to read-only.

Syntax

Syntax for setting database options:

```
ALTER DATABASE { <database name> } SET { <option> [ ,...n ] };
```

Examples

Set a database to read-only and use ARITHABORT by default.

```
ALTER DATABASE Demo SET READ_ONLY, ARITHABORT ON;
```

Set a database to use automatic statistic creation.

```
ALTER DATABASE Demo SET AUTO_CREATE_STATISTICS ON;
```

Set a database offline immediately.

```
ALTER DATABASE DEMO SET OFFLINE WITH ROLLBACK IMMEDIATE;
```

For more information, see https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql-set-options

Migrate to Aurora MySQL Database Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• SQL Server's database options are inapplicable to Aurora MySQL

Overview

The concept of a database in Aurora MySQL is different than SQL Server. In Aurora MySQL, a database is synonymous with a schema. Therefore, the notion of database options is not applicable to Aurora MySQL.

> **Note:** Aurora MySQL has two settings that are saved with the database/schema: the default character set, and the default collation for creating new objects.

Migration Considerations

For migration considerations, see Server Options.

Migrate from SQL Server Server Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Cluster and Database Parameter Groups

Overview

SQL Server provides server-level settings that affect all databases and all sessions. You can modify these settings using the sp_configure system stored procedure.

You can use Server Options to perform the following configuration tasks:

- Define hardware utilization such as memory management, affinity mask, priority boost, network packet size, and soft Non-Uniform Memory Access (NUMA) .
- Alter run time global values such as recovery interval, remote login timeout, optimization for ad-hoc workloads, and cost threshold for parallelism.
- Enable and disable global features such as C2 Audit, OLE, procedures, CLR procedures, and allow trigger recursion.
- Configure global security settings such as server authentication mode, remote access, shell access with xp_cmdshell, CLR access level, and database chaining.
- Set default values for sessions such as user options, default language, backup compression, and fill factor.

Some settings require an explicit RECONFIGURE command to apply the changes to the server. High risk settings require RECONFIGURE WITH OVERRIDE for the changes to be applied. Some advanced options are hidden by default. To view and modify these settings, set *show advanced options* to *1* and re-execute sp_configure.

> **Note:** Server audits are managed via the T-SQL commands CREATE and ALTER SERVER AUDIT.

Syntax

```
EXECUTE sp_configure <option>, <value>;
```

Examples

Limit server memory usage to 4GB.

```
EXECUTE sp_configure 'show advanced options', 1;
```

```
RECONFIGURE;
```

```
sp_configure 'max server memory', 4096;
```

```
RECONFIGURE;
```

Allow command shell access from T-SQL.

```
EXEC sp_configure 'show advanced options', 1;
```

```
RECONFIGURE;
```

```
EXEC sp_configure 'xp_cmdshell', 1;
```

```
RECONFIGURE;
```

Viewing current values.

```
EXECUTE sp_configure
```

For more information, see https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/server-configuration-options-sql-server

Migrate to Aurora MySQL Server Options

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Cluster and Database Parameter Groups

Overview

The concept of an database in Aurora MyQL is different than SQL Server. For Aurora MySQL, the terms database and schema are synonymous. Therefore, the concept of database options does is not applicable to Aurora MySQL.

The Aurora MySQL equivalent of SQL Server's database and server options are Server System Variables, which are run time settings you can modify using one of the following approaches:

- MySQL command line utility
- Aurora DB Cluster and DB Instance Parameters
- System variables used by the SQL SET command

Compared to SQL Server, Aurora MySQL provides a much wider range of server settings and configurations.
For a full list of the options available in Aurora MySQL, see the links at the end of this section. The Aurora MySQL default parameter group lists more than 250 different parameters.

> **Note:** Unlike stand alone installations of MySQL, Amazon Aurora does not provide file system access to the configuration file. Cluster-level parameters are managed in database cluster parameter groups. Instance-level parameters are managed in database parameter groups. Also, in Aurora MySQL some parameters from the full base set of standalone MySQL installations can not be modified and others were removed. Many parameters are viewable but not modifiable.

SQL Server and Aurora MySQL are completely different engines. Except for a few obvious settings such as max server memory which has an equivalent of innodb_buffer_pool_size, most of the Aurora MySQL parameter settings are not compatible with SQL Server.

In most cases, you should use the default parameter groups because they are optimized for common use cases.

Amazon Aurora is a cluster of DB instances and, as a direct result, some of the MySQL parameters apply to the entire cluster while other parameters apply only to particular database instances in the cluster. The following table describes how Aurora MySQL parameters are controlled:

Aurora MySQL Parameter Class	Controlled Via
Cluster-level parameters	Managed via cluster parameter groups For example:

Aurora MySQL Parameter Class	Controlled Via
Single cluster parameter group per Amazon Aurora Cluster	aurora_load_from_s3_role, default_password_lifetime, default_storage_engine
Database Instance-Level parameters Every instance in your Amazon Aurora cluster can be associated with a unique database parameter group	Managed via database parameter groups For example: autocommit, connect_timeout, innodb_change_buffer_max_size

Syntax

Server-level options are set with the SET GLOBAL command.

```
SET GLOBAL <option> = <Value>;
```

Examples

Modify Compression Level

Decrease compression level to reduce CPU usage.

```
SET GLOBAL innodb_compression_level = 5;
```

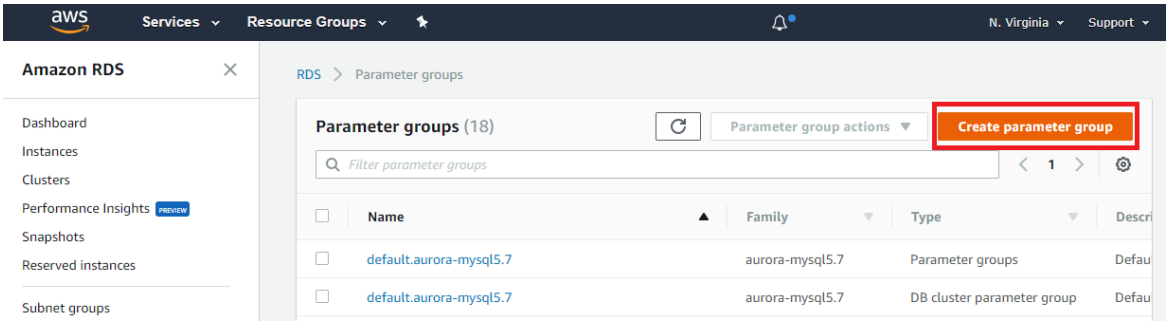
Create Parameter Groups

The following walkthrough demonstrates how to create and configure the Amazon Aurora database and cluster parameter groups:

Navigate to Parameter Group in the RDS Service of the AWS Console.

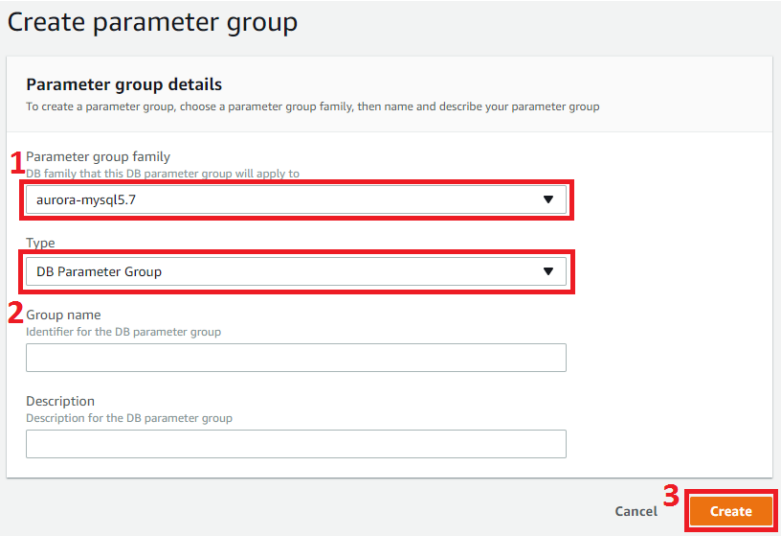
Click **Create Parameter Group**.

> **Note:** You cannot edit the default parameter group. Create a custom parameter group to apply changes to your Amazon Aurora cluster and its database instances.



On the new page:

- Select **aurora-mysql5.7** from the **Parameter group family** dropdown list.
- Select **DB Parameter Group** from the **Type** dropdown list. Another option is to select **Cluster Parameter Group** to modify cluster parameters.
- Click **Create**.



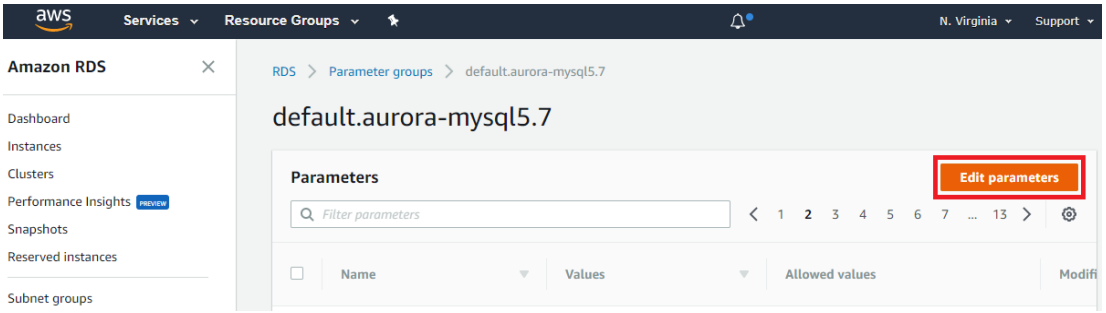
Modify a Parameter Group

The following walkthrough demonstrates how to modify an existing parameter group:

Navigate to the Parameter group section in the RDS Service of the AWS Console.

Click the name of the parameter group to edit.

On the new page, click the **Edit parameters** button.



Change parameter values and click **Save changes**.

For more information, see:

- *https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithParamGroups.html*
- *https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html*

High Availability and Disaster Recovery (HADR)

Migrate from SQL Server Backup and Restore

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Backup	• Storage level backup managed by Amazon RDS

Overview

The term *Backup* refers to both the process of copying data and to the resulting set of data created by the processes that copy data for safekeeping and disaster recovery. Backup processes copy SQL Server data and transaction logs to media such as tapes, network shares, cloud storage, or local files. These "backups" can then be copied back to the database using a *restore* process.

SQL Server uses files, or filegroups, to create backups for an individual database or subset of a database. Table backups are not supported.

When a database uses the FULL recovery model, transaction logs also need to be backed up. Transaction logs allow backing up only database changes since the last full backup and provide a mechanism for point-in-time restore operations.

Recovery Model is a database-level setting that controls transaction log management. The three available recovery models are SIMPLE, FULL, and BULK LOGGED. For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/recovery-models-sql-server.

The SQL Server RESTORE process copies data and log pages from a previously created backup back to the database. It then triggers a recovery process that rolls forward all committed transactions not yet flushed to the data pages when the backup took place. It also rolls back all uncommitted transactions written to the data files.

SQL Server supports the following types of backups:

- **Copy-Only Backups** are independent of the standard chain of SQL Server backups. They are typically used as "one-off" backups for special use cases and do not interrupt normal backup operations.
- **Data Backups** copy data files and the transaction log section of the activity during the backup. A Data Backup may contain the whole database (Database Backup) or part of the database. The parts can be a Partial Backup or a file/filegroup.
- A **Database Backup** is a Data Backup representing the entire database at the point in time when the backup process finished.
- A **Differential Backup** is a data backup containing only the data structures (extents) modified since the last full backup. A differential backup is dependent on the previous full backup and can not be used alone.

- A **Full Backup** is a data backup containing a Database Backup and the transaction log records of the activity during the backup process.
- **Transaction Log Backups** do not contain data pages. They contain the log pages for all transaction activity since the last Full Backup or the previous Transaction Log Backup.
- **File Backups** consist of one or more files or filegroups.

SQL Server also supports Media Families and Media Sets that can be used to mirror and stripe backup devices. For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/media-sets-media-families-and-backup-sets-sql-server

SQL Server 2008 Enterprise edition and later versions support Backup Compression. Backup Compression provides the benefit of a smaller backup file footprint, less I/O consumption, and less network traffic at the expense of increased CPU utilization for executing the compression algorithm. For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/-backup-compression-sql-server

A database backed up in the SIMPLE recovery mode can only be restored from a full or differential backup. For FULL and BULK LOGGED recovery models, transaction log backups can be restored also to minimize potential data loss.

Restoring a database involves maintaining a correct sequence of individual backup restores. For example, a typical restore operation may include the following steps:

1. Restore the most recent Full Backup.
2. Restore the most recent Differential Backup.
3. Restore a set of uninterrupted Transaction Log Backups, in order.
4. Recover the database.

For large databases, a full restore, or a complete database restore, from a full database backup is not always a practical solution. SQL Server supports Data File Restore that restores and recovers a set of files and a single Data Page Restore, except for databases using the SIMPLE recovery model.

Syntax

Backup syntax:

```
Backing Up a Whole Database
BACKUP DATABASE <Database Name> [ <Files / Filegroups> ] [ READ_WRITE_FILEGROUPS ]
  TO <Backup Devices>
  [ <MIRROR TO Clause> ]
  [ WITH [DIFFERENTIAL ]
  [ <Option List> ][;]
```

```
BACKUP LOG <Database Name>
  TO <Backup Devices>
  [ <MIRROR TO clause> ]
  [ WITH <Option List> ][;]
```

```
<Option List> =
COPY_ONLY | {COMPRESSION | NO_COMPRESSION } | DESCRIPTION = <Description>
```

```
| NAME = <Backup Set Name> | CREDENTIAL | ENCRYPTION | FILE_SNAPSHOT | { EXPIREDATE =
<Expiration Date> | RETAINDAYS = <Retention> }
{ NOINIT | INIT } | { NOSKIP | SKIP } | { NOFORMAT | FORMAT } |
{ NO_CHECKSUM | CHECKSUM } | { STOP_ON_ERROR | CONTINUE_AFTER_ERROR }
{ NORECOVERY | STANDBY = <Undo File for Log Shipping> } | NO_TRUNCATE
ENCRYPTION ( ALGORITHM = <Algorithm> | SERVER CERTIFICATE = <Certificate> | SERVER
ASYMMETRIC KEY = <Key> );
```

Restore Syntax:

```
RESTORE DATABASE <Database Name>  [ <Files / Filegroups> ] | PAGE = <Page ID>
FROM <Backup Devices>
[ WITH   [ RECOVERY | NORECOVERY | STANDBY = <Undo File for Log Shipping> } ]
[, <Option List>]
[;]
```

```
RESTORE LOG <Database Name>  [ <Files / Filegroups> ] | PAGE = <Page ID>
[ FROM <Backup Devices>
[ WITH   [ RECOVERY | NORECOVERY | STANDBY = <Undo File for Log Shipping> } ]
[, <Option List>]
[;]
```

```
<Option List> =
MOVE <File to Location>
| REPLACE | RESTART | RESTRICTED_USER | CREDENTIAL
| FILE = <File Number> | PASSWORD = <Passord>
| { CHECKSUM | NO_CHECKSUM } | { STOP_ON_ERROR | CONTINUE_AFTER_ERROR }
| KEEP_REPLICATION | KEEP_CDC
| { STOPAT = <Stop Time>
| STOPATMARK = <Log Sequence Number>
| STOPBEFOREMARK = <Log Sequence Number>
```

Examples

Perform a full compressed database backup.

```
BACKUP DATABASE MyDatabase TO DISK='C:\Backups\MyDatabase\FullBackup.bak'
WITH COMPRESSION;
```

Perform a log backup.

```
BACKUP DATABASE MyDatabase TO DISK='C:\Backups\MyDatabase\LogBackup.bak'
WITH COMPRESSION;
```

Perform a partial differential backup.

```
BACKUP DATABASE MyDatabase
   FILEGROUP = 'FileGroup1',
   FILEGROUP = 'FileGroup2'
   TO DISK='C:\Backups\MyDatabase\DB1.bak'
   WITH DIFFERENTIAL;
```

Restore a database to a point in time.

```
RESTORE DATABASE MyDatabase
   FROM DISK='C:\Backups\MyDatabase\FullBackup.bak'
   WITH NORECOVERY;

RESTORE LOG AdventureWorks2012
   FROM DISK='C:\Backups\MyDatabase\LogBackup.bak'
   WITH NORECOVERY, STOPAT = '20180401 10:35:00';

RESTORE DATABASE AdventureWorks2012 WITH RECOVERY;
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/backup-overview-sql-server*
- *https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/restore-and-recovery-overview-sql-server*

Migrate to Aurora MySQL Backup and Restore

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Backup	• Storage level backup managed by Amazon RDS

Overview

Aurora MySQL continuously backs up all cluster volumes and retains restore data for the duration of the backup retention period. The backups are incremental and can be used to restore the cluster to any point in time within the backup retention period. You can specify a backup retention period from one to 35 days when creating or modifying a database cluster. Backups incur no performance impact and do not cause service interruptions.

Additionally, you can manually trigger data snapshots in a cluster volume that can be saved beyond the retention period. You can use Snapshots to create new database clusters.

> **Note:** Manual snapshots incur storage charges for Amazon RDS.

Restoring Data

You can recover databases from Aurora's automatically retained data or from a manually saved snapshot. Using the automatically retained data significantly reduces the need to take frequent snapshots and maintain Recovery Point Objective (RPO) policies.

The RDS console displays the available time frame for restoring database instances in the Latest Restorable Time and Earliest Restorable Time fields. The Latest Restorable Time is typically within the last five minutes. The Earliest Restorable Time is the end of the backup retention period.

> **Note:** The Latest Restorable Time and Earliest Restorable Time fields display when a database cluster restore has been completed. Both display NULL until the restore process completes.

Database Cloning

Database cloning is a fast and cost-effective way to create copies of a database. You can create multiple clones from a single DB cluster and additional clones can be created from existing clones. When first created, a cloned database requires only minimal additional storage space.

Database cloning uses a copy-on-write protocol. Data is copied only when it changes either on the source or cloned database.

Data cloning is useful for avoiding impacts on production databases. For example:

- Testing schema or parameter group modifications.
- Isolating intensive workloads. For example, exporting large amounts of data and running high

resource-consuming queries.

- Development and testing with a copy of a production database.

Copying and Sharing Snapshots

Database snapshots can be copied and shared within the same AWS Region, across AWS Regions, and across AWS accounts. Snapshot sharing allows an authorized AWS account to access and copy snapshots. Authorized users can restore a snapshot from its current location without first copying it.

Copying an automated snapshot to another AWS account requires two steps:

1. Create a manual snapshot from the automated snapshot.
2. Copy the manual snapshot to another account.

Backup Storage

In all RDS regions, Backup Storage is the collection of both automated and manual snapshots for all database instances and clusters. The size of this storage is the sum of all individual instance snapshots.

When an Aurora MySQL database instance is deleted, all automated backups of that database instance are also deleted. However, Amazon RDS provides the option to create a final snapshot before deleting a database instance. This final snapshot is retained as a manual snapshot. Manual snapshots are not automatically deleted.

The Backup Retention Period

Retention periods for Aurora MySQL DB cluster backups are configured when creating a cluster. If not explicitly set, the default retention is one day when using the Amazon RDS API or the AWS CLI. The retention period is seven days if using the AWS Console. You can modify the backup retention period at any time with a value between one and 35 days.

Disabling Automated Backups

You cannot disable automated backups on Aurora MySQL. The backup retention period for Aurora MySQL is managed by the database cluster.

Saving Data from an Amazon Aurora MySQL Database to Amazon S3

Aurora MySQL supports a proprietary syntax for dumping and loading data directly from and to an Amazon S3 bucket.

The SELECT ... INTO OUTFILE S3 statement is used to export data out of Aurora MySQL, and its counterpart LOAD DATA FROM S3 statement is used for loading data directly from Amazon S3 text files.

> **Note:** This integration enables very efficient dumps since there is no need for an intermediate client application to handle the data export, import, and save.

The syntax for the SELECT ... INTO OUTFILE S3 statement is:

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
```

```
        [HIGH_PRIORITY]
        [STRAIGHT_JOIN]
        [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
        [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
        [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
        [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
         [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
INTO OUTFILE S3 'S3-URI'
[CHARACTER SET charset_name]
    [export_options]
    [MANIFEST {ON | OFF}]
    [OVERWRITE {ON | OFF}]

export_options:
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
```

The syntax for the LOAD DATA FROM S3 statement is:

```
LOAD DATA FROM S3 [FILE | PREFIX | MANIFEST] 'S3-URI'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name,...)]
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_user_var,...)]
    [SET col_name = expr,...]
```

For more information on loading data from S3, see https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/AuroraMySQL.Integrating.LoadFromS3.html.

As can be seen from the syntax, Aurora MySQL offers various options for easy control of saving and loading data directly from an SQL statement without needing to configure options or external services.

The MANIFEST option of the export allows you to create an accompanying JSON file that lists the text files created by the SELECT ... INTO OUTFILE S3 statement. Later, the LOAD DATA FROM S3 statement can use this manifest to load the data files back into the database tables.

Migration Considerations

Migrating from a self managed backup policy to a Platform as a Service (PaaS) environment such as Aurora MySQL is a complete paradigm shift. You no longer need to worry about transaction logs, file groups, disks running out of space, and purging old backups.

Amazon RDS provides guaranteed continuous backup with point-in-time restore up to 35 days.

Managing an SQL Server backup policy with similar RTO and RPO is a challenging task. With Aurora MySQL, all you need to do set is the retention period and take manual snapshots for special use cases.

Considerations for Exporting Data to S3

By default, each file created in an S3 bucket as a result of the export has a maximal size of 6GB. The system rolls over to a new file once this limit is exceeded. However, Aurora MySQL guarantees that rows will not span multiple files, and therefore slight variations from this max size are possible.

The SELECT ... INTO OUTFILE S3 statement is an atomic transaction. Large or complicated SELECT statements may take a significant amount of time to complete. In the event of an error, the statement rolls back and must be executed again. However, if some of the data has already been uploaded to the S3 bucket, it is not deleted as part of the rollback and you can use a differential approach to upload only the remaining data.

> **Note:** For exports larger than 25GB, it is recommended to split the SELECT ... INTO OUTFILE S3 statement into multiple, smaller batches.

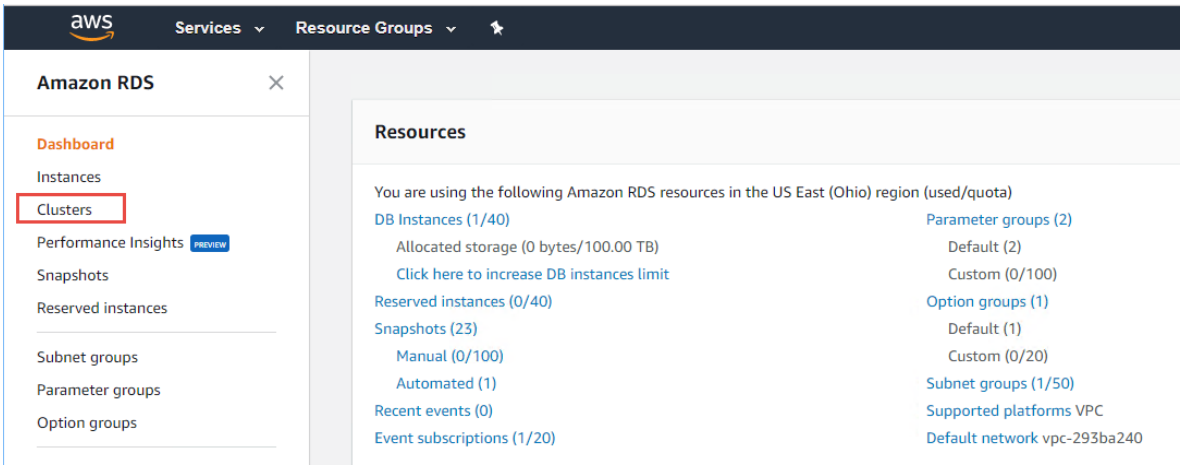
Metadata, such as table schema or file metadata, is not uploaded by Aurora MySQL to Amazon S3.

Examples

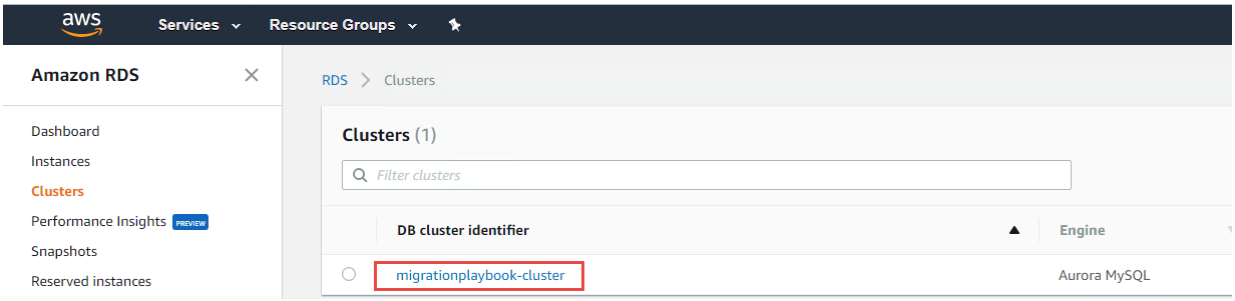
Change the Retention Policy to Seven Days

The following walkthrough describes how to change Aurora MySQL DB cluster retention settings from one day to seven days using the RDS console.

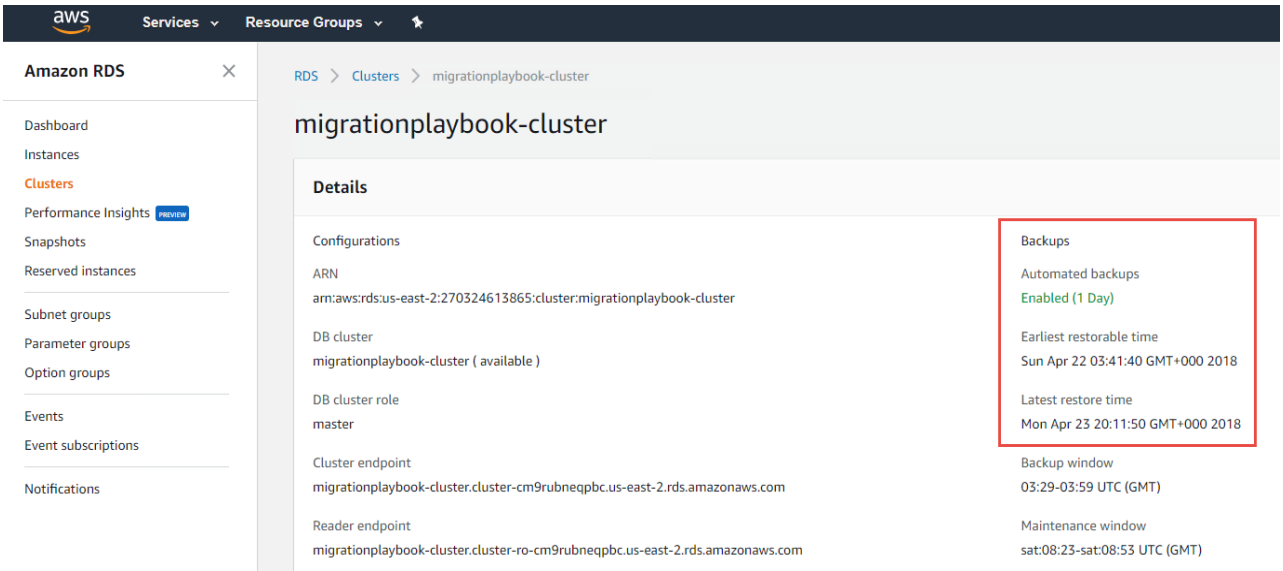
Login to the RDS Console and click **Clusters**.



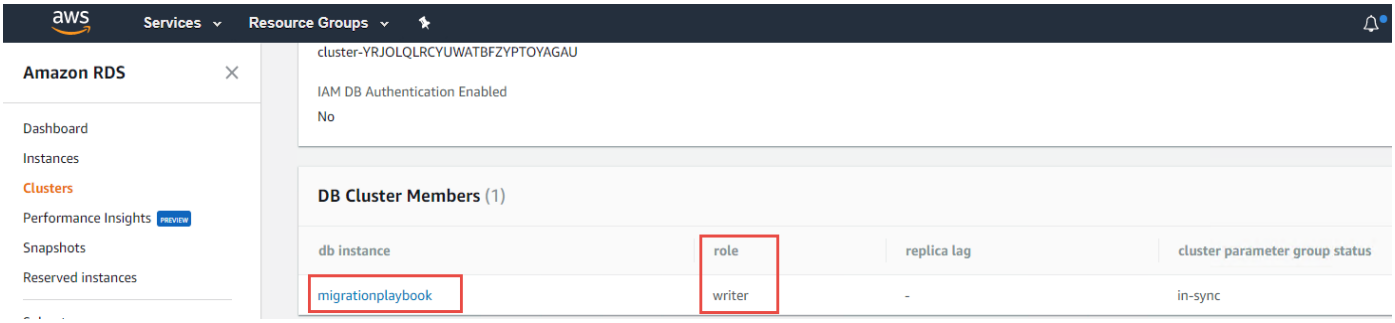
Click the **DB cluster identifier**.



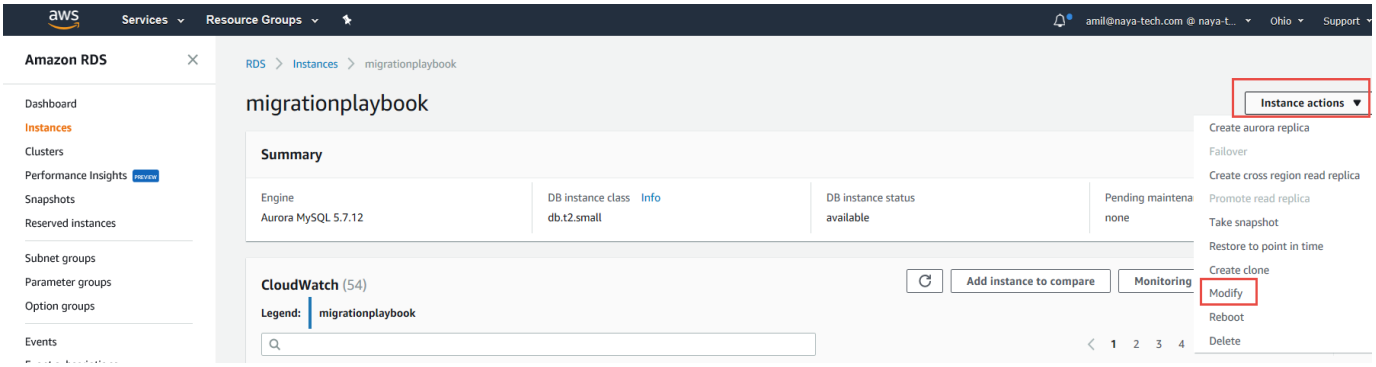
Verify the current automatic backup settings.



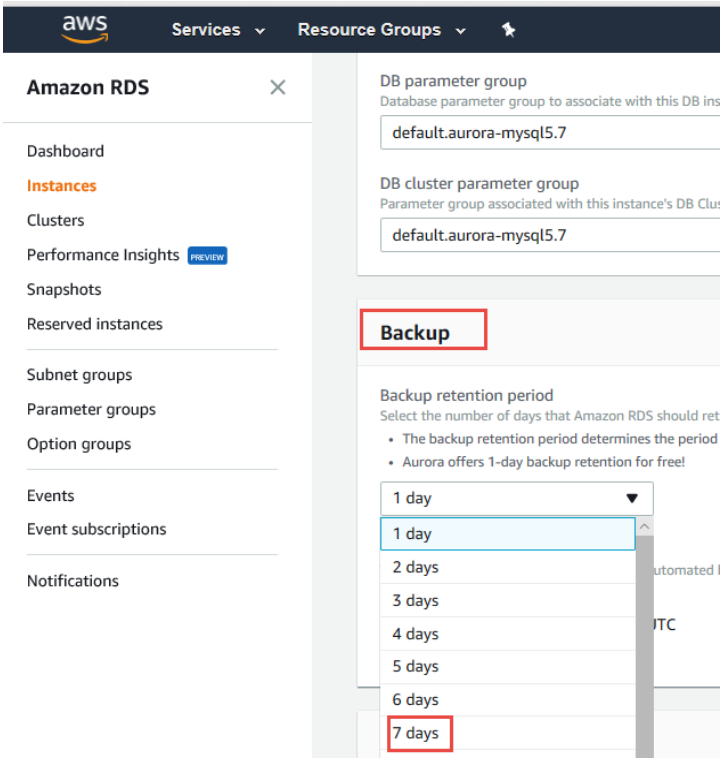
Scroll down to the **DB Cluster Members** section and click the database instance with the **writer** role.



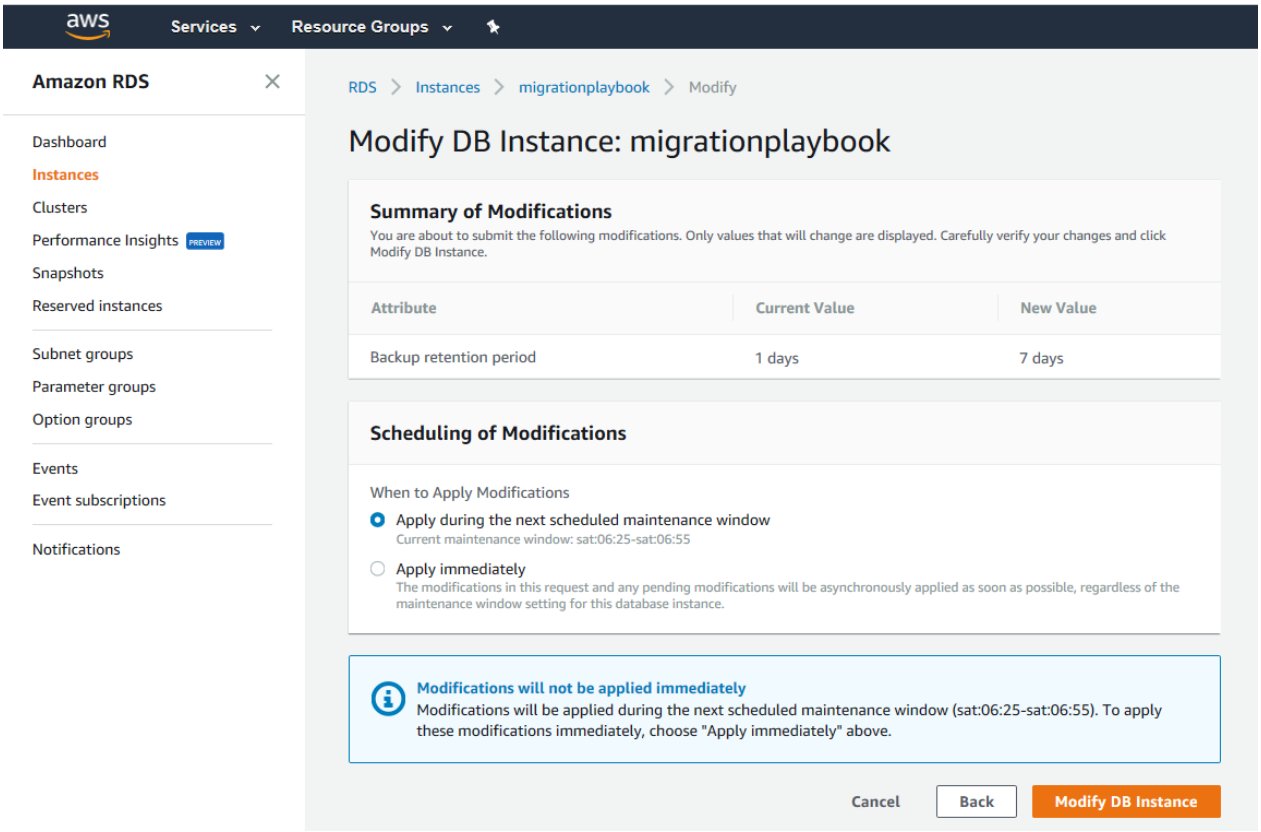
On the top left, click **Instance Actions > Modify**.



Scroll down to the **Backup** section. Select **7 Days** from the drop-down list.



Click **Continue**, review the summary, select a **Schedule of Modifications**, and click **Modify DB Instance**.



For more information and an example of creating a manual snapshot, see Maintenance Plans.

Exporting Data to Amazon S3

For a detailed example with all the necessary preliminary steps required to export data from Aurora MySQL to an Amazon S3 bucket, see https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/AuroraMySQL.Integrating.SaveIntoS3.html.

Summary

Feature	SQL Server	Aurora MySQL	Comments
Recovery Model	`SIMPLE, BULK LOGGED, FULL`	N/A	The functionality of Aurora MySQL backups is equivalent to the FULL recovery model.
Backup Database	`BACKUP DATABASE`	Automatic and continuous	
Partial Backup	`BACKUP DATABASE ... FILE= ... | FILEGROUP = ...`	N/A	
Log Backup	`BACKUP LOG`	N/A	Backup is at the storage level.

Feature	SQL Server	Aurora MySQL	Comments
Differential Backups	`BACKUP DATABASE ... WITH DIFFERENTIAL`	N/A	
Database Snapshots	`BACKUP DATABASE ... WITH COPY_ ONLY`	RDS console or API	The terminology is inconsistent between SQL Server and Aurora MySQL. A database snapshot in SQL Server is similar to database cloning in Aurora MySQL. Aurora MySQL database snapshots are similar to a COPY_ONLY backup in SQL Server.
Database Clones	`CREATE DATABASE... AS SNAPSHOT OF...`		The terminology is inconsistent between SQL Server and Aurora MySQL. A database snapshot in SQL Server is similar to database cloning in Aurora MySQL. Aurora MySQL database snapshots are similar to a COPY_ONLY backup in SQL Server.
Point in time restore	`RESTORE DATABASE | LOG ... WITH STOPAT...`	Any point within the retention period using RDS console or API	
Partial Restore	`RESTORE DATABASE... FILE= ... | FILEGROUP = ...`	N/A	
Export and import table data	DTS, SSIS, BCP, Linked Servers to files	`SELECT INTO ... OUTFILE S3` `LOAD DATA FROM S3`	

For more information, see

- https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Managing.html#Aurora.Managing.Backups
- https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/AuroraMySQL.Integrating.SaveIntoS3.html

Migrate from SQL Server High Availability Essentials

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Multi replica, scale out solution using Amazon Aurora clusters and Availability Zones

Overview

SQL Server provides several solutions to support high availability and disaster recovery requirements including Always On Failover Cluster Instances (FCI), Always On Availability Groups, Database Mirroring, and Log Shipping. The following sections describe each solution.

> **Note:** This section does not cover backup and restore. See Backup and Restore.

Always On Failover Cluster Instances (FCI)

Always On Failover Cluster Instances use the Windows Server Failover Clustering (WSFC) operating system framework to deliver redundancy at the server instance level.

An FCI is an instance of SQL Server installed across two or more WSFC nodes. For client applications, the FCI is transparent and appears to be a normal instance of SQL Server running on a single server. The FCI provides failover protection by moving the services from one WSFC node Windows server to another WSFC node windows server in the event the current "active" node becomes unavailable or degraded.

FCIs target scenarios where a server fails due to a hardware malfunction or a software hangup. Without FCI, a significant hardware or software failure would render the service unavailable until the malfunction is corrected. With FCI, another server can be configured as a "stand by" to replace the original serverif it stops servicing requests.

For each service or cluster resource, there is only one node that actively services client requests (known as "owning a resource group"). A monitoring agent constantly monitors the resource owners and can transfer ownership to another node in the event of a failure or planned maintenance such as installing service packs or security patches. This process is completely transparent to the client application, which may continue to submit requests as normal when the failover or ownership transfer process completes.

FCI can significantly minimize downtime due to hardware or software general failures. The main benefits of FCI are:

- Full instance level protection.
- Automatic failover of resources from one node to another.
- Supports a wide range of storage solutions. WSFC cluster disks can be iSCSI, Fiber Channel, SMB file shares, and others.

- Supports multi-subnet.
- No need client application configuration after a failover.
- Configurable failover policies.
- Automatic health detection and monitoring.

For more information, see https://docs.microsoft.com/en-us/sql/sql-server/failover-clusters/windows/always-on-failover-cluster-instances-sql-server

Always On Availability Groups

Always On Availability Groups is the most recent high availability and disaster recovery solution for SQL Server. It was introduced in SQL Server 2012 and supports high availability for one or more user databases. Because it can be configured and managed at the database level rather than the entire server, it provides much more control and functionality. As with FCI, Always On Availability Groups relies on the framework services of WSFC nodes.

Always On Availability Groups utilize real-time log record delivery and apply mechanism to maintain near real-time, readable copies of one or more databases.
These copies can also be used as redundant copies for resource usage distribution between servers (a scale-out read solution).

The main characteristics of Always On Availability Groups are:

- Supports up to nine availability replicas: One primary replica and up to eight secondary readable replicas.
- Supports both asynchronous-commit and synchronous-commit availability modes.
- Supports automatic failover, manual failover, and a forced failover. Only the latter can result in data loss.
- Secondary replicas allow both read-only access and offloading of backups.
- Availability Group Listener may be configured for each availability group. It acts as a virtual server address where applications can submit queries. The listener may route requests to a read-only replica or to the primary replica for read-write operations. This configuration also facilitates fast failover as client applications do not need to be reconfigured post failover.
- Flexible failover policies.
- The automatic page repair feature protects against page corruption.
- Log transport framework uses encrypted and compressed channels.
- Rich tooling and APIs including Transact-SQL DDL statements, management studio wizards, Always On Dashboard Monitor, and Powershell scripting.

For more information, see https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/always-on-availability-groups-sql-server

Database Mirroring.

> **Note:** Microsoft recommends avoiding Database Mirroring for new development. This feature is deprecated and will be removed in a future release. It is recommended to use Always On Availability Groups instead.

Database mirroring is a legacy solution to increase database availability by supporting near instantaneous failover. It is similar in concept to Always On Availability Groups, but can only be configured for one database at a time and with only one "standby" replica.

For more information, see https://docs.microsoft.com/en-us/sql/database-engine/database-mirroring/database-mirroring-sql-server

Log Shipping

Log shipping is one of the oldest and well tested high availability solutions. It is configured at the database level similar to Always On Availability Groups and Database Mirroring. Log shipping can be used to maintain one or more standby (secondary) databases for a single master (primary) database.

The Log shipping process involves three steps:

1. Backing up the transaction log of the primary database instance.
2. Copying the transaction log backup file to a secondary server.
3. Restoring the transaction log backup to apply changes to the secondary database.

Log shipping can be configured to create multiple secondary database replicas by repeating steps 2 and 3 above for each secondary server. Unlike FCI and Always On Availability Groups, log shipping solutions do not provide automatic failover.

In the event the primary database becomes unavailable or unusable for any reason, an administrator must configure the secondary database to serve as the primary and potentially reconfigure all client applications to connect to the new database.

> **Note:** Secondary databases can be used for read-only access, but require special handling. For more information, see https://docs.microsoft.com/en-us/sql/database-engine/log-shipping/configure-log-shipping-sql-server

The main characteristics of Log Shipping solutions are:

- Provides sredundancy for a single primary database and one or more secondary databases. Log Shipping is considered less of a high availability solution due to the lack of automatic failover.
- Supports limited read-only access to secondary databases.
- Administrators have control over the timing and delays of the primary server log backup and secondary server restoration.
- Longer delays can be useful if data is accidentally modified or deleted in the primary database.

For more information about log shipping, see https://docs.microsoft.com/en-us/sql/database-engine/log-shipping/about-log-shipping-sql-server

Examples

Configure an Always On Availability Group.

```
CREATE DATABASE DB1;
```

```
ALTER DATABASE DB1 SET RECOVERY FULL;
```

```
BACKUP DATABASE DB1 TO DISK = N'\\MyBackupShare\DB1\DB1.bak' WITH FORMAT;
```

```
CREATE ENDPOINT DBHA STATE=STARTED
AS TCP (LISTENER_PORT=7022) FOR DATABASE_MIRRORING (ROLE=ALL);
```

```
CREATE AVAILABILITY GROUP AG_DB1
   FOR
      DATABASE DB1
   REPLICA ON
      'SecondarySQL' WITH
         (
         ENDPOINT_URL = 'TCP://SecondarySQL.MyDomain.com:7022',
         AVAILABILITY_MODE = ASYNCHRONOUS_COMMIT,
         FAILOVER_MODE = MANUAL
         );
```

```
-- On SecondarySQL
ALTER AVAILABILITY GROUP AG_DB1 JOIN;
```

```
RESTORE DATABASE DB1 FROM DISK = N'\\MyBackupShare\DB1\DB1.bak'
WITH NORECOVERY;
```

```
-- On Primary
BACKUP LOG DB1
TO DISK = N'\\MyBackupShare\DB1\DB1_Tran.bak'
    WITH NOFORMAT
```

```
-- On SecondarySQL
RESTORE LOG DB1
    FROM DISK = N'\\MyBackupShare\DB1\DB1_Tran.bak'
    WITH NORECOVERY
```

```
ALTER DATABASE MyDb1 SET HADR AVAILABILITY GROUP = MyAG;
```

For more information, see https://docs.microsoft.com/en-us/sql/sql-server/failover-clusters/high-availability-solutions-sql-server

Migrate to Aurora MySQL High Availability Essentials

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Multi replica, scale out solution using Amazon Aurora clusters and Availability Zones

Overview

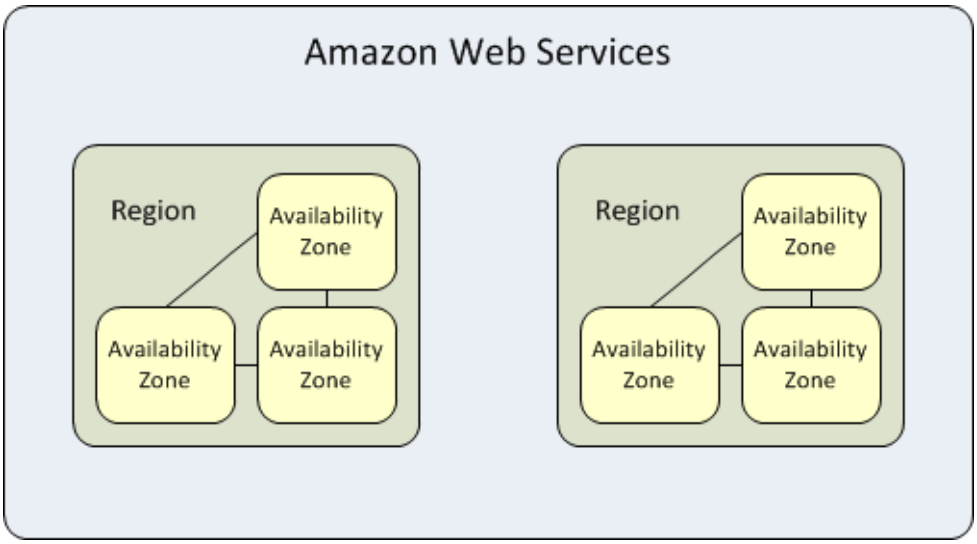
Aurora MySQL is a fully managed Platform as a Service (PaaS) providing high availability capabilities. Amazon RDS provides database and instance administration functionality for provisioning, patching, backup, recovery, failure detection, and repair.

New Aurora MySQL database instances are always created as part of a cluster. If you don't specify replicas at creation time, a single-node cluster is created. You can add database instances to clusters later.

Regions and Availability Zones

Amazon RDS is hosted in multiple global locations. Each location is composed of Regions and Availability Zones. Each Region is a separate geographic area having multiple, isolated Availability Zones. Amazon RDS supports placement of resources such as database instances and data storage in multiple locations. By default, resources are not replicated across regions.

Each Region is completely independent and each Availability Zone is isolated from all others. However, the main benefit of Availability Zones within a Region is that they are connected through low-latency, high bandwidth local network links.



Resources may have different scopes. A resource may be global, associated with a specific region (region level) , or associated with a specific Availability Zone within a region. For more information, see https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/resources.html

When creating a database instance, you can specify an availability zone or use the default "No Preference", in which case Amazon chooses the availability zone for you.

Aurora MySQL instances can be distributed across multiple availability zones. Applications can be designed to take advantage of failover such that in the event of an instance in one availability zone failing, another instance in different availability zone will take over and handle requests.

Elastic IP addresses can be used to abstract the failure of an instance by remapping the virtual IP address to one of the available database instances in another Availability Zone. For more information, see https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

An Availability Zone is represented by a region code followed by a letter identifier. For example, us-east-1a.

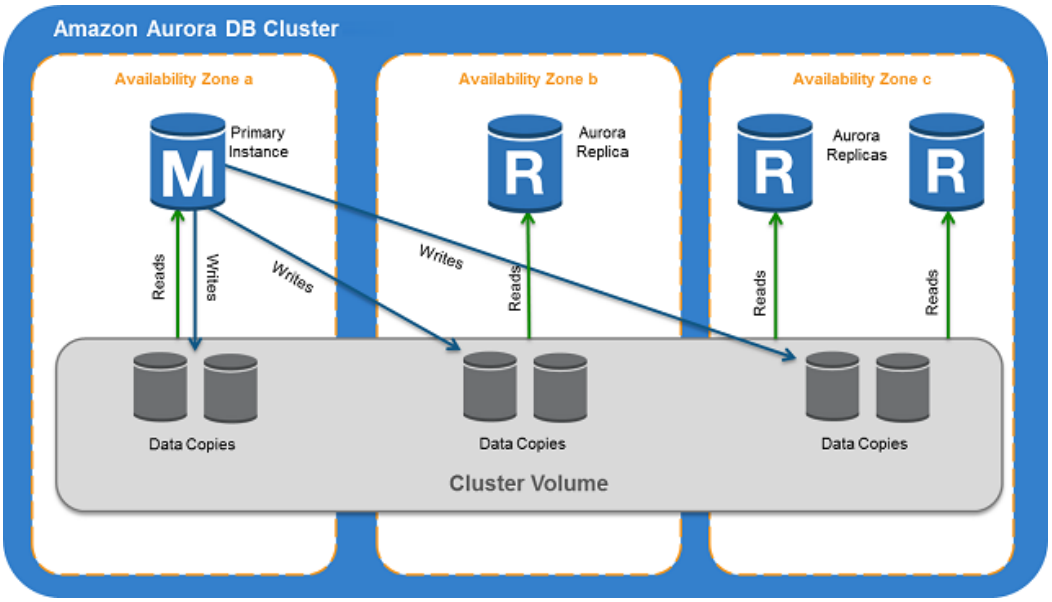
> **Note:** To guarantee even resource distribution across Availability Zones for a region, Amazon RDS independently maps Availability Zones to identifiers for each account. For example, the Availability Zone us-east-1a for one account might not be in the same location as us-east-1a for another account. Users cannot coordinate Availability Zones between accounts.

Aurora MySQL DB Cluster

A DB cluster consists of one or more DB instances and a cluster volume that manages the data for those instances. A cluster volume is a virtual database storage volume that may span multiple Availability Zones with each holding a copy of the database cluster data.

An Aurora database cluster is made up of one of more of the following types of instances:

- A **Primary instance** that supports both read and write workloads. This instance is used for all DML transactions. Every Aurora DB cluster has one, and only, one primary instance.
- An **Aurora Replica** that supports read-only workloads. Every Aurora MySQL database cluster may contain from zero to 15 Aurora Replicas in addition to the primary instance for a total maximum of 16 instances. Aurora Replicas enable scale-out of read operations by offloading reporting or other read-only processes to multiple replicas. Place aurora replicas in multiple availability Zones to increase availability of the databases.



Endpoints

Endpoints are used to connect to Aurora MySQL databases. An endpoint is a Universal Resource Locator (URL) comprised of a host address and port number.

- A **Cluster Endpoint** is an endpoint for an Aurora database cluster that connects to the current primary instance for that database cluster regardless of the availability zone in which the primary resides. Every Aurora MySQL DB cluster has one cluster endpoint and one primary instance. The cluster endpoint should be used for transparent failover for either read or write workloads.

  > **Note:** Use the cluster endpoint for all write operations including all DML and DDL statements.

  If the primary instance of a DB cluster fails for any reason, Aurora automatically fails over server requests to a new primary instance. A example of a typical Aurora MySQL DB Cluster endpoint is: `mydbcluster.cluster-123456789012.us-east-1.rds.amazonaws.com:3306`

- A **Reader Endpoint** is an endpoint that is used to connect to one of the Aurora read-only replicas in the database cluster. Each Aurora MySQL database cluster has one reader endpoint. If there are more than one Aurora Replicas in the cluster, the reader endpoint redirects the connection to one of the available replicas. Use the Reader Endpoint to support load balancing for read-only connections. If the DB cluster contains no replicas, the reader endpoint redirects the connection to the primary instance. If an Aurora Replica is created later, the Reader Endpoint starts directing connections to the new Aurora Replica with minimal interruption in service. An example of a typical Aurora MySQL DB Reader Endpoint is: `mydbcluster.cluster-ro-123456789012.us-east-1.rds.amazonaws.com:3306`

- An **Instance Endpoint** is a specific endpoint for every database instance in an Aurora DB cluster. Every Aurora MySQL DB instance regardless of its role has its own unique instance endpoint. Use the Instance Endpoints only when the application handles failover and read workload scale-out on its own. For example, you can have certain clients connect to one replica and others to another. An example of a typical Aurora MySQL DB Reader Endpoint is: `mydbinstance.123456789012.us-east-1.rds.amazonaws.com:3306`

Some general considerations for using endpoints:

- Consider using the cluster endpoint instead of individual instance endpoints because it supports high-availability scenarios. In the event that the primary instance fails, Aurora MySQL automatically fails over to a new primary instance. This configuration can be accomplished by either promoting an existing Aurora Replica to be the new primary or by creating a new primary instance.
- If you use the cluster endpoint instead of the instance endpoint, the connection is automatically redirected to the new primary.
- If you choose to use the instance endpoint, you must use the RDS cosole or the API to discover which database instances in the database cluster are available and their current roles. Then, connect using that instance endpoint.

- Be aware that the reader endpoint load balances connections to Aurora Replicas in an Aurora database cluster, but it does not load balance specific queries or workloads. If your application requires custom rules for distributing read workloads, use instance endpoints.
- The reader endpoint may redirect connection to a primary instance during the promotion of an Aurora Replica to a new primary instance.

Amazon Aurora Storage

Aurora MySQL data is stored in a cluster volume. The Cluster volume is a single, virtual volume that uses fast solid state disk (SSD) drives.The cluster volume is comprised of multiple copies of the data distributed between availability zones in a region. This configuration minimizes the chances of data loss and allows for the failover scenarios mentioned above.

Aurora cluster volumes automatically grow to accommodate the growth in size of your databases. An Aurora cluster volume has a maximum size of 64 tebibytes (TiB). Since table size is theoretically limited to the size of the cluster volume, the maximum table size in an Aurora DB cluster is 64 TiB.

Storage Auto-Repair

The chance of data loss due to disk failure is greatly minimize due to the fact that Aurora MySQL maintains multiple copies of the data in three Availability Zones. Aurora MySQL detects failures in the disks that make up the cluster volume. If a disk segment fails, Aurora repairs the segment automatically. Repairs to the disk segments are made using data from the other cluster volumes to ensure correctness. This process allows Aurora to significantly minimize the potential for data loss and the subsequent need to restore a database.

Survivable Cache Warming

When a database instance starts, Aurora MySQL performs a "warming" process for the buffer pool. Aurora MySQL pre-loads the buffer pool with pages that have been frequently used in the past. This approach improves performance and shortens the natural cache filling process for the initial period when the database instance starts servicing requests. Aurora MySQL maintains a separate process to manage the cache, which can stay alive even when the database process restarts. The buffer pool entries remain in memory regardless of the database restart providing the database instance with a fully "warm" buffer pool.

Crash Recovery

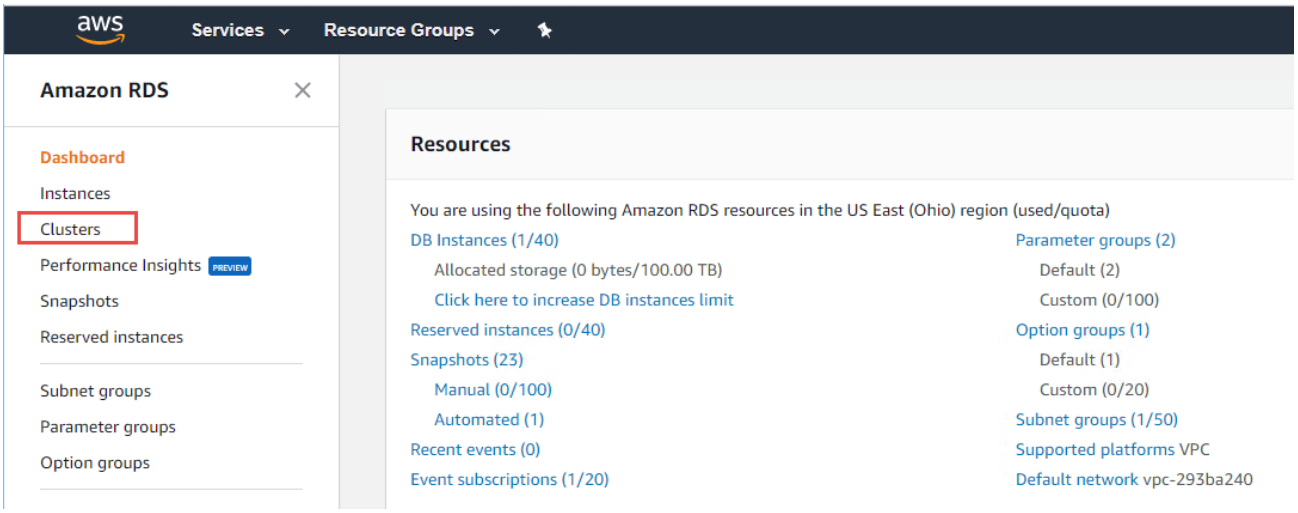
Aurora MySQL can instantaneously recover from a crash and continue to serve requests. Crash recovery is performed asynchronously using parallel threads enabling the database to remain open and available immediately after a crash.

For more information about crash recovery, see https://-doc-s.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Managing.html#Aurora.Managing.FaultTolerance.

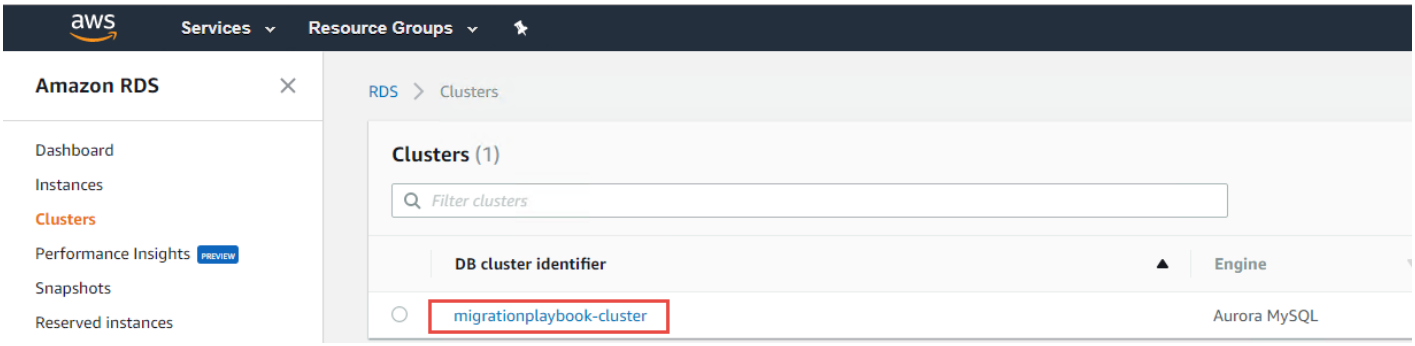
Examples

The following walkthrough describes how to configure a read-only replica using the AWS RDS console:

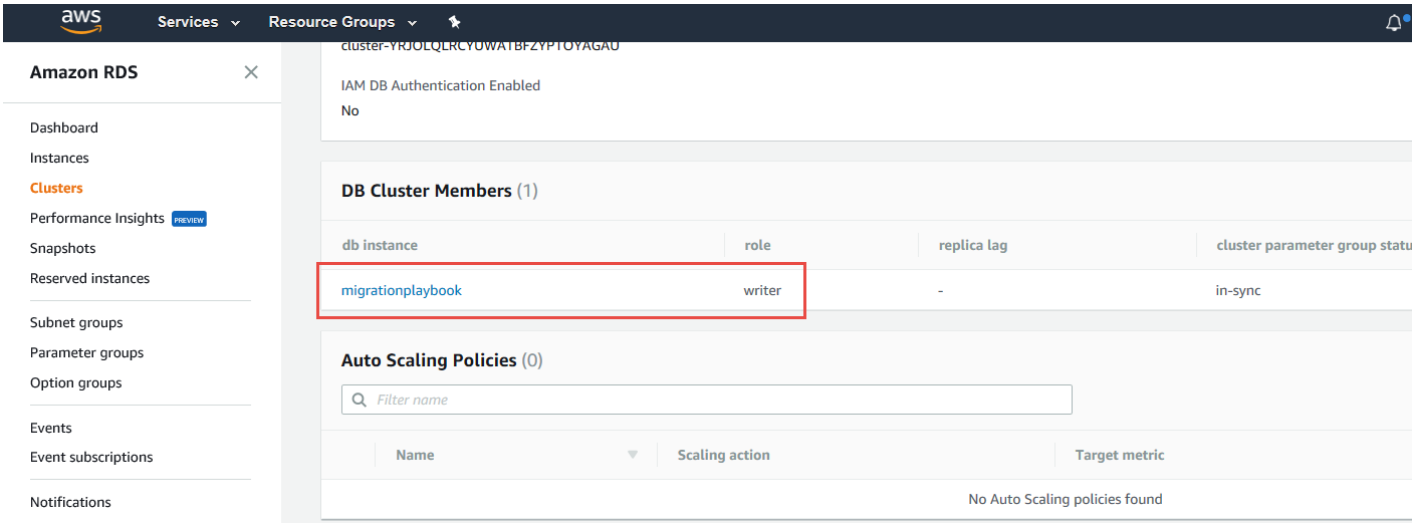
Login to the RDS Console and click **Clusters**.



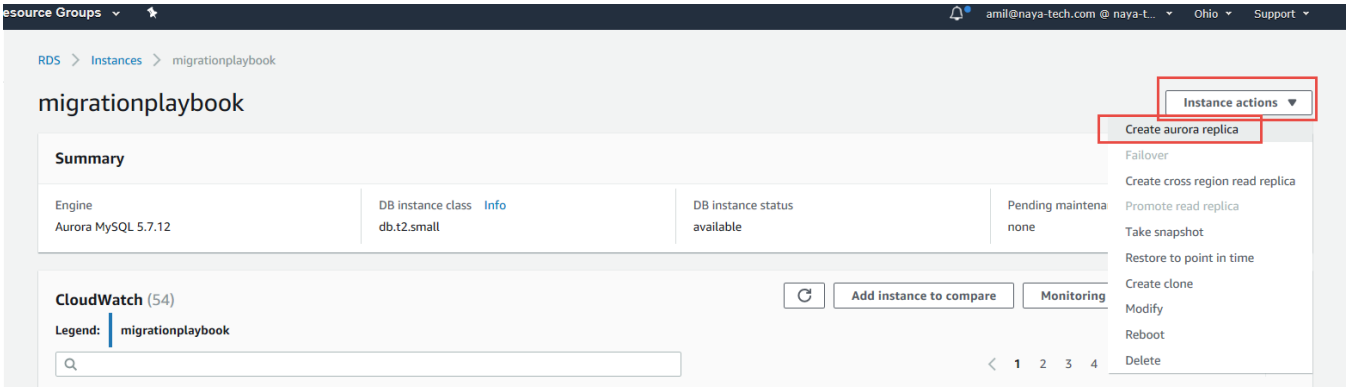
Click the cluster name.



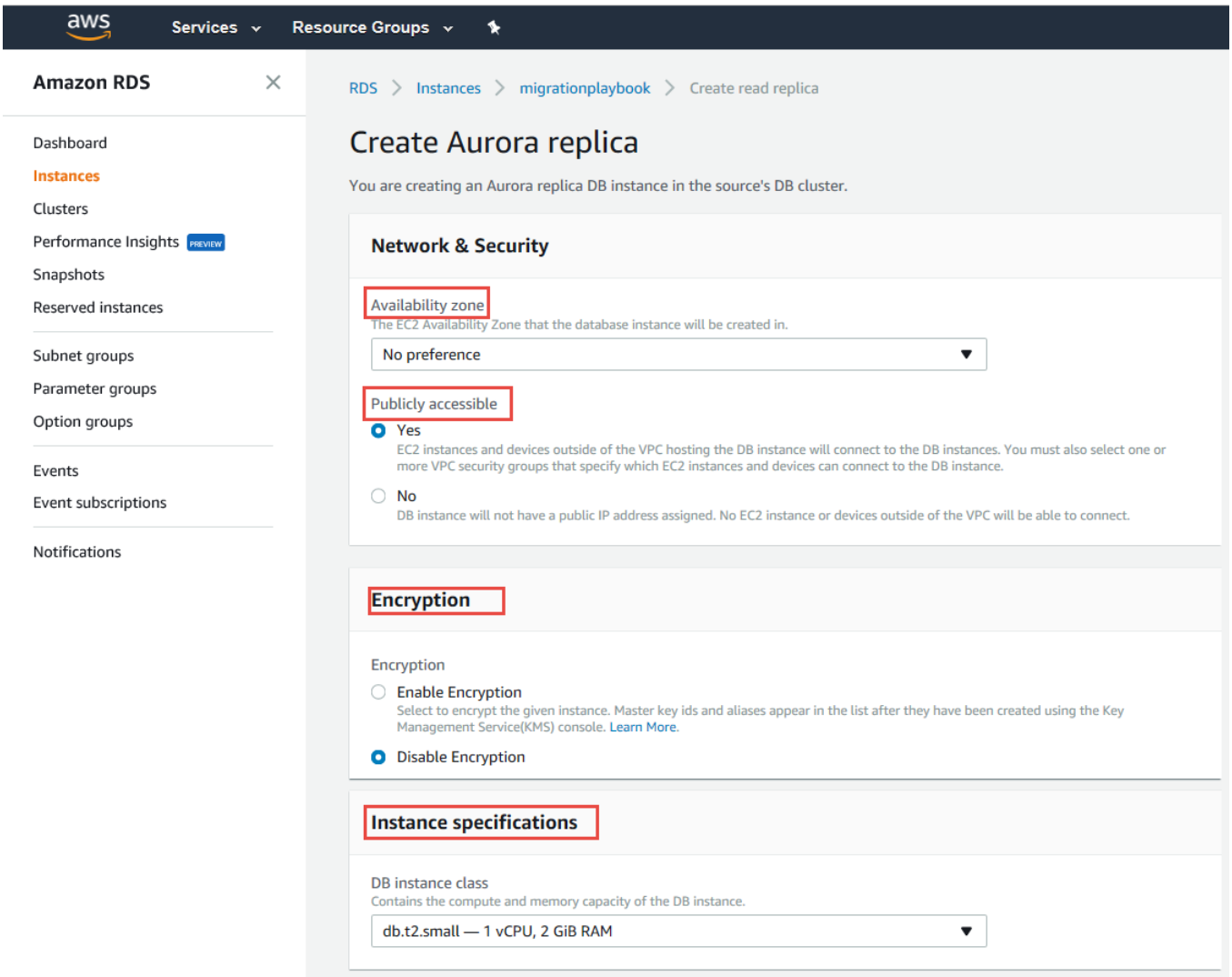
Scroll down to the **DB Cluster Members** section and click the current primary instance (writer role).



Click **Instance actions** and select **Create Aurora Replica**.



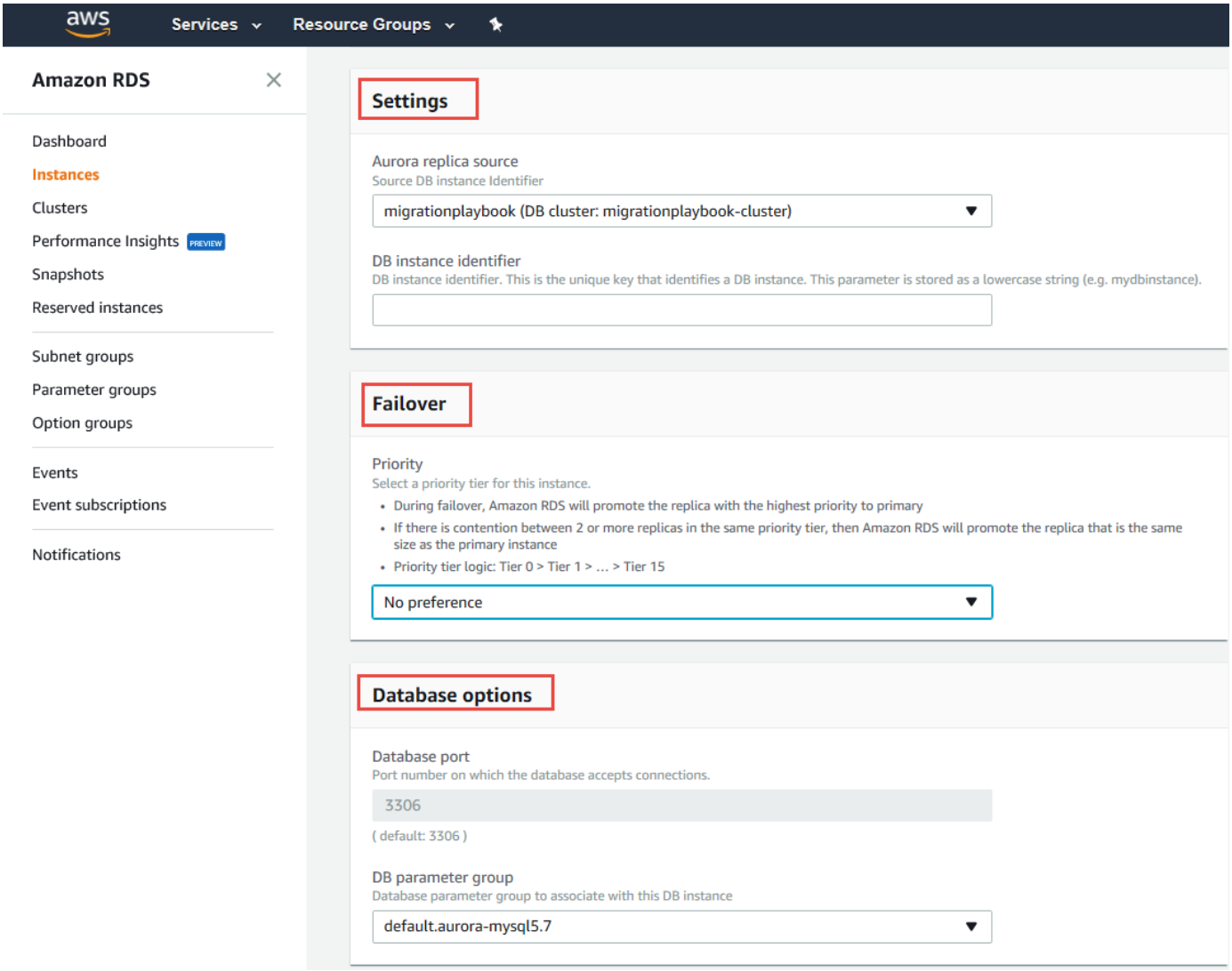
On the **Create Aurora Replica** page, select the **Availability zone**, or leave it blank to let AWS to select the availability zone. For **Publicly accessible**, select **No** if you don't want to assign a public IP address to this instance. For **Encryption**, select Enable Encryption if you want the instance to be encrypted. For **Instance specifications**, select the Instance size.



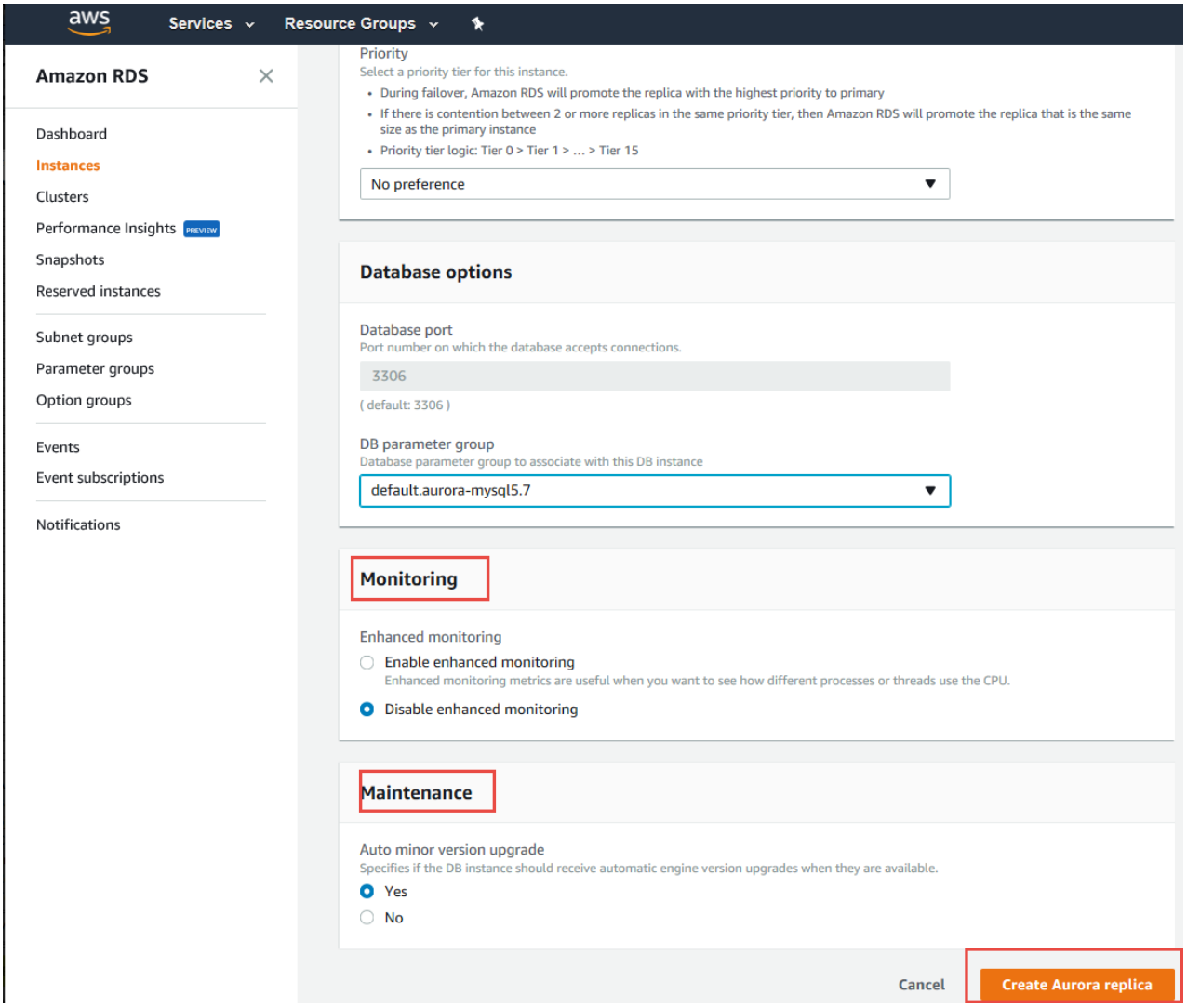
Scroll down to continue specifying the Aurora Replica settings. Leave the **Aurora replica source**default setting. For **Failover**, select the failover priority setting for this instance (1 to 15). If the

primary instance fails and multiple Aurora Replicas are available, the next-in-line primary is the Aurora Replicas that will have the highest priority. You can also select a specific parameter group for this replica.

> **Note:** The Aurora Replica can be a different size than the Primary Instance and can use a different parameter group.



Scroll down and select values for **Monitoring** and **Maintenance**.



Click **Create Aurora Replica** and return to the dashboard to view the creation progress.

Summary

Feature	SQL Server	Aurora MySQL	Comments
Server level failure protection	Failover Cluster Instances	N/A	Not applicable. Clustering is handled by Aurora MySQL.
Database level failure protection	Always On Availability Groups	Aurora Replicas	
Log replication	Log Shipping	N/A	Not applicable. Aurora MySQL handles data replication at the storage level.
Disk error protection	RESTORE... PAGE=	Automatically	

Feature	SQL Server	Aurora MySQL	Comments
Maximum Read Only replicas	8 + Primary	15 + Primary	
Failover address	Availability Group Listener	Cluster Endpoint	
Read Only work-loads	READ INTENT con-nection	Read Endpoint	

For more information, see:

- *https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Overview.html*
- *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html*

Indexes

Migrate from SQL Server Clustered and Non Clustered Indexes

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Indexes	• Clustered primary keys only • Filtered indexes and included columns not supported

Overview

Indexes are physical disk structures used to optimize data access. They are associated with tables or materialized views and allow the query optimizer to access rows and individual column values without scanning an entire table.

An index consists of index keys, which are columns from a table or view. They are sorted in ascending or descending order providing quick access to individual values for queries that use equality or range predicates. Database indexes are similar to book indexes that list page numbers for common terms. Indexes created on multiple columns are called Composite Indexes.

SQL Server implements indexes using the Balanced Tree algorithm (B-tree).

> **Note:** SQL Server supports additional index types such as hash indexes (for memory-optimized tables), spatial indexes, full text indexes, and XML indexes.

Indexes are created automatically to support table primary keys and unique constraints. They are required to efficiently enforce uniqueness. Up to 250 indexes can be created on a table to support common queries.

SQL Server provides two types of B-Tree indexes: Custered Indexes and Non-Clustered Indexes.

Clustered Indexes

Clustered indexes include all the table's column data in their leaf level. The entire table data is sorted and logically stored in order on disk. A Clustered Index is similar to a phone directory index where the entire data is contained for every index entry. Clustered indexes are created by default for Primary Key constraints. However, a primary key doesn't necessarily need to use a clustered index if it is explicitly specified as non-clustered.

Clustered indexes are created using the CREATE CLUSTERED INDEX statement. Only one clustered index can be created for each table because the index itself is the table's data. A table having a clustered index is called a "clustered table" (also known as an "index organized table" in other relational database management systems). A table with no clustered index is called a "heap".

Examples

Create a Clustered Index as part of table definition.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL
 PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
);
```

Create an explicit clustered index using CREATE INDEX.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL
 PRIMARY KEY NONCLUSTERED,
Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE CLUSTERED INDEX IDX1
ON MyTable(Col2);
```

Non-Clustered Indexes

Non clustered indexes also use the B-Tree algorithm but consist of a data structure separate from the table itself. They are also sorted by the index keys, but the leaf level of a non-clustered index contains pointers to the table rows; not the entire row as with a clustered index.

Up to 999 non-clustered indexes can be created on a SQL Server table. The type of pointer used at the lead level of a non-clustered index (also known as a row locator) depends on whether the table has a clustered index (clustered table) or not (heap). For heaps, the row locators use a physical pointer (RID). For clustered tables, row locators use the clustering key plus a potential uniquifier. This approach minimizes non-clustered index updates when rows move around, or the clustered index key value changes.

Both clustered and non clustered indexes may be defined as UNIQUE using the CREATE UNIQUE INDEX statement. SQL Server maintains indexes automatically for a table or view and updates the relevant keys when table data is modified.

Examples

Create a unique non-clustered index as part of table definition.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL
 PRIMARY KEY,
Col2 VARCHAR(20) NOT NULL
 UNIQUE
);
```

Create a unique non-clustered index using CREATE INDEX.

```
CREATE TABLE MyTable
(
Col1 INT NOT NULL
```

```
 PRIMARY KEY CLUSTERED,
Col2 VARCHAR(20) NOT NULL
);
```

```
CREATE UNIQUE NONCLUSTERED INDEX IDX1 ON MyTable(Col2);
```

Filtered Indexes and Covering Indexes

SQL Server also supports two special options for non clustered indexes. Filtered indexes can be created to index only a subset of the table's data. They are useful when it is known that the application will not need to search for specific values such as NULLs.

For queries that typically require searching on particular columns but also need additional column data from the table, non-clustered indexes can be configured to include additional column data in the index leaf level in addition to the row locator. This may prevent expensive lookup operations, which follow the pointers to either the physical row location (in a heap) or traverse the clustered index key in order to fetch the rest of the data not part of the index. If a query can get all the data it needs from the non-clustered index leaf level, that index is considered a "covering" index.

Examples

Create a filtered index to exclude NULL values.

```
CREATE NONCLUSTERED INDEX IDX1
ON MyTable(Col2)
WHERE Col2 IS NOT NULL;
```

Create a covering index for queries that search on `col2` but also need data from `col3`.

```
CREATE NONCLUSTERED INDEX IDX1
ON MyTable (Col2)
INCLUDE (Col3);
```

Indexes On Computed Columns

SQL Server allows creating indexes on persisted computed columns. Computed columns are table or view columns that derive their value from an expression based on other columns in the table. They are not explicitly specified when data is inserted or updated. This feature is useful when a querie's filter predicates are not based on the column table data as-is but on a function or expression.

Examples

For example, consider the following table that stores phone numbers for customers, but the format is not consistent for all rows; some include country code and some do not:

```
CREATE TABLE PhoneNumbers
(
PhoneNumber VARCHAR(15) NOT NULL
 PRIMARY KEY,
Customer VARCHAR(20) NOT NULL
);
```

```
INSERT INTO PhoneNumbers
VALUES
('+1-510-444-3422','Dan'),
('644-2442-3119','John'),
('1-402-343-1991','Jane');
```

The following query to look up the owner of a specific phone number must scan the entire table because the index cannot be used due to the preceding % wild card:

```
SELECT Customer
FROM PhoneNumbers
WHERE PhoneNumber LIKE '%510-444-3422';
```

A potential solution would be to add a computed column that holds the phone number in reverse order:

```
ALTER TABLE PhoneNumbers
ADD ReversePhone AS REVERSE(PhoneNumber)
PERSISTED;
```

```
CREATE NONCLUSTERED INDEX IDX1
ON PhoneNumbers (ReversePhone)
INCLUDE (Customer);
```

Now, the following query can be used to search for the customer based on the reverse string, which places the wild card at the end of the LIKE predicate. This approach provides an efficient index seek to retrieve the customer based on the phone number value:

```
DECLARE @ReversePhone VARCHAR(15) = REVERSE('510-444-3422');
SELECT Customer
FROM PhoneNumbers
WHERE ReversePhone LIKE @ReversePhone + '%';
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql*

Migrate to Aurora MySQL Clustered and Non Clustered Indexes

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Indexes	• Clustered primary keys only • Filtered indexes and included columns not supported

Overview

Aurora MySQL supports Balanced Tree (b-tree) indexes similar to SQL Server. However, the terminology, use, and options for these indexes are different.

Primary Key Indexes

Primary key indexes are created automatically by Aurora MySQL to support Primary Key constraints. They are the equivalent of SQL Server clustered indexes and contain the entire row in the leaf level of the index. Unlike SQL Server, primary key indexes are not configurable; you cannot use a non-clustered index to support a primary key. In Aurora MySQL, a primary key index consisting of multiple columns is called "Multiple Column index". It is the equivalent of an SQL Server composite index.

The MySQL query optimizer can use b-tree indexes to efficiently filter equality and range predicates. The Aurora MySQL optimizer considers using b-tree indexes to access data especially when queries use one or more of the following operators: >, >=, <, <=, =, or IN, BETWEEN, IS NULL, or IS NOT NULL predicates.

Primary key indexes in Aurora MySQL cannot be created with the CREATE INDEX statement. Since they are part of the primary key, they can only be created as part of the CREATE TABLE statement or with the ALTER TABLE... ADD CONSTRAINT... PRIMARY KEY statement. To drop a primary key index, use the ALTER TABLE... DROP PRIMARY KEY statement.

The relational model specifies that every table must have a primary key, but Aurora MySQL and most other relational database systems do not enforce it. If a table does not have a primary key specified, Aurora MySQL locates the first unique index where all key columns are specified as NOT NULL and uses that as the clustered index.

> **Note:** If no primary key or suitable unique index can be found, Aurora MySQL creates a "hidden" GEN_CLUST_INDEX clustered index with internally generated row ID values. These auto-generated row IDs are based on a six-byte field that increases monotonically (similar to IDENTITY or SEQUENCE).

Examples

Create a Primary Key index as part of the table definition.

```
CREATE TABLE MyTable (Col1 INT NOT NULL PRIMARY KEY, Col2 VARCHAR(20) NOT NULL);
```

Create a Primary key index for an existing table with no primary key.

```
ALTER TABLE MyTable ADD CONSTRAINT PRIMARY KEY (Col1);
```

> **Note:** Constraints in Aurora MySQL do not need to be explicitly named like in SQL Server.

Column and Multiple Column Secondary Indexes

Aurora MySQL Single column indexes are called "Column Indexes" and are the equivalent of SQL Server single column non-clustered indexes. Multiple column indexes are the equivalent of composite non-clustered indexes in SQL Server. They can be created as part of the table definition when creating unique constraints or explicitly using the INDEX or KEY keywords. For more information, see Creating Tables.

Multiple column indexes are useful when queries filter on all or leading index key columns. Specifying the optimal order of columns in a multiple column index can improve the performance of multiple queries accessing the table with similar predicates.

Examples

Create a unique b-tree index as part of the table definition.

```
CREATE TABLE MyTable (Col1 INT NOT NULL PRIMARY KEY, Col2 VARCHAR(20) UNIQUE);
```

Create a non-unique multiple column index on an existing table.

```
CREATE INDEX IDX1 ON MyTable (Col1, Col2) USING BTREE;
```

> **Note:** The USING clause is not mandatory. The default index type for Aurora MySQL is BTREE.

Secondary Indexes on Generated Columns

Aurora MySQL supports creating indexes on generated columns. They are the equivalent of SQL Server computed columns. Generated columns derive their values from the result of an expression. Creating an index on a generated column enables generated columns to be used as part of a filter predicate and may use the index for data access.

Generated columns can be created as STORED or VIRTUAL, but indexes can only be created on STORED generated columns.

Generated expressions cannot exceed 64 KB for the entire table. For example, you can create a single generated column with an expression length of 64K or create 12 fields with a length of 5K each. For more information, see Creating Tables.

Prefix Indexes

Aurora MySQL also supports indexes on partial string columns. Indexes can be created that use only the leading part of column values using the following syntax:

```
CREATE INDEX <Index Name> ON <Table Name> (<col name>(<prefix length>));
```

Prefixes are optional for CHAR, VARCHAR, BINARY, and VARBINARY column indexes, but must be specified for BLOB and TEXT column indexes.

Index prefix length is measured in bytes. The prefix length for CREATE TABLE, ALTER TABLE, and CREATE INDEX statements is interpreted as the number of characters for non-binary string types (CHAR, VARCHAR, TEXT) or the number of bytes for binary string types (BINARY, VARBINARY, BLOB).

Examples

Create a prefix index for the first ten characters of a customer name.

```
CREATE INDEX PrefixIndex1 ON Customers (CustomerName(10));
```

Summary

The following table summarizes the key differences to consider when migrating b-tree indexes from SQL Server to Aurora MySQL

Index Feature	SQL Server	Aurora MySQL	Comments
Clustered indexes supported for	Table keys, composite or single column, unique and non-unique, null or not null	Primary keys only	
Non clustered index supported for	Table keys, composite or single column, unique and non unique, null or not null	Unique constraints, single column and multi-column	
Max number of non clustered indexes	999	64	
Max total index key size	900 bytes	3072 bytes for a 16 KB page size, 1536 bytes for a 8 KB page size 768 bytes for a 4 KB page size	
Max columns per index	32	16	
Index Prefix	N/A	Optional for CHAR, VARCHAR, BINARY, and VARBINARY Mandatory for BLOB	

Index Feature	SQL Server	Aurora MySQL	Comments
		and TEXT	
Filtered Indexes	Supported	N/A	
Included columns	Supported	N/A	Add the required columns as index key columns instead of included
Indexes on BLOBS	N/A	Supported, limited by maximal index key size	

For more information see:

- *https://dev.mysql.com/doc/refman/5.7/en/create-index.htm*
- *https://dev.mysql.com/doc/refman/5.7/en/column-indexes.htm*
- *https://dev.mysql.com/doc/refman/5.7/en/multiple-column-indexes.html*

Management

Migrate from SQL Server SQL Server Agent

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Agent	• See Alerting and Maintenance Plans

Overview

SQL Server Agent provides two main functions: Scheduling automated maintenance and backup jobs, and for alerting.

> **Note:** Other SQL built-in frameworks such as replication, also use SQL Agent jobs under the covers.

Maintenance Plans, backups and Alerting are covered in separate sections:

- Maintenance Plans
- Backups
- Alerting

For more information about SQL Server Agent, see https://docs.microsoft.com/en-us/sql/ssms/agent/sql-server-agent

Migrate to Aurora MySQL Agent

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Agent	• See Alerting and Maintenance Plans

Overview

SQL Server Agent provides two main functions: Scheduling automated maintenance jobs and alerting.

> **Note:** Other SQL built-in frameworks such as replication also use SQL Agent jobs.

Maintenance Plans and Alerting are covered in separate sections:

- Maintenance Plans
- Alerting

Aurora MySQL does provide a native, in-database scheduler. It is limited to the cluster scope and can't be used to manage multiple clusters. There are no native alerting capabilities in Aurora MySQL similar to SQL Server Agent's Alerts.

Although AWS RDS does not currently provide an external scheduling agent like SQL Server's Agent, CloudWatch Events provides the ability to specify a “cron-like” schedule to execute Lambda functions. This approach requires writing custom code in C#, NodeJS, Java, or Python. Additionally, any task that runs longer than 5 minutes will not work due to the Lambda time out limit. For example, this limit may pose a challenge for index rebuild operations. Other options include:

1. Running an SQL Server for the sole purpose of using the Agent.
2. Using a t2 or container to schedule your code (C#, NodeJS, Java, Python) with Cron. A t2.nano is simple to deploy and can run tasks indefinitely at a very modest cost. For most scheduling applications, the low resources should not be an issue.

Aurora MySQL Database Events

Aurora MySQL also provides a native, in-database scheduling framework that can be used to trigger scheduled operations including maintenance tasks.

Events are executed by a dedicated thread, which can be seen in the process list. The global event_ scheduler must be turned on explicitly from it's default state of OFF for the event thread to run. Event errors are written to the error log. Event metadata can be viewed using the INFORMATION_ SCHEMA.EVENTS view.

Syntax

```
CREATE EVENT <Event Name>
    ON SCHEDULE <Schedule>
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'string']
    DO <Event Body>;

<Schedule>:
    AT <Time Stamp> [+ INTERVAL <Interval>] ...
  | EVERY <Interval>
    [STARTS <Time Stamp> [+ INTERVAL <Interval>] ...]
    [ENDS <Time Stamp> [+ INTERVAL <Interval>] ...]

<Interval>:
    quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
              WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
              DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

Examples

Create an event to collect login data statistics that runs once five hours after creation.

```
CREATE EVENT Update_T1_In_5_Hours
    ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 5 HOUR
    DO
      INSERT INTO LoginStatistics
        SELECT UserID,
               COUNT(*) AS LoginAttempts
        FROM Logins AS L
        GROUP BY UserID
        WHERE LoginData = '20180502';
```

Create an event to run every hour and delete session information older than four hours.

```
CREATE EVENT Clear_Old_Sessions
    ON SCHEDULE
      EVERY 4 HOUR
    DO
      DELETE FROM Sessions
      WHERE LastCommandTime < CURRENT_TIMESTAMP - INTERVAL 4 HOUR;
```

Schedule weekly index rebuilds and pass parameters.

```
CREATE EVENT Rebuild_Indexes
    ON SCHEDULE
      EVERY 1 WEEK
    DO
      CALL IndexRebuildProcedure(1, 80)
```

For more information, see

- https://dev.mysql.com/doc/refman/5.7/en/create-event.html
- https://dev.mysql.com/doc/refman/5.7/en/events-configuration.html
- https://aws.amazon.com/cloudwatch
- https://aws.amazon.com/lambda/

Migrate from SQL Server Alerting

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Event Notifications Subscription with Amazon Simple Notification Service (SNS)

Overview

SQL Server provides *SQL Server Agent* to generate alerts. When running, SQL Server Agent constantly monitors SQL Server windows application log messages, performance counters, and Windows Management Instrumentation (WMI) objects. When a new error event is detected, the agent checks the MSDB database for configured alerts and executes the specified action.

You can define SQL Server Agent alerts for the following categories:

- SQL Server events
- SQL Server performance conditions
- WMI events

For SQL Server events, the alert options include the following settings:

- **Error Number:** Alert when a specific error is logged.
- **Severity Level:** Alert when any error in the specified severity level is logged.
- **Database:** Filter the database list for which the event will generate an alert.
- **Event Text:** Filter specific text in the event message.

  Note: SQL Server agent is pre-configured with several high severity alerts. It is highly recommended to enable these alerts.

To generate an alert in response to a specific performance condition, specify the performance counter to be monitored, the threshold values for the alert, and the predicate for the alert to occur. The following list identifies the performance alert settings:

- **Object:** The Performance counter *category* or the monitoring area of performance.
- **Counter:** A counter is a specific attribute value of the object.
- **Instance:** Filter by SQL Server instance (multiple instances can share logs).
- **Alert if counter and Value:** The threshold for the alert and the predicate. The threshold is a number. Predicates are *Falls below*, *becomes equal* to, or *rises above* the threshold.

WMI events require the WMI namespace and the WMI Query Language (WQL) query for specific events.

Alerts can be assigned to specific operators with schedule limitations and multiple response types including:

- Execute an SQL Server Agent Job.
- Send Email, Net Send command, or a pager notification.

You can configure Alerts and responses with SQL Server Management Studio or with a set of system stored procedures.

Examples

Configure an alert for all errors with severity 20.

```
EXEC msdb.dbo.sp_add_alert
 @name = N'Severity 20 Error Alert',
 @severity = 20,
 @notification_message = N'A severity 20 Error has occurred. Initiating emergency pro-
cedure',
 @job_name = N'Error 20 emergency response';
```

For more information, see https://docs.microsoft.com/en-us/sql/ssms/agent/alerts

Migrate to Aurora MySQL Alerting

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Event Notifications Subscription with Amazon Simple Notification Service (SNS)

Overview

Aurora MySQL does not support direct configuration of engine alerts.
Use the *Event Notifications Infrastructure* to collect history logs or receive event notifications in near real-time.

Amazon Relational Database Service (RDS) uses Amazon Simple Notification Service (SNS) to provide notifications for events. SNS can send notifications in any form supported by the region including email, text messages, or calls to HTTP endpoints for response automation.

Events are grouped into categories. You can only subscribe to event categories, not individual events. SNS sends notifications when any event in a category occurs.

You can subscribe to alerts for database instances, database clusters, database snapshots, database cluster snapshots, database security groups and database parameter groups. For example, a subscription to the *Backup* category for a specific database instance sends notifications when backup related events occur on that instance. A subscription to a *Configuration Change* category for a database security group sends notifications when the security group changes.

> **Note:** For Amazon Aurora, some events occur at the cluster rather than instance level. You will not receive those events if you subscribe to an Aurora DB instance.

SNS sends event notifications to the address specified when the subscription was created. Typically, administrators create several subscriptions. For example, one subscription to receive logging events and another to receive only critical events for a production environment requiring immediate responses.

You can disable notifications without deleting a subscription by setting the *Enabled* radio button to *No* in the Amazon RDS console. Alternatively, use the Command Line Interface (CLI) or RDS API to change the *Enabled* setting.

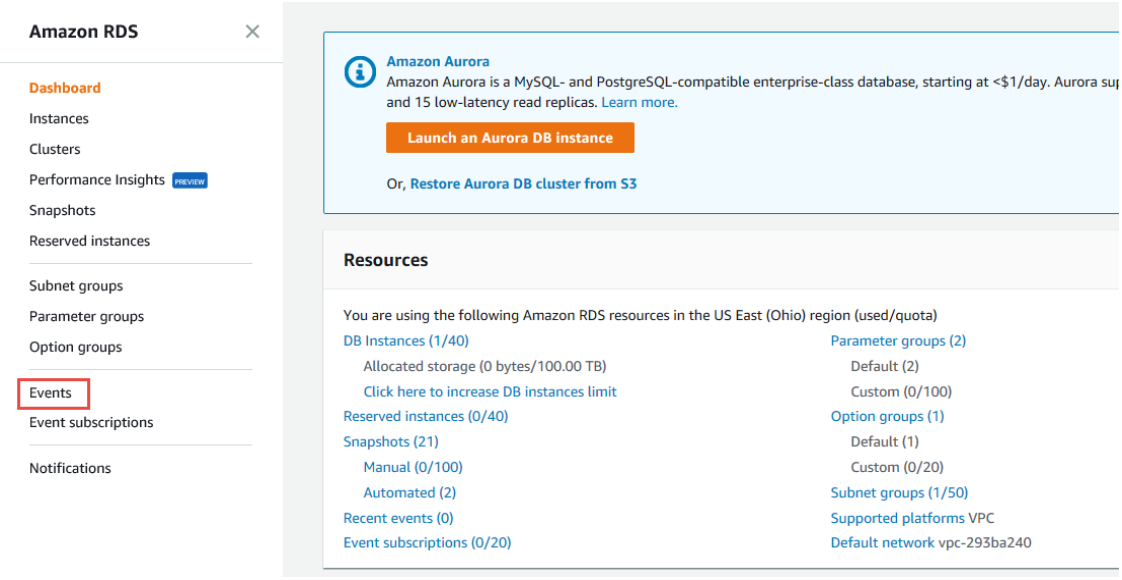
Subscriptions are identified by the Amazon Resource Name (ARN) of an Amazon SNS topic. The Amazon RDS console creates ARNs when subscriptions are created. When using the CLI or API, you must create the ARN using the Amazon SNS console or the Amazon SNS API.

Examples

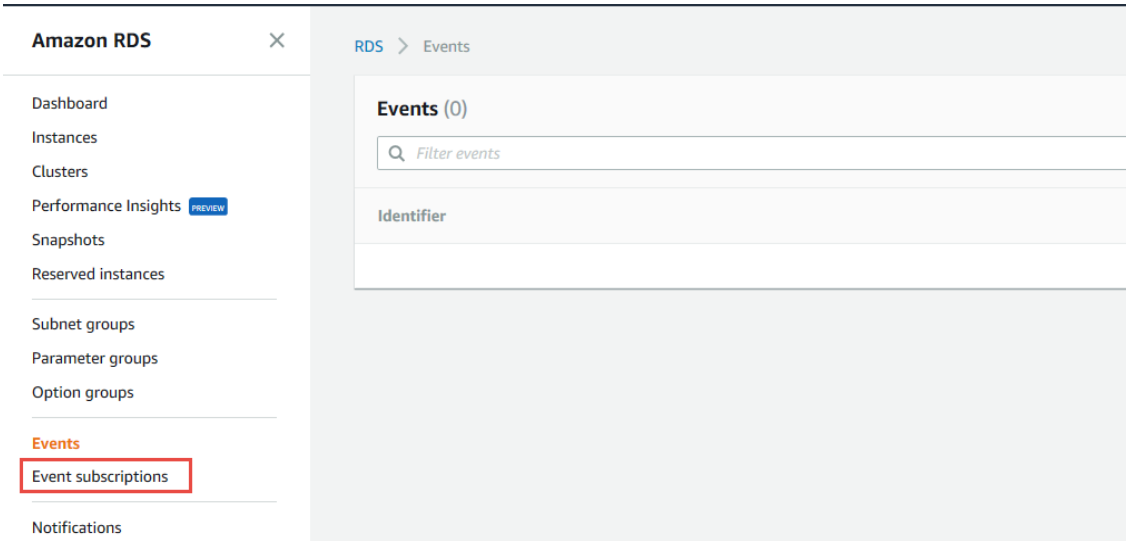
The follow walkthrough demonstrates how to create an Event Notification Subscription:

Sign into an Amazon AWS account, open the AWS Console, and navigate to the Amazon RDS page.

Click **Events** on the left navigation pane.

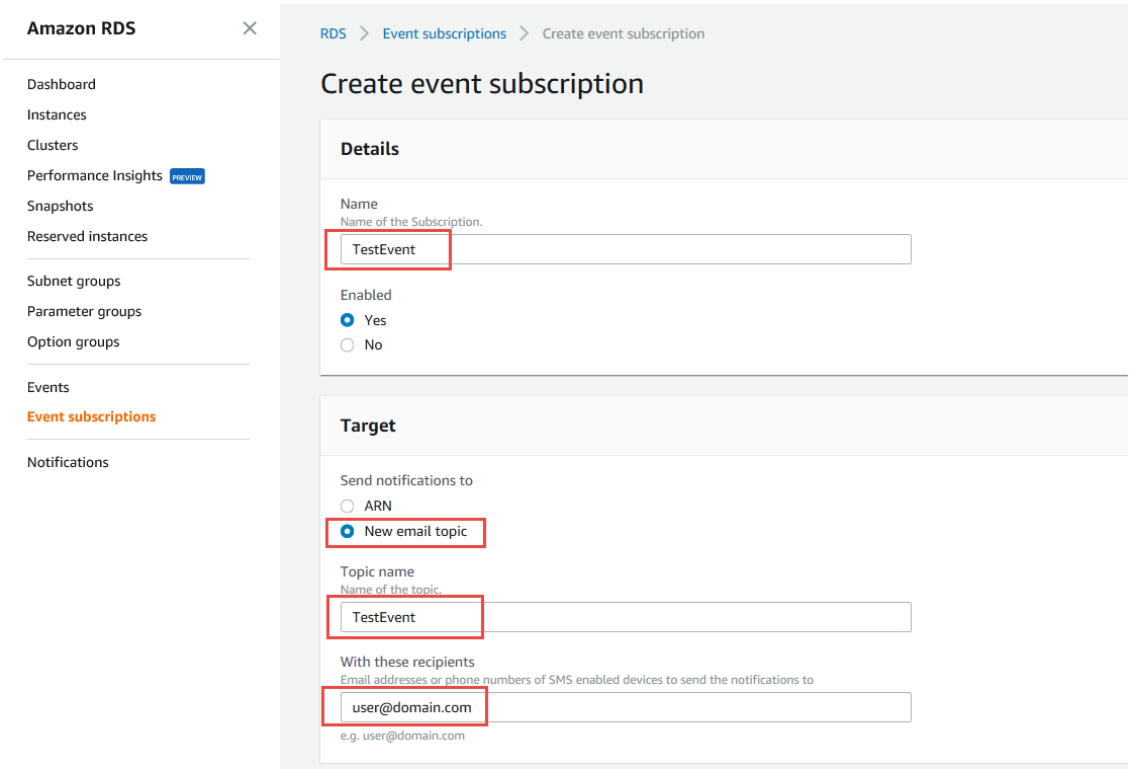


If you have not previously subscribed to events, the screen will show zero events.

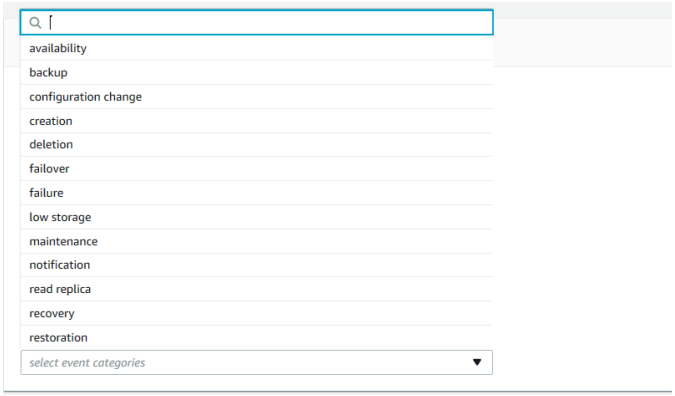


Click **Event Subscriptions** and then click **CREATE EVENT SUBSCRIPTION** on the top right side.

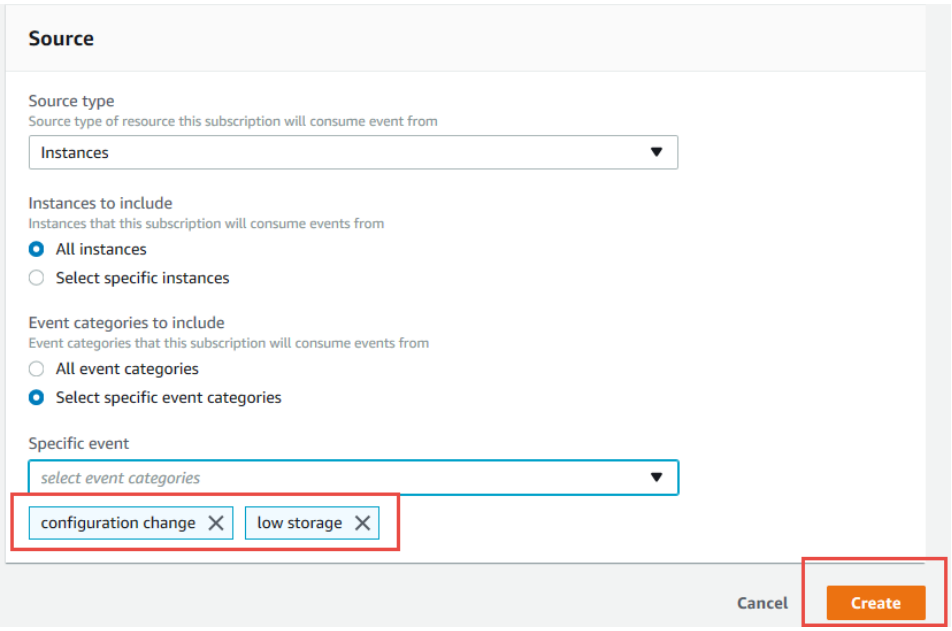
Enter **Name of the subscription** and select a **Target** of ARN or Email. For email subscriptions, enter values for **Topic name** and **With these recipients**.



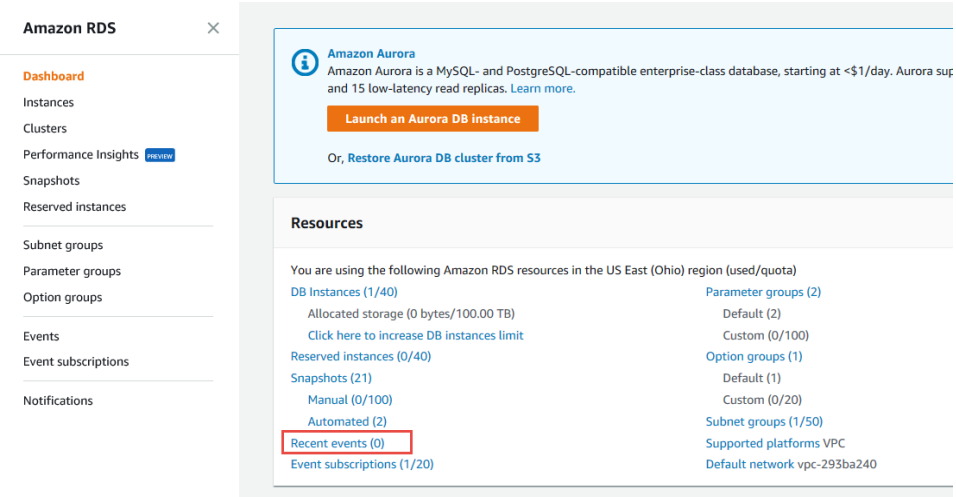
Select the event source and choose specific event categories. Click the drop-down menu to view the list of available categories.



Choose the event categories to be monitored and click **Create**.



From the AWS RDS Dashboard, click the **View Recent Events** button.



For more information, see https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_Events.html

Migrate from SQL Server Database Mail

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Mail	• Use Lambda Integration

Overview

The Database Mail framework is an email client solution for sending messages directly from SQL Server. Email capabilities and APIs within the database server provide easy management of the following messages:

- Server administration messages such as alerts, logs, status reports, and process confirmations.
- Application messages such as user registration confirmation and action verifications.

  Note: Database Mail is turned off by default.

The main features of the Database Mail framework are:

- Database Mail sends messages using the standard and secure Simple Mail Transfer Protocol (SMTP) .
- The email client engine runs asynchronously and sends messages in a separate process to minimize dependencies.
- Database Mail supports multiple SMTP Servers for redundancy.
- Full support and awareness of Windows Server Failover Cluster for high availability environments.
- Multi-profile support with multiple failover accounts in each profile.
- Enhanced security management with separate roles in MSDB.
- Security is enforced for mail profiles.
- Attachment sizes are monitored and can be capped by the administrator.
- Attachment file types can be blacklisted.
- Email activity can be logged to SQL Server, the Windows application event log, and to a set of system tables in MSDB.
- Supports full auditing capabilities with configurable retention policies.
- Supports both plain text and HTML messages.

Architecture

Database Mail is built on top of the Microsoft SQL Server Service Broker queue management framework.

The system stored procedure sp_send_dbmail sends email messages. When this stored procedure is executed, it inserts an row to the mail queue and records the Email message.

The queue insert operation triggers execution of the Database Mail process (DatabaseMail.exe). The Database Mail process then reads the Email information and sends the message to the SMTP servers.

When the SMTP servers acknowledge or reject the message, the Database Mail process inserts a status row into the status queue, including the result of the send attempt. This insert operation triggers the execution of a system stored procedure that updates the status of the Email message send attempt.

Database Mail records all Email attachments in the system tables. SQL Server provides a set of system views and stored procedures for troubleshooting and administration of the Database Mail queue.

Deprecated SQL Mail framework

The old SQL Mail framework using xp_sendmail has been deprecated as of SQL Server 2008R2 in accordance with https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms143729(v=sql.105).

The legacy mail system has been completely replaced by the greatly enhanced DB mail framework described here. The old system has been out-of-use for many years because it was prone to synchronous execution issues and windows mail profile quirks.

Syntax

```
EXECUTE sp_send_dbmail
    [[,@profile_name =] '<Profile Name>']
    [,[,@recipients =] '<Recipients>']
    [,[,@copy_recipients =] '<CC Recipients>']
    [,[,@blind_copy_recipients =] '<BCC Recipients>']
    [,[,@from_address =] '<From Address>']
    [,[,@reply_to =] '<Reply-to Address>']
    [,[,@subject =] '<Subject>']
    [,[,@body =] '<Message Body>']
    [,[,@body_format =] '<Message Body Format>']
    [,[,@importance =] '<Importance>']
    [,[,@sensitivity =] '<Sensitivity>']
    [,[,@file_attachments =] '<Attachments>']
    [,[,@query =] '<SQL Query>']
    [,[,@execute_query_database =] '<Execute Query Database>']
    [,[,@attach_query_result_as_file =] <Attach Query Result as File>]
    [,[,@query_attachment_filename =] <Query Attachment Filename>]
    [,[,@query_result_header =] <Query Result Header>]
    [,[,@query_result_width =] <Query Result Width>]
    [,[,@query_result_separator =] '<Query Result Separator>']
    [,[,@exclude_query_output =] <Exclude Query Output>]
    [,[,@append_query_error =] <Append Query Error>]
    [,[,@query_no_truncate =] <Query No Truncate>]
    [,[,@query_result_no_padding =] @<Parameter for Query Result No Padding>]
    [,[,@mailitem_id =] <Mail item id>] [,OUTPUT]
```

Examples

Create a Database Mail account.

```
EXECUTE msdb.dbo.sysmail_add_account_sp
    @account_name = 'MailAccount1',
    @description = 'Mail account for testing DB Mail',
    @email_address = 'Address@MyDomain.com',
    @replyto_address = 'ReplyAddress@MyDomain.com',
    @display_name = 'Mailer for registration messages',
    @mailserver_name = 'smtp.MyDomain.com' ;
```

Create a Database Mail profile.

```
EXECUTE msdb.dbo.sysmail_add_profile_sp
    @profile_name = 'MailAccount1 Profile',
    @description = 'Mail Profile for testing DB Mail' ;
```

Associate the account with the profile.

```
EXECUTE msdb.dbo.sysmail_add_profileaccount_sp
    @profile_name = 'MailAccount1 Profile',
    @account_name = 'MailAccount1',
    @sequence_number =1 ;
```

Grant the profile access to DBMailUsers role.

```
 EXECUTE msdb.dbo.sysmail_add_principalprofile_sp
    @profile_name = 'MailAccount1 Profile',
    @principal_name = 'ApplicationUser',
    @is_default = 1 ;
```

Send a message with sp_db_sendmail.

```
EXEC msdb.dbo.sp_send_dbmail
    @profile_name = 'MailAccount1 Profile',
    @recipients = 'Recipient@Mydomain.com',
    @query = 'SELECT * FROM fn_WeeklySalesReport(GETDATE())',
    @subject = 'Weekly Sales Report',
    @attach_query_result_as_file = 1 ;
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/database-mail/database-mail

Migrate to Aurora MySQL Database Mail

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Mail	• Use Lambda Integration

Overview

Aurora MySQL does not provide native support sending mail from the database.

For alerting purposes, use the Event Notification Subscription feature to send email notifications to operators.
For more information, see Alerting.

For application email requirements, consider using a dedicated email framework. If the code generating email messages must be in the database, consider using a queue table. Replace all occurrences of sp_send_dbmail with an INSERT into the queue table. Design external applications to connect, read the queue, send email an message, and then update the status periodically. With this approach, messages can be populated with a query result similar to sp_send_dbmail with the query option.

The only way to sent Email from the database, is to use the LAMBDA integration.

For more information about Lambda, see https://aws.amazon.com/lambda.

Examples

Sending an Email from Aurora MySQL via Lambda Integration

See the walkthrough on https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/AuroraMySQL.Integrating.Lambda.html

Migrate from SQL Server ETL

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Amazon Glue for ETL

Overview

SQL Server offers a native Extract, Transform, and Load (ETL) framework of tools and services to support enterprise ETL requirements. The legacy Data Transformation Services (DTS) has been deprecated as of SQL Server 2008 (see https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/cc707786(v=sql.105)) and replaced with SQL Server Integration Services (SSIS), which was introduced with SQL Server 2005.

DTS

DTS was introduced in SQL Server version 7 in 1998. It was significantly expanded in SQL Server 2000 with features such as FTP, database level operations, and Microsoft Message Queuing (MSMQ) integration. It included a set of objects, utilities, and services that enabled easy, visual construction of complex ETL operations across heterogeneous data sources and targets.

DTS supported OLE DB, ODBC, and text file drivers. It allowed transformations to be scheduled using SQL Server Agent. DTS also provided version control and backup capabilities with version control systems such as Microsoft Visual SourceSafe.

The fundamental entity in DTS was the DTS Package. Packages were the logical containers for DTS objects such as connections, data transfers, transformations, and notifications. The DTS framework also included the following tools:

- DTS Wizards
- DTS Package Designers
- DTS Query Designer
- DTS Run Utility

SSIS

The SSIS framework was introduced in SQL Server 2005, but was limited to the top-tier editions only, unlike DTS which was available with all editions.

SSIS has evolved over DTS to offer a true modern, enterprise class, heterogeneous platform for a broad range of data migration and processing tasks. It provides a rich workflow oriented design with features for all types of enterprise data warehousing. It also supports scheduling capabilities for multi-dimensional cubes management.

SSIS Provides the following tools:

- **SSIS Import/Export Wizard** is an SQL Server Management Studio extension that enables quick creation of packages for moving data between a wide array of sources and destinations. However, it has limited transformation capabilities.
- **SQL Server Business Intelligence Development Studio (BIDS)** is a developer tool for creating complex packages and transformations. It provides the ability to integrate procedural code into package transformations and provides a scripting environment. Recently, BIDS has been replaced by **SQL Server Data Tools - Business intelligence (SSDT-BI)**.

SSIS objects include:

- Connections
- Event handlers
- Workflows
- Error handlers
- Parameters (Beginning with SQL Server 2012)
- Precedence constraints
- Tasks
- Variables

SSIS packages are constructed as XML documents and can be saved to the file system or stored within a SQL Server instance using a hierarchical name space.

For more information, see

- https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services
- https://en.wikipedia.org/wiki/Data_Transformation_Services

Migrate to Aurora MySQL ETL

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Amazon Glue for ETL

Overview

Aurora MySQL provides Amazon Glue for enterprise class Extract, Transform, and Load (ETL). It is a fully managed service that performs data cataloging, cleansing, enriching, and movement between heterogeneous data sources and destinations. Being a fully managed service, the user does not need to be concerned with infrastructure management.

Amazon Glue Key Features

Integrated Data Catalog

The Amazon Glue Data Catalog is a persistent metadata store,that can be used to store all data assets, whether in the cloud or on-premises. It stores table schemas, job steps, and additional meta data information for managing these processes. Amazon Glue can automatically calculate statistics and register partitions in order to make queries more efficient. It maintains a comprehensive schema version history for tracking changes over time.

Automatic Schema Dfiscovery

Amazon Glue provides automatic crawlers that can connect to source or target data providers. The crawler uses a prioritized list of classifiers to determine the schema for your data and then generates and stores the metadata in the Amazon Glue Data Catalog. Crawlers can be scheduled or executed on-demand. You can also trigger a crawler when an event occurs to keep metadata current.

Code Generation

Amazon Glue automatically generates the code to extract, transform, and load data. All you need to do is point Glue to your data source and target. The ETL scripts to transform, flatten, and enrich data are created automatically. Amazon Glue scripts can be generated in Scala or Python and are written for Apache Spark.

Developer Endpoints

When interactively developing Glue ETL code, Amazon Glue provides development endpoints for editing, debugging, and testing. You can use any IDE or text editor for ETL development. Custom readers, writers, and transformations can be imported into Glue ETL jobs as libraries. You can also use and share code with other developers in the Amazon Glue GitHub repository (see https://-github.com/awslabs/aws-glue-libs).

Flexible Job Scheduler

Amazon Glue jobs can be triggered for execution either on a pre-defined schedule, on-demand, or as a response to an event.

Multiple jobs can be started in parallel and dependencies can be explicitly defined across jobs to build complex ETL pipelines. Glue handles all inter-job dependencies, filters bad data, and retries failed jobs. All logs and notifications are pushed to Amazon CloudWatch; you can monitor and get alerts from a central service.

Migration Considerations

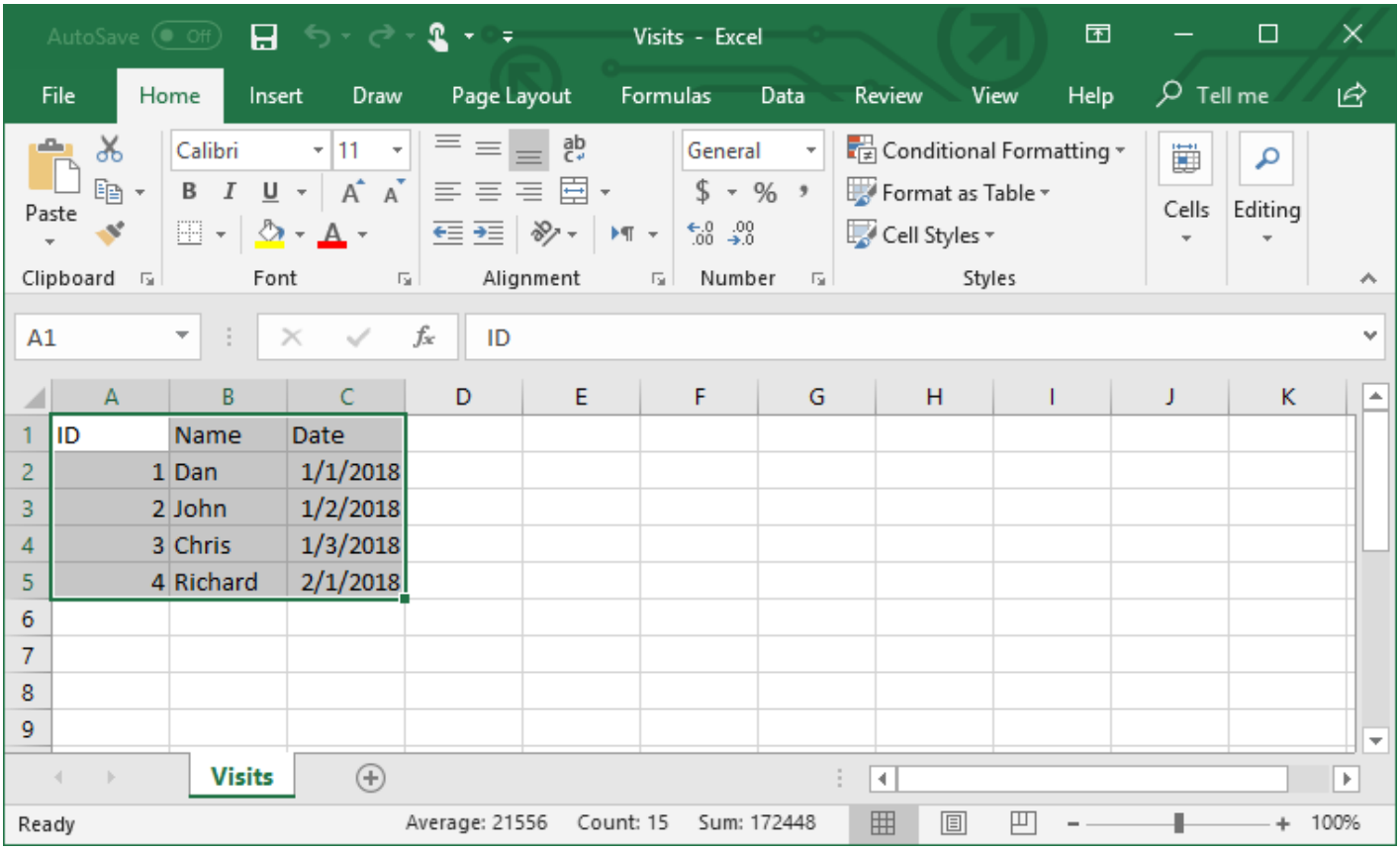
Currently, there are no automatic tools for migrating ETL packages from DTS or SSIS into Amazon Glue. Migration from SQL Server to Aurora MySQL requires rewriting ETL processes to use Amazon Glue.

Alternatively, consider using an EC2 SQL Server instance to run the SSIS service as an interim solution. The connectors and tasks must be revised to support Aurora MySQL instead of SQL Server, but this approach allows gradual migration to Amazon Glue.

Examples

The following walkthrough describes how to create an Amazon Glue job to upload a CSV file from S3 to Aurora MySQL.

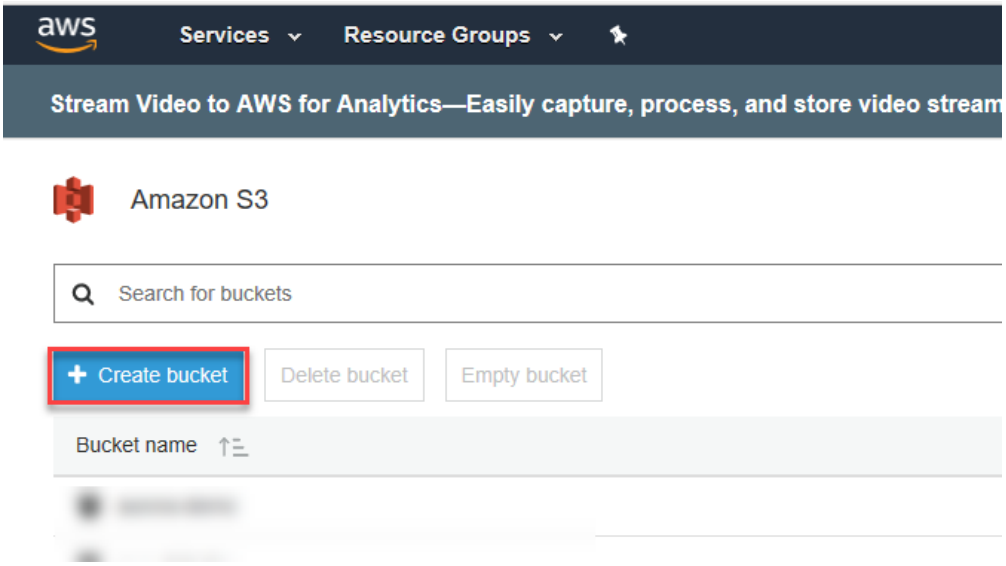
The source file for this walkthrough is a simple Visits table in CSV format. The objective is to upload this file to an S3 bucket and create a Glue job to discover and copy it into an Aurora MySQL database.



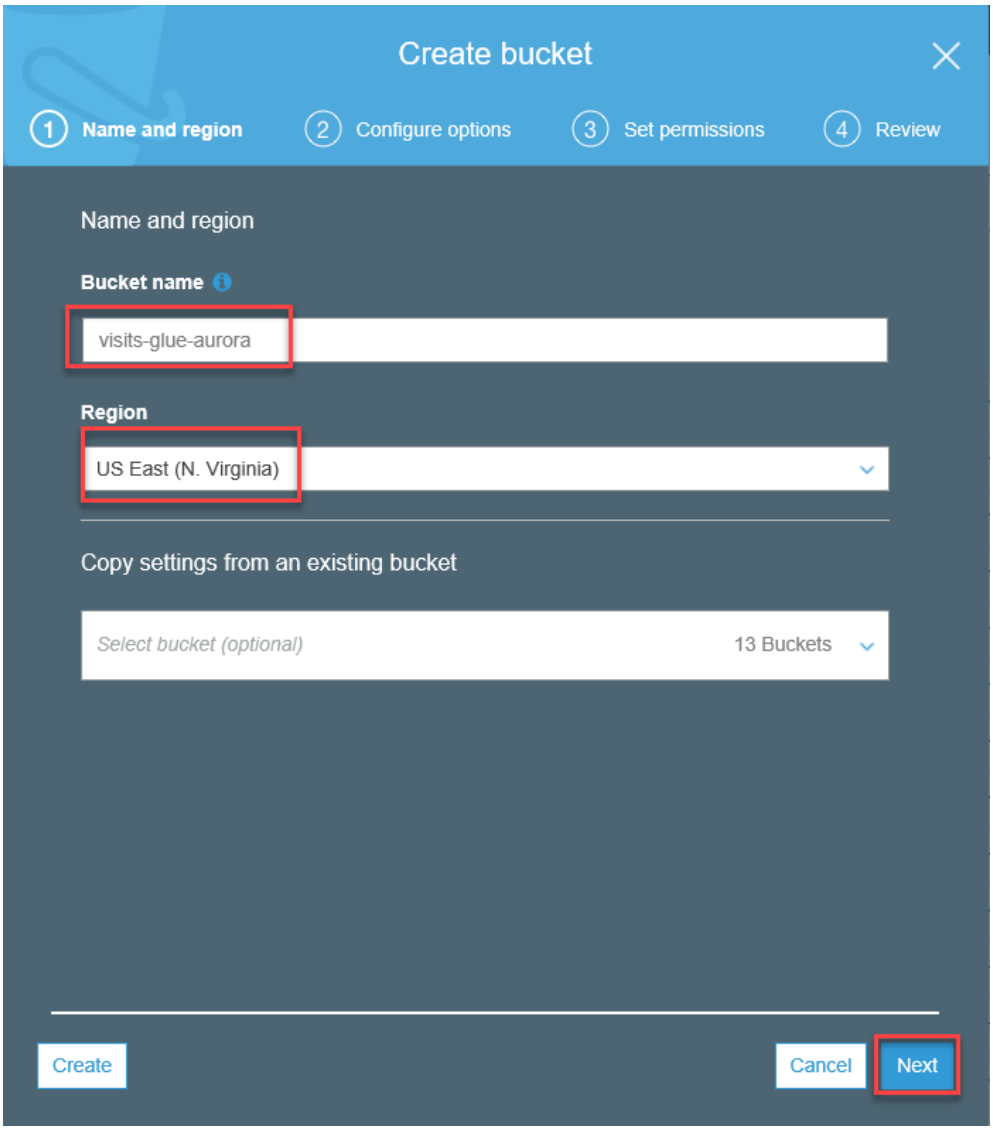
Step 1 - Create a Bucket in Amazon S3 and Upload the CSV File

Navigate to the S3 management console page https://s3.console.aws.amazon.com/s3/home and click **Create Bucket**.

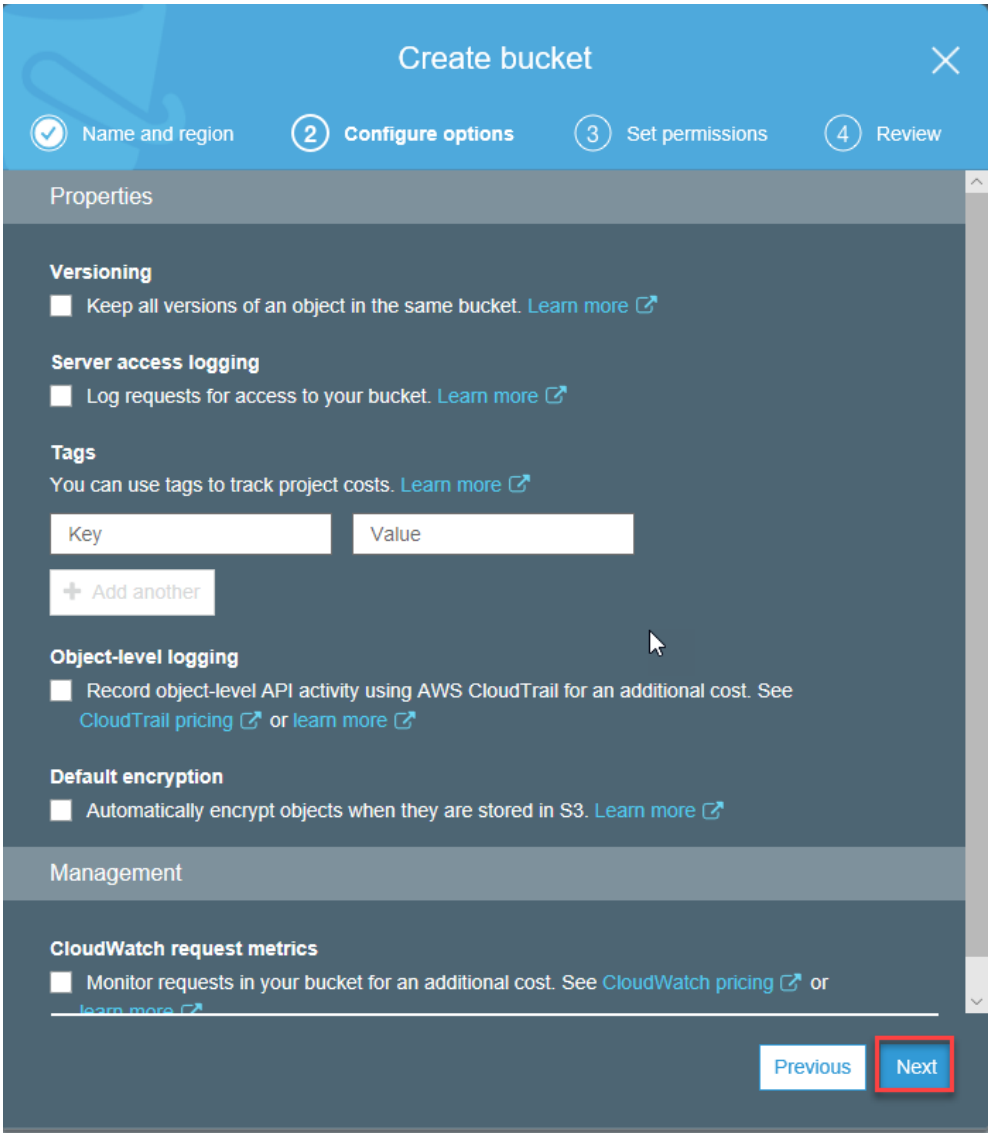
> **Note:** This walkthrough demonstrates how to create the buckets and upload the files manually, which is automated using the S3 API for production ETLs. Using the console to manually execute all the settings will help you get familiar with the terminology, concepts, and work flow.



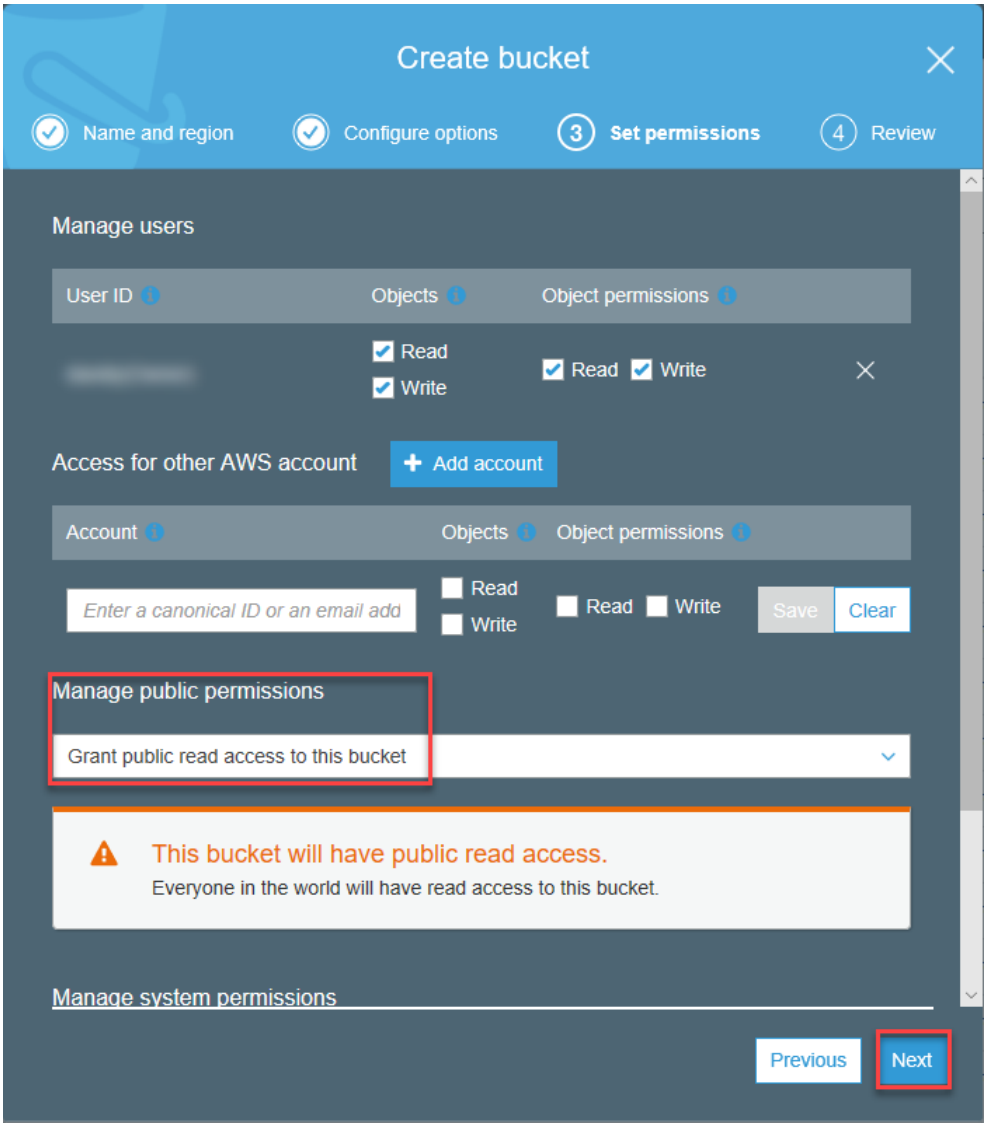
In the create bucket wizard, enter a unique name for the bucket, select a region and click **Next**.



Configure the options. For this walkthrough, skip this phase and click **Next**.

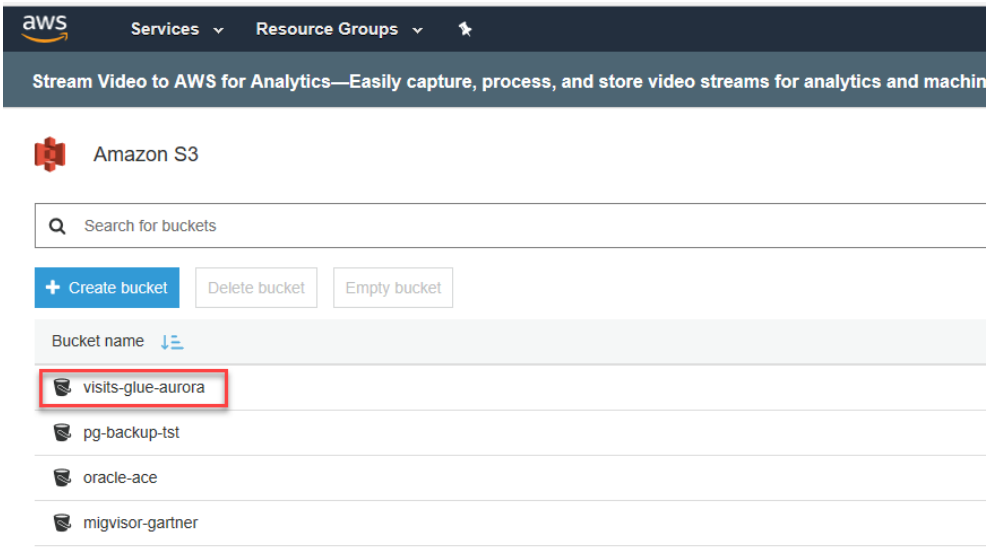


On the **Set Permissions** page, configure access permissions to allow public access to the visits file and click **Next**.

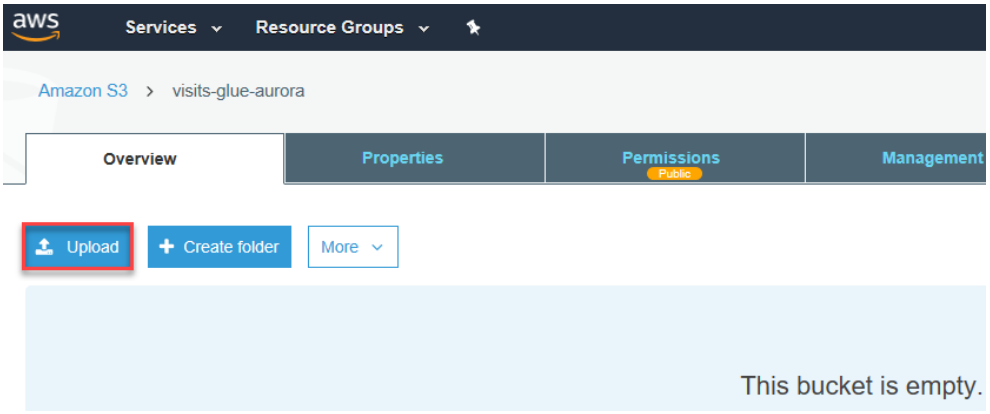


Review your settings and click **Create Bucket**.

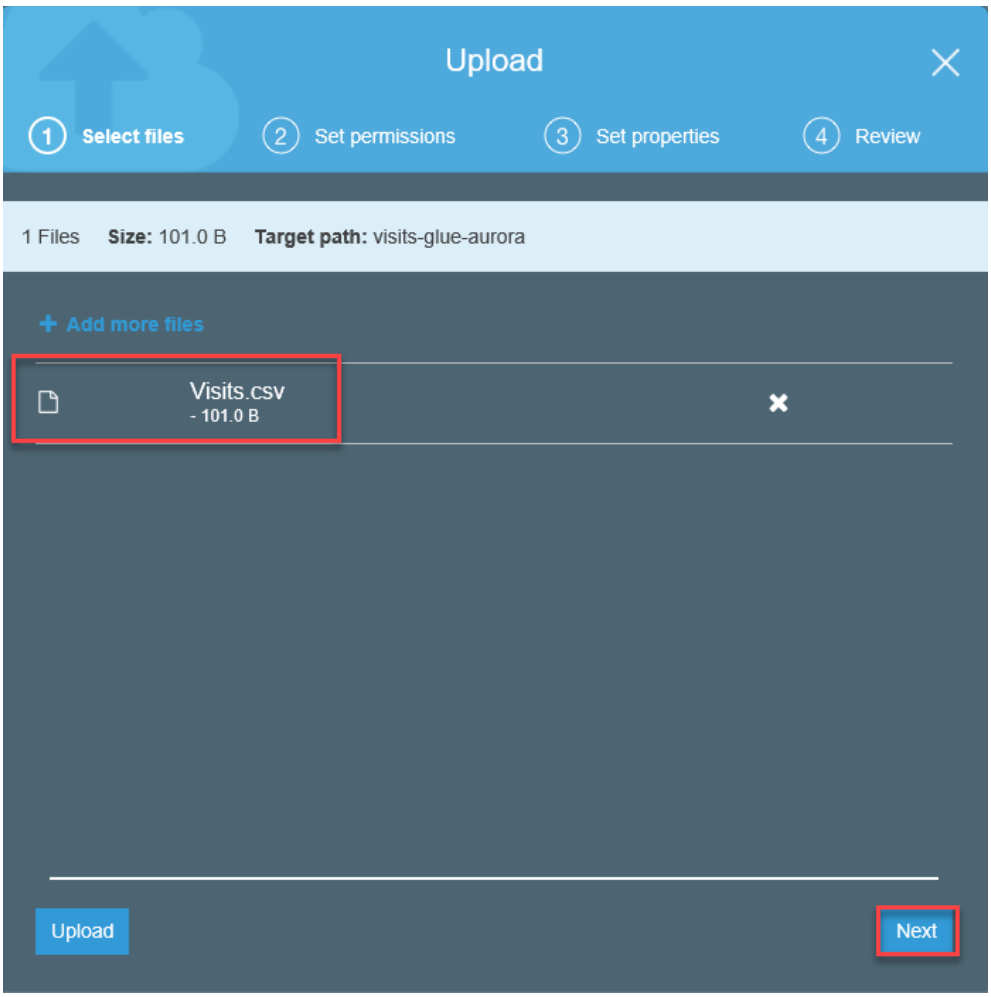
On the S3 Management Console, click the newly created bucket.



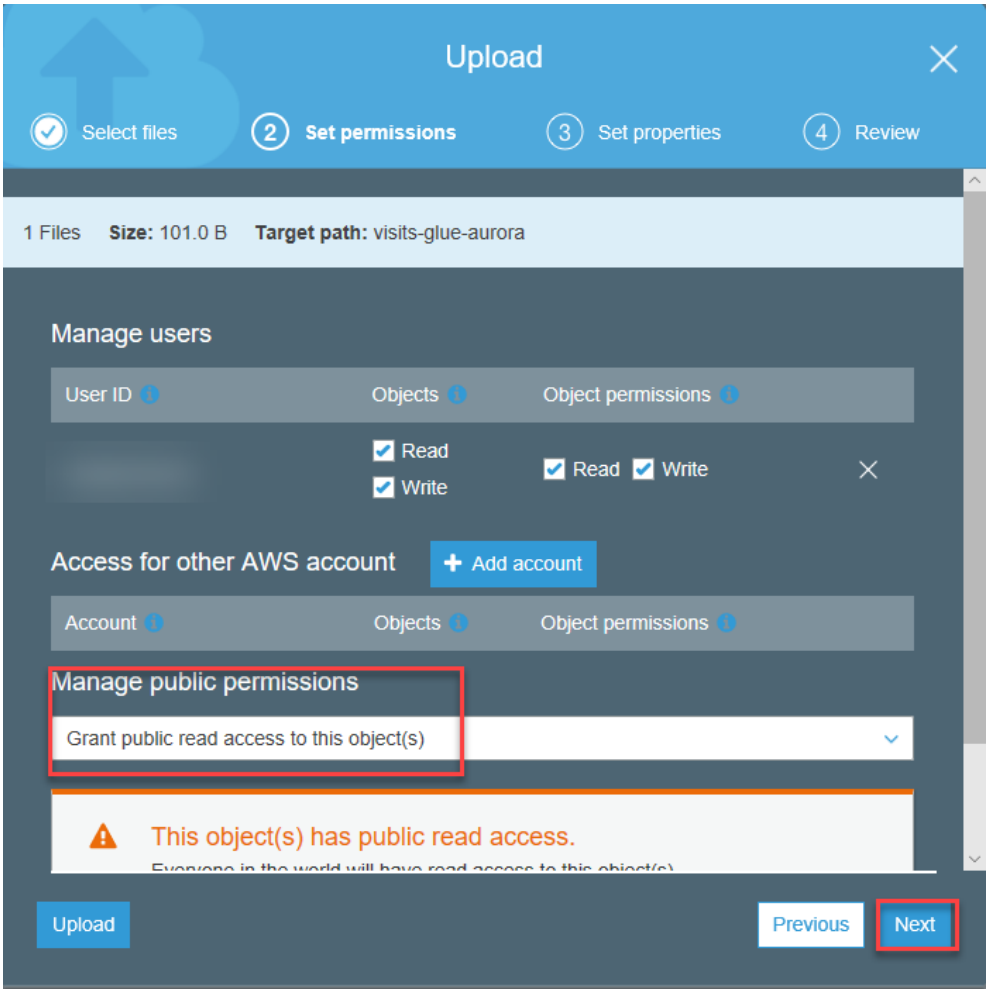
On the bucket page, click **Upload**.



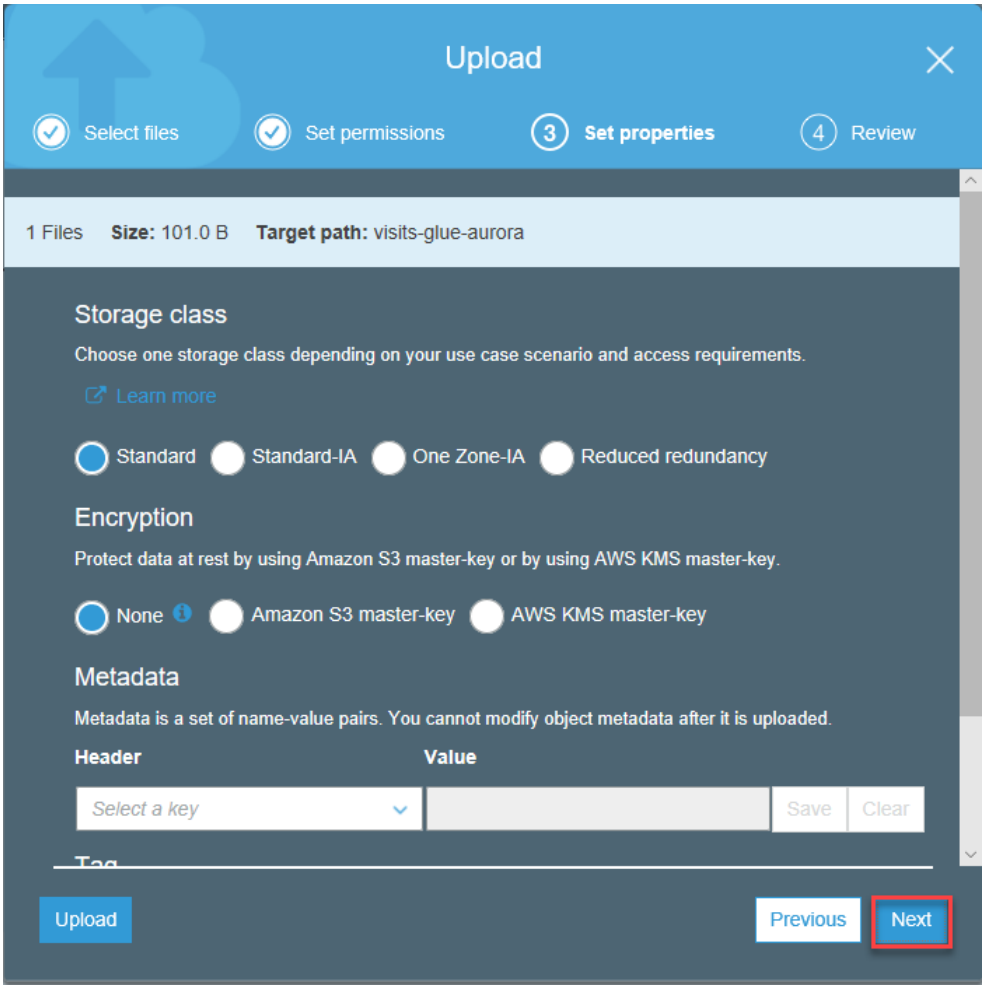
On the upload page, either "drag and drop" or use the **Add Files** button to select the upload files. Click **Next**.



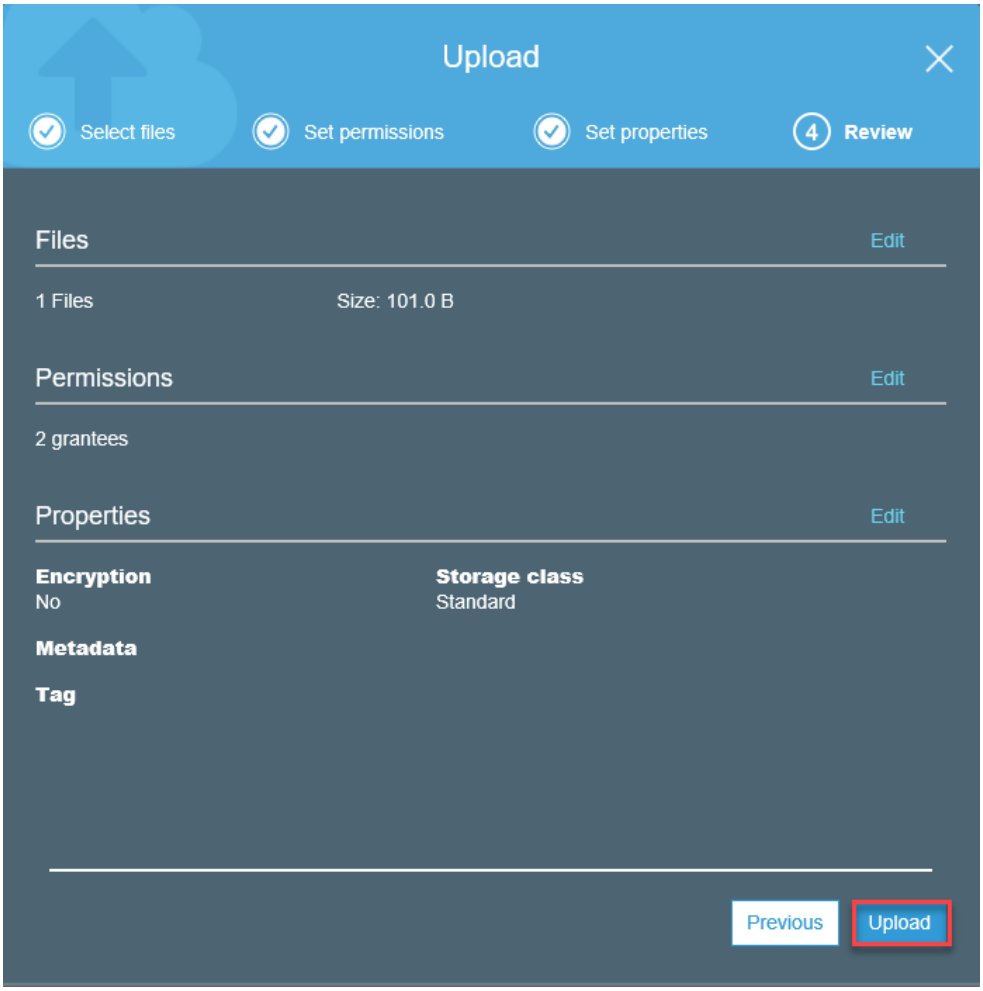
Set the required permissions and click **Next**. For this walk-through, allow public access to the visits file. Click **Next**.



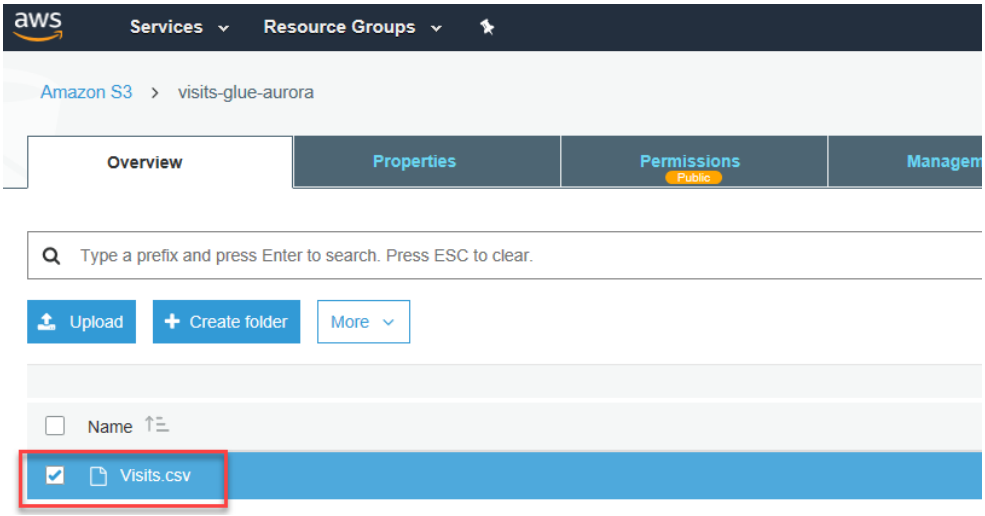
On the **Set Properties** page, select the required options. For this walk-through, skip this step. Click **Next**.



Review the settings and click **Upload**.



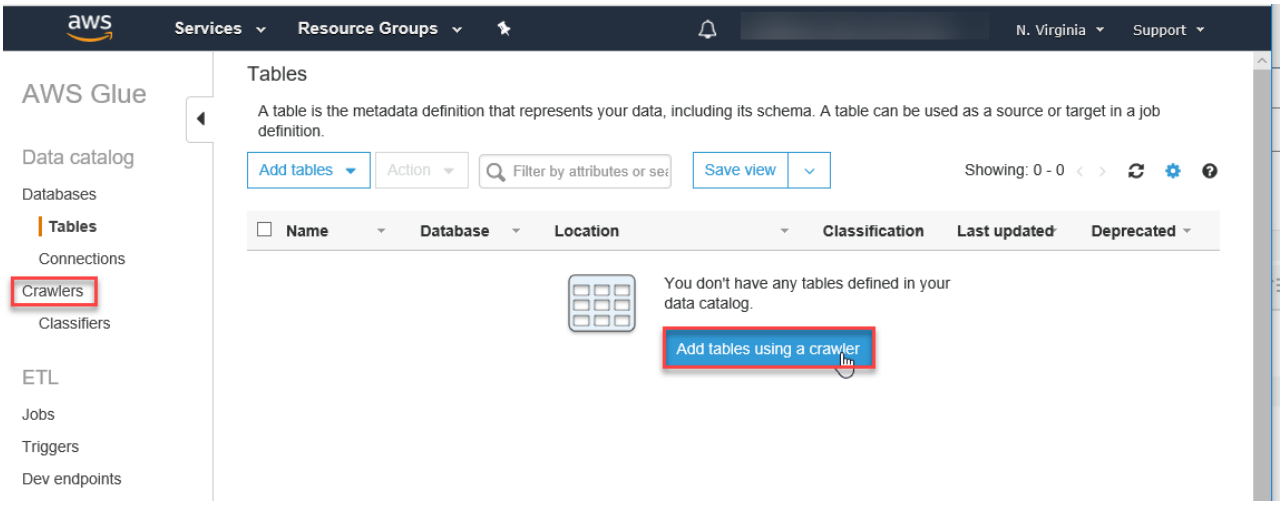
Ensure the file uploads successfully and appears in the file list for the bucket.



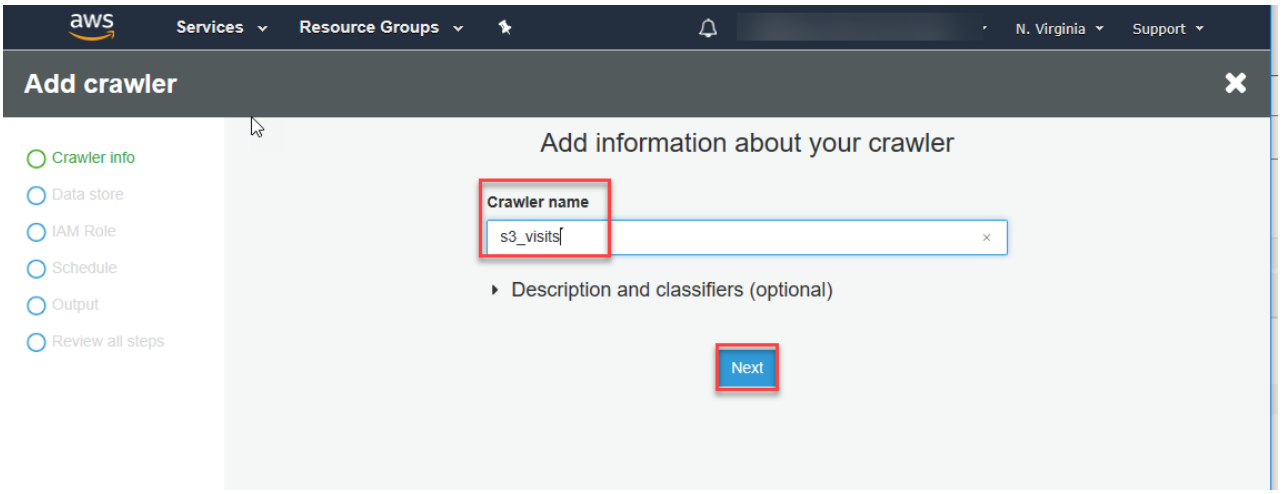
Step 2 - Add an Amazon Glue Crawler to Discover and Catalog the Visits File

Navigate to the Amazon Glue management console page at https://-console.aws.amazon.com/glue/home.

Click **Add tables using a crawler**. Alternatively, click the **Crawlers** navigation link on the left and then click **Add Crawler**.

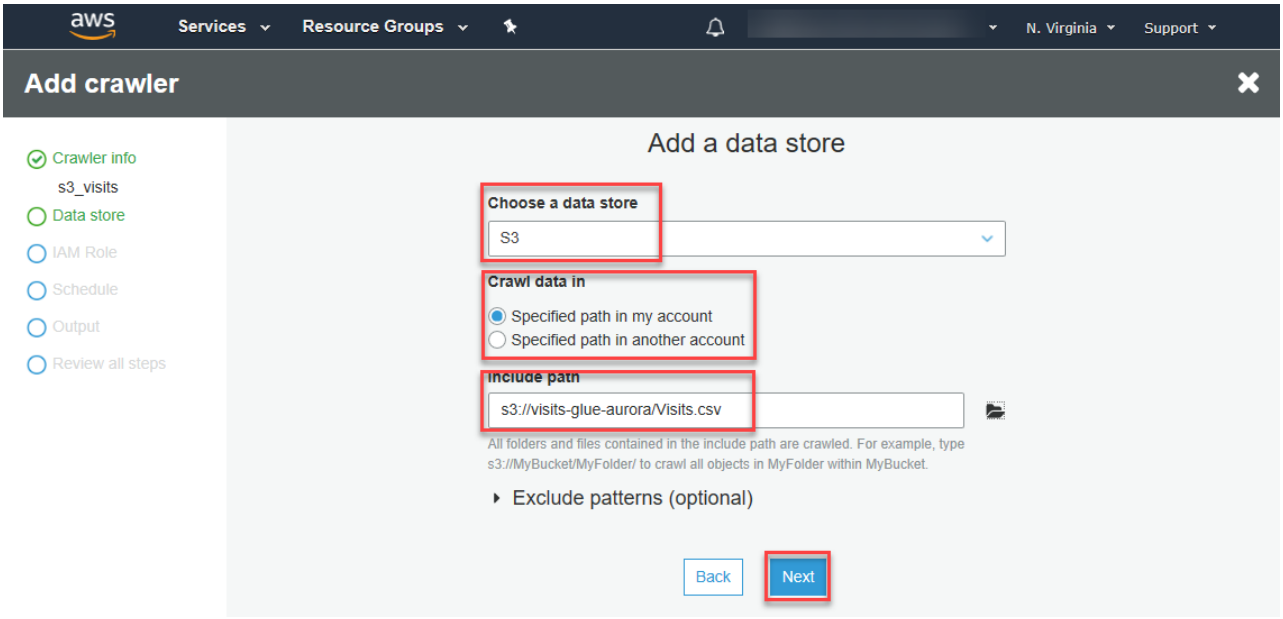


Provide a descriptive name for the crawler and click **Next**.

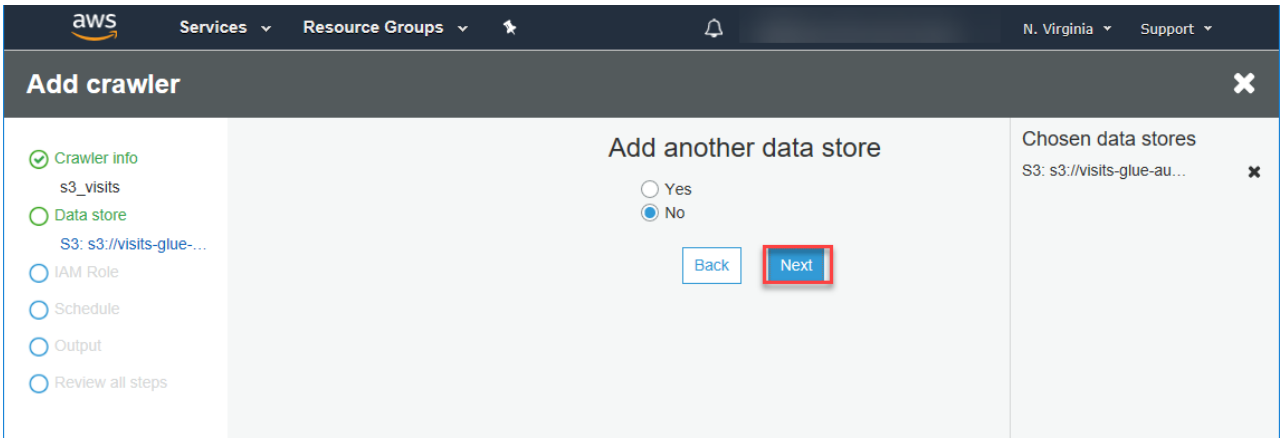


Leave the default S3 data store and choose whether the file is in a path in your account or another account. For this example, the path is in my account and specified in the **Include path** text box. Click **Next**.

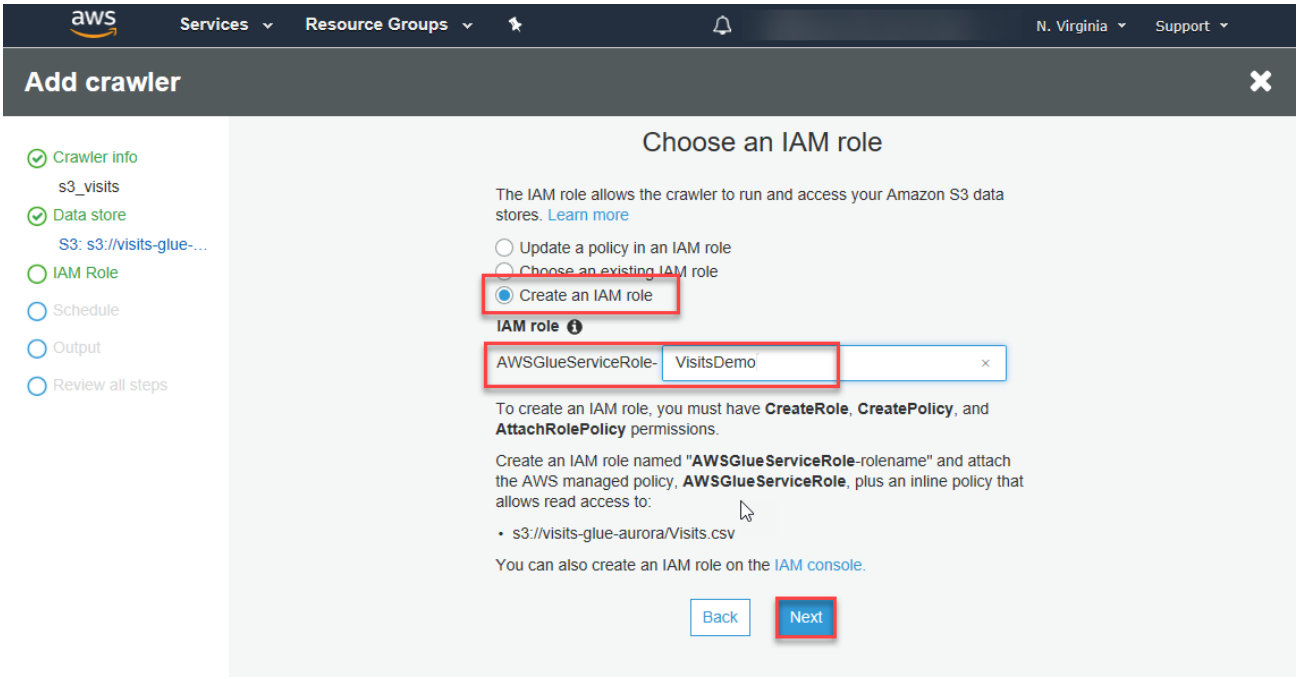
> **Note:** Click the small folder icon to the right of the **Include path** text box to open a visual folder hierarchy navigation window.



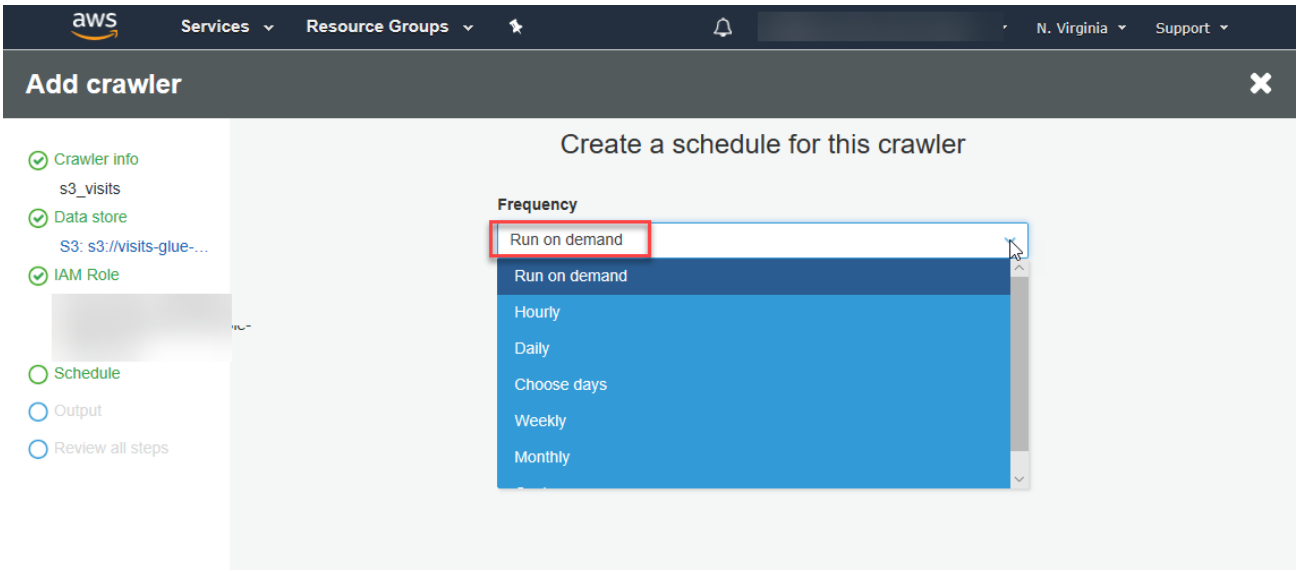
Select whether the crawler accesses another data store or not. For this example only uses the visits file. Click **Next**.



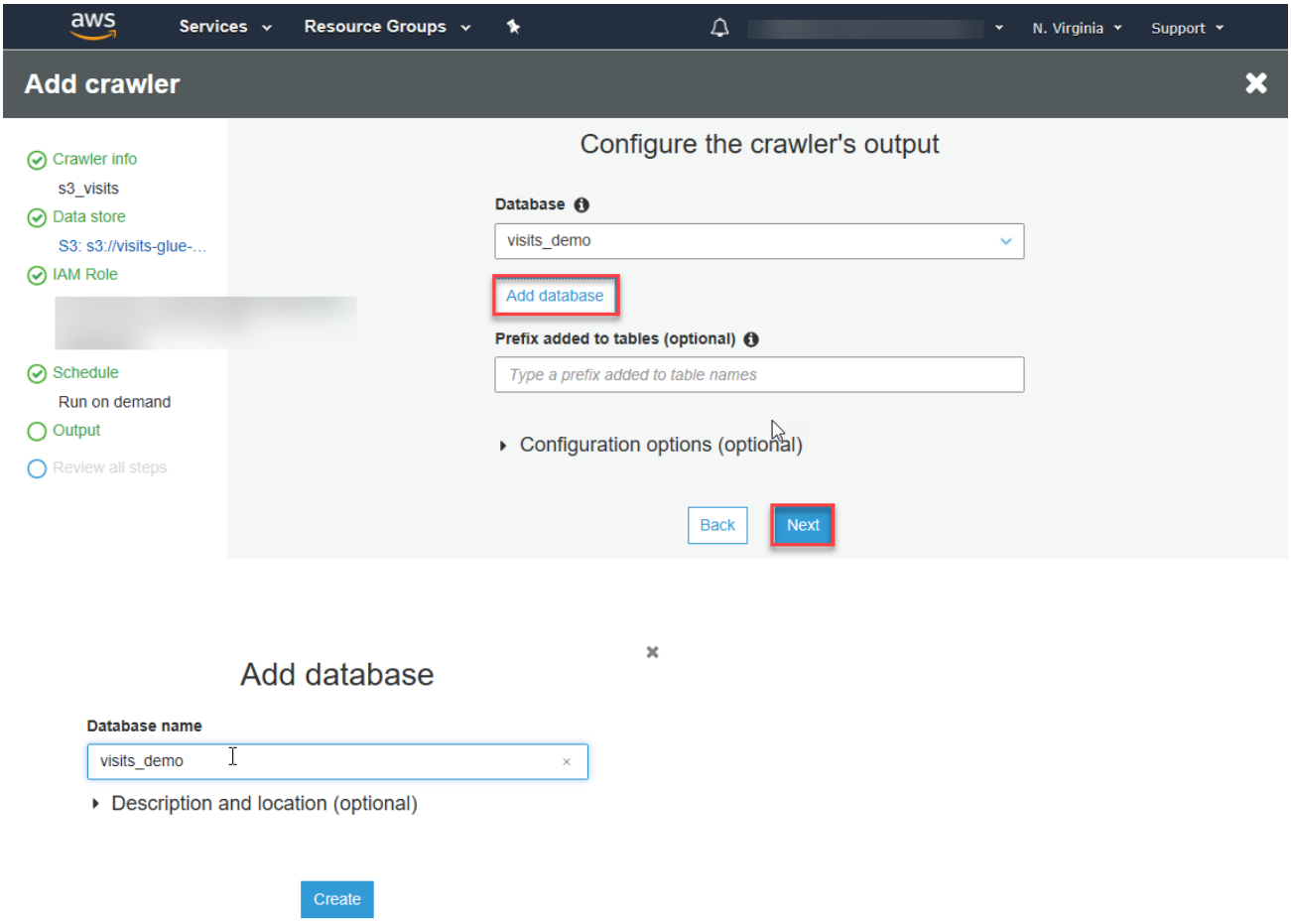
The IAM role window allows selection of the security context the crawler uses to execute. You can choose an existing role, update an existing policy, or create a new role. For this example, create a new role. Click **Next**.



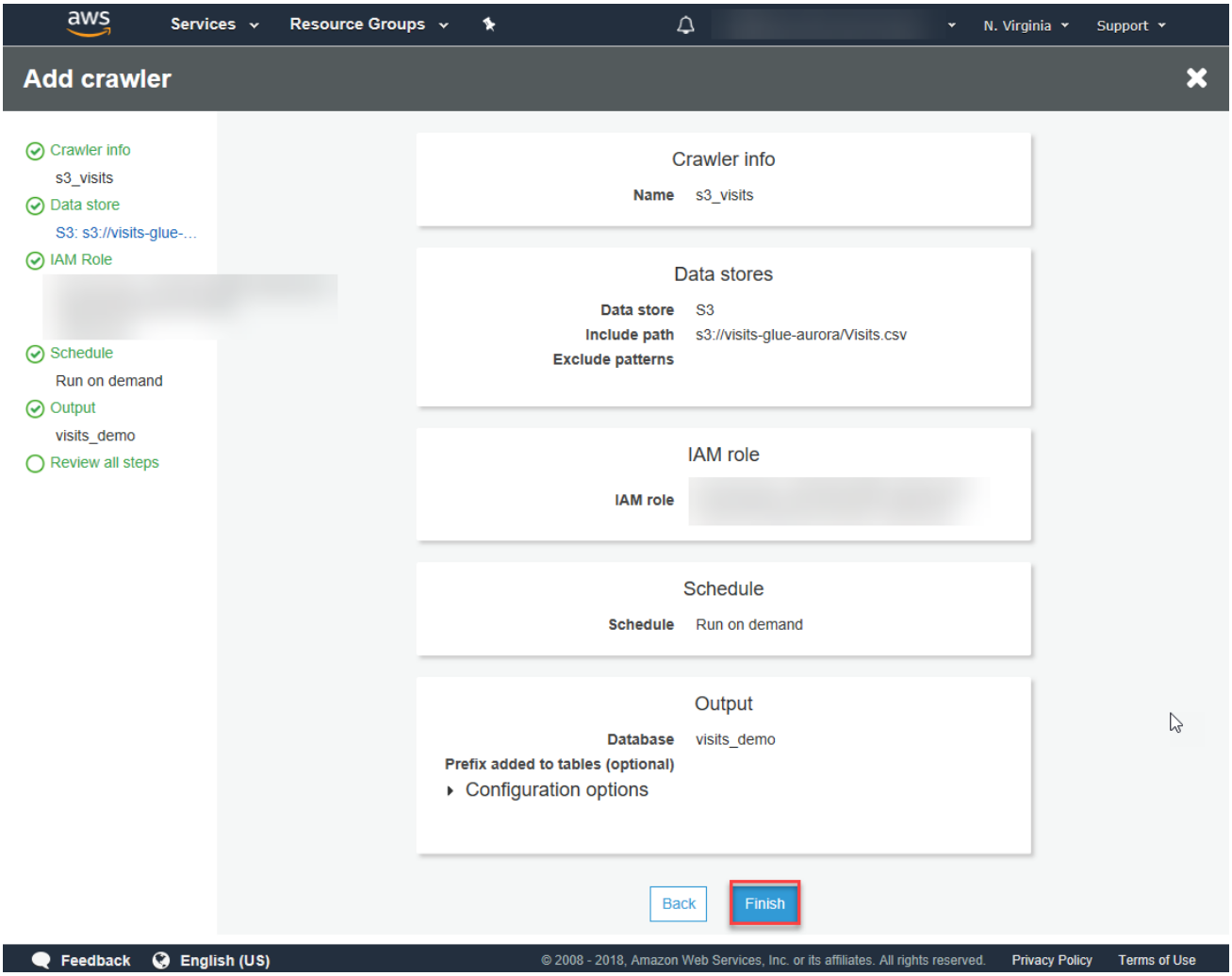
Choose the crawler schedule and frequency. For this example, use **Run on demand**. Click **Next**.



Click **Add database** and provide a name for the new catalog database. Enter an optional table prefix for easy reference. Click **Next**.



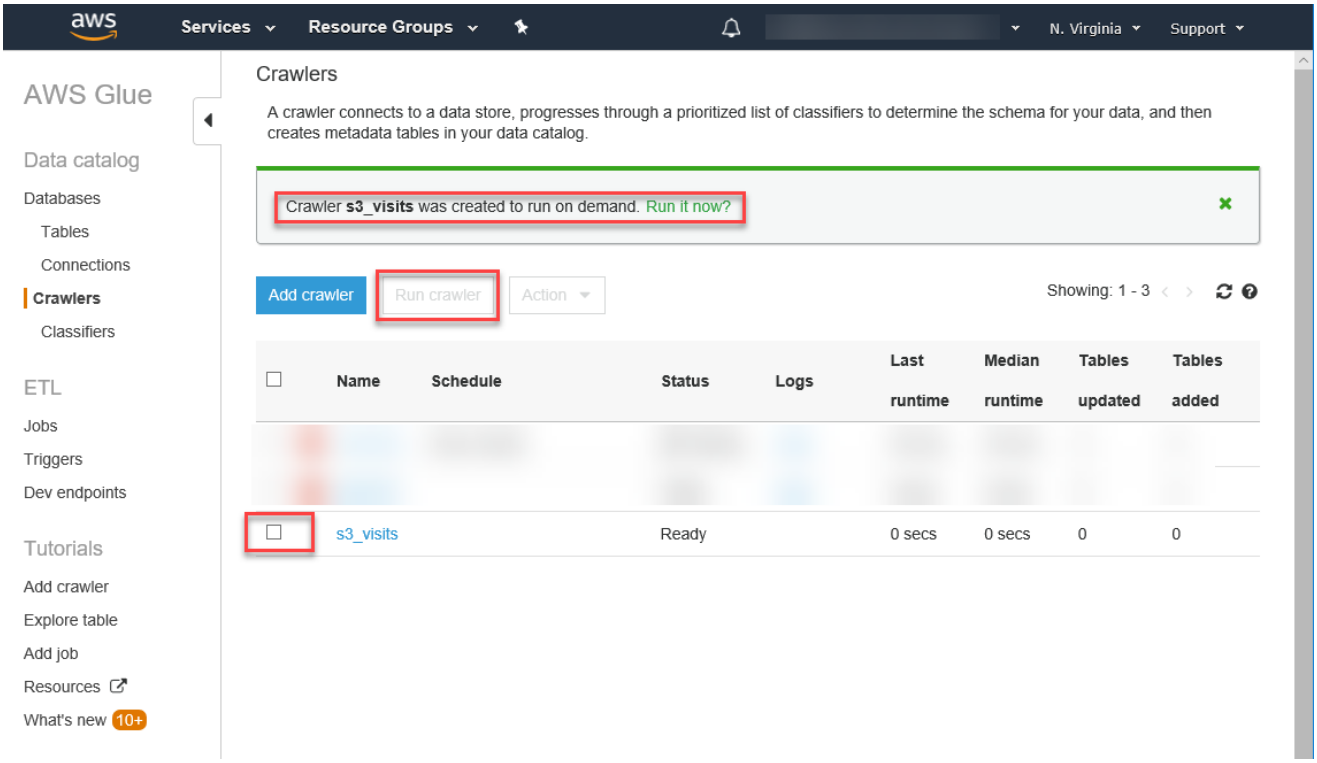
Review your entries and click **Finish** to create the crawler.



Step 3 - Run the Crawler

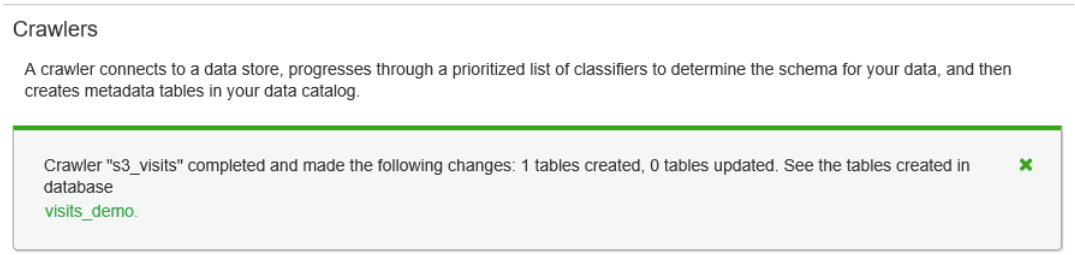
Navigate to the **Crawlers** page on the glue management console https://-console.aws.amazon.com/glue/home?catalog:tab=crawlers.

Since you just created a new crawler, a message box asks if you want to run it now. You can click the link or check the check-box near the crawler's name and click the **Run crawler** button.

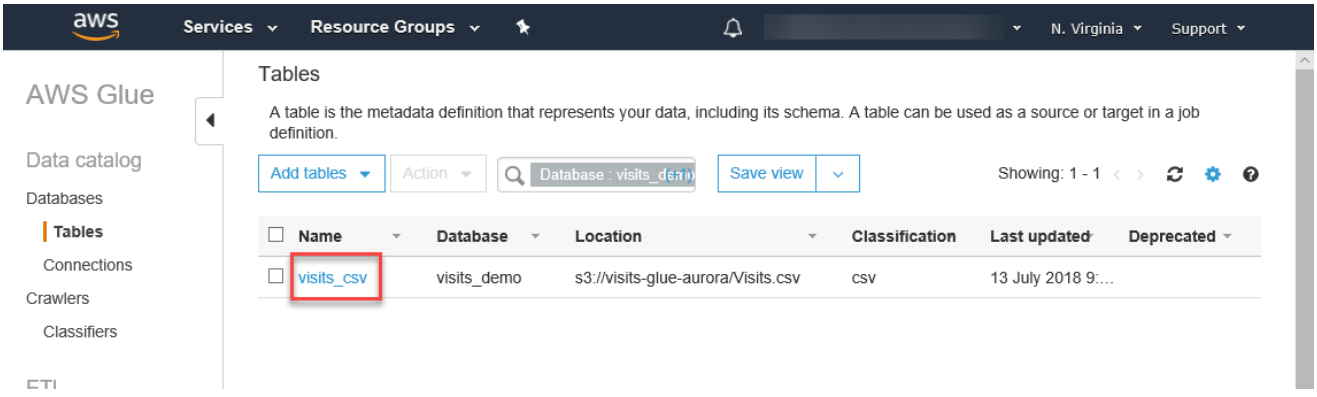


After the crawler completes, the Visits table should be discovered and recorded in the catalog in the table specified.

The following message box appears on the page:

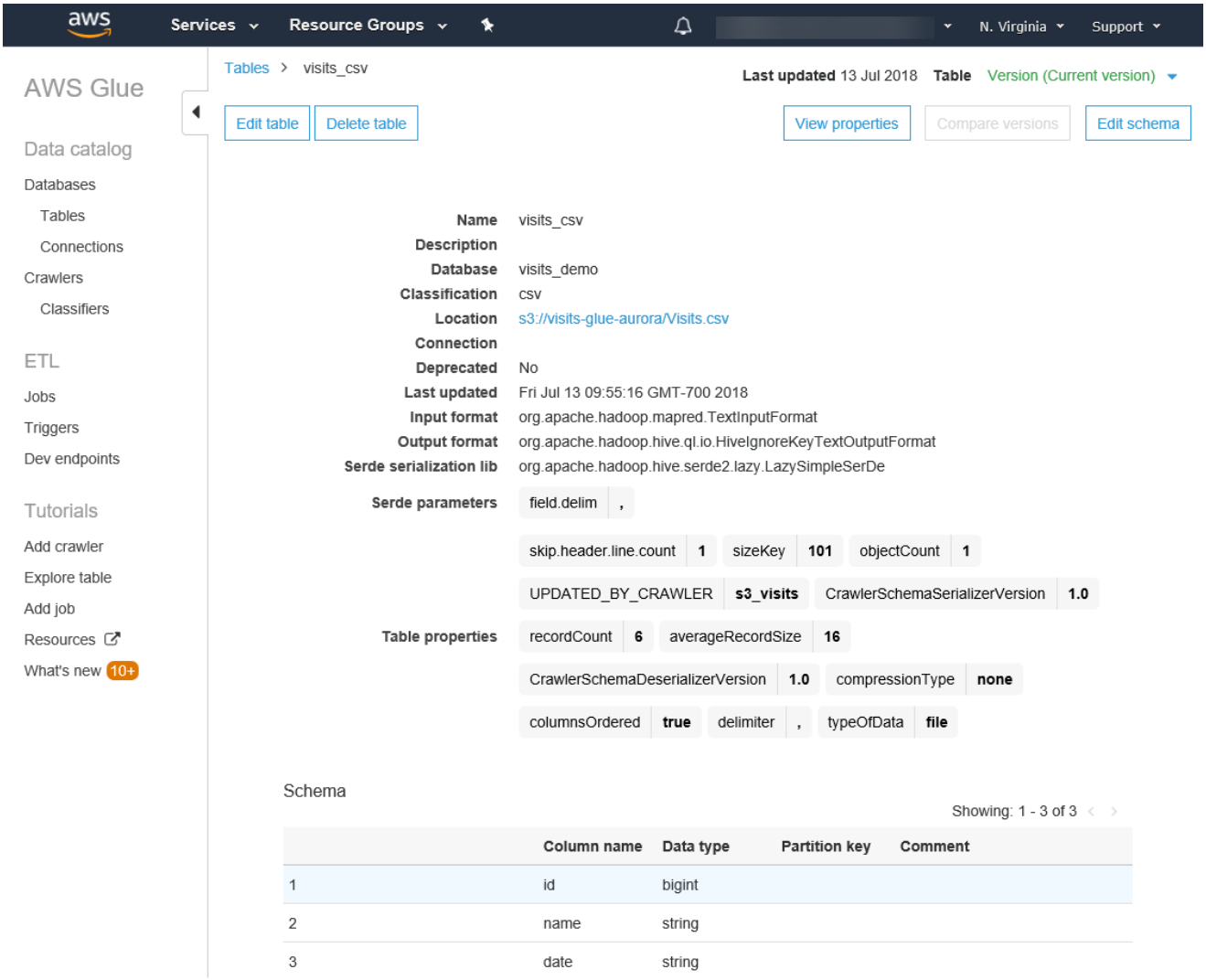


Click the link to get to the table that was just discovered and then click the table name.



Verify the crawler identified the table's properties and schema correctly.

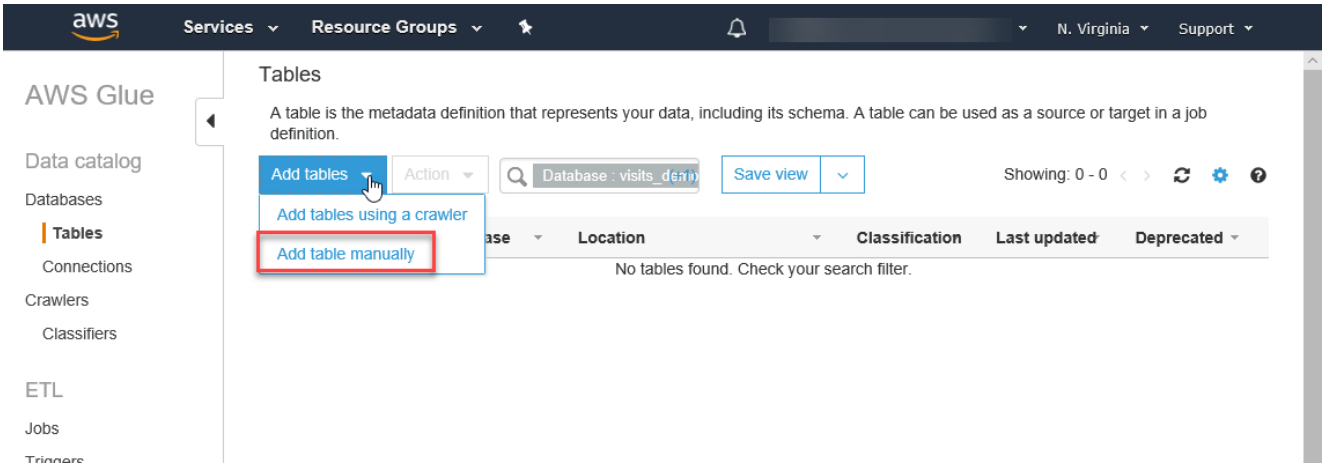
Note: You can manually adjust the properties and schema JSON files using the buttons on the top right.



Optional - Add Tables Manually

If you don't want to add a crawler, you can add tables manually.

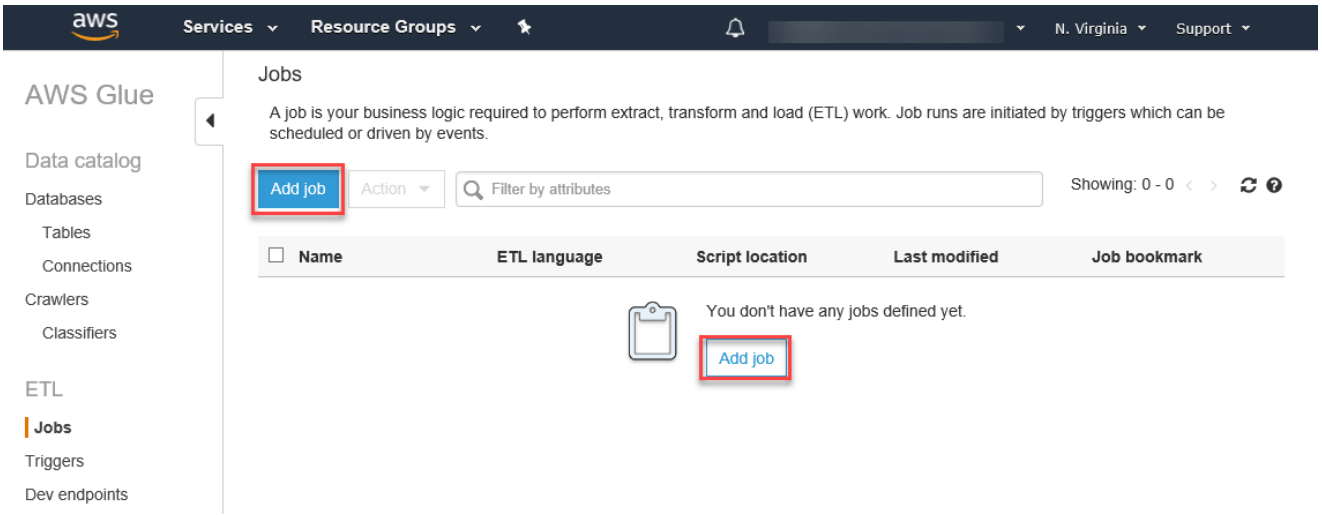
Navigate to https://console.aws.amazon.com/glue/home, the default page is the Tables page. Click **Add tables** and select **Add table manually**.



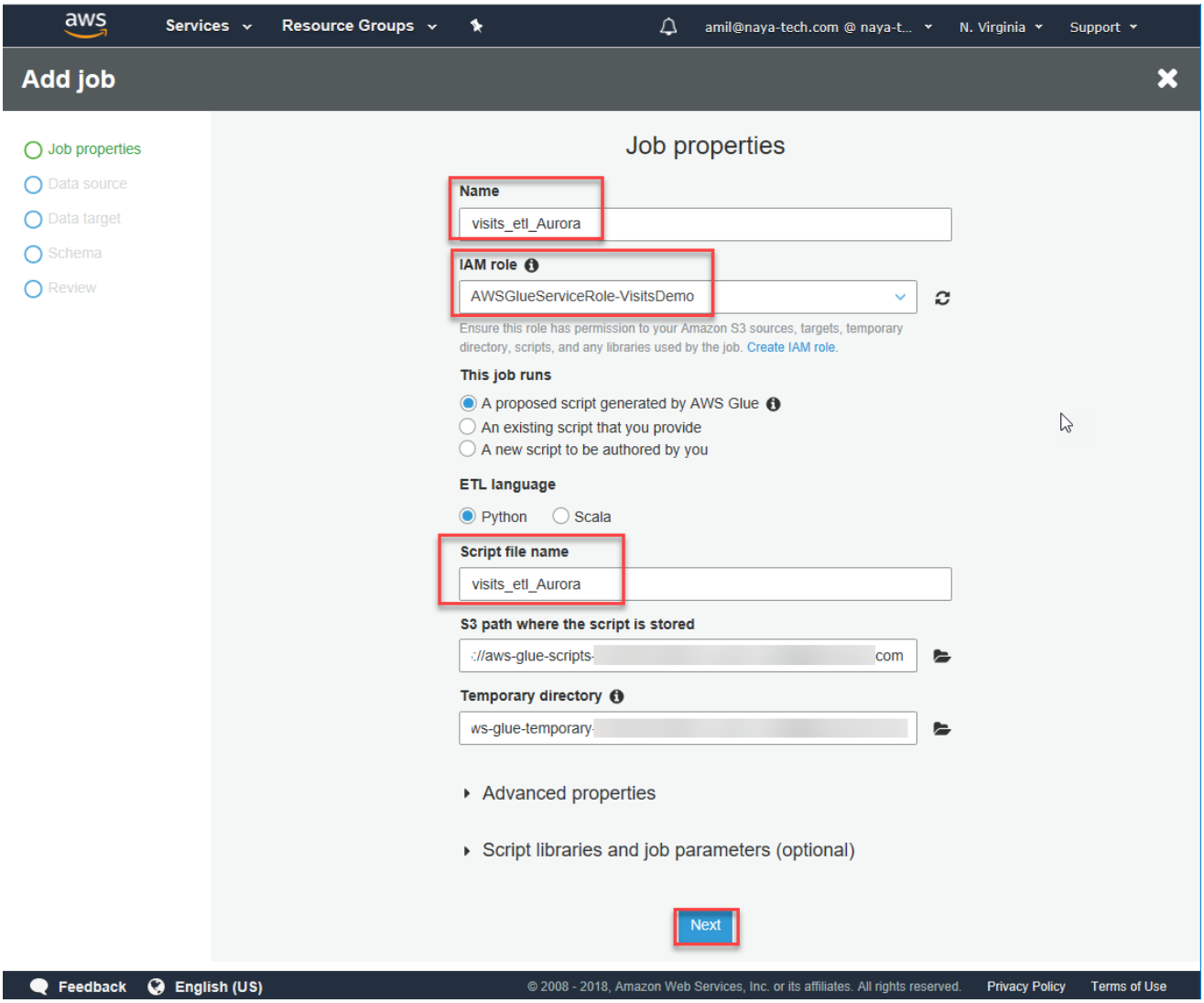
The process is similar the one used for the crawler.

Step 4 - Create an ETL Job to Copy the Visits Table to an Aurora MySQL Database.

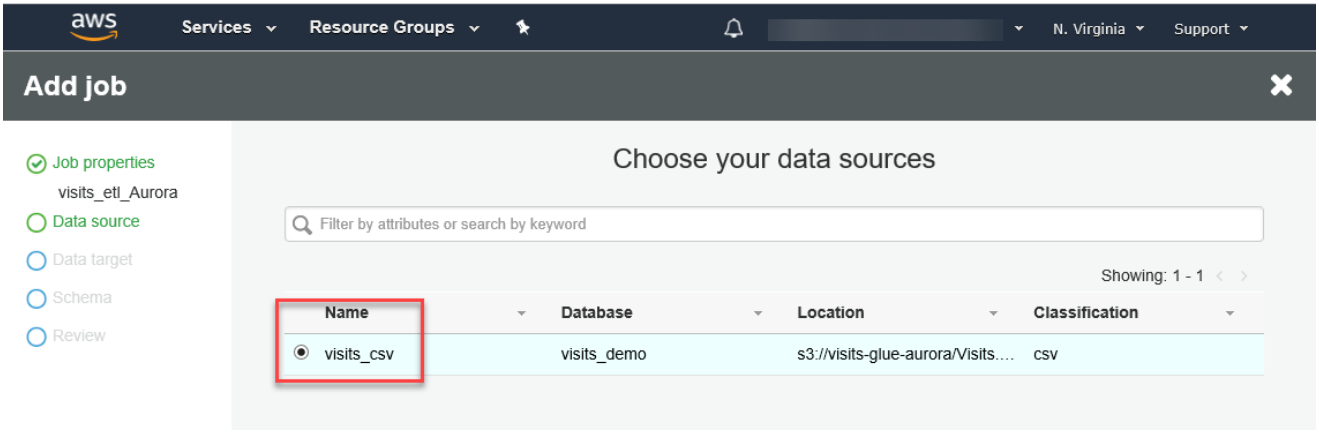
Navigate to the Amazon Glue ETL Jobs page at https://-console.aws.amazon.com/glue/home?etl:tab=jobs. Since this is the first job, the list is empty. Click Add Job.



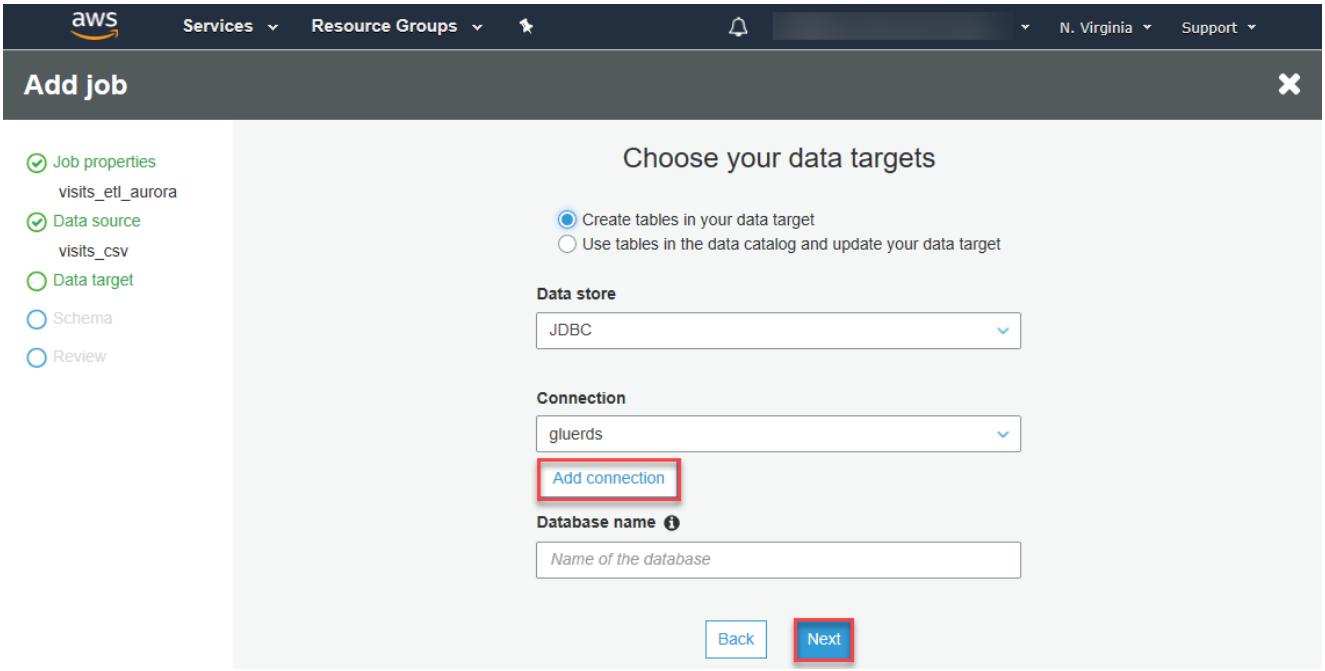
Enter a name for the ETL job and pick a role for the security context. For this example, use the same role created for the crawler. The job may consist of a pre-existing ETL script, a manually-authored script , or an automatic script generated by Amazon Glue. For this example, use Amazon Glue. Enter a name for the script file or accept the default, which is also the job's name. Configure advanced properties and parameters if needed and click **Next**.



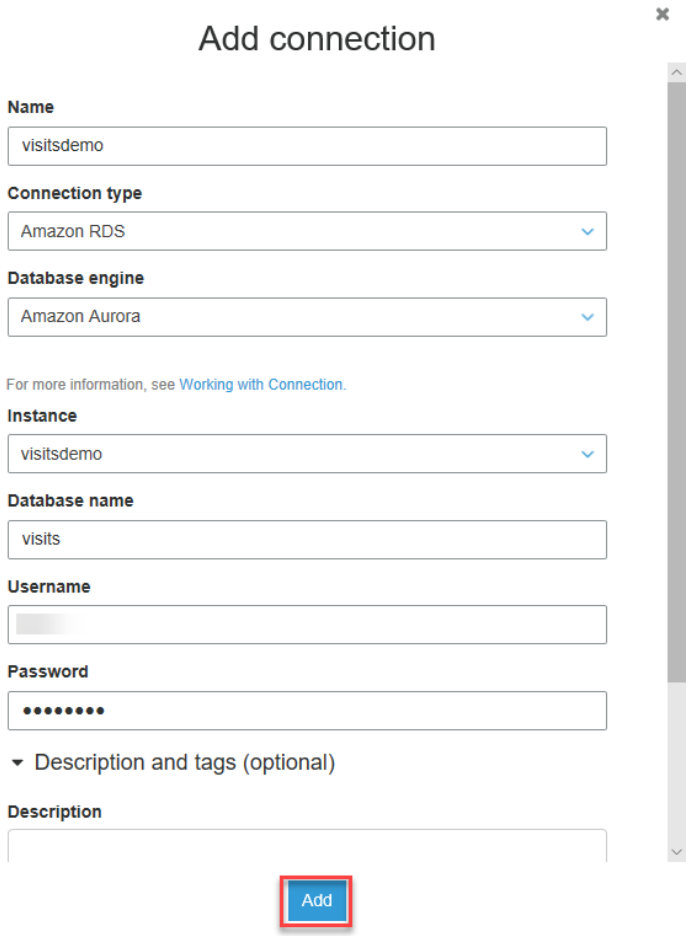
Select the data source for the job (in this example, there is only one). Click **Next**.



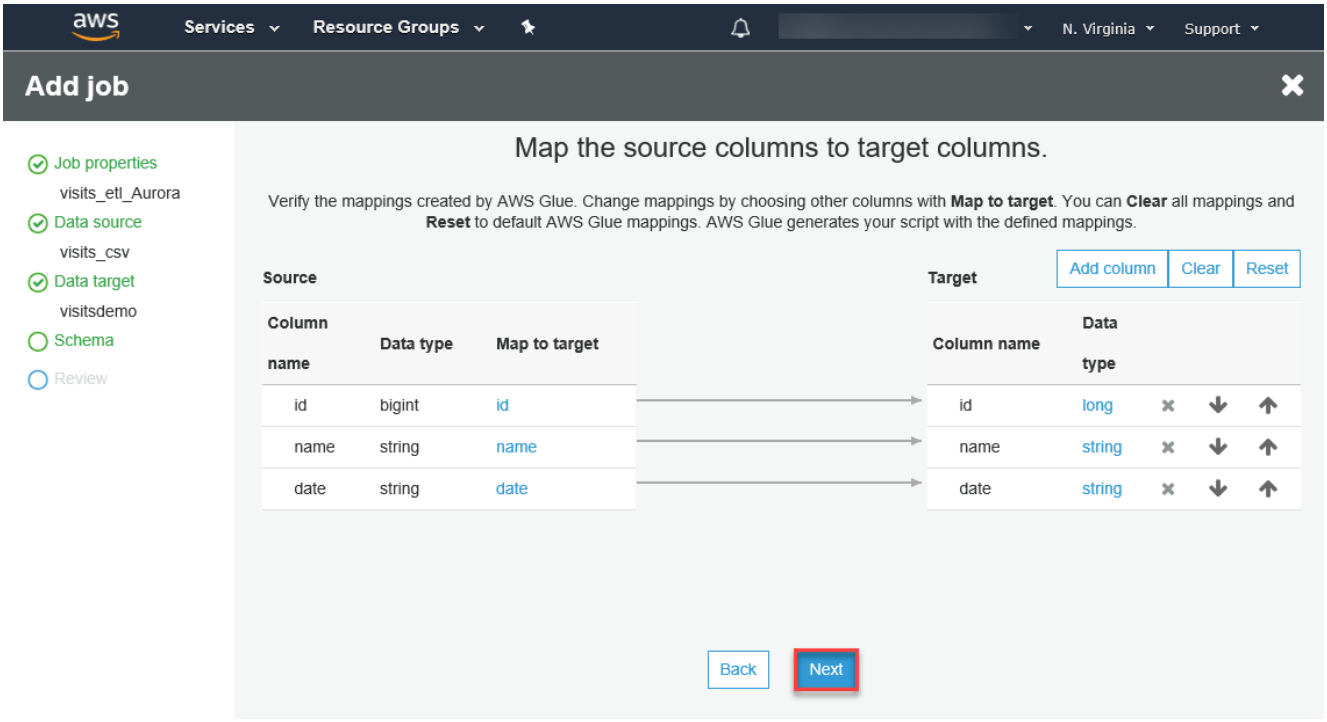
On the **Data Target** page, select **Create tables in your data target**, use the JDBC Data store, and the gluerds connection type. **Click Add Connection**.



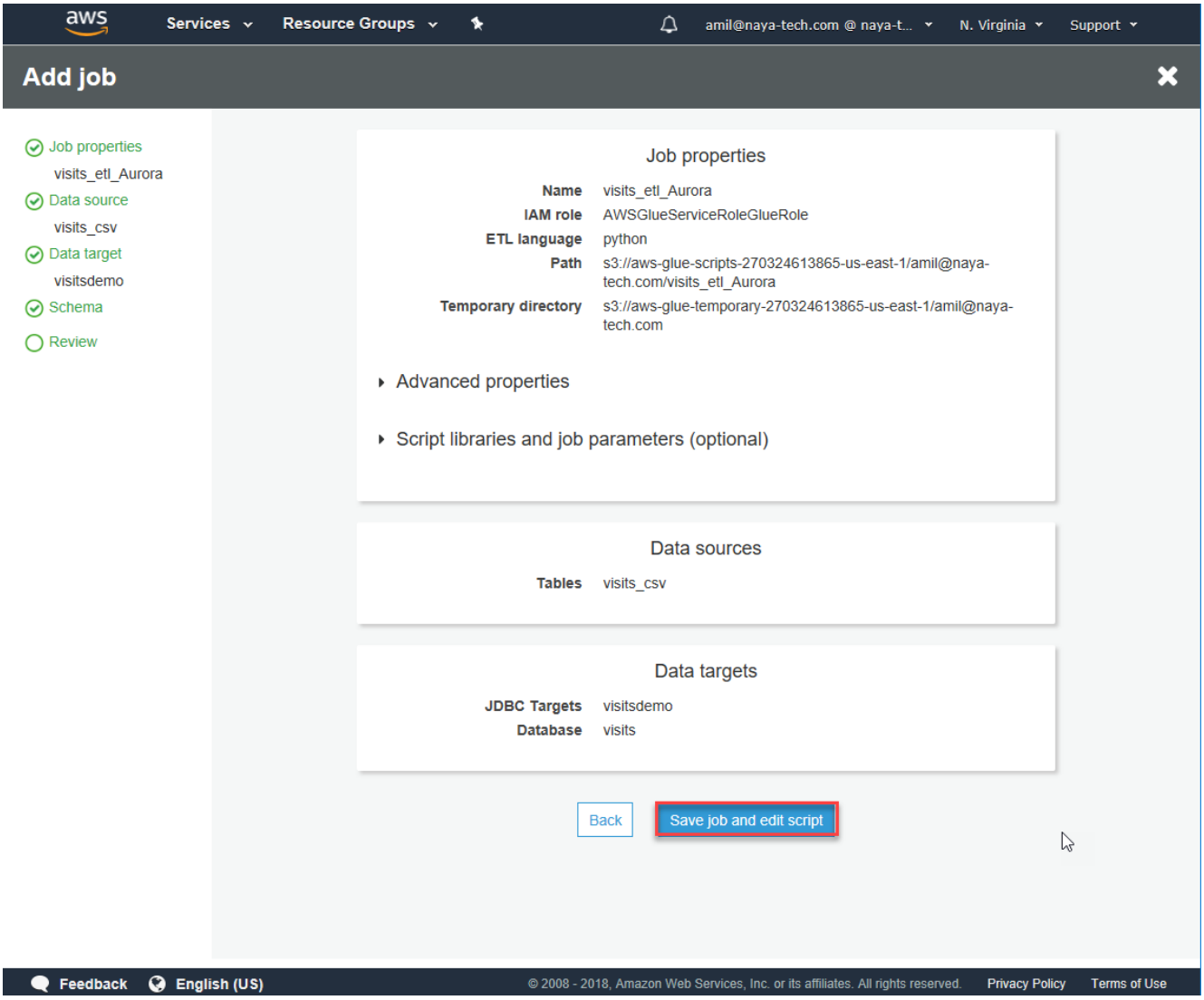
On the **Add connection** page, enter the access details for the Aurora Instance and lick **Add**.



Click **Next** to display the column mapping between the source and target. For this example, leave the default mapping and data types. Click **Next**.

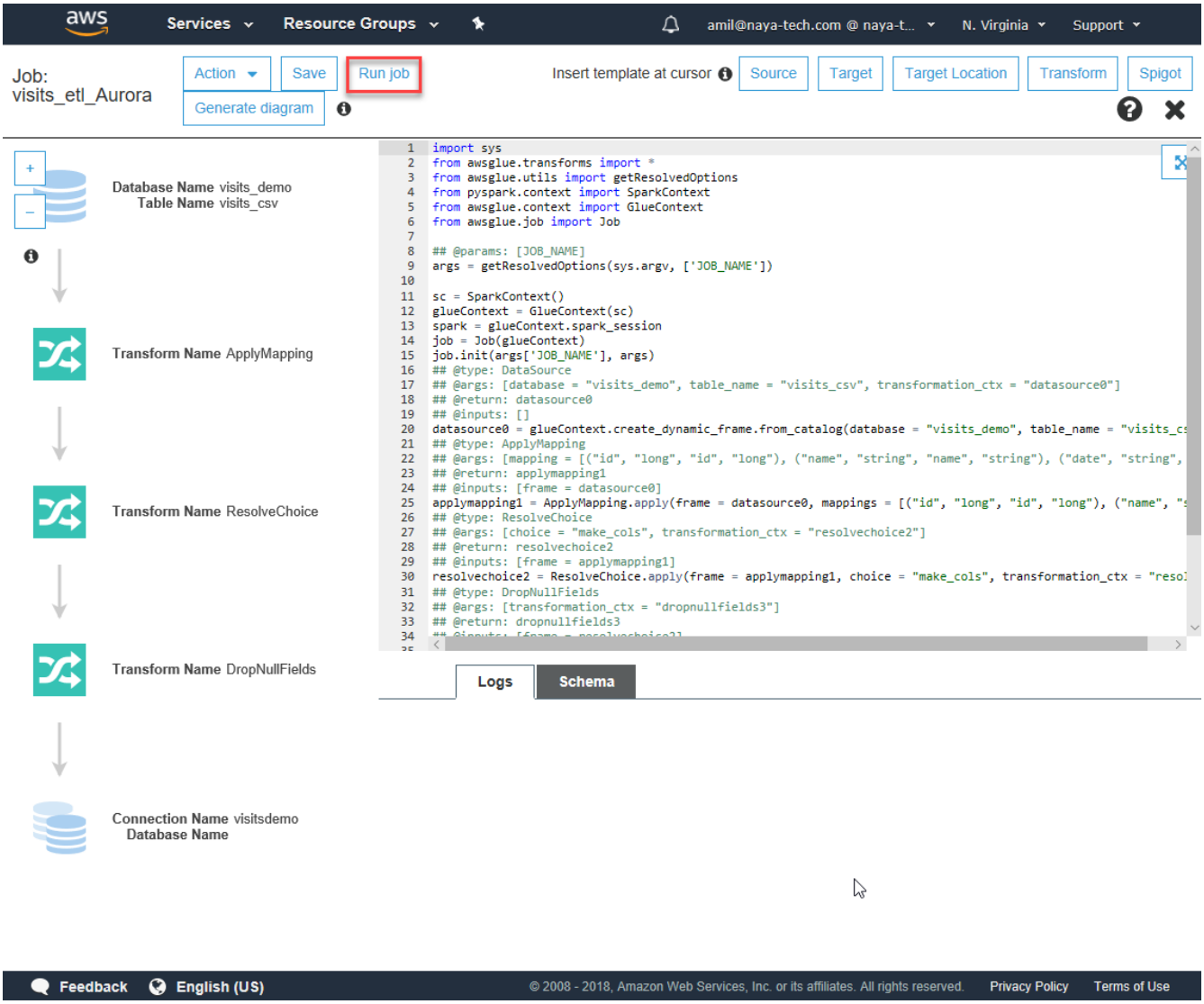


Review the job properties and click **Save job and edit script**.

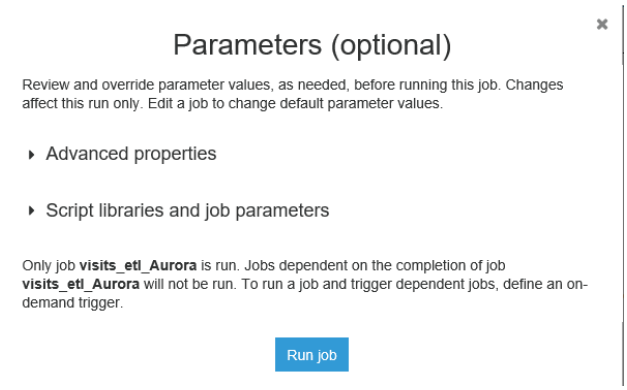


Review the generated script and make manual changes as needed. You can use the built-in templates for source, target, target location, transform, and spigot using the buttons at the top right section of the screen.

For this example, run the script as-is. Click **Run Job**.

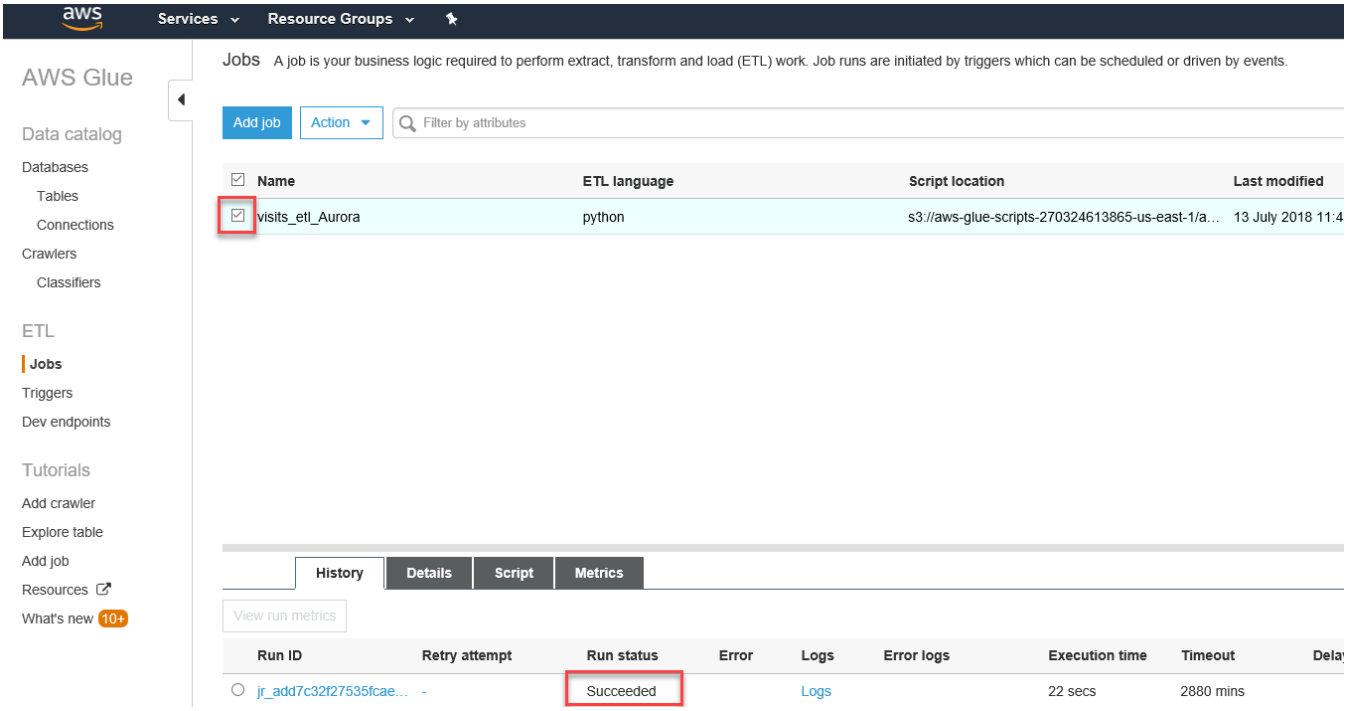


The optional parameters window displays. Click **Run Job**.

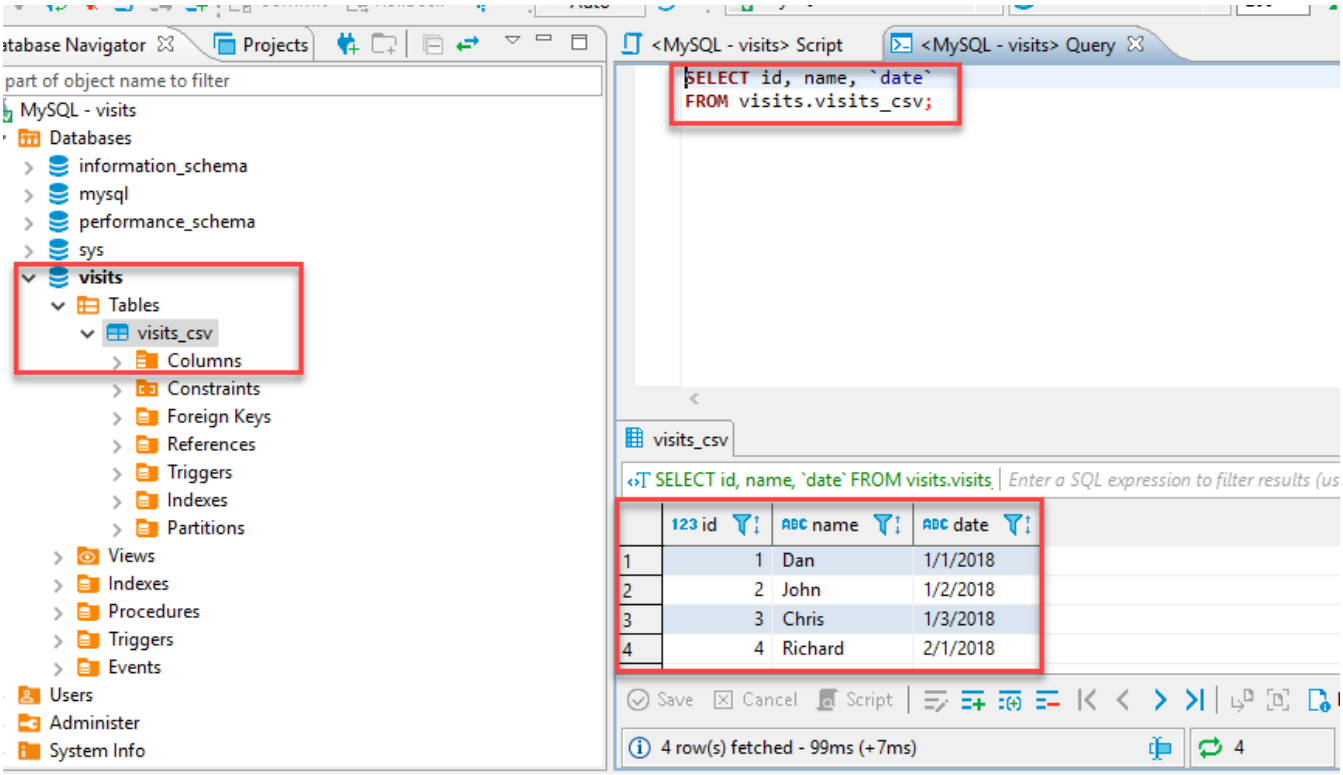


Navigate back to the glue management console jobs page at https://-console.aws.amazon.com/glue/home?etl:tab=jobs.

On the history tab, verify the job status as **Succeeded** and view the logs if needed.



Now open your query IDE, connect to the Aurora MySQL cluster, and query the visits database to make sure the data has been transferred successfully.



For more information, see

- https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html
- https://aws.amazon.com/glue/developer-resources/

Migrate from SQL Server Viewing Server Logs

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• View logs from the Amazon RDS console, the Amazon RDS API, the AWS CLI, or the AWS SDKs

Overview

SQL Server logs system and user generated events to the *SQL Server Error Log* and to the *Windows Application Log*. It logs recovery messages, kernel messages, security events, maintenance events, and other general server level error and informational messages. The Windows Application Log contains events from all windows applications including SQL Server and SQL Server agent.

SQL Server Management Studio Log Viewer unifies all logs into a single consolidated view. You can also view the logs with any text editor.

Administrators typically use the SQL Server Error Log to confirm successful completion of processes, such as backup or batches, and to investigate the cause of run time errors. These logs can help detect current risks or potential future problem areas.

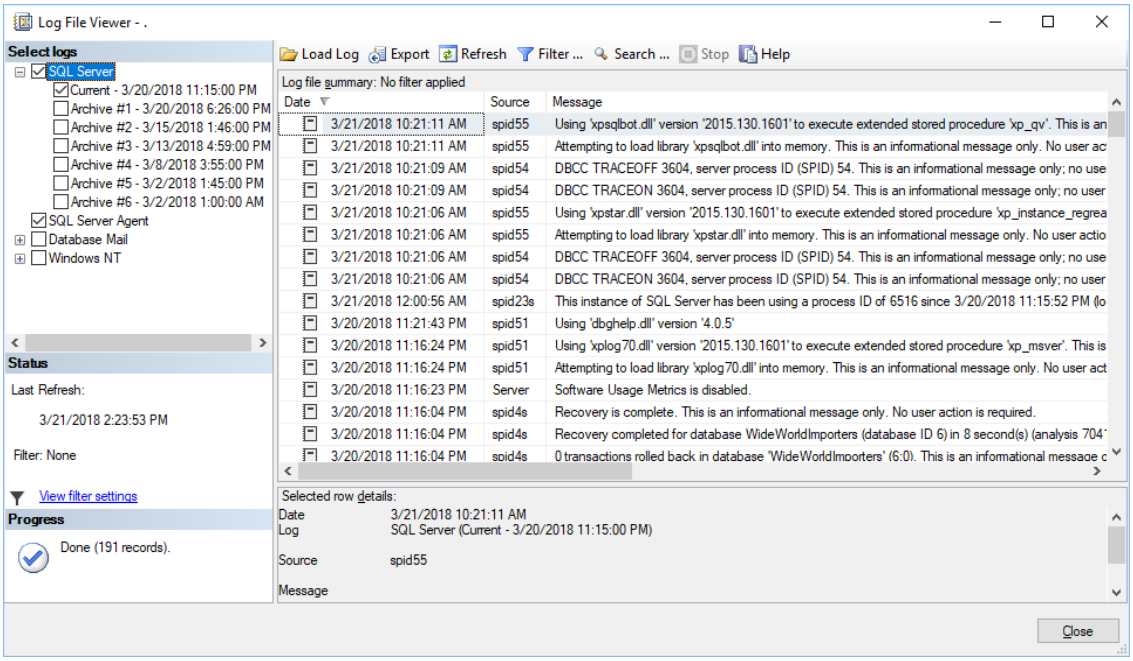
To view the log for SQL Server, SQL Server Agent, Database Mail, and Windows applications, open the SQL Server Management Studio Object Explorer pane, navigate to **Management > SQL Server Logs** , and double click the current log.

The following table identifies some common error codes database administrators typically look for in the error logs:

Error Code	Error Message
1105	Could not allocate space
3041	Backup Failed
9002	Transaction Log Full
14151	Replication agent failed
17053	Operating System Error
18452	Login Failed
9003	Possible database corruption

Examples

The following screenshot shows typical Log File Viewer content:



For more information, see https://docs.microsoft.com/en-us/sql/tools/configuration-manager/monitoring-the-error-logs

Migrate to Aurora MySQL Viewing Server Logs

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• View logs from the Amazon RDS console, the Amazon RDS API, the AWS CLI, or the AWS SDKs

Overview

Aurora MySQL provides administrators with access to the MySQL error log, slow query log, and the general log.

The MySQL Error Log is generated by default. To generate the slow query and general logs, set the corresponding parameters in the database parameter group.
For more details about parameter groups, see Server Options.

You can view Aurora MySQL logs directly from the Amazon RDS console, the Amazon RDS API, the AWS CLI, or the AWS SDKs. You can also direct the logs to a database table in the main database and use SQL queries to view the data. To download a binary log, use the *mysqlbinlog* utility.

The system writes error events to the the *mysql-error.log* file, which you can view using the Amazon RDS console. Alternatively, you can use the Amazon RDS API, the Amazon RDS CLI, or the AWS SDKs-retrieve to retrieve the log.

mysql-error.log buffers are flushed every five minutes and are appended to the file*mysql-error-running.log*. The *mysql-error-running.log* file is rotated every hour and retained for 24 hours.

Aurora MySQL writes to the error log only on server startup, server shutdown, or when an error occurs. A database instance may run for long periods without generating log entries.

You can enable and configure the Aurora MySQL Slow Query and general logs to write log entries to a file or a database table by setting the corresponding parameters in the database parameter group. The following list identifies he parameters that control the log options:

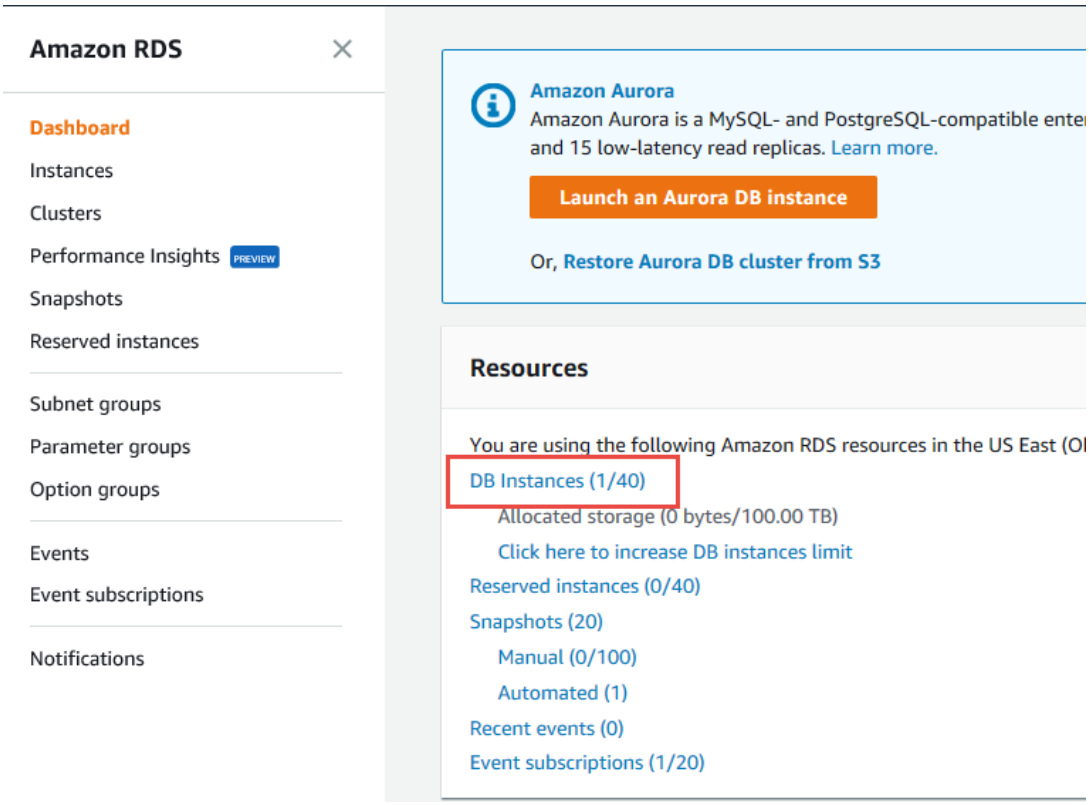
- **slow_query_log:** Set to 1 to create the Slow Query Log. The default is 0.
- **general_log:** Set to 1 to create the General Log. The default is 0.
- **long_query_time:** Specify a value in seconds for the shortest query execution time to be logged. The default is 10 seconds; the minimum is 0.
- **log_queries_not_using_indexes:** Set to 1 to log all queries not using indexes to the slow query log. The default is 0. Queries using indexes are logged even if their execution time is less than the value of the *long_query_time* parameter.
- **log_output option:** Specify one of the following options:
  - **TABLE** (default)**:**Write general queries to the mysql.general_log table and slow queries to the mysql.slow_log table

- **FILE:** Write both general and slow query logs to the file system. Log files are rotated hourly.
- **NONE:** Disable logging.

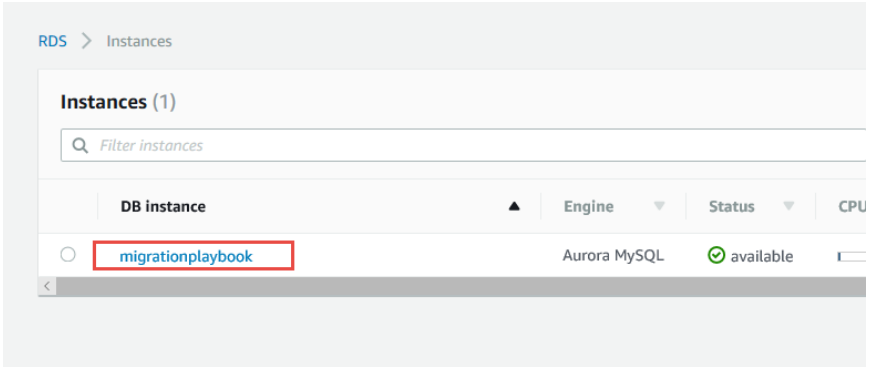
Examples

The following walkthrough demonstrates how to view the Aurora MySQL error logs in the RDS console.

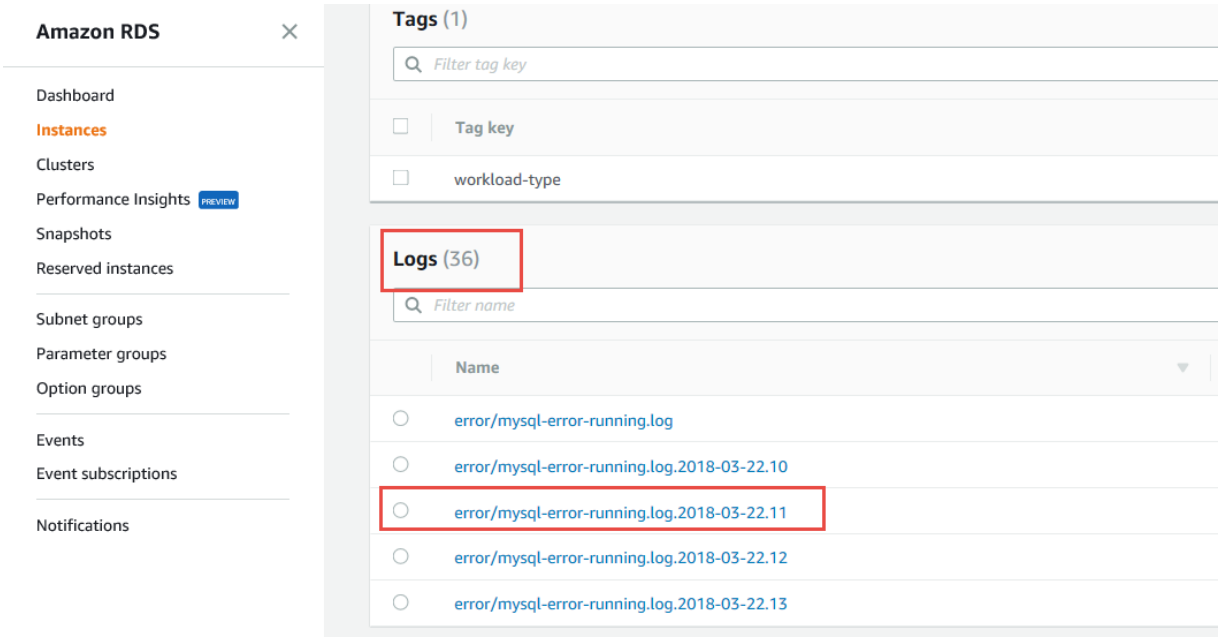
Using a web browser, navigate to https://console.aws.amazon.com/rds/home and click **Instances**.



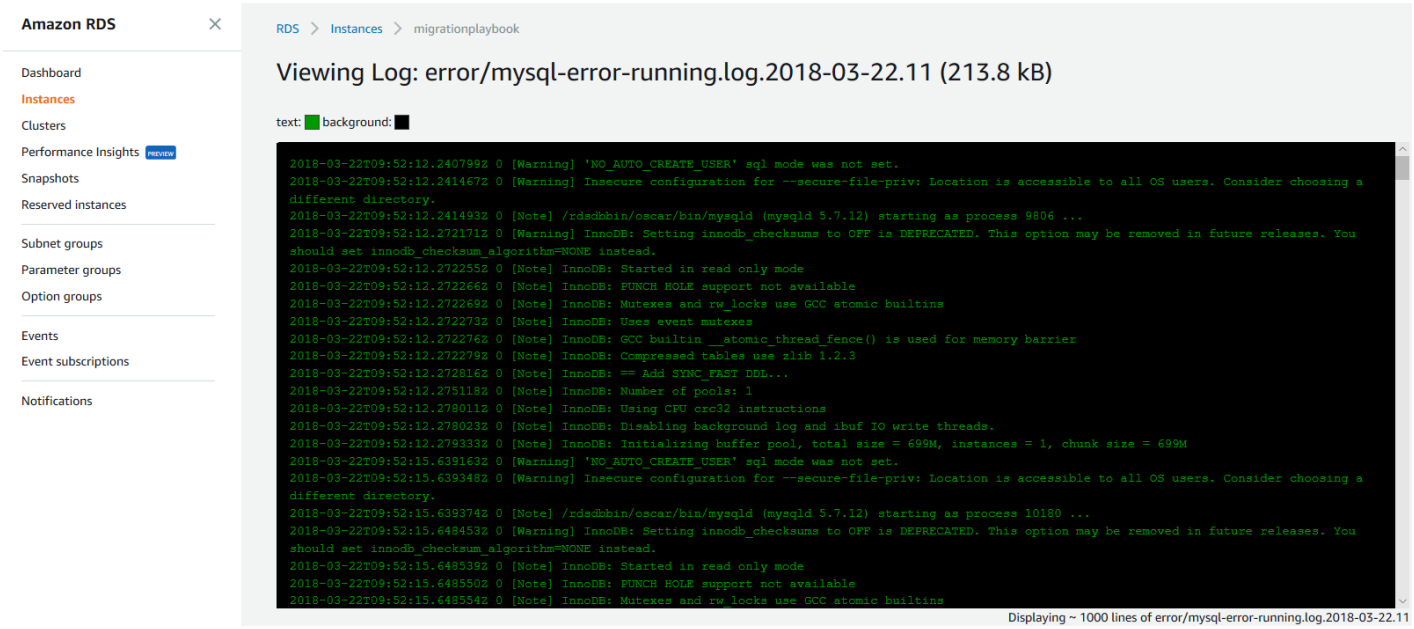
Click the instance for which you want to view the error log.



Scroll down to the logs section and click the log name.



The log viewer displays the log content.



For more information, see https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_LogAccess.Concepts.MySQL.html

Migrate from SQL Server Maintenance Plans

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Backups via the RDS services • Table maintenance via SQL

Overview

A Maintenance plan is a set of automated tasks used to optimize a database, performs regular backups, and ensure it is free of inconsistencies. Maintenance plans are implemented as SQL Server Integration Services (SSIS) packages and are executed by SQL Server Agent jobs. They can be run manually or automatically at scheduled time intervals.

SQL Server provides a variety of pre-configured maintenance tasks. You can create custom tasks using T-SQL scripts or operating system batch files.

Maintenance plans are typically used for the following tasks:

- Backing up database and transaction log files.
- Performing cleanup of database backup files in accordance with retention policies.
- Performing database consistency checks.
- Rebuilding or reorganizing indexes.
- Decreasing data file size by removing empty pages (shrink a database).
- Updating statistics to help the query optimizer obtain updated data distributions.
- Running SQL Server Agent jobs for custom actions.
- Executing a T-SQL task.

Maintenance plans can include tasks for operator notifications and history/maintenance cleanup. They can also generate reports and output the contents to a text file or the maintenance plan tables in msdb.

Maintenance plans can be created and managed using the maintenance plan wizard in SQL Server Management Studio, Maintenance Plan Design Surface (provides enhanced functionality over the wizard), Management Studio Object Explorer, and T-SQL system stored procedures.

For more information about SQL Server Agent migration, see SQL Server Agent.

Deprecated DBCC Index and Table Maintenance Commands

The DBCC DBREINDEX, INDEXDEFRAG, and SHOWCONTIG commands have been deprecated as of SQL Server 2008R2 in accordance with https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-

2008-r2/ms143729(v=sql.105).

In place of the deprecated DBCC, SQL Server provides newer syntax alternatives as detailed in the following table.

Deprecated DBCC Command	Use Instead
DBCC DBREINDEX	ALTER INDEX ... REBUILD
DBCC INDEXDEFRAG	ALTER INDEX ... REORGANIZE
DBCC SHOWCONTIG	sys.dm_db_index_physical_stats

For the Aurora MySQL alternatives to these maintenance commands, see Aurora MySQL Maintenance Plans.

Examples

Enable Agent XPs, which are disabled by default.

```
EXEC [sys].[sp_configure] @configname = 'show advanced options', @configvalue = 1
RECONFIGURE ;
```

```
EXEC [sys].[sp_configure] @configname = 'agent xps', @configvalue = 1
RECONFIGURE;
```

Create a T-SQL maintenance plan for a single index rebuild .

```
USE msdb;
```

Add the `Index Maintenance IDX1` job to SQL Server Agent.

```
EXEC dbo.sp_add_job @job_name = N'Index Maintenance IDX1', @enabled = 1, @description
= N'Optimize IDX1 for INSERT' ;
```

Add the T-SQL job step `Rebuild IDX1 to 50 percent fill`.

```
EXEC dbo.sp_add_jobstep @job_name = N'Index Maintenance IDX1', @step_name = N'Rebuild
IDX1 to 50 percent fill', @subsystem = N'TSQL',
@command = N'Use MyDatabase; ALTER INDEX IDX1 ON Shcema.Table REBUILD WITH ( FILL_
FACTOR = 50), @retry_attempts = 5, @retry_interval = 5;
```

Add a schedule to run every day at 01:00 AM.

```
EXEC dbo.sp_add_schedule @schedule_name = N'Daily0100', @freq_type = 4, @freq_interval
= 1, @active_start_time = 010000;
```

Associate the schedule `Daily0100` with the job Index Maintenance *IDX1*.

```
EXEC sp_attach_schedule @job_name = N'Index Maintenance IDX1' @schedule_name =
N'Daily0100' ;
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/maintenance-plans/-maintenance-plans

Migrate to Aurora MySQL Maintenance Plans

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Backups via the RDS services • Table maintenance via SQL

Overview

Amazon RDS performs automated database backups by creating storage volume snapshots that back up entire instances, not individual databases.

RDS creates snapshots during the backup window for individual database instances and retains snapshots in accordance with the backup retention period. You can use the snapshots to restore a database to any point in time within the backup retention period.

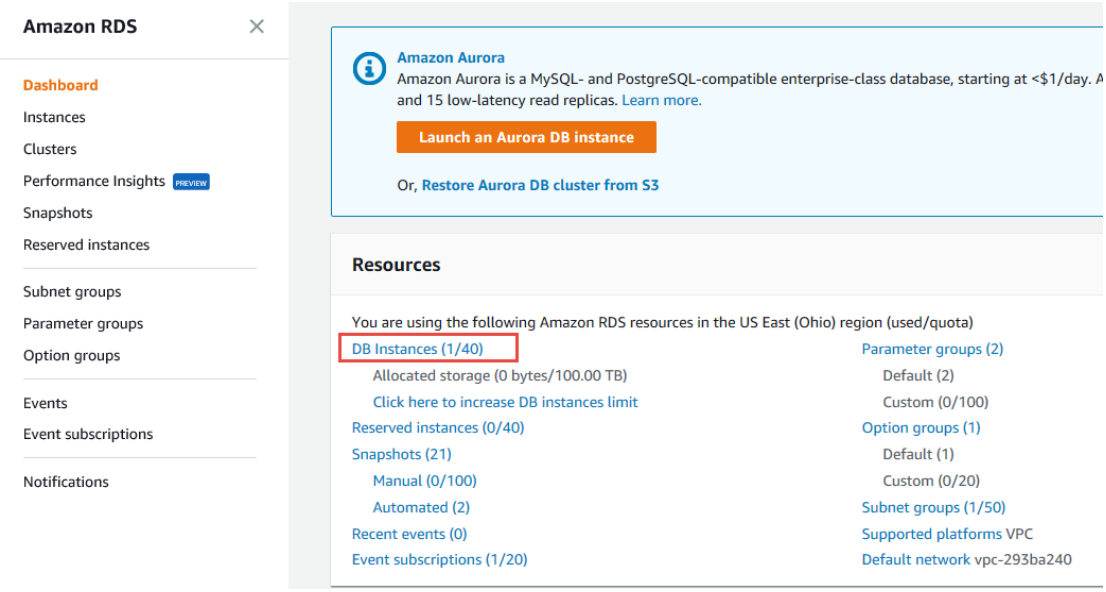
> **Note:** The state of a database instance must be ACTIVE for automated backups to occur.

You can backup database instances manually by creating an explicit database snapshot. Use the AWS console, the AWS CLI, or the AWS API to take manual snapshots.

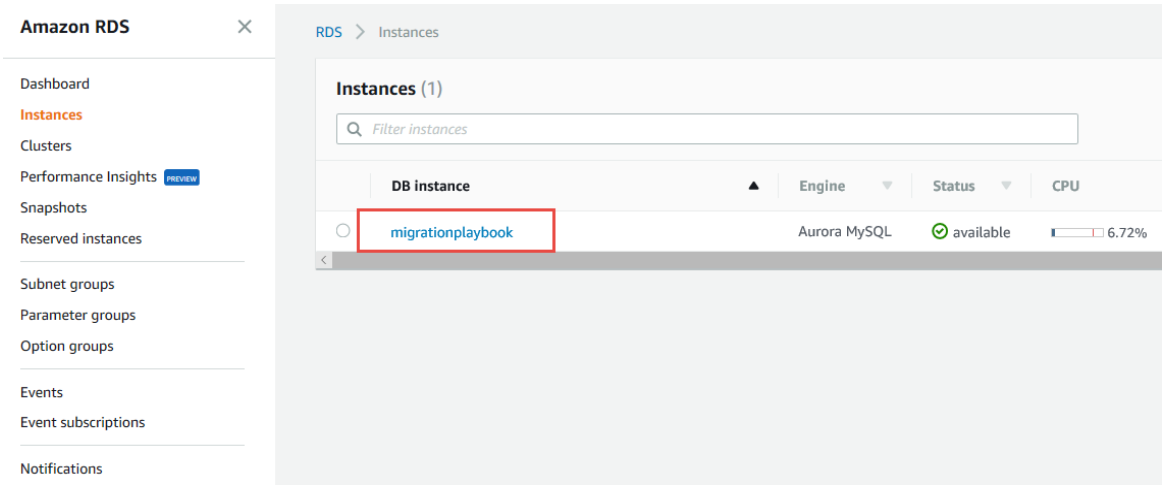
Examples

Create a Manual Database Snapshot Using the RDS Console

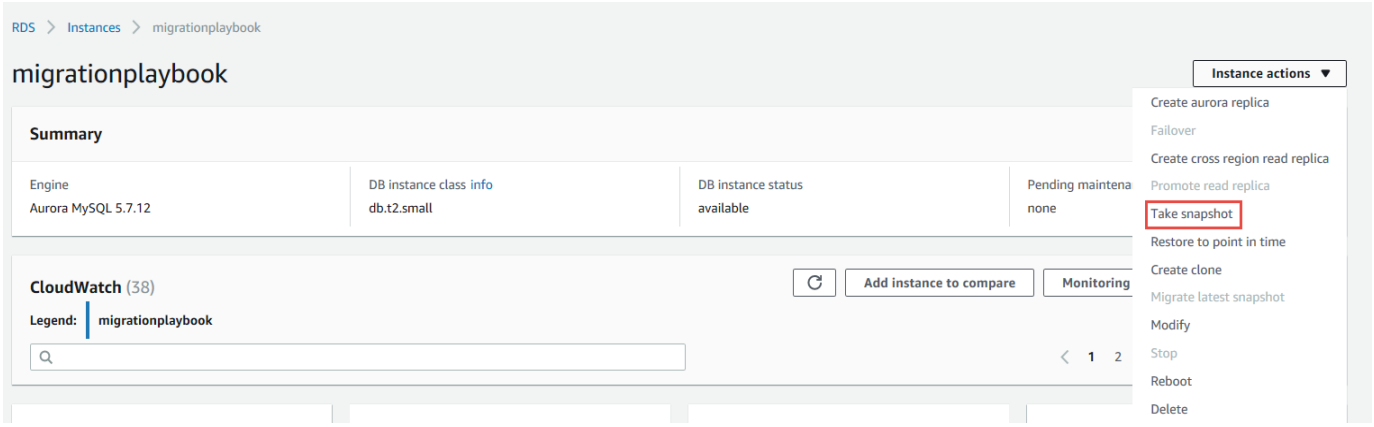
Login to the RDS console and click **DB Instances**.



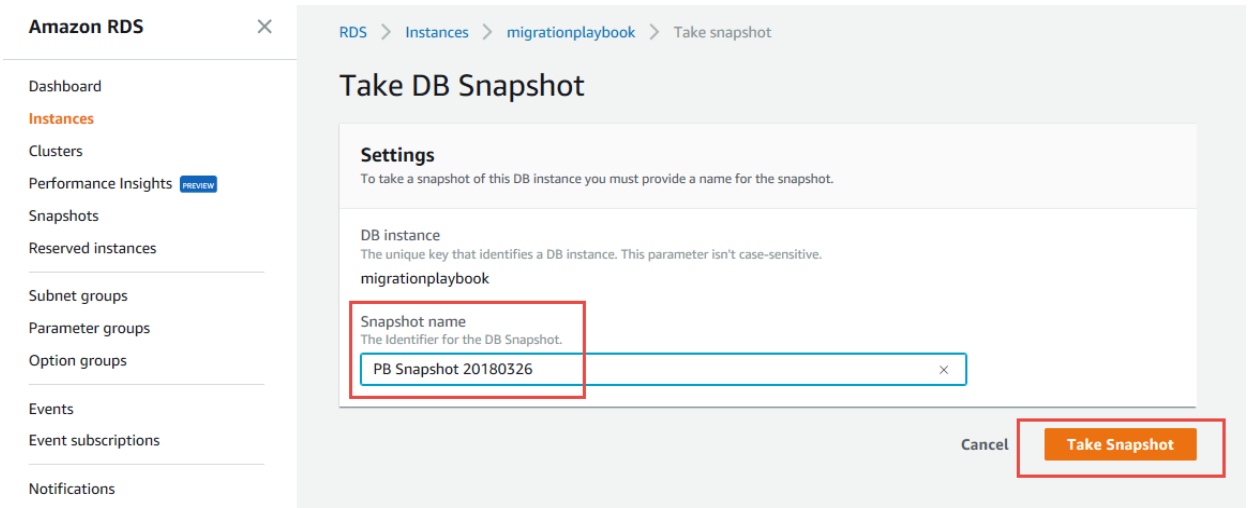
Click the Instance to create a snapshot of.



Click **Instance Actions** and select **Take Snapshot**.

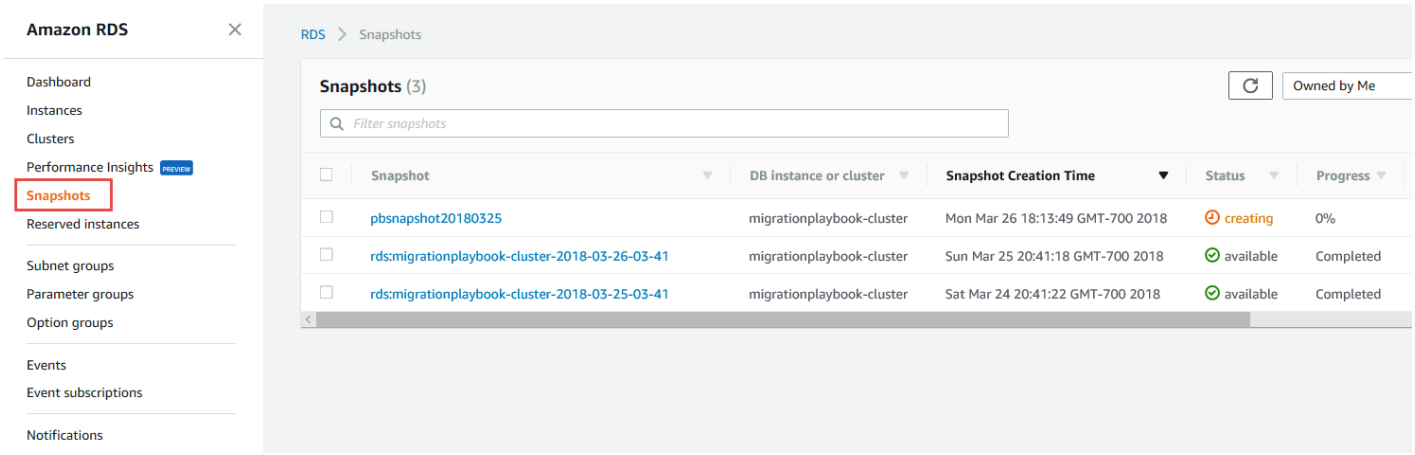


Enter **Snapshot name** and click the **Take Snapshot** button.

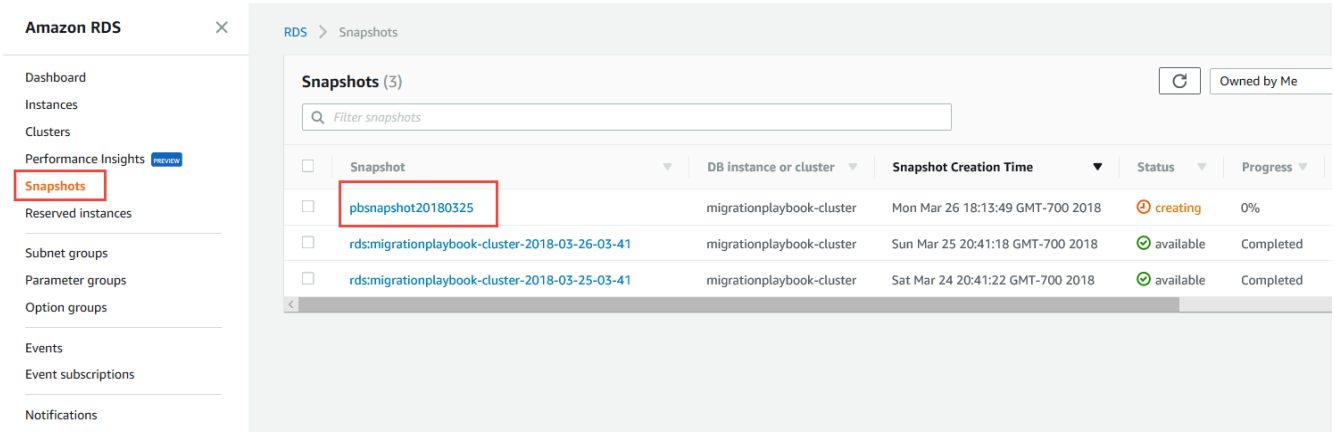


Viewing and Restoring Snapshots on RDS console

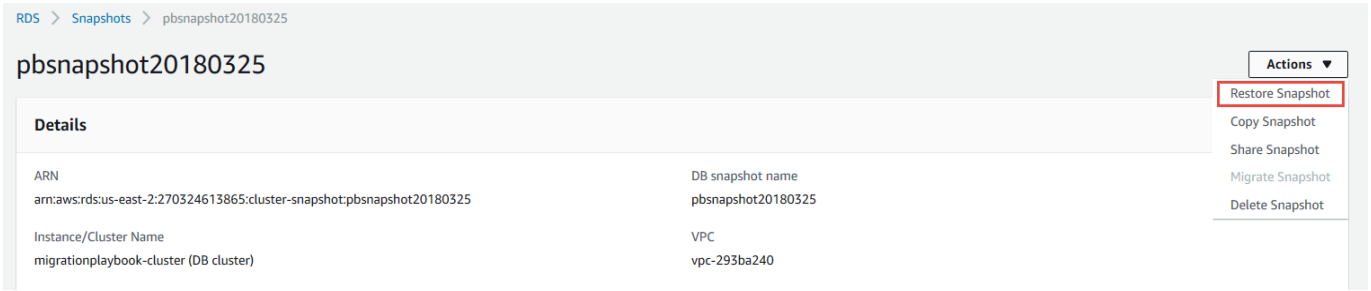
Login to the RDS console and click **Snapshots**.



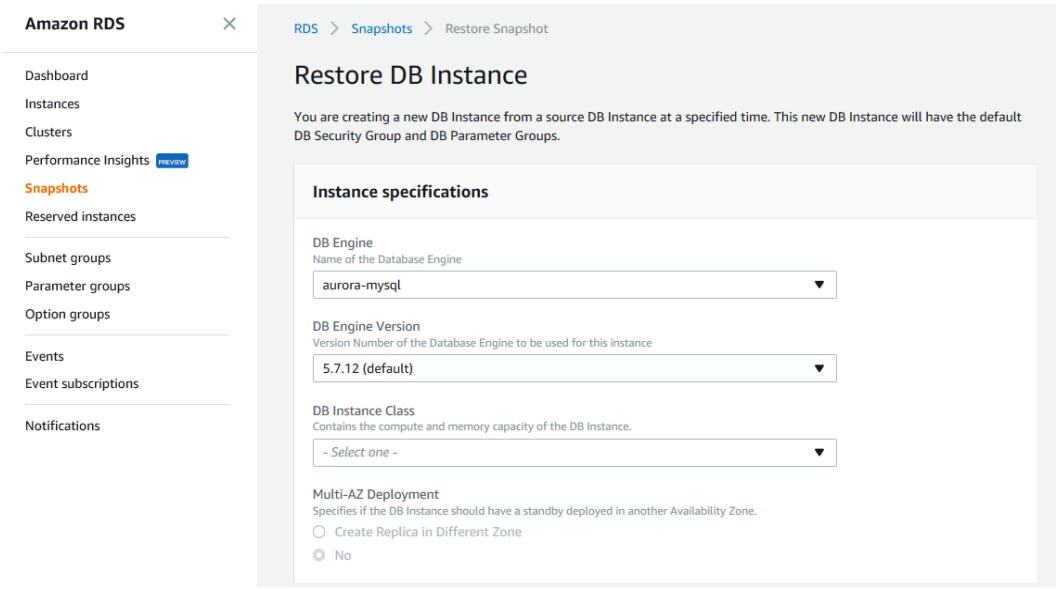
Click the snapshot to restore.



Click **Actions** and then click **Restore Snapshot**. A snapshot restore operation does not overwrite the database instance; it creates a new snapshot.



Enter the **Instance specifications** and click the **Restore DB instance** button at the bottom of the page.



You can also restore a database instance to a point-in-time. For more details see Backup and Restore.

For all other tasks, use a third-party or a custom application scheduler.

Rebuild and Reorganize an Index

Aurora MySQL supports the OPTIMIZE TABLE command, which is similar to the REORGANIZE option of SQL Server indexes.

```
OPTIMIZE TABLE MyTable;
```

To perform a full table rebuild with all secondary indexes, perform a null altering action using either ALTER TABLE <table> FORCE or ALTER TABLE <table> ENGINE = <current engine>.

```
ALTER TABLE MyTable FORCE;
```

```
ALTER TABLE MyTable ENGINE = InnoDB
```

Perform Database Consistency Checks

Use the CHECK TABLE command to perform a database consistency check.

```
CHECK TABLE <table name> [FOR UPGRADE | QUICK]
```

The FOR UPGRADE option checks if the table is compatible with the current version of MySQL fto determine whether there have been any incompatible changes in any of the table's data types or indexes since the table was created. The QUICK options does not scan the rows to check for incorrect links.

For routine checks of a table, use the QUICK option.

> **Note:** In most cases, Aurora MySQL will find all errors in the data file. When an error is found, the table is marked as “corrupted” and cannot be used until it is repaired.

Converting Deprecated DBCC Index and Table Maintenance Commands

Deprecated DBCC Command	Aurora MySQL Equivalent
DBCC DBREINDEX	ALTER TABLE ... FORCE
DBCC INDEXDEFRAG	OPTIMIZE TABLE
DBCC SHOWCONTIG	CHECK TABLE

Decrease Data File Size by Removing Empty Pages (shrink database)

Unlike SQL Server that uses a single set of files for an entire database, Aurora MySQL uses one file for each database table. Therefore you do not need to shrink an entire database.

Update Statistics to Help the Query Optimizer Get Updated Data Distribution

Aurora MySQL uses both persistent and non-persistent table statistics. Non-persistent statistics are deleted on server restart and after some operations. The statistics are then recomputed on the next table access. Therefore, different estimates could be produced when recomputing statistics leading to different choices in execution plans and variations in query performance.

Persistent optimizer statistics survive server restarts and provide better plan stability resulting in more consistent query performance. Persistent optimizer statistics provide the following control and flexibility options:

- Set the innodb_stats_auto_recalc configuration option to control whether statistics are updated automatically when changes to a table cross a threshold.
- Set the STATS_PERSISTENT, STATS_AUTO_RECALC, and STATS_SAMPLE_PAGES clauses with CREATE TABLE and ALTER TABLE statements to configure custom statistics settings for individual tables.
- View optimizer statistics in the mysql.innodb_table_stats and mysql.innodb_index_stats tables.
- View the last_update column of the mysql.innodb_table_stats and mysql.innodb_index_stats tables to see when statistics were last updated.
- Modify the mysql.innodb_table_stats and mysql.innodb_index_stats tables to force a specific query optimization plan or to test alternative plans without modifying the database.

For more information, see Managing Statistics.

Summary

The following table summarizes the key tasks that use SQL Server maintenance Plans and a comparable Aurora MySQL solutions.

Task	SQL Server	Aurora MySQL	Comments
Rebuild or reorganize indexes	ALTER INDEX / ALTER TABLE	OPTIMIZE TABLE / ALTER TABLE	

Task	SQL Server	Aurora MySQL	Comments
Decrease data file size by removing empty pages	DBCC SHRINKDATABASE / DBCC SHRINKFILE	Files are per table; not per database. Rebuilding a table optimizes file size.	Not needed
Update statistics to help the query optimizer get updated data distribution	UPDATE STATISTICS / sp_updatestats	Set innodb_stats_auto_recalc to ON in the instance global parameter group.	
Perform database consistency checks	DBCC CHECKDB / DBCC CHECKTABLE	CHECK TABLE	
Back up the database and transaction log files	BACKUP DATABASE / BACKUP LOG	Automated backups + snapshots	See Backup and Restore
Run SQL Server Agent jobs for custom actions	sp_start_job, scheduled	Not supported	

For more information, see:

- *https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithAutomatedBackups.html*
- *https://dev.mysql.com/doc/refman/5.6/en/check-table.html*

Migrate from SQL Server Monitoring

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Amazon Cloud Watch service

Overview

Monitoring server performance and behavior is a critical aspect of maintaining service quality and includes ad hoc and ongoing collection of data; root cause analysis; and preventative or reactive actions. SQL Server provides an array of interfaces to monitor and collect server data.

Windows Operating System Level Tools

The Windows Scheduler can be used to trigger execution of script files to collect, store, and process performance data.

System Monitor is a graphical tool for measuring and recording performance of SQL Server and other windows related metrics using the Windows Management Interface (WMI) performance objects.

> **Note:** Performance objects can also be accessed directly from T-SQL using the system function sys.dm_os_performance_counters.

Performance counters exist for both real time measurements such as CPU Utilization and for aggregated history such as average active transactions.
For a full list of the object hierarchy, see https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/use-sql-server-objects

SQL Server Extended Events

SQL Server's latest tracing framework provides very lightweight and robust event collection and storage. SQL Server management Studio features the New Session Wizard and New Session graphic user interfaces for managing and analyzing captured data. SQL Server Extended Events consists of the following items:

- **SQL Server Extended Events Package** is a logical container for Extended Events objects.
- **SQL Server Extended Events Targets** are consumers of events. Targets include Event File, which writes data to the file Ring Buffer for retention in memory, or for processing aggregates such as Event Counters and Histograms.
- **SQL Server Extended Events Engine** is a collection of services and tools that comprise the framework.
- **SQL Server Extended Events Sessions** are logical containers mapped many-to-many with packages events, and filters.

The following example creates a session that logs lock escalations and lock timeouts to a file:

```
CREATE EVENT SESSION Locking_Demo
ON SERVER
    ADD EVENT sqlserver.lock_escalation,
    ADD EVENT sqlserver.lock_timeout,
    ADD TARGET package0.etw_classic_sync_target
        (SET default_etw_session_logfile_path = N'C:\ExtendedEvents\Locking\Demo_
20180502.etl')
    WITH (MAX_MEMORY=8MB, MAX_EVENT_SIZE=2MB);
GO
```

SQL Server Tracing Framework and the SQL Server Profiler Tool

The SQL Server trace framework is the predecessor to the Extended Events framework and remains popular among database administrators. The lighter and more flexible Extended Events Framework is recommended for development of new monitoring functionality.

SQL Server Management Studio

SQL Server management studio provides several monitoring extensions:

- **SQL Server Activity Monitor** is an in-process, real-time, basic high level information graphical tool.
- **Query Graphical Show Plan** provides easy exploration of estimated and actual query execution plans.
- **Query Live Statistics** displays query execution progress in real time.
- **Replication Monitor** and **Log Shipping Monitor**
- **Standard Performance Reports**

T-SQL

From the T-SQL interface, SQL Server provides many system stored procedures, system views, and functions for monitoring data.

System stored procedures such as sp_who and sp_lock provide real-time information. sp_monitor provides aggregated data.

Built in functions such as @@CONNECTIONS, @@IO_BUSY, @@TOTAL_ERRORS, and others provide high level server information.

A rich set of System Dynamic Management functions and views are provided for monitoring almost every aspect of the server. These functions reside in the sys schema and are prefixed with dm_string. For more information about Dynamic Management Views, see https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/system-dynamic-management-views

Trace Flags

Trace flags can be set to log events. For example, set trace flag 1204 to log deadlock information.

SQL Server Query Store

Query Store is a database level framework supporting automatic collection of queries, execution plans, and run time statistics. This data is stored in system tables and can be used to diagnose performance issues, understand patterns, and understand trends. It can also be set to automatically revert plans when a performance regression is detected.

In addition, it is common for SQL Server administrators to use third-party tools — most of which build on the existing native monitoring framework — and add historical, analytical, exploratory features, and automatic advisers.

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitor-and-tune-for-performance

Migrate to Aurora MySQL Monitoring

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Amazon Cloud Watch service

Overview

The native features for monitoring MySQL databases such as innodb logging and the performance schema are disabled for Aurora MySQL. Most third-party tools that rely on these features cannot be used. Some vendors provide monitoring services specifically for Aurora MySQL.

However, Amazon RDS provide a very rich monitoring infrastructure for Aurora MySQL clusters and instances with the native Cloud Watch service.

These services are improved frequently.

See the following up-to-date articles, which include examples and walkthroughs for monitoring Aurora MySQL clusters and instances:

- https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Monitoring.html
- https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_Monitoring.OS.html

Migrate from SQL Server Resource Governor

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Per User Resource limit

Overview

SQL Server Resource Governor provides the capability to control and manage resource consumption. Administrators can specify and enforce workload limits on CPU, physical I/O, and Memory. Resource configurations are dynamic and can be changed in real time.

Use Cases

The following list identifies typical Resource Governor use cases:

- **Minimize performance bottlenecks and inconsistencies** to better support Service Level Agreements (SLA) for multiple workloads and users.
- **Protect against runaway queries** that consume a large amount of resources or explicitly throttle I/O intensive operations. For example, consistency checks with DBCC that may bottleneck the I/O subsystem and negatively impact concurrent workloads.
- **Allow tracking and control for resource-based pricing** scenarios to improve predictability of user charges.

Concepts

The three basic concepts in Resource Governor are *Resource Pools*, *Workload Groups*, and *Classification*.

- **Resource Pools** represent physical resources. Two built-in resource pools, internal and default, are created when SQL Server is installed. You can create custom user-defined resource pools for specific workload types.
- **Workload Groups** are logical containers for session requests with similar characteristics. Workload Groups allow aggregate resource monitoring of multiple sessions. Resource limit policies are defined for a Workload Group. Each Workload Group belongs to a Resource Pool.
- **Classification** is a process that inspects incoming connections and assigns them to a specific Workload Group based on the common attributes. User-defined functions are used to implement Classification. For more information, see User Defined Functions.

Examples

Enable the Resource Governor.

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a Resource Pool.

```
CREATE RESOURCE POOL ReportingWorkloadPool
WITH (MAX_CPU_PERCENT = 20);
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a Workload Group.

```
CREATE WORKLOAD GROUP ReportingWorkloadGroup USING poolAdhoc;
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Create a classifier function.

```
CREATE FUNCTION dbo.WorkloadClassifier()
RETURNS sysname WITH SCHEMABINDING
AS
BEGIN
 RETURN (CASE
          WHEN HOST_NAME()= 'ReportServer'
           THEN 'ReportingWorkloadGroup'
          ELSE 'Default'
         END)
END;
```

Register the classifier function.

```
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION = dbo.WorkloadClassifier);
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor

Migrate to Aurora MySQL Resource Governor

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Use Per User Resource limit

Overview

Aurora MySQL does not support a server-wide, granular, resource-based, workload resource isolation and management capability similar to SQL Server's Resource Governor. However, Aurora MySQL does support the feature *User Resource Limit Options* that you can use to achieve similar high-level functionality for limiting resource consumption of user connections.

You can specify User Resource Limit Options as part of the CREATE USER statement to place the following limits on users:

- The number of total queries/hour an account is allowed to issue.
- The number of updates/hour an account is allowed to issue .
- The number of times/hour an account can establish a server connection.
- The total number of concurrent server connections allowed for the account.

For more information about Aurora MySQL users and roles, see Users and Roles.

Syntax

```
CREATE USER <User Name> ...
WITH
MAX_QUERIES_PER_HOUR count |
MAX_UPDATES_PER_HOUR count |
MAX_CONNECTIONS_PER_HOUR count |
MAX_USER_CONNECTIONS count
```

Migration Considerations

Although both SQL Server Resource Manager and Aurora MySQL User Resource Limit Options provide the same basic function — limiting the amount of resources for distinct types of workloads — they differ significantly in scope and flexibility.

SQL Server's Resource Manager is a dynamically configured independent framework based on actual run-time resource consumption. User Resource Limit Options are defined as part of the security objects and requires application connection changes to map to limited users. To modify these limits, you must alter the user object.

User Resource Limit Options do not allow limiting workload activity based on actual resource consumption, but rather provides a quantitative limit for the number of queries or number of connections. A runaway query that consumes a large amount of resources may slow down the server.

Another important difference is how exceeded resource limits are handled. SQL Server Resource Governor throttles execution; Aurora MySQL raises errors.

Examples

Create a resource-limited user.

```
CREATE USER 'ReportUsers'@'localhost'
IDENTIFIED BY 'ReportPassword'
WITH
MAX_QUERIES_PER_HOUR 60
MAX_UPDATES_PER_HOUR 0
MAX_CONNECTIONS_PER_HOUR 5
MAX_USER_CONNECTIONS 2;
```

Summary

Feature	SQL Server Resource Governor	Aurora MySQL User Resource Limit Options	Comments
Scope	Dynamic workload pools and workload groups, mapped to aclassifier function	Per user	Application connection strings need to use specific limited users.
Limited resources	IO, CPU, and Memory	Number of queries, number of connections	
Modifying limits	ALTER RESOURCE POOL	ALTER USER	Application may use a dynamic connection string.
When resource threshold limit is reached	Throttles and queues execution	Raises an error	Application retry logic may need to be added.

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/create-user.html#create-user-resource-limits*
- *https://dev.mysql.com/doc/refman/5.7/en/user-resources.html*

Migrate from SQL Server Linked Servers

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Linked Servers	• Data transfer across schemas only, use a custom application solution to access remote instances

Overview

Linked servers in SQL Server are used to enable the database engine to connect to external Object Linking and Embedding for Data Bases (OLE-DB) sources.These are typically used to execute T-SQL commands and include tables in other instances of SQL Server, or other RDBMS engines such as Oracle. SQL Server supports multiple types of OLE-DB sources as linked servers, including Microsoft Access, Microsoft Excel, text files and others.

The main benefits of using linked servers are

- Read external data for import or processing
- Execute distributed queries, data modifications, and transactions for enterprise-wide data sources
- Heterogeneous data source querying using the same familiar T-SQL API

Linked servers can be configured using either SQL Server Management Studio, or the system stored procedure sp_addlinkedserver.
The available functionality and the specific requirements vary significantly between the various OLE-DB sources. Some sources may allow read only access, others may require specific security context settings, etc.

The Linked Server Definition contains the linked server alias, the OLE DB provider, and all the parameters needed to connect to a specific OLE-DB data source.

The OLE-DB provider is a .Net Dynamic Link Library (DLL) that handles the interaction of SQL Server with all data sources of its type. For example, OLE-DB Provider for Oracle.
The OLE-DB data source is the specific data source to be accessed, using the specified OLE-DB provider.

> **Note:** SQL Server distributed queries can be used with any custom OLE DB provider, as long as the required interfaces are implemented correctly.

SQL Server parses the T-SQL commands that access the linked server, and sends the appropriate requests to the OLE-DB provider.
There are several access methods for remote data, including opening the base table for read, or issuing SQL queries against the remote data source.

Managing linked servers can be done using SQL Server Management Studio graphical user interface, or using T-SQL system stored procedres.

- `EXECUTE sp_addlinkedserver` to add new server definitions
- `EXECUTE sp_addlinkedserverlogin` to define security context
- `EXECUTE sp_linkedservers or SELECT * FROM sys.servers` system catalog view to retrieve meta data
- `EXECUTE sp_dropserver` to delete a linked server

Linked server data sources are accessed from T-SQL using a fully qualified, four part naming scheme.

<Server Name>.<Database Name>.<Schema Name>.<Object Name>

Additionally, the OPENQUERY row set function can be used to explicitly invoke pass-through queries on the remote linked server, and the OPENROWSET and OPENDATASOURCE row set functions can be used for ad-hoc remote data access without defining the linked server in advance.

Syntax

```
EXECUTE sp_addlinkedserver
    [ @server= ] <Linked Server Name>
[ , [ @srvproduct= ] <Product Name>]
[ , [ @provider= ] <OLE DB Provider>]
[ , [ @datasrc= ] <Data Source>]
[ , [ @location= ] <Data Source Address>]
[ , [ @provstr= ] <Provider Connection String>]
[ , [ @catalog= ] <Database>];
```

Examples

Create a linked server to a local text file

```
EXECUTE sp_addlinkedserver MyTextLinkedServer, N'Jet 4.0',
   N'Microsoft.Jet.OLEDB.4.0',
   N'D:\TextFiles\MyFolder',
   NULL,
   N'Text';
```

Define security context

```
EXECUTE sp_addlinkedsrvlogin MyTextLinkedServer, FALSE, Admin, NULL;
```

Use sp_tables_ex to list tables in folder

```
EXEC sp_tables_ex MyTextLinkedServer;
```

Issue a SELECT query using 4 part name

```
SELECT *
FROM MyTextLinkedServer...[FileName#text];
```

For more information, see

- https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-addlinkedserver-transact-sql
- https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/distributed-queries-stored-procedures-transact-sql

Migrate to Aurora MySQL Linked Servers

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Linked Servers	• Data transfer across schemas only, use a custom application solution to access remote instances

Overview

Aurora MySQL does not support remote data access.

Connectivity between schemas is trivial, connectivity to other instances will require an application custom solution.

Migrate from SQL Server Scripting

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Non compatible tool sets and scripting languages • Use MySQL Workbench, Amazon RDS API, AWS Management Console, and Amazon CLI

Overview

SQL Server supports T-SQL and XQuery scripting within multiple execution frameworks such as SQL Server Agent, and stored procedures.

The SQLCMD command line utility can also be used to execute T-SQL scripts. However, the most extensive and feature-rich scripting environment is PowerShell.

SQL Server provides two PowerShell snap-ins that implement a provider that exposes the entire SQL Server Management Object Model (SMO) as PowerShell paths. Additionally, a set of SQL Server cmdlets can be used to execute specific SQL Server commands.

> **Note:** Invoke-Sqlcmd can be used to execute scripts using the SQLCMD utility.

The sqlps utility launches the PowerShell scripting environment and automatically loads the SQL Server modules. sqlps can be launched from a command prompt or from the Object Explorer pane of SQL Server Management Studio. You can execute ad-hoc PowerShell commands and script files (for example, .\SomeFolder\SomeScript.ps1).

> **Note:** SQL Server Agent supports executing PowerShell scripts in job steps.
> For more information, see SQL Server Agent.

SQL Server also supports three types of direct database engine queries: T-SQL, XQuery, and the SQLCMD utility.T-SQL and XQuery can be called from stored procedures, SQL Server Management Studio (or other IDE), and SQL Server agent Jobs. The SQLCMD utility also supports commands and variables.

Examples

Backup a database with PowerShell using the default backup options.

```
PS C:\> Backup-SqlDatabase -ServerInstance "MyServer\MySQLServerInstance" -Database
"MyDB"
```

Get all rows from the MyTable table in they MyDB database

```
PS C:\> Read-SqlTableData -ServerInstance MyServer\MySQLServerInstance" -DatabaseName
"MyDB"  -TableName "MyTable"
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/powershell/sql-server-powershell*
- *https://docs.microsoft.com/en-us/sql/relational-databases/scripting/database-engine-scripting*
- *https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility*

Migrate to Aurora MySQL Scripting

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Non compatible tool sets and scripting languages • Use MySQL Workbench, Amazon RDS API, AWS Management Console, and Amazon CLI

Overview

As a Platform as a Service (PaaS), Aurora MySQL accepts connections from any compatible client, but you cannot access the MySQL command line utility typically used for database administration. However, you can use MySQL tools installed on a network host and the Amazon RDS API. The most common tools for Aurora MySQL scripting and automation include MySQL Workbench, MySQL Utilities, and the Amazon RDS API. The following sections describe each tool.

MySQL Workbench

MySQL Workbench is the most commonly used tool for development and administration of MySQL servers. It is available as a free Community Edition and a paid Commercial Edition that adds enterprise features such as database documentation features. MySQL Workbench is an integrated IDE with the following features:

- **SQL Development:** Manage and configure connections to aurora MySQL clusters and execute SQL queries using the SQL editor.
- **Data Modeling:** Reverse and forward engineer graphical database schema models and manage schemas with the Table Editor.
- **Server Administration:** Not applicable to Aurora MySQL. Use the Amazon RDS console to administer servers.

The MySQL Workbench also supports a Python scripting shell that you can use interactively and programmatically.

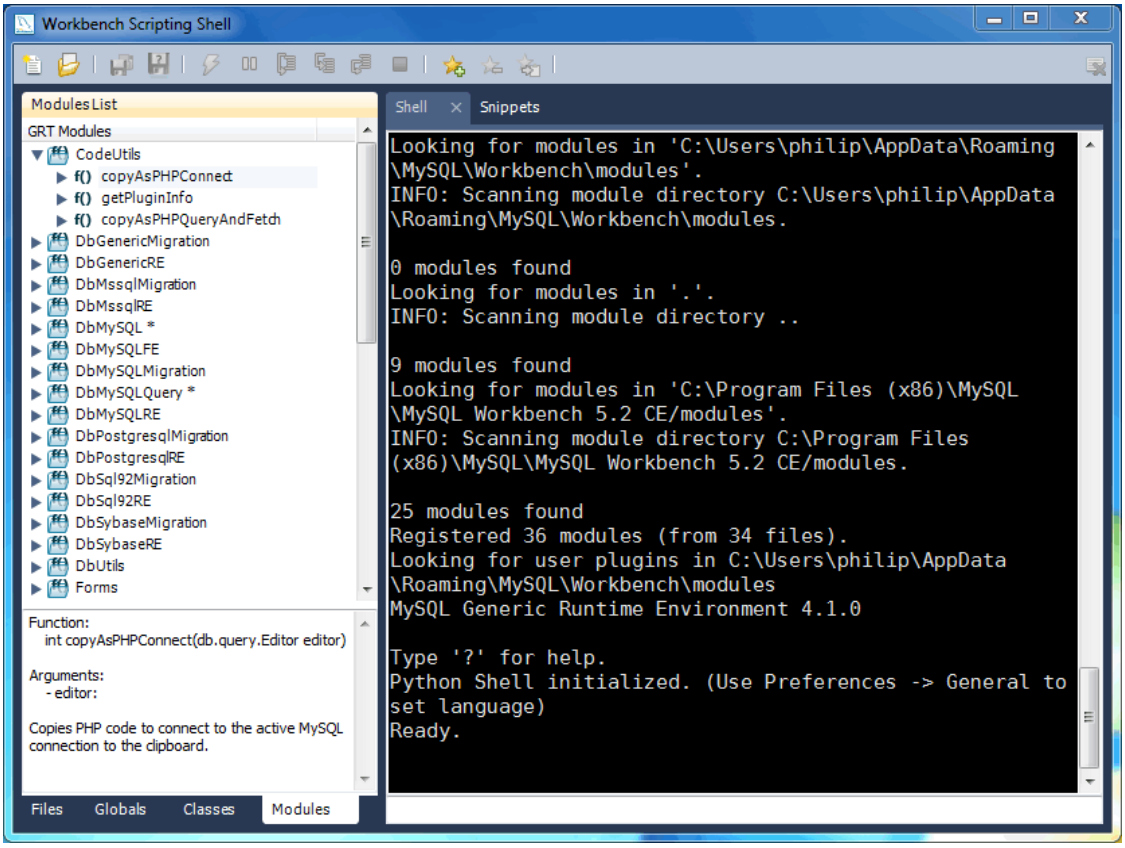


Image from the MySQL documentation at https://dev.mysql.com/doc/workbench/en/wb-exploring-scripting-shell.html

MySQL Utilities

MySQL Utilities are a set of Python command line tools used for common maintenance and administration of MySQL servers tasks. They can reduce the need to write custom code for common tasks and can be easily customized. The following tools are included in the MySQL Utilities set. Note that some tools will not work with Aurora MySQL because you don't have root access to the underlying server.

- **Admin Utilities:**Clone, Copy, Compare, Diff, Export, Import, and User Management
- **Replication Utilities:** Setup, Configuration, and Verification
- **General Utilities:** Disk Usage, Redundant Indexes, Manage Metadata, and Manage Audit Data

Amazon RDS API

The Amazon RDS API is a web service for managing and maintaining Aurora MySQL (and other) relational databases. It can be used to setup, operate, scale, backup, and perform many common administration tasks. The RDS API supports multiple database platforms and can integrate administration seamlessly for heterogeneous environments.

> **Note:** The Amazon RDS API is asynchronous. Some interfaces may require polling or callback functions to receive command status and results.

You can access Amazon RDS using the AWS Management Console, the AWS Command Line Interface (CLI), and the Amazon RDS Progammatic API as described in the following sections.

AWS Management Console

The AWS Management Console is a simple web-based set of tools for interactive management of Aurora MySQL and other RDS services. It can be accessed at https://console.aws.amazon.com/rds/

AWS Command Line Interface (CLI)

The Amazon AWS Command Line Interface is an open source tool that runs on Linux, Windows, or MacOS having Python 2 version 2.6.5 and higher or Python 3 version 3.3 and higher.

The AWS CLI is built on top of the AWS SDK for Python (Boto), which provides commands for interacting with AWS services. With minimal configuration, you can start using all AWS Management Console functionality from your favorite terminal application.

- **Linux shells:** Use common shell programs such as Bash, Zsh, or tsch.
- **Windows command line:** Run commands in PowerShell or the Windows Command Processor
- **Remotely:** Run commands on Amazon EC2 instances through a remote terminal such as PuTTY or SSH.

The AWS Tools for Windows PowerShell and AWS Tools for PowerShell Core are PowerShell modules built on the functionality exposed by the AWS SDK for .NET. These Tools enable scripting operations for AWS resources using the PowerShell command line.

> **Note:** You cannot use SQL Server cmdlets in power shell.

Amazon RDS Programmatic API

The Amazon RDS API can be used to automate management of DB instances and other Amazon RDS objects.

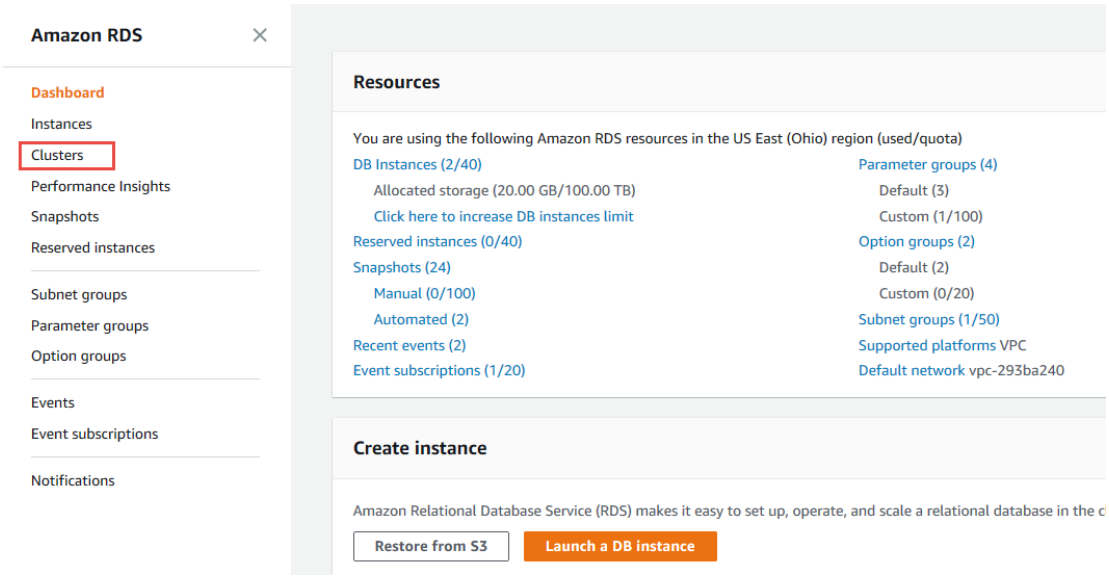
For more information on using Amazon RDS API, see:

- ***API actions:*** *http://docs.aws.amazon.com/AmazonRDS/latest/APIReference/API_Operations.html*
- ***Data Types:*** *http://docs.aws.amazon.com/AmazonRDS/latest/APIReference/API_Types.html*
- ***Common query parameters:*** *http://-docs.aws.amazon.com/AmazonRDS/latest/APIReference/CommonParameters.html*
- ***Error codes:*** *http://docs.aws.amazon.com/AmazonRDS/latest/APIReference/CommonErrors.html*

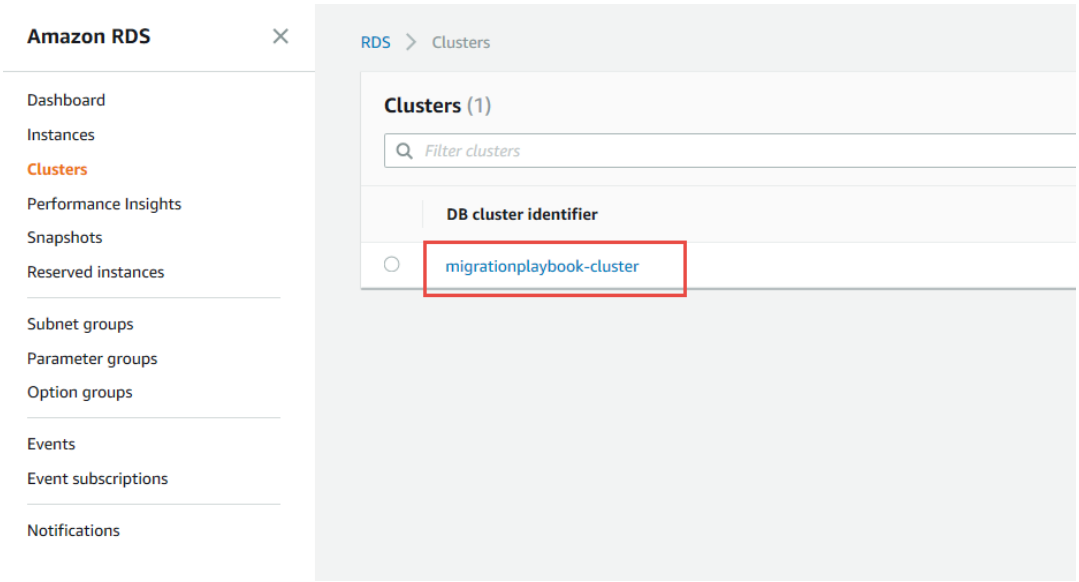
Examples

The following walkthrough describes how to connect to an Aurora MySQL DB instance using the MySQL Utility:

Log on to the Amazon RDS Console and click **Clusters**.

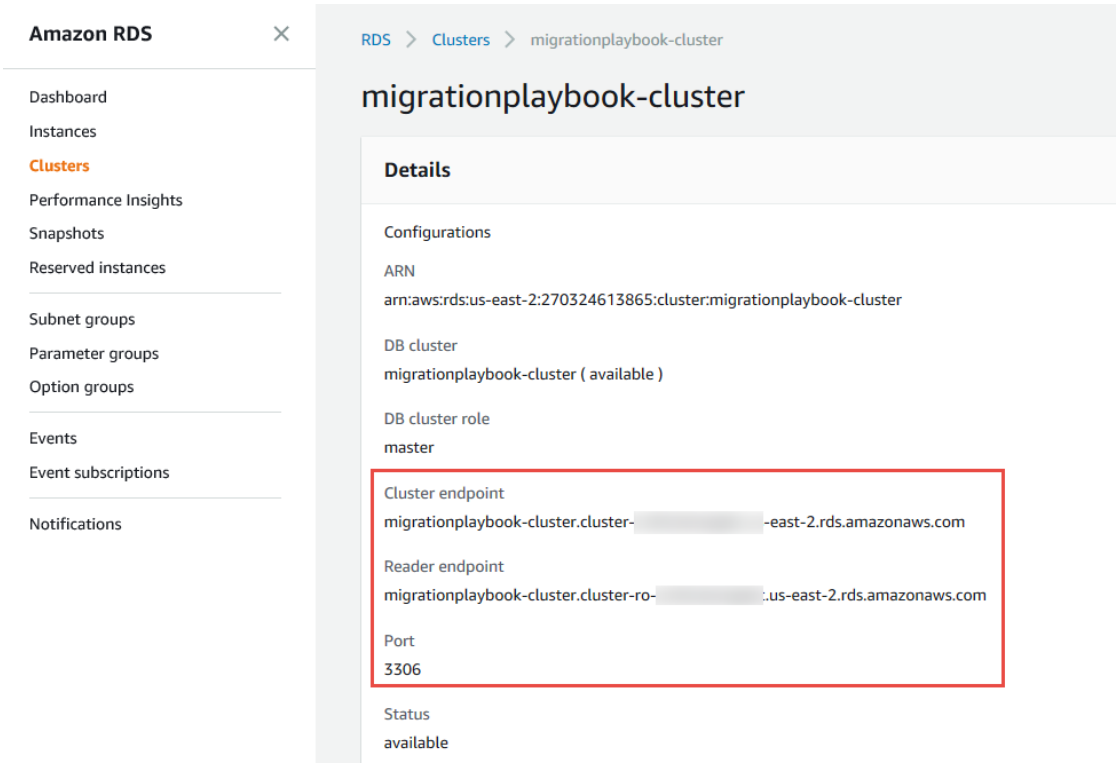


Click the name of the cluster to which you wish to connect.



Copy the cluster endpoint address.

> **Note:** You can also connect to individual DB instances. For more information, see [High Availability Essentials].]



From a command shell, type the following:

```
mysql -h mysql-instance1.123456789012.us-east-2.rds.amazonaws.com -P 3306 -u Master-
User
```

- The -h parameter is the endpoint DNS name of the Aurora MySQL DB cluster.
- The -P parameter is the port number .

Provide the password when prompted. The system displays the following (or similar) message.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 350
Server version: 5.6.27-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

For more information, see

- *https://dev.mysql.com/downloads/utilities/*
- *https://dev.mysql.com/downloads/workbench/*
- *https://docs.aws.amazon.com/cli/latest/reference/*
- *https://docs.aws.amazon.com/AmazonRDS/latest/APIReference/Welcome.html*

Performance Tuning

Migrate from SQL Server Execution Plans

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Syntax differences • Completely different optimizer with different operators and rules

Overview

Execution plans provide users detailed information about the data access and processing methods chosen by the SQL Server Query Optimizer. They also provide estimated or actual costs of each operator and sub tree. Execution plans provide critical data for troubleshooting query performance challenges.

SQL Server creates execution plans for most queries and returns them to client applications as plain text or XML documents. SQL Server produces an execution plan when a query executes, but it can also generate estimated plans without executing a query.

SQL Server Management Studio provides a graphical view of the underlying XML plan document using icons and arrows instead of textual information. This graphical view is extremely helpful when investigating the performance aspects of a query.

To request an estimated execution plan, use the SET SHOWPLAN_XML, SHOWPLAN_ALL, or SHOWPLAN_TEXT statements.

Examples

Show the estimated execution plan for a query.

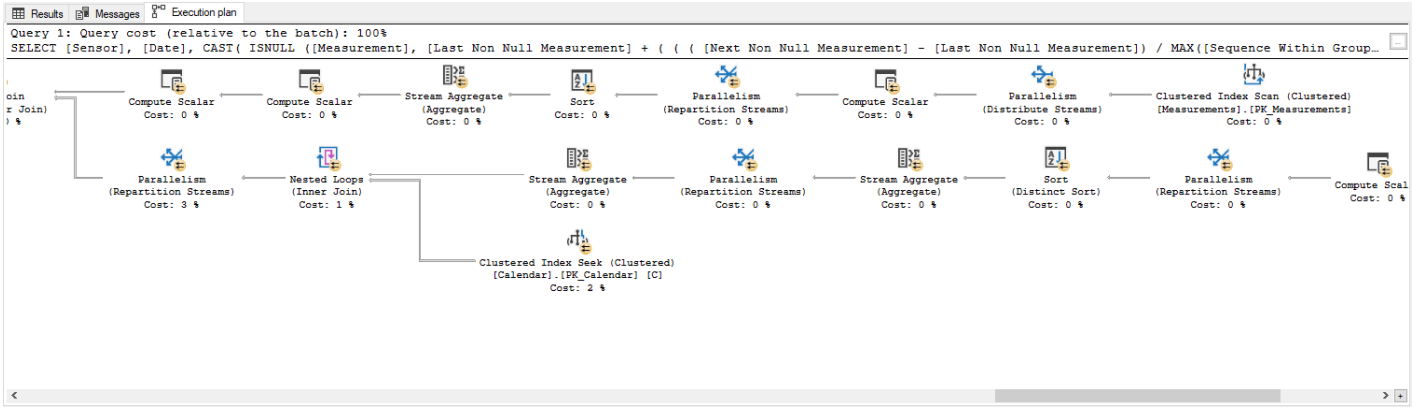
```
SET SHOWPLAN_XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET SHOWPLAN_XML OFF;
```

Actual execution plans return after execution of the query, or batch of queries, completes and include run-time statistics about resource usage and warnings. To request the actual execution plan, use the SET STATISTICS XML statement to return the XML document object. Alternatively, use the STATISTICS PROFILE statement, which returns an additional result set containing the query execution plan.

Show the actual execution plan for a query.

```
SET STATISTICS XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET STATISTICS XML OFF;
```

An example of a (partial) graphical execution plan from SQL Server Management Studio



For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/performance/display-and-save-execution-plans

Migrate to Aurora MySQL Execution Plans

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Syntax differences • Completely different optimizer with different operators and rules

Overview

Aurora MySQL provides the EXPLAIN/DESCRIBE statement to display execution plan and used with the SELECT, DELETE, INSERT, REPLACE, and UPDATE statements.

> **Note:** You can use the EXPLAIN/DESCRIBE statement to retrieve table and column metadata. See the link at the end of this section for more details.

When EXPLAIN is used with a statement, MySQL returns the execution plan generated by the query optimizer. MySQL explains how the statement will be processed including information about table joins and order.
When EXPLAIN is used with the FOR CONNECTION option, it returns the execution plan for the statement executing in the named connection. You can use the FORMAT option to select either a TRADITIONAL tabular format or a JSON format.

The EXPLAIN statement requires SELECT permissions for all tables and views accessed by the query directly or indirectly. For views, EXPLAIN requires the SHOW VIEW permission. EXPLAIN can be extremely valuable for improving query performance when used to find missing indexes. You can also use EXPLAIN to determine if the optimizer joins tables in an optimal order. MySQL Workbench includes an easy to read visual explain feature similar to SQL Server Management Studio graphical execution plans.

Syntax

Simplified syntax for the EXPLAIN statement:

```
{EXPLAIN | DESCRIBE | DESC} [EXTENDED | FORMAT = TRADITIONAL | JSON]
[SELECT statement | DELETE statement | INSERT statement | REPLACE statement | UPDATE
statement | FOR CONNECTION <connection id>]
```

Examples

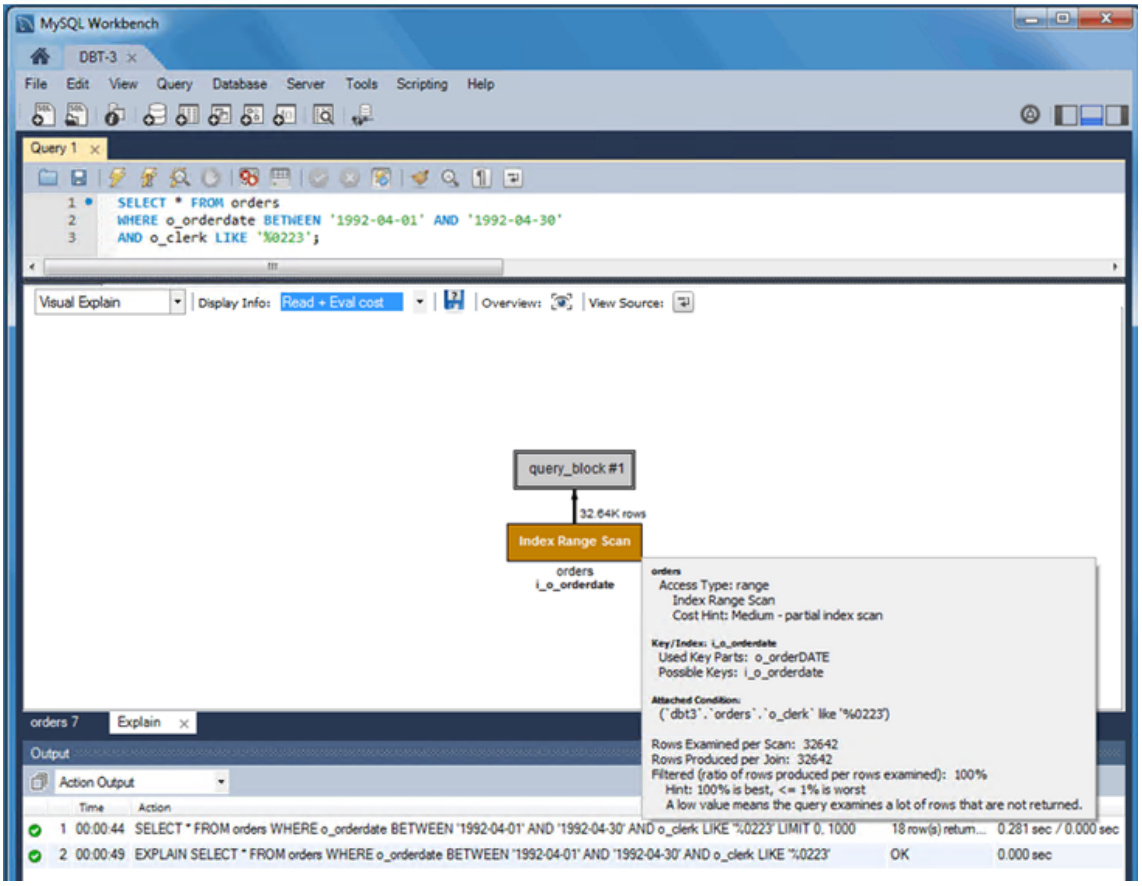
View the execution plan for a statement.

```
CREATE TABLE Employees
(
EmployeeID INT NOT NULL PRIMARY KEY,
Name VARCHAR(100) NOT NULL,
INDEX USING BTREE(Name)
);
```

```
EXPLAIN SELECT *
        FROM Employees
        WHERE Name = 'Jason';
```

```
id select_type  table      partitions    type    possible_keys    key      key_len ref      rows
Extra
-- -----------  -----      ----------    ----    -------------    ---      ------- ---
     ----    ---------       -----
1  SIMPLE       Employees                ref     Name             Name     102     const    1
```

View the MySQL Workbench graphical execution plan.



> **Note:** To instruct the optimizer to use a join order corresponding to the order in which the tables are specified in a SELECT statement, use SELECT STRAIGHT_JOIN. For more details, see Query Hints and Plan Guides.

For more information, see https://dev.mysql.com/doc/refman/5.7/en/explain.html

Migrate from SQL Server Query Hints and Plan Guides

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Hints	• Very limited set of hints - Index hints and optimizer hints as comments • Syntax differences

Overview

SQL Server *hints* are instructions that override automatic choices made by the query processor for DML and DQL statements. The term *hint* is misleading because, in reality, it forces an override to any other choice of execution plan.

JOIN Hints

LOOP, HASH, MERGE, and REMOTE hints can be explicitly added to a JOIN. For example, `... Table1 INNER LOOP JOIN Table2 ON ...`. These hints force the optimizer to use Nested Loops, Hash Match, or Merge physical join algorithms. REMOTE enables processing a join with a remote table on the local server.

Table Hints

Table hints override the default behavior of the query optimizer. Table hints are used to explicitly force a particular locking strategy or access method for a table operation clause. These hints do not modify the defaults and apply only for the duration of the DML or DQL statement.

Some common table hints are INDEX = <Index value>, FORCESEEK, NOLOCK, and TABLOCKX.

Query Hints

Query hints affect the entire set of query operators, not just the individual clause in which they appear. Query hints may be JOIN Hints, Table Hints, or from a set of hints that are only relevant for Query Hints.

Some common table hints include OPTIMIZE FOR, RECOMPILE, FORCE ORDER, FAST <rows>.

Query hints are specified after the query itself following the WITH options clause.

Plan Guides

Plan guides provide similar functionality to query hints in the sense they allow explicit user intervention and control over query optimizer plan choices. Plan guides can use either query hints or a full fixed, pre-generated plan attached to a query. The difference between query hints and plan guides is the way they are associated with a query.

While query or table hints need to be explicitly stated in the query text, they are not an option if you have no control over the source code generating these queries. If an application uses ad-hoc queries

instead of stored procedures, views, and functions, the only way to affect query plans is to use plan guides. They are often used to mitigate performance challenges with third-party software

A plan guide consists of the statement whose execution plan needs to be adjusted and either an OPTION clause that lists the desired query hints or a full XML query plan that is enforced as long it is valid.

At run time, SQL Server matches the text of the query specified by the guide and attaches the OPTION hints. Or, it assigns the provided plan for execution.

SQL Server supports three types of Plan Guides.

- **Object Plan Guides** target statements that run within the scope of a code object such as a stored procedure, function, or trigger. If the same statement is found in another context, the plan guide is not be applied.
- **SQL Plan Guides** are used for matching general ad-hoc statements not within the scope of code objects. In this case, any instance of the statement regardless of the originating client is assigned the plan guide.
- **Template Plan Guides** can be used to abstract statement templates that differ only in parameter values. It can be used to override the PARAMETERIZATION database option setting for a family of queries.

Syntax

Query Hints:

> **Note:** The following syntax is for SELECT. Query hints can be used in all DQL and DML statements.

```
SELECT <statement>
OPTION
(
{{HASH|ORDER} GROUP
|{CONCAT |HASH|MERGE} UNION
|{LOOP|MERGE|HASH} JOIN
|EXPAND VIEWS
|FAST <Rows>
|FORCE ORDER
|{FORCE|DISABLE} EXTERNALPUSHDOWN
|IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX
|KEEP PLAN
|KEEPFIXED PLAN
|MAX_GRANT_PERCENT = <Percent>
|MIN_GRANT_PERCENT = <Percent>
|MAXDOP <Number of Processors>
|MAXRECURSION <Number>
|NO_PERFORMANCE_SPOOL
|OPTIMIZE FOR (@<Variable> {UNKNOWN|= <Value>}[,...])
|OPTIMIZE FOR UNKNOWN
|PARAMETERIZATION {SIMPLE|FORCED}
|RECOMPILE
```

```
|ROBUST PLAN
|USE HINT ('<Hint>' [,...])
|USE PLAN N'<XML Plan>'
|TABLE HINT (<Object Name> [,<Table Hint>[[,...]])
});
```

Create a Plan Guide:

```
EXECUTE sp_create_plan_guide @name = '<Plan Guide Name>'
    ,@stmt = '<Statement>'
    ,@type = '<OBJECT|SQL|TEMPLATE>'
    ,@module_or_batch = 'Object Name>'|'<Batch Text>'| NULL
    ,@params = '<Parameter List>'|NULL }
    ,@hints = 'OPTION(<Query Hints>'|'<XML Plan>'|NULL;
```

Examples

Limit parallelism for a sales report query.

```
EXEC sp_create_plan_guide
    @name = N'SalesReportPlanGuideMAXDOP',
    @stmt = N'SELECT *
              FROM dbo.fn_SalesReport(GETDATE())
    @type = N'SQL',
    @module_or_batch = NULL,
    @params = NULL,
    @hints = N'OPTION (MAXDOP 1)';
```

Use table and query hints.

```
SELECT *
FROM MyTable1 AS T1
    WITH (FORCESCAN)
    INNER LOOP JOIN
    MyTable2 AS T2
    WITH (TABLOCK, HOLDLOCK)
    ON T1.Col1 = T2.Col1
WHERE T1.Date BETWEEN DATEADD(DAY, -7, GETDATE()) AND GETDATE()
```

For more information, see:

- *https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql*
- *https://docs.microsoft.com/en-us/sql/relational-databases/performance/plan-guides*

Migrate to Aurora MySQL Query Hints and Plan Guides

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Hints	• Very limited set of hints - Index hints and optimizer hints as comments • Syntax differences

Overview

Aurora MySQL supports two types of hints: Optimizer Hints and Index Hints. Unlike SQL Server, it does note provide a feature similar to Plan Guides.

Index Hints

The index hints should appear familiar to SQL Server users although the syntax is somewhat different. Index hints are placed directly after the table name as with SQL Server, but the keywords are different.

Syntax

```
SELECT ...
FROM <Table Name>
    USE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (<Index List>)
  | IGNORE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (<Index List>)
  | FORCE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (<Index List>)
...n
```

The USE INDEX hint limits the optimizer's choice to one of the indexes listed in the <Index List> white list. Alternatively, indexes can be black listed using the IGNORE keyword.

The FORCE INDEX hint is similar to USE INDEX (index_list), but with strong favor towards seek vs. scan. This hint is more like SQL Server's FORCESEEK hint although the Aurora MySQL optimizer can choose a scan if other options are not valid.

The hints use the actual index names; not column names. You can refer to Primary keys using the keyword PRIMARY.

> **Note:** In Aurora MySQL, the primary key is the clustered index.
> For more information see Clustered and Non Clustered Indexes .

The syntax for index Aurora MySQL hints has the following characteristics:

- Omitting the <Index List> is allowed for USE INDEX only. It translates to "don't use any indexes", which is equivalent to a clustered index scan.

- Index hints can be further scoped down using the FOR clause. Use FOR JOIN, FOR ORDER BY or FOR GROUP BY to limit the hint applicability to that specific query processing phase.
- Multiple index hints can be specified for the same or different scope.

Optimizer Hints

Optimizer hints give developers or administrators control over some of the optimizer decision tree. They are specified within the statement text as a comment with the prefix "+".

Optimizer hints may pertain to different scopes and are valid in only one or two scopes. The available scopes for optimizer hints in descending scope width order are:

- **Global** hints affect the entire statement. Only MAX_EXECUTION TIME is a Global Optimizer Hint.
- **Query** level hints affect a query block within a composed statement such as UNION or a sub-query.
- **Table** level hints affect a table within a query block.
- **Index** level hints affect an index of a table.

Syntax

```
SELECT /*+ <Optimizer Hints> */ <Select List>...
```

```
INSERT /*+ <Optimizer Hints> */ INTO <Table>...
```

```
REPLACE /*+ <Optimizer Hints> */ INTO <Table>...
```

```
UPDATE /*+ <Optimizer Hints> */ <Table> SET...
```

```
DELETE /*+ <Optimizer Hints> */ FROM <Table>...
```

The following optimizer hints are available in Aurora MySQL:

Hint Name	Description	Applicable Scopes
BKA, NO_BKA	Enables or disables Batched Key Access join processing	Query block, table
BNL ,NO_BNL	Enables or disables Block Nested-Loop join processing	Query block, table
MAX_EXECUTION_TIME	Limits statement execution time	Global
MRR, NO_MRR	Enables or disables Multi-Range Read optimization	Table, index
NO_ICP	Disables Index Condition Push-down optimization	Table, index
NO_RANGE_OPTIMIZATION	Disables range optimization	Table, index

Hint Name	Description	Applicable Scopes
QB_NAME	Assigns a logical name to a query block	Query block
SEMIJOIN, NO_SEMIJOIN	Enables or disables semi-join strategies	Query block
SUBQUERY	Determines MATERIALIZATION, and INTOEXISTS processing	Query block

Query block names (using `QB_NAME`) are used to distinguish a block for limiting the scope of the table hint. Add "@ " to indicate a hint scope for one or more named subqueries. For example:

```
SELECT /*+ SEMIJOIN(@SubQuery1 FIRSTMATCH, LOOSESCAN) */ *
FROM Table1
WHERE Col1 IN (SELECT /*+ QB_NAME(SubQuery1) */ Col1
               FROM t3);
```

Values for `MAX_EXECUTION_TIME` values are measured in seconds and are always global for the entire query.

> **Note:** This option does not exist in SQL Server where the execution time limit is for the session scope.

For more information the functionality of individual hints, see the links at the end of this section.

Migration Considerations

In general, the Aurora MySQL hint framework is relatively limited compared to the granular control provided by SQL Server. The specific optimizations used for SQL Server may be completely inapplicable to a new query optimizer. It is recommended to start migration testing with all hints removed. Then, selectively apply hints as a last resort if other means such as schema, index, and query optimizations have failed.

Aurora MySQL uses a list of indexes and hints, both white list (USE) and black list (IGNORE), as opposed to SQL Server's explicit index approach.

Index hints are not mandatory instructions. Aurora MySQL has some room to choose alternatives if it cannot use the hinted index. In SQL Server, forcing an invalid index or access method raises an error.

Examples

Force an index access.

```
SELECT * FROM Table1 USE INDEX (Index1) ORDER BY Col1;
```

Specify multiple index hints.

```
SELECT * FROM Table1 USE INDEX (Index1) INNER JOIN Table2 IGNORE INDEX(Index2) ON
Table1.Col1 = Table2.Col1  ORDER BY Col1;
```

Specify optimizer hints.

```
SELECT /*+ NO_RANGE_OPTIMIZATION(Table1 PRIMARY, Index2) */ Col1 FROM Table1 WHERE
Col2 = 300;
```

```
SELECT /*+ BKA(t1) NO_BKA(t2) */ * FROM Table1 INNER JOIN Table2 ON ...;
```

```
SELECT /*+ NO_ICP(t1, t2) */ * FROM Table1 INNER JOIN Table2 ON ...;
```

Summary

Feature	SQL Server	Aurora MySQL	Comments
Force a specific plan	Plan Guides	N/A	
Apply hints to a query at run time	Plan Guides	N/A	
Join hints	LOOP | MERGE | HASH	BNL | NO_BNL (Block Nested Loops)	
Locking Hints	Supported	N/A	
Force seek or scan	FORCESEEK, FORCESCAN	USE with no index list forces a clustered index scan	
Force an index	INDEX=	USE	
White list and black list indexes	N/A	Supported with USE and IGNORE	
Parameter value hints	OPTIMIZE FOR	N/A	
Compilation hints	RECOMPILE	N/A	

For more information, see:

- *https://dev.mysql.com/doc/refman/5.7/en/controlling-optimizer.html*
- *https://dev.mysql.com/doc/refman/5.7/en/optimizer-hints.html*
- *https://dev.mysql.com/doc/refman/5.7/en/index-hints.html*
- *https://dev.mysql.com/doc/refman/5.7/en/subquery-optimization.html*

Physical Storage

Migrate from SQL Server Partitioning

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Partitioning	• More partition types in Aurora MySQL with more restrictions on partitioned tables

Overview

SQL Server provides a logical and physical framework for partitioning table and index data. Each table and index are partitioned, but may have only one partition. SQL Server 2017 supports up to 15,000 partitions.

Partitioning separates data into logical units that can be stored in more than one file group. SQL Server partitioning is horizontal, where data sets of rows are mapped to individual partitions. A partitioned table or index is a single object and must reside in a single schema within a single database. Composing objects of disjointed partitions is not allowed.

All DQL and DML operations are partition agnostic except for the special predicate $partition, which can be used for explicit partition elimination.

Partitioning is typically needed for very large tables to ease the following management and performance challenges:

- Deleting or inserting large amounts of data in a single operation, with partition switching instead of individual row processing, while maintaining logical consistency.
- Maintenance operations can be split and customized per partition. For example, older data partitions can be compressed and more active partitions can be rebuilt or reorganized more frequently.
- Partitioned tables may use internal query optimization techniques such as collocated and parallel partitioned joins.
- Physical storage performance optimization by distributing IO across partitions and physical storage channels
- Concurrency improvements due to the engine's ability to escalate locks to the partition level and not the whole table.

Partitioning in SQL Server uses the following three objects:

- **Partitioning Column:** A Partitioning column is the column (or columns) being used by the partition function to partition the table or index. The value of this column determines the logical partition to which it belongs. You can use computed columns in a partition function as long as they are explicitly PERSISTED. Partitioning columns may be any data type that is a valid index column with less than 900 bytes per key, except timestamp and LOB data types.

- **Partition Function:** A Partition function is a database object that defines how the values of the partitioning columns for individual tables or index rows are mapped to a logical partition. The partition function describes the partitions for the table or index and their boundaries.
- **Partition Scheme:** A partition scheme is a database object that maps individual logical partitions of a table or an index to a set of file groups, which in turn consist of physical operating system files. Placing individual partitions on individual file groups enables backup operations for individual partitions (by backing their associated file groups).

Syntax

```
CREATE PARTITION FUNCTION <Partition Function>(<Data Type>)
AS RANGE [ LEFT | RIGHT ]
FOR VALUES (<Boundary Value 1>,...)[;]
```

```
CREATE PARTITION SCHEME <Partition Scheme>
AS PARTITION <Partition Function>
[ALL] TO (<File Group> | [ PRIMARY ] [,...])[;]
```

```
CREATE TABLE <Table Name> (<Table Definition>)
ON <Partition Schema> (<Partitioning Column>);
```

Examples

Create a partitioned table.

```
CREATE PARTITION FUNCTION PartitionFunction1 (INT)
AS RANGE LEFT FOR VALUES (1, 1000, 100000);
```

```
CREATE PARTITION SCHEME PartitionScheme1
AS PARTITION PartitionFunction1
ALL TO (PRIMARY);
```

```
CREATE TABLE PartitionTable (
Col1 INT NOT NULL PRIMARY KEY,
Col2 VARCHAR(20)
)
ON PartitionScheme1 (Col1);
```

For more information, see

- *https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-partition-scheme-transact-sql*
- *https://docs.microsoft.com/en-us/sql/t-sql/statements/create-partition-function-transact-sql*

Migrate to Aurora MySQL Partitioning

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		SCT Action Codes - Partitioning	• More partition types in Aurora MySQL with more restrictions on partitioned tables

Overview

Aurora MySQL supports a much richer framework for table partitioning than SQL Server with many additional options such as hash partitioning, sub partitioning and other features. However, it also introduces many restrictions on the tables that participate in partitioning.

> **Note:** The maximum number of partitions for a table is 8,192, including subpartitions. Although smaller than SQL Server's 15,000 theoretical partition count, practical partitioning rarely contains more than a few hundred partitions.

The following sections describe the types of partitions supported by Aurora MySQL.

RANGE Partitioning

RANGE Partitions are the equivalent of SQL Server's RANGE partition functions, which are the only type currently supported. A RANGE partitioned table has explicit boundaries defined. Each partition contains only rows for which the partitioning expression value lies within the boundaries. Value ranges must be contiguous and can not overlap. Partition boundaries are defined using the VALUES LESS THAN operator.

LIST Partitioning

List partitioning somewhat resembles range partitioning. Similar to RANGE, each partition must be defined explicitly. The main difference between LIST and RANGE partitioning is that LIST partitions are defined using a set of value lists instead of a contiguous range.

Use the PARTITION BY LIST(<Column Expression>) to define the type and the partitioning column. <Column Expression> must return an integer value.
Afterward, every partition is defined using the VALUES IN (<Value List>) where <Value List> consists of a comma-separated list of integer values.

RANGE | LIST COLUMNS Partitioning

COLUMNS partitioning is a variant of both RANGE and LIST partitioning. However, COLUMNS partitioning allows multiple columns in partitioning keys. All column values are considered for matching to a particular partition.

Both RANGE COLUMNS partitioning and LIST COLUMNS partitioning allow the use of non-integer values for defining both value ranges and value lists. The following data types are supported for COLUMNS partitioning:

- All integer types
- DATE and DATETIME
- CHAR, VARCHAR, BINARY, and VARBINARY

HASH Partitioning

HASH partitioning is typically used to guarantee even distribution of rows for a desired number of partitions. When using either RANGE or LIST partitioning (and their variants), the boundaries are explicitly defined and associate a row to a partition based on the column value or set of values.

With HASH partitioning, Aurora MySQL manages the values and individual partitions. You only need to specify the column or column expression to be hashed and the total number of partitions.

Subpartitioning

With Subpartitioning, or composite partitioning, each primary partition is further partitioned to create a two-layer partitioning hierarchy. Subpartitions must use either HASH or KEY partitioning and only RANGE or LIST partitions may be subpartitioned. SQL Server does not support subpartitions.

Partition Management

Aurora MySQL provides several mechanisms for managing partitioned tables including adding, dropping, redefining, merging, and splitting existing partitioned tables. These management operations can use the Aurora MySQL partitioning extensions to the ALTER TABLE statement.

Dropping Partitions

For tables using either RANGE or LIST partitioning, drop a partition using the ALTER TABLE ... DROP PARTITION statement option.

When a partition is dropped from a RANGE partitioned table, all the data in the current partition is deleted and new rows with values that would have fit the partition go to the immediate neighbor partition.

When a partition is dropped from a LIST partitioned table, data is also deleted but new rows with values that would have fit the partition cannot be INSERTED or UPDATED because they no longer have a logical container.

For HASH and KEY partitions, use the ALTER TABLE ... COALESCE PARTITION <Number of Partitions>. This approach reduces the current total number of partitions by the <Number of Partitions> value.

Adding and Splitting Partitions

To add a new range boundary, or partition for a new list of values, use the ALTER TABLE ... ADD PARTITION statement option.

For RANGE partitioned tables, your can only add a new range to the end of the list of existing partitions.

If you need to split an existing RANGE partition into two partitions, use the ALTER TABLE ... REORGANIZE PARTITION statement.

Switching and Exchanging Partitions

Aurora MySQL supports the exchange of a table partition, or a subpartition, with another table. Use the `ALTER TABLE <Partitioned Table> EXCHANGE PARTITION <Partition> WITH TABLE <Non Partitioned Table>` statement option.

The non-partitioned table can not be a temporary table and the schema of both tables must be identical. The non partitioned table can not have a foreign key being referenced, or referencing it. It is critical that all rows in the non-partitioned table are within the partition boundaries, unless the WITHOUT VALIDATION option is used.

> **Note:** ALTER TABLE ... EXCHANGE PARTITION requires the ALTER, INSERT, CREATE, and DROP privileges.

Executing the ALTER TABLE ... EXCHANGE PARTITION statement does not trigger the execution of triggers on the partitioned table or the exchanged non-partitioned table.

> **Note:** AUTO_INCREMENT columns in the exchanged table are reset when ALTER TABLE ... EXCHANGE PARTITION is executed.
> For more information, see Sequences and Identity.

Syntax

Create a partitioned table.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <Table Name>
(<Table Definition>) [<Table Options>]
PARTITION BY
        { [LINEAR] HASH(<Expression>)
        | [LINEAR] KEY [ALGORITHM={1|2}] (<Column List>)
        | RANGE{(expr) | COLUMNS(<Column List>)}
        | LIST{(expr) | COLUMNS(<Column List>)} }
    [PARTITIONS <Number>]
    [SUBPARTITION BY
        { [LINEAR] HASH(<Expression>)
        | [LINEAR] KEY [ALGORITHM={1|2}] (<Column List>) }
      [SUBPARTITIONS <Number>]
```

Reorganize/Split a partition.

```
ALTER TABLE <Table Name>
REORGANIZE PARTITION <Partition> INTO (
PARTITION <New Partition 1> VALUES LESS THAN (<New Range Boundary>),
PARTITION <New Partition 2> VALUES LESS THAN (<Range Boundary>)
);
```

Exchange a partition.

```
ALTER TABLE <Partitioned Table> EXCHANGE PARTITION <Partition> WITH TABLE <Non Partitioned Table>;
```

DROP a partition.

```
ALTER TABLE <Table Name> DROP PARTITION <Partition>;
```

Migration Considerations

Because Aurora MySQL stores each table in its own file and since file management is performed by AWS and cannot be modified, some of the physical aspects of partitioning in SQL Server do not apply to Aurora MySQL. For example, the concept of file groups and assigning partitions to file groups.

Aurora MySQL does not support foreign keys partitioned tables. Neither the referencing table nor referenced table can use partitioning. Partitioned tables can not have foreign keys referencing other tables or be referenced from other tables. Partitioning keys or expressions in Aurora MySQL must be INT data types. They cannot be ENUM types. The expression may result in a NULL state. The exceptions to this rule are:

- Partitioning by RANGE COLUMNS or LIST COLUMNS. It is possible to use strings, DATE, and DATETIME columns.
- Partitioning by [LINEAR] KEY. Allows use of any valid MySQL data type (except TEXT and BLOB) for partitioning keys. Aurora MySQL's key-hashing functions result in the correct data type.

Partitioned tables support neither FULLTEXT indexes nor spatial data types such as POINT and GEOMETRY.

Unlike SQL Server, exchanging partitions in Aurora MySQL is only supported between a partitioned and a non-partitioned table. In SQL server, SWITCH PARTITION can be used to switch any partition between partitions tables because technically all tables are partitioned (to one or more partitions).

Examples

Create a RANGE partitioned table.

```
CREATE TABLE MyTable (
    Col1 INT NOT NULL PRIMARY KEY,
    Col2 VARCHAR(20) NOT NULL
)
PARTITION BY RANGE (Col1)
(
    PARTITION p0 VALUES LESS THAN (100000),
    PARTITION p1 VALUES LESS THAN (200000),
    PARTITION p2 VALUES LESS THAN (300000),
    PARTITION p3 VALUES LESS THAN (400000)
);
```

Create subpartitions.

```
CREATE TABLE MyTable (Col1 INT NOT NULL, DateCol DATE NOT NULL, )
PARTITION BY RANGE(YEAR(DateCol))
SUBPARTITION BY HASH(TO_DAYS(<DateCol>))
SUBPARTITIONS 2 (
     PARTITION p0 VALUES LESS THAN (1990),
     PARTITION p1 VALUES LESS THAN (2000),
     PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

Drop a RANGE partition.

```
ALTER TABLE MyTable DROP PARTITION p2
```

Reduce the number of HASH partitions by four.

```
ALTER TABLE <Table Name> COALESCE PARTITION 4;
```

Add RANGE partitions.

```
ALTER TABLE MyTable ADD PARTITION (PARTITION p4 VALUES LESS THAN (50000));
```

Summary

The following table identifies similarities, differences, and key migration considerations.

Feature	SQL Server	Aurora MySQL	Comments
Partition types	RANGE only	RANGE, LIST, HASH, KEY	
Partitioned tables scope	All tables are partitioned, some have more than one partition	All tables are not partitioned, unless explicitly partitioned	
Partition boundary direction	LEFT or RIGHT	RIGHT only	Only determines to which partition the boundary value itself will go.
Dynamic Range Partition	N/A — literal values must be explicitly set in partition function		
Exchange partition	Any partition to any partition	Partition to table (non-partitioned table)	Only partition to table, no partition to partition switch.
Partition function	Abstract function object, Independent of individual	Defined per partitioned table	

Feature	SQL Server	Aurora MySQL	Comments
	column		
Partition scheme	Abstract partition storage mapping object	N/A	In Aurora MySQL, physical storage is managed by AWS RDS.
Limitations on partitioned tables	None — all tables are partitioned	Extensive — No FK, no Full text	For more information, see https://dev.mysql.-com/doc/refman/5.7/en/partitioning-limitations.html

For more information, see

- *https://dev.mysql.com/doc/refman/5.7/en/partitioning-overview.html*
- *https://dev.mysql.com/doc/refman/5.7/en/partitioning-management.html*
- *https://dev.mysql.com/doc/refman/5.7/en/partitioning-types.html*

Security

Migrate from SQL Server Column Encryption

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Syntax • Encryption hierarchy much simpler

Overview

SQL Server provides encryption and decryption functions to secure the content of individual columns. The following list identifies common encryption functions:

- EncryptByKey and DecryptByKey
- EncryptByCert and DecruptByCert
- EncryptByPassPhrase and DecruptByPassPhrase
- EncryptByAsymKey and DecryptByAsymKey

You can use these functions anywhere in your code; they are not limited to encrypting table columns. A common use case is to increase run time security by encrypting of application user security tokens passed as parameters.

These functions follow the general SQL Server encryption hierarchy, which in turn use the Windows Server Data Protection API.

Symmetric encryption and decryption consume minimal resources and can be used for large data sets.

> **Note:** This section does not cover Transparent Data Encryption (TDE) or AlwaysEncrypted end-to-end encryption.

Syntax

General syntax for EncryptByKey and DecryptByKey:

```
EncryptByKey ( <key GUID> , { 'text to be encrypted' }, { <use authenticator flag>}, { <authenticator> }  );
```

```
DecryptByKey ( 'Encrypted Text' , <use authenticator flag>, { <authenticator> )
```

Examples

The following example demonstrates how to encrypt an employee Social Security Number:

Create a database master key.

```
USE MyDatabase;
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = '<MyPassword>';
```

Create a certificate and a key.

```
CREATE CERTIFICATE Cert01
WITH SUBJECT = 'SSN';
```

```
CREATE SYMMETRIC KEY SSN_Key
WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE Cert01;
```

Create an employees table.

```
CREATE TABLE Employees
(
EmployeeID INT PRIMARY KEY,
SSN_encrypted VARBINARY(128)  NOT NULL
);
```

Open the symmetric key for encryption.

```
OPEN SYMMETRIC KEY SSN_Key
DECRYPTION BY CERTIFICATE Cert01;
```

Insert the encrypted data.

```
INSERT INTO Employees (EmployeeID, SSN_encrypted)
VALUES
(1, EncryptByKey(Key_GUID('SSN_Key') , '1112223333', 1, HashBytes('SHA1', CONVERT
(VARBINARY, 1)));
```

```
SELECT EmployeeID,
       CONVERT(CHAR(10), DecryptByKey(SSN, 1 , HashBytes('SHA1', CONVERT(VARBINARY,
EmployeeID)))) AS SSN
FROM Employees;

EmployeeID      SSN_Encrypted                       SSN
----------      ---------------------               ----------
1               0x00F983FF436E32418132...           1112223333
```

For more information, see:

- https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/encrypt-a-column-of-data
- https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/encryption-hierarchy

Migrate to Aurora MySQL Column Encryption

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Syntax • Encryption hierarchy much simpler

Overview

Aurora MySQL provides encryption and decryption functions similar to SQL Server with a much less elaborate security hierarchy that is easier to manage.

The encryption functions require the actual key as a string, so you must take extra measures to protect the data. For example, hashing the key values on the client.

Aurora MySQL supports the AES and DES encryption algorithms. You can use the following functions data encryption and decryption:

- AES_DECRYPT
- AES_ENCRYPT
- DES_DECRYPT
- DEC_ENCRYPT

  Note: The ENCRYPT, DECRYPT, ENCODE and DECODE functions are deprecated beginning with MySQL version 5.7.2 and 5.7.6. Asymmetric encryption is not supported in Aurora MySQL

Syntax

General syntax for the encryption functions:

```
[A|D]ES_ENCRYPT(<string to be encrypted>, <key string> [,<initialization vector>])
[A|D]ES_DECRYPT(<encrypted string>, <key string> [,<initialization vector>])
```

It is highly recommended to use the optional initialization vector to circumvent whole value replacement attacks. When encrypting column data, it is common to use an immutable key as the initialization vector. With this approach, decryption fails if a whole value moves to another row.

Consider using SHA2 instead of SHA1 or MD5 because there are known exploits available for the SHA1 and MD5. Passwords, keys, or any sensitive data passed to these functions from the client are not encrypted unless you are using an SSL connection. One benefit of using Amazon AWS IAM is that database connections are encrypted with SSL by default.
For more information, see Users and Roles.

Examples

The following example demonstrates how to encrypt an employee Social Security Number:

Create an employees table.

```
CREATE TABLE Employees
(
EmployeeID INT NOT NULL PRIMARY KEY,
SSN_Encrypted BINARY(32) NOT NULL
);
```

Insert the encrypted data.

```
INSERT INTO Employees (EmployeeID, SSN_Encrypted)
VALUES (1, AES_ENCRYPT('1112223333', UNHEX(SHA2('MyPassword',512)), 1));
```

Note: Use the UNHEX function for more efficient storage and comparisons.

Verify decryption.

```
SELECT EmployeeID,
       SSN_Encrypted,
       AES_DECRYPT(SSN_Encrypted, UNHEX(SHA2('MyPassword',512)), EmployeeID) AS SSN
FROM Employees

EmployeeID      SSN_Encrypted           SSN
----------      ------------------      ----------
1               ` ©> +yp°øýNZ~Gø        1112223333
```

For more information, see https://dev.mysql.com/doc/refman/5.7/en/encryption-functions.html

Migrate from SQL Server Data Control Language

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Simpler permission hierarchy

The ANSI standard specifies, and most Relational Database Management Systems (RDBMS) use, GRANT and REVOKE commands to control permissions.

However, SQL Server also provides a DENY command to explicitly restrict access to a resource. DENY takes precedence over GRANT and is needed to avoid potentially conflicting permissions for users having multiple logins. For example, if a user has DENY for a resource through group membership but GRANT access for a personal login, the user is denied access to that resource.

SQL Server allows granting permissions at multiple levels from lower-level objects such as columns to higher level objects such as servers. Permissions are categorized for specific services and features such as the service broker.

Permissions are used in conjunction with database users and roles.
See Users and Roles for more details.

Syntax

Simplified syntax for SQL Server DCL commands:

```
GRANT { ALL [ PRIVILEGES ] } | <permission>  [ ON <securable> ] TO <principal>
```

```
DENY   { ALL [ PRIVILEGES ] } | <permission> [ ON <securable> ] TO <principal>
```

```
REVOKE [ GRANT OPTION FOR ] {[ ALL [ PRIVILEGES ] ]|<permission>} [ ON <securable> ] {
TO | FROM } <principal>
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/security/permissions-hierarchy-database-engine

Migrate to Aurora MySQL Data Control Language

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
		N/A	• Simpler permission hierarchy

Overview

Aurora MySQL supports the ANSI Data Control Language (DCL) commands GRANT and REVOKE.

Administrators can grant or revoke permissions for individual objects such as a column, a stored function, or a table. Permissions can be granted to multiple objects using wild cards.

Only explicitly GRANTED permissions can be revoked. For example, if a user was granted SELECT permissions for the entire database using the following command:

```
GRANT SELECT
ON database.*
TO UserX;
```

it is not possible to REVOKE the permission for a single table. You must revoke the SELECT permission for all tables using the following command:

```
REVOKE SELECT
ON database.*
FROM UserX;
```

Aurora MySQL provides a GRANT permission option, which is very similar to SQL Server's WITH GRANT OPTION clause. This permission gives a user permission to further grant the same permission to other users.

```
GRANT EXECUTE
ON PROCEDURE demo.Procedure1
TO UserY
WITH GRANT OPTION;
```

> **Note:** Aurora MySQL users can have resource restrictions associated with their accounts similar to the SQL Server resource governor. See Resource Governor.

The following table identifies Aurora MySQL privileges:

Permissions	Use to
ALL [PRIVILEGES]	Grant all privileges at the specified access level except GRANT OPTION and PROXY.
ALTER	Enable use of ALTER TABLE. Levels: Global, database, table.

Permissions	Use to
ALTER ROUTINE	Enable stored routines to be altered or dropped. Levels: Global, database, procedure.
CREATE	Enable database and table creation. Levels: Global, database, table.
CREATE ROUTINE	Enable stored routine creation. Levels: Global, database.
CREATE TEMPORARY TABLES	Enable the use of CREATE TEMPORARY TABLE. Levels: Global, database.
CREATE USER	Enable the use of CREATE USER, DROP USER, RENAME USER, and REVOKE ALL PRIVILEGES. Level: Global.
CREATE VIEW	Enable views to be created or altered. Levels: Global, database, table.
DELETE	Enable the use of DELETE. Level: Global, database, table.
DROP	Enable databases, tables, and views to be dropped. Levels: Global, database, table.
EVENT	Enable the use of events for the Event Scheduler. Levels: Global, database.
EXECUTE	Enable the user to execute stored routines. Levels: Global, database, table.
GRANT OPTION	Enable privileges to be granted to or removed from other accounts. Levels: Global, database, table, procedure, proxy.
INDEX	Enable indexes to be created or dropped. Levels: Global, database, table.
INSERT	Enable the use of INSERT. Levels: Global, database, table, column.
LOCK TABLES	Enable the use of LOCK TABLE Son tables for which you have the SELECT privilege. Levels: Global, database.
PROXY	Enable user proxying. Level: From user to user.
REFERENCES	Enable foreign key creation. Levels: Global, database, table, column.
REPLICATION CLIENT	Enable the user to determine the location of master and slave servers. Level: Global.
REPLICATION SLAVE	Enable replication slaves to read binary log events from the master. Level: Global.
SELECT	Enable the use of SELECT. Levels: Global, database, table, column.
SHOW DATABASES	Enable SHOW DATABASES to show all databases. Level: Global.
SHOW VIEW	Enable the use of SHOW CREATE VIEW. Levels: Global, database, table.

Permissions	Use to
TRIGGER	Enable trigger operations. Levels: Global, database, table.
UPDATE	Enable the use of UPDATE. Levels: Global, database, table, column.

Syntax

```
GRANT <privilege type>...
ON [object type] <privilege level>
TO <user> ...
```

```
REVOKE <privilege type>...
ON [object type] <privilege level>
FROM <user> ...
```

> **Note:** Table, Function, and Procedure object types can be explicitly stated but are not mandatory.

Examples

Attempt to REVOKE a partial permission that was granted as a wild card permission.

```
CREATE USER TestUser;
GRANT SELECT
 ON Demo.*
 TO TestUser;
REVOKE SELECT ON Demo.Invoices
 FROM TestUser
```

The command above displays the following error:

```
SQL ERROR [1147][42000]: There is no such grant defined for user TestUser on host '%'
on table 'Invoices'
```

Grant SELECT Permission to a user on all tables in the demo database.

```
GRANT SELECT
ON Demo.*
TO 'user'@'localhost';
```

Revoke EXECUTE permissions from a user on the `EmployeeReport` stored procedure.

```
REVOKE EXECUTE
ON Demo.EmployeeReport
FROM 'user'@'localhost';
```

For more information, seehttps://dev.mysql.com/doc/refman/5.7/en/grant.html

Migrate from SQL Server Transparent Data Encryption

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Enable encryption when creating the database instance

Overview

Transparent Data Encryption (TDE) is an SQL Server feature designed to protect data at-rest in the event an attacker obtains the physical media containing database files.

TDE does not require application changes and is completely transparent to users. The storage engine encrypts and decrypts data on-the-fly. Data is not encrypted while in memory or on the network. TDE can be turned on or off individually for each database.

TDE encryption uses a Database Encryption Key (DEK) stored in the database boot record, making it available during database recovery. The DEK is a symmetric key signed with a server certificate from the master system database.

In many instances, security compliance laws require TDE for data at rest.

Examples

The following example demonstrates how to enable TDE for a database:

Create a master key and certificate.

```
USE master;
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'MyPassword';
CREATE CERTIFICATE TDECert WITH SUBJECT = 'TDE Certificate';
```

Create a database encryption key.

```
USE MyDatabase;
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE TDECert;
```

Enable TDE.

```
ALTER DATABASE MyDatabase SET ENCRYPTION ON;
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption

Migrate to Aurora MySQL Transparent Data Encryption

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• Enable encryption when creating the database instance

Overview

Amazon Aurora MySQL provides the ability to encrypt data at rest (data stored in persistent storage) for new database instances. When data encryption is enabled, Amazon Relational Database Service (RDS) automatically encrypts the database server storage, automated backups, read replicas, and snapshots using the AES-256 encryption algorithm.

You can manage the keys used for RDS encrypted instances from the Identity and Access Management (IAM) console using the AWS Key Management Service (AWS KMS). If you require full control of a key, you must manage it yourself. You cannot delete, revoke, or rotate default keys provisioned by AWS KMS.

The following limitations exist for Amazon RDS encrypted instances:

- You can only enable encryption for an Amazon RDS database instance when you create it, not afterward. It is possible to encrypt an existing database by creating a snapshot of the database instance and then creating an encrypted copy of the snapshot. You can restore the database from the encrypted snapshot (see https://-docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CopySnapshot.html)
- Encrypted database instances cannot be modified to disable encryption.
- Encrypted Read Replicas must be encrypted with the same key as the source databse instance.
- An unencrypted backup or snapshot can not be restored to an encrypted database instance.
- KMS encryption keys are specific to the region where they are created. Copying an encrypted snapshot from one region to another requires the KMS key identifier of the destination region.

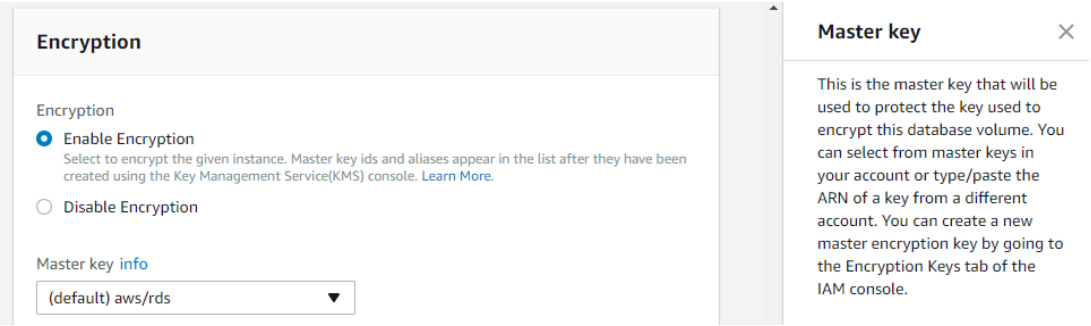
  Note: Disabling the key for an encrypted database instance prevents reading from, or writing to, that instance. When Amazon RDS encounters a database instance encrypted by a key to which Amazon RDS does not have access, it puts the database instance into a terminal state. In this state, the database instance is no longer available and the current state of the database can't be recovered. To restore the database instance, you must re-enable access to the encryption key for Amazon RDS and then restore the database instance from a backup.

Examples

The following walk-through demonstrates how to enable TDE.

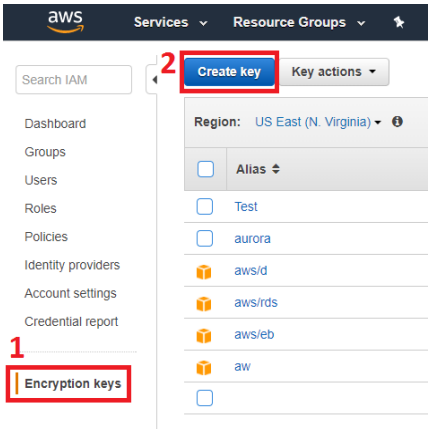
Enable Encryption

In the database settings, enable encryption and choose a master key. You can choose the default key provided for the account or define a specific key based on an IAM KMS ARN from your account or a different account.

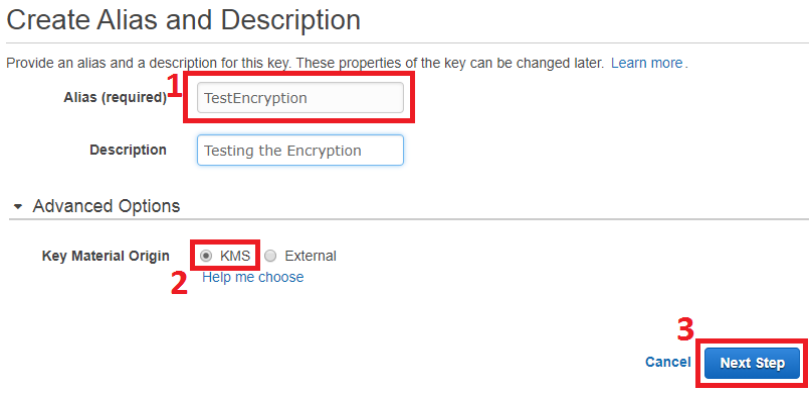


Create an Encryption Key

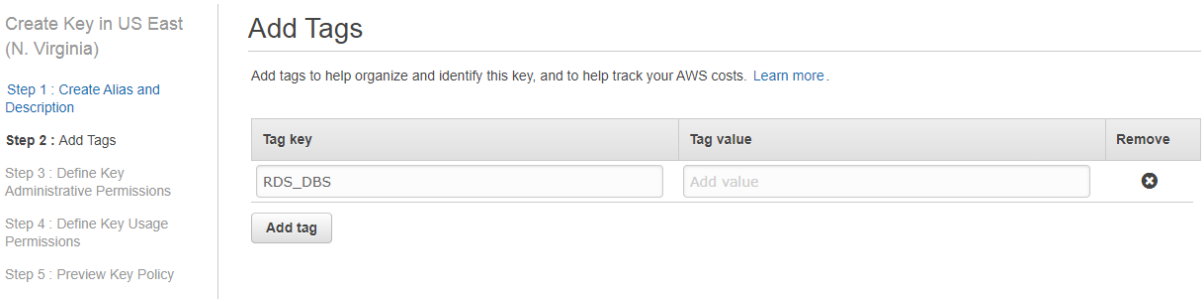
Navigate to the IAM and click **Encryption keys** and then **CREATE KEY**.



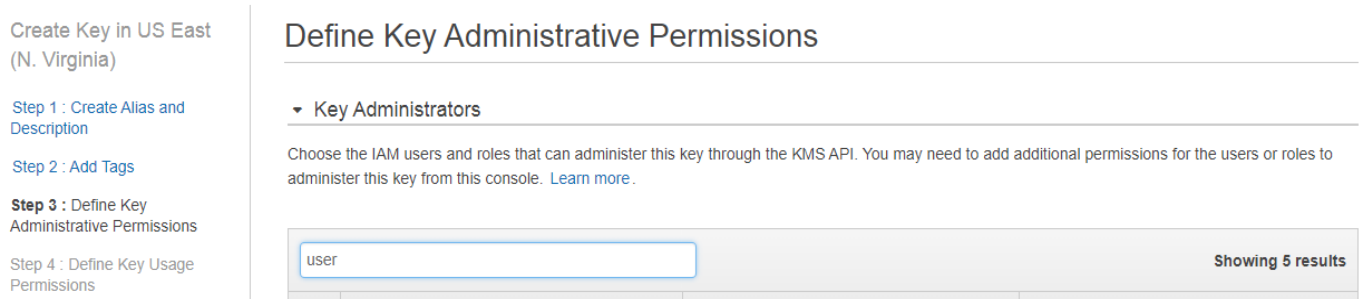
Enter the **Alias** and **Description**. Under **Advanced Options**, Select **KMS**. Click **Next**.



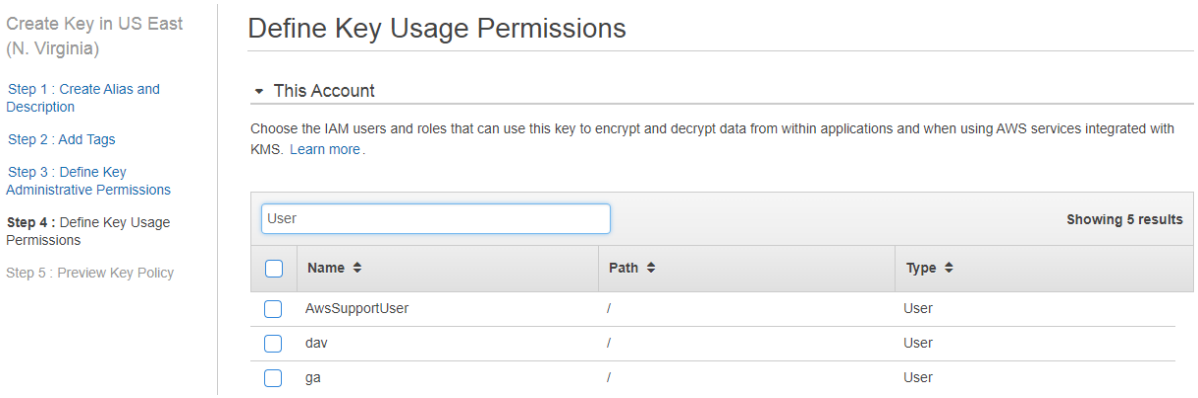
Add a tag specifying the key's name. Click **Next**.



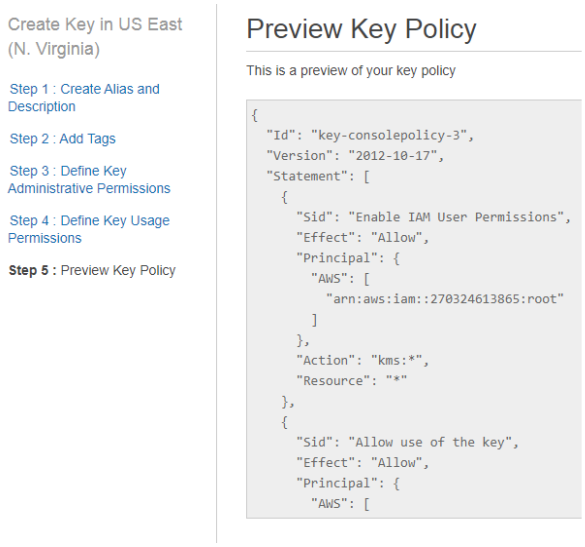
Click **Next** (skip **Step-3: Define Key Administrative Permissions**).



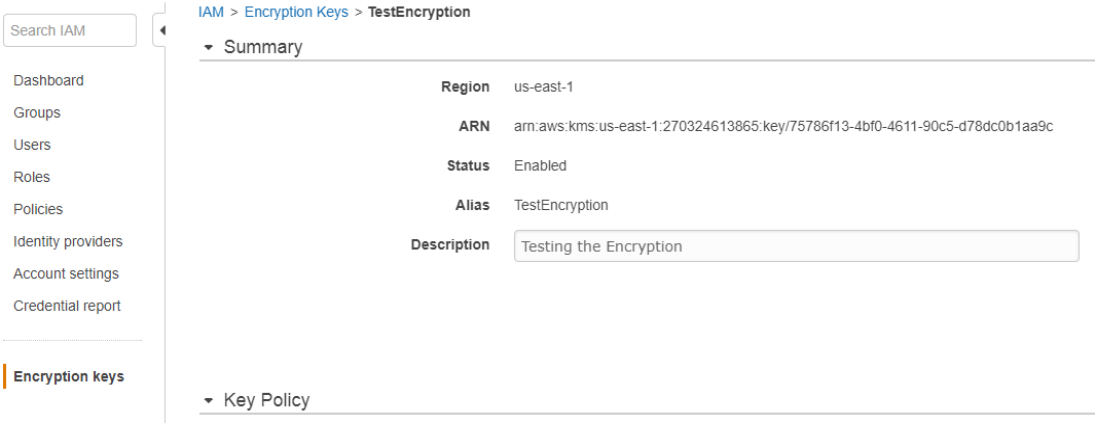
Assign the key to the users who will access Aurora MySQL. For more information about users, see [Users and Roles](#).



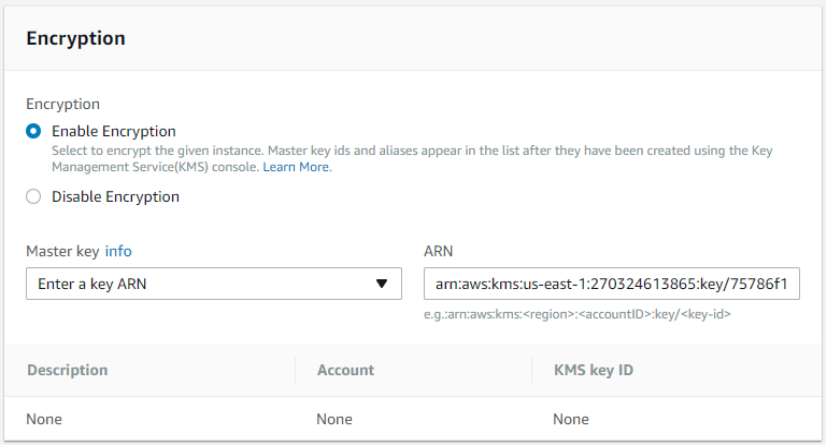
The system displays the ARN of the key and its account.



Click **Finish**. Click the new key to display the ARN.



Set the master encryption key using the newly created ARN.



Launch the instance.

For more information, see http://docs.aws.amazon.com/AmazonS3/latest/dev/SSEUsingRESTAPI.html and http://docs.aws.amazon.com/cli/latest/reference/s3/cp.htm

Migrate from SQL Server Users and Roles

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• No native role support in the database • Use AWS IAM accounts with the AWS Authentication Plugin

Overview

SQL Server provides two layers of security principals: *Logins* at the server level and *Users* at the database level. Logins are mapped to users in one or more databases. Administrators can grant logins server-level permissions that are not mapped to particular databases such as Database Creator, System Administrator and Security Administrator .

SQL Server also supports *Roles* for both the server and the database levels. At the database level, administrators can create custom roles in addition to the general purpose built-in roles.

For each database, administrators can create users and associate them with logins. At the database level, the built-in roles include *db_owner*, *db_datareader*, *db_securityadmin* and others. A database user can belong to one or more roles (users are assigned to the *public* role by default and can't be removed). Administrators can grant permissions to roles and then assign individual users to the roles to simplify security management.

Logins are authenticated using either Windows Authentication, which uses the Windows Server Active Directory framework for integrated single sign-on, or SQL authentication, which is managed by the SQL Server service and requires a password, certificate, or asymmetric key for identification. Logins using windows authentication can be created for individual users and domain groups.

In previous versions of SQL server, the concepts of user and schema were interchangeable. For backward compatibility, each database has several existing schemas, including a default schema named *dbo* which is owned by the *db_owner* role. Logins with system administrator privileges are automatically mapped to the dbo user in each database. Typically, you do not need to migrate these schemas.

Examples

Create a login.

```
CREATE LOGIN MyLogin WITH PASSWORD = 'MyPassword'
```

Create a database user for `MyLogin`.

```
USE MyDatabase; CREATE USER MyUser FOR LOGIN MyLogin;
```

Assign `MyLogin` to a server role.

```
ALTER SERVER ROLE dbcreator ADD MEMBER 'MyLogin'
```

Assign `MyUser` to the `db_datareader` role.

```
ALTER ROLE db_datareader ADD MEMBER 'MyUser';
```

For more information, see https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles

Migrate to Aurora MySQL Users and Roles

Feature Compatibility	SCT Automation Level	SCT Action Code Index	Key Differences
	N/A	N/A	• No native role support in the database • Use AWS IAM accounts with the AWS Authentication Plugin

Overview

Aurora MySQL supports only *Users*; *Roles* are not supported. Database administrators must specify privileges for individual users. Aurora MySQL uses database user accounts to authenticate sessions and authorize access to specific database objects.

> **Note:** When granting privileges, you have the option to use wild-card characters for specifying multiple privileges for multiple objects.
> See the Data Control Language for more details.

When using Identity and Access Management (IAM) database authentication, roles are available as part of the IAM framework and can be used for authentication. This authentication method uses tokens in place of passwords. AWS Signature Version 4 generates authentication tokens with a lifetime of 15 minutes. You do not need to store user credentials in the database because authentication is managed externally. You can use IAM in conjunction with standard database authentication.

> **Note:** In Aurora MySQL, a *database* is equivalent to an SQL Server *schema*.

The AWS Authentication Plugin works seamlessly with Aurora MySQL instances. Users logged in with AWS IAM accounts use access tokens to authenticate. This mechanism is similar to the SQL Server windows authentication option.

IAM database authentication provides the following benefits:

- Supports roles for simplifying user and access management.
- Provides a single sign on experience that is safer than using MySQL managed passwords.
- Encrypts network traffic to and from the database using Secure Sockets Layer (SSL) protocol.
- Provides centrally managed access to your database resources, alleviating the need to manage access individually for each database instance or database cluster.

  > **Note:** IAM database authentication limits the number of new connections to 20 connections/second.

Syntax

Simplified syntax for CREATE USER in Aurora MySQL:

```
CREATE USER <user> [<authentication options>] [REQUIRE {NONE | <TLS options>] }]
[WITH <resource options> ] [<Password options> | <Lock options>]
```

```
<Authentication option>:
{IDENTIFIED BY 'auth string'|PASSWORD 'hash string'|WITH auth plugin|auth plugin BY
'auth_string'|auth plugin AS 'hash string'}
<TLS options>: {SSL| X509| CIPHER 'cipher'| ISSUER 'issuer'| SUBJECT 'subject'}
<Resource options>: { MAX_QUERIES_PER_HOUR | MAX_UPDATES_PER_HOUR | MAX_CONNECTIONS_
PER_HOUR | MAX_USER_CONNECTIONS count}
<Password options>: {PASSWORD EXPIRE | DEFAULT | NEVER | INTERVAL N DAY}
<Lock options>: {ACCOUNT LOCK | ACCOUNT UNLOCK}
```

Note: Aurora MySQL allows you to assign resource limitations to specific users, similar to SQL Server Resource Governor. See Resource Governor for more details.

Examples

Create a user, force a password change, and impose resource limits.

```
CREATE USER 'Dan'@'localhost'
IDENTIFIED WITH mysql_native_password BY 'Dan''sPassword'
WITH MAX_QUERIES_PER_HOUR 500
PASSWORD EXPIRE;
```

Create a user with IAM authentication.

```
CREATE USER LocalUser
IDENTIFIED WITH AWSAuthenticationPlugin AS 'IAMUser';
```

Summary

The following table summarizes common security tasks and the differences between SQL Server and Aurora MySQL.

Task	SQL Server	Aurora MySQL
View database users	`SELECT Name FROM sys.sysusers`	`SELECT User FROM mysql.user`
Create a user and password	`CREATE USER <User Name> WITH PASSWORD = <PassWord>;`	`CREATE USER <User Name> IDENTIFIED BY <Password>`
Create a role	`CREATE ROLE <Role Name>`	`Use AWS IAM Roles`
Change a user's password	`ALTER LOGIN <SQL Login> WITH PASSWORD = <PassWord>;`	`ALTER USER <User Name> IDENTIFIED BY <Password>`
External authentication	`Windows Authentication`	`AWS IAM (Identity and Access Managemet)`
Add a user to a role	`ALTER ROLE <Role Name> ADD MEMBER <User Name>`	`Use AWS IAM Roles`
Lock a user	`ALTER LOGIN <Login Name> DISABLE`	`ALTER User <User Name> ACCOUNT LOCK`

Task	SQL Server	Aurora MySQL
Grant SELECT on a schema	`GRANT SELECT ON SCHEMA::<Schema Name> to <User Name>`	`GRANT SELECT ON <Schema Name>.* TO <User Name>`

For more information, see

- https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
- https://docs.aws.amazon.com/IAM/latest/UserGuide/id.html

Appendix A: SQL Server 2018 Deprecated Feature List

SQL Server 2018 Deprecated Feature	Section
TEXT, NTEXT, and IMAGE data types	SQL Server Data Types topic and Aurora MySQL Data Types topic
SET ROWCOUNT for DML	SQL Server Session Options topic and Aurora MySQL Session Options topic
TIMESTAMP syntax for CREATE TABLE	SQL Server Creating Tables topic and Aurora MySQL Creating Tables topic
DBCC DBREINDEX, INDEXDEFRAG, and SHOWCONTIG	SQL Server Maintenance Plans topic
Old SQL Mail	SQL Server Database Mail
IDENTITY Seed, Increment, non PK, and compound	SQL Server Sequences and Identity and Aurora MySQL Sequences and Identity
Stored Procedures RETURN Values	SQL Server Stored Procedures and Aurora MySQL Stored Procedures
GROUP BY ALL, Cube, and Compute By	SQL Server GROUP BY and Aurora MySQL GROUP BY
DTS	SQL Server ETL and Aurora MySQL ETL
Old outer join syntax *= and =*	SQL Server Table JOIN and Aurora MySQL Table JOIN
'String Alias' = Expression	Migration Tips
DEFAULT keyword for INSERT statements	Migration Tips

Appendix B: Migration Quick Tips

This section provides migration tips that can help save time as you transition from SQL Server to Aurora MySQL. They address many of the challenges faced by administrators new to Aurora MySQL. Some of these tips describe functional differences in similar features between SQL Server and Aurora MySQL.

Management

- The concept of a DATABASE in MySQL is not the same as SQL Server. A DATABASE in MySQL is synonymous with SCHEMA. See Databases and Schemas.
- You cannot create explicit statistics objects in Aurora MySQL. Statistics are collected and maintained for indexes only.
- The equivalent of SQL Server's CREATE DATABASE... AS SNAPSHOT OF... resembles Aurora MySQL Database cloning. However, unlike SQL Server snapshots, which are read only, Aurora MySQL cloned databases are updatable, .
- In Aurora MySQL, the term "Database Snapshot" is equivalent to SQL Server's BACKUP DATABASE... WITH COPY_ONLY.
- Partitioning in Aurora MySQL supports more partition types than SQL Server. However, be aware that partitioning in Aurora MySQL restricts the use of many other fundamental features such as foreign keys.
- Partition SWITCH in SQL Server can be performed between any two partitions of any two tables. In Aurora MySQL, you can only EXCHANGE a table partition with a full table.
- Unlike SQL Server's statistics, Aurora MySQL does not collect detailed key value distribution; it relies on selectivity only. When troubleshooting execution, be aware that parameter values are insignificant to plan choices.

SQL

- Triggers work differently in Aurora MySQL. Triggers are executed for each row (not just once). The syntax for inserted and deleted is new and old. They always contain 0, or 1 row.
- Triggers in Aurora MySQL can not be modified using the ALTER command. Drop and replace instead.
- Aurora MySQL does not support the @@FETCH_STATUS system parameter for cursors. When declaring cursors in Aurora MySQL, you must create an explicit HANDLER object, which can set a variable based on the "row not found in cursor" event. See the example in Stored Procedures.
- To execute a stored procedure, use CALL instead of EXECUTE.
- To execute a string as a query, use Aurora MySQL Prepared Statements instead of either sp_executesql, or EXECUTE(<String>) syntax.
- Aurora MySQL supports AFTER and BEFORE triggers. There is no equivalent to INSTEAD OF triggers. The only difference between BEFORE and INSTEAD OF triggers is that DML statements are applied (row by row) to the base table when using BEFORE and does not require an explicit

action in the trigger. If you need to make changes to data affected by a trigger, you can UPDATE the new and old tables; the changes are persisted.

- Aurora MySQL does not support user defined types. Use base types instead and add column constraints as needed.
- The CASE keyword in Aurora MySQL is not only a conditional expression as in SQL Server. Depending on the context where it appears, CASE can also be used for flow control similar to IF <condition> BEGIN <Statement block> END ELSE BEGIN <statement block> END.
- In Aurora MySQL, IF blocks must be terminated with END IF. WHILE loops must be terminated with END WHILE. The same rule applies to REPEAT - END REPEAT and LOOP - END LOOP.
- Cursors cannot be deallocated in Aurora MySQL. Closing them provides the same behavior.
- Aurora MySQL syntax for opening a transaction is START TRANSACTION as opposed to BEGIN TRANSACTION. COMMIT and ROLLBACK are used without the TRANSACTION keyword.
- The default isolation level in Aurora MySQL is REPEATABLE READ as opposed to SQL Server's READ COMMITTED. By default, it also uses consistent reads similar to SQL Server's READ COMMITTED SNAPSHOT.
- Aurora MySQL supports Boolean expressions in SELECT lists using the "=" operator. In SQL Server, "=" operators in select lists are used to assign aliases. SELECT Col1 = 1 FROM T in Aurora MySQL returns a column with the alias Col1 = 1, and the value 1 for the rows where Col1 = 1, and 0 for the rows where Col1 <> 1 OR Col1 IS NULL.
- Aurora MySQL does not use special data types for UNICODE data. All string types may use any character set and any relevant collation including multiple types of character sets not supported by SQL Server such as UTF-8, UTF-32, and others. A VARCHAR column can be of a UTF-8 character set, and have a latin1_CI collation for example. Similarly, there is no "N" prefix for string literals.
- Collations can be defined at the server, database, and column level similar to SQL Server. They can also be defined at the table level.
- SQL Server's DELETE <Table Name> syntax, which allows omitting the FROM keyword, is invalid in Aurora MySQL. Add the FROM keyword to all delete statements.
- UPDATE expressions in Aurora MySQL are evaluated in order from left to right. This behavior is different from SQL Server and the ANSI standard which require an"'all at once" evaluation. For example, in the statement UPDATE Table SET Col1 = Col1 + 1, Col2 = Col1, Col2 is set to the new value of Col1. The end result is Col1 = Col2.
- Aurora MySQL allows multiple rows with NULL for a UNIQUE constraint; SQL Server allows only one. Aurora MySQL follows the behavior specified in the ANSI standard.
- Although Aurora MySQL supports the syntax for CHECK constraints, they are parsed, but ignored.
- Aurora MySQL AUTO_INCREMENT column property is similar to IDENTITY in SQL Server. However, there is major difference in the way sequences are maintained. While SQL Server caches a set of values in memory, the last allocation is recorded on disk. When the service restarts, some values may be lost, but the sequence continues from where it left off. In Aurora MySQL, each time the service is restarted, the seed value to the AUTO_INCREMET is reset to one

increment interval larger than the largest existing value. Sequence position is not maintained across service restarts.

- Parameter names in Aurora MySQL do not require a preceding "@". You can declare local variables such as DECLARE MyParam1 INTEGER.
- Parameters that use the @sign do not have to be declared first. You can assign a value directly, which implicitly declares the parameter. For example SET @MyParam = 'A'.
- Local parameter scope is not limited to an execution scope. You can define or set a parameter in one statement, execute it, and then query it in the following batch.
- Error handling in Aurora MySQL is called "condition handling". It uses explicitly created objects, named conditions, and handlers. Instead of THROW and RAISERROR, it uses the SIGNAL and RESIGNAL statements.
- Aurora MySQL does not support the MERGE statement. Use the REPLACE statement and the INSERT... ON DUPLICATE KEY UPDATE statement as alternatives.
- You cannot concatenate strings in Aurora MySQL using the "+" operator. 'A' + 'B' is not a valid expression. Use the CONCAT function instead. For example, CONCAT('A', 'B').
- Aurora MySQL does not support aliasing in the select list using the 'String Alias' = Expression. Aurora MySQL treats it as a logical predicate, returns 0 or FALSE, and will alias the column with the full expression. USE the AS syntax instead. Also note that this syntax has been deprecated as of SQL Server 2008 R2.
- Aurora MySQL does not support using the DEFAULT keyword for INSERT statements. Use explicit NULL instead.
  Also note that this syntax has been deprecated as of SQL Server 2008 R2.
- Aurora MySQL has a large set of string functions that is much more diverse than SQL Server. Some of the more useful string functions are:
  - TRIM is not limited to full trim or spaces. The syntax is TRIM([{BOTH | LEADING | TRAILING} [<remove string>] FROM] <source string>)).
  - LENGTH in MySQL is equivalent to DATALENGTH in T-SQL. CHAR_LENGTH is the equivalent of T-SQL LENGTH.
  - SUBSTRING_INDEX returns a substring from a string before the specified number of occurrences of the delimiter.
  - FIELD returns the index (position) of the first argument in the subsequent arguments.
  - FIND_IN_SET returns the index position of the first argument within the second argument.
  - REGEXP and RLIKE provide support for regular expressions.
  - STRCMP provides string comparison.
  - For more string functions, see https://dev.mysql.com/doc/refman/5.7/en/string-functions.html.
- Aurora MySQL Date and Time functions differ from SQL Server's and can cause confusion during migration. For example:

- DATEADD is supported, but is only used to add dates. Use TIMESTAMPADD, DATE_ADD, or DATE_SUB. There is similar behavior for DATEDIFF.
- Do not use CAST and CONVERT for date formatting styles. In Aurora MySQL, use DATE_FORMAT and TIME_FORMAT.
- If your application uses the ANSI CURRENT_TIMESTAMP syntax, conversion is not required. Use NOW in place of GETDATE.

- Object identifiers are case sensitive by default in Aurora MySQL. If you get an 'Object not found error', verify object name case.
- In Aurora MySQL variables cannot be declared interactively in a script but only within stored routines such as stored procedures, functions, and triggers
- Aurora MySQL is much stricter than SQL Server in terms of statement terminators. Be sure to always use a semicolons at the end of statements.
- The syntax for CREATE PROCEDURE requires parenthesis after the procedure name, similar to SQL Server User Defined Functions. It does not allow the AS keyword before the procedure body.
- Beware of control characters when copying and pasting a script to Aurora MySQL clients. Aurora MySQL is much more sensitive to these than SQL Server, and they result in frustrating syntax errors that are hard to spot.

Glossary

ACID
Atomicity, Consistency, Isolation, Durability

AES
Advanced Encryption Standard

ANSI
American National Standards Institute

API
Application Programming Interface

ARN
Amazon Resource Name

AWS
Amazon Web Services

BLOB
Binary Large Object

CDATA
Character Data

CLI
Command Line Interface

CLOB
Character Large Object

CLR
Common Language Runtime

CPU
Central Processing Unit

CRI
Cascading Referential Integrity

CSV
Comma Separated Values

CTE
Common Table Expression

DB
Database

DBCC
Database Console Commands

DDL
Data Definition Language

DEK
Database Encryption Key

DES
Data Encryption Standard

DML
Data Manipulation Language

DQL
Data Query Language

FCI
Failover Cluster Instances

HADR
High Availability and Disaster Recovery

IAM
Identity and Access Management

IP
Internet Protocol

ISO
International Organization for Standardization

JSON
JavaScript Object Notation

KMS
Key Management Service

NUMA
Non-Uniform Memory Access

OLE
Object Linking and Embedding

OLTP
Online Transaction Processing

PaaS
Platform as a Service

PDF
Portable Document Format

QA
Quality Assurance

RDMS
Relational Database Management System

RDS
Amazon Relational Database Service

REGEXP
Regular Expression

SCT
Schema Conversion Tool

SHA
Secure Hash Algorithm

SLA
Service Level Agreement

SMB
Server Message Block

SQL
Structured Query Language

SQL/PSM
SQL/Persistent Stored Modules

SSD
Solid State Disk

SSH
Secure Shell

T-SQL
Transact-SQL

TDE
Transparent Data Encryption

UDF
User Defined Function

UDT
User Defined Type

UTC
Universal Time Coordinated

WMI
Windows Management Instumentation

WQL
Windows Management Instrumentation Query Language

WSFC
Windows Server Failover Clustering

XML
Extensible Markup Language