Microsoft SQL Server 2019 to Amazon Aurora MySQL Migration Playbook

SQL Server to Aurora MySQL Migration Playbook

SQL Server to Aurora MySQL Migration Playbook: Microsoft SQL Server 2019 to Amazon Aurora MySQL Migration Playbook

Table of Contents

Overview

The first section of this document provides an overview of AWS Schema Conversion Tool (AWS SCT) and AWS Database Migration Service (AWS DMS) tools for automating the migration of schema, objects and data. The remainder of the document contains individual sections for the source database features and their Aurora 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 AWS SCT, or to explore a variety of topics where you expect to have some incompatibility issues. When you use 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 AWS 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 the source database and Aurora. It also includes a graphical compatibility indicator and links to the actual sections in the playbook.

The Migration Quick Tips section provides a list of tips for administrators or developers who have little experience with Aurora (PostgreSQL or MySQL). It briefly highlights key differences between the source database and Aurora that they are likely to encounter.

Note that not all of the source database features are fully compatible with Aurora or have simple workarounds. From a migration perspective, this document doesn't yet cover all source database features and capabilities.

This database migration playbook covers the following topics:

- Performance Tuning
- Storage
- Security
- SQL Server 2018 Deprecated Features List
- Migration Quick Tips

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

Automation level icon	Description
	**Very high compatibility**. None or minimal low-risk and low-effort rewrites needed.
	**High compatibility**. Some low-risk rewrites needed, easy workarounds exist for incompatible features.
	**Medium compatibility**. More involved low-medium risk rewrites needed, some redesign may be needed for incompatible features.
	**Low compatibility**. Medium to high risk rewrites needed, some incompatible features

Automation level icon	Description
	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 architect ural solution to work around incompatibilities.

AWS SCT and AWS DMS Automation Level Legend

Automation level icon	Description
	**Full automation**. AWS SCT performs fully automatic conversion, no manual conversion needed.
	**High automation**. Minor, simple manual conversions may be needed.
	**Medium automation**. Low-medium complexit y manual conversions may be needed.
	**Low automation**. Medium-high complexity manual conversions may be needed.

Automation level icon	Description
	**Very low automation**. High risk or complex manual conversions may be needed.
	No automation. Not currently supported by AWS SCT, manual conversion is required for this feature.

Migration Tools and Services

Topics

AWS Schema Conversion Tool

The AWS Schema Conversion Tool (AWS SCT) is a Java utility that connects to source and target databases, scans the source database schema objects (tables, views, indexes, procedures, and so on), 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.

We recommend 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 couldn't be migrated automatically. For more information, see the AWS Schema Conversion Tool User Guide.

> **Note**
>
> This walkthrough uses the AWS Database Migration Service Sample Database. You can download it from GitHub.

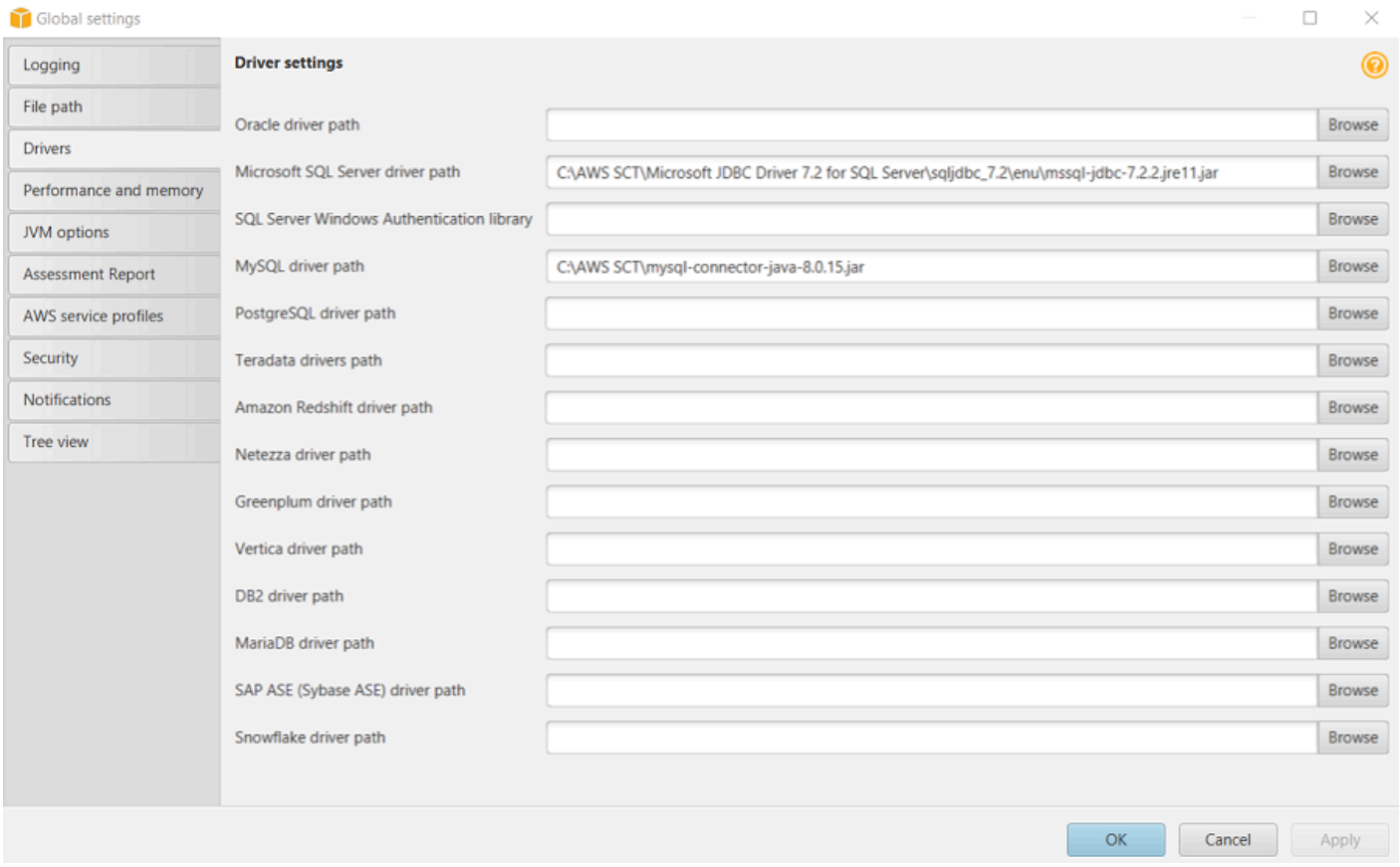
Download the Software and Drivers

Download and install AWS SCT. For more information, see Installing, verifying, and updating in the AWS Schema Conversion Tool User Guide.

Download the Microsoft SQL Server and MySQL drivers. For more information, see Installing the required database drivers in the AWS Schema Conversion Tool User Guide.

Configure AWS SCT

1. Start AWS Schema Conversion Tool (AWS SCT).
2. Choose **Settings** and then choose **Global settings**.
3. On the left navigation bar, choose **Drivers**.
4. Enter the paths for the SQL Server and MySQL drivers downloaded in the first step.



5. Choose **Apply** and then **OK**.

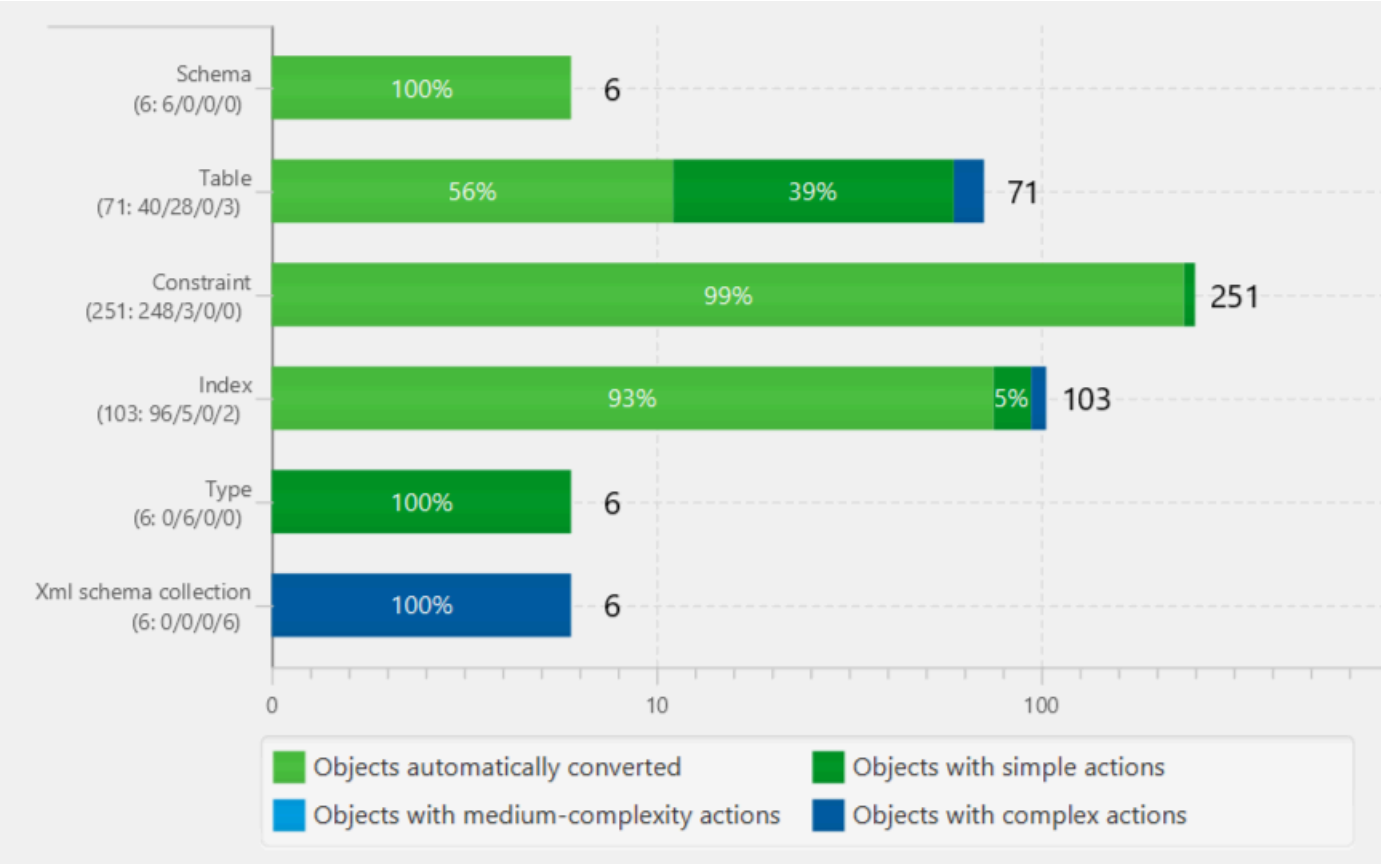
Create a New Migration Project

1. In AWS SCT, choose **File**, and then choose **New project wizard**. Alternatively, use the keyboard shortcut **Ctrl+W**.
2. Enter a project name and select a location for the project files. For **Source engine**, choose **Microsoft SQL Server**, and then choose **Next**.
3. Enter connection details for the source SQL Server database and choose **Test connection** to verify. Choose **Next**.
4. Select the schema or database to migrate and choose **Next**.

The progress bar displays the objects that AWS SCT analyzes. When AWS SCT completes the analysis, the application displays the database migration assessment report. 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, choose **Save to PDF** at the top right and open the PDF document.

Target platform	Auto or minimal changes			Complex actions			
				Storage objects		Code objects	
	Storage objects	Code objects	Conversion actions	Objects count	Conversion actions	Objects count	Conversion actions
Amazon RDS for MySQL	432 (98%)	24 (46%)	204	11 (2%)	11	28 (54%)	117
Amazon Aurora (MySQL compatible)	432 (98%)	24 (46%)	204	11 (2%)	11	28 (54%)	117
Amazon RDS for PostgreSQL	431 (97%)	26 (50%)	471	12 (3%)	15	26 (50%)	117
Amazon Aurora (PostgreSQL compatible)	431 (97%)	26 (50%)	471	12 (3%)	15	26 (50%)	117
Amazon RDS for MariaDB	430 (97%)	24 (46%)	282	13 (3%)	11	28 (54%)	117
Amazon Redshift	423 (95%)	26 (50%)	389	20 (5%)	16	26 (50%)	31
Amazon Glue	0 (0%)	10 (100%)	0	0 (0%)	0	0 (0%)	0
Babelfish for Aurora PostgreSQL	396 (89%)	22 (42%)	30	47 (11%)	56	30 (58%)	137

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



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

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

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.

Choose the schema, open the context (right-click) menu, and then choose **Create report** to create a report tailored for the target database type. You can view this report in AWS SCT.

The progress bar updates while the report is generated.

AWS SCT displays the executive summary page of the database migration assessment report.

Choose **Action items**. 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.

Choose the database name, open the context (right-click) menu, and choose **Convert schema**. Make sure that you uncheck the `sys` and `information_schema` system schemas. This step doesn't make any changes to the target database.

On the right pane, AWS SCT displays the new virtual schema 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.

Choose the database on the right pane, open the context (right-click) menu, and choose either **Apply to database** to automatically run the conversion script against the target database, or choose **Save as SQL** to save to an SQL file.

We recommend saving to an SQL file because you can verify and QA the converted code. Also, you can make the adjustments needed for objects that couldn't be automatically converted.

For more information, see the AWS Schema Conversion Tool User Guide.

AWS SCT Action Code Index

The following table shows the icons we use to describe the automation levels of AWS Schema Conversion Tool (AWS SCT) and AWS Database Migration Service (AWS DMS).

Automation level icon	Description
	**Full automation** — AWS 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.

Automation level icon	Description
	**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 AWS SCT, manual conversion is required for this feature.

The following sections list the AWS 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 Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) support the entry level American National Standards Institute (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, `UNIQUE`, and foreign keys. Some changes may be required for computed columns and global temporary tables.

For more information, see Creating Tables.

Action code	Action message
659	If you use recursion, make sure that table variables in your source database and temporary tables in your target database have the same scope.
679	AWS SCT replaced computed columns with triggers.
680	MySQL doesn't support global temporary tables.

Constraints

AWS Schema Conversion Tool (AWS SCT automatically converts most constraints because SQL Server and Amazon Aurora MySQL-Compatible Edition (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. AWS SCT replaces check constraints with triggers, and some default expressions for `DateTime` columns aren't supported for automatic conversion. AWS SCT can't automatically convert complex expressions for other default values.

For more information, see Constraints.

Action code	Action message
676	MySQL doesn't support the `SET DEFAULT` referential constraint action.
677	MySQL doesn't support functions or expressions as a default value for `BLOB` and `TEXT` columns.

Action code	Action message
678	MySQL doesn't support check constraints.
825	AWS SCT removed the default value of the ë column.
826	AWS SCT can't convert the default value of the ë variable.
827	AWS SCT can't convert default values.

Data Types

Data type syntax and rules are very similar between SQL Server and Aurora MySQL. AWS SCT automatically converts most of data type syntax and rules. Note that date and time handling paradigms are different for SQL Server and Aurora MySQL and require manual verification or conversion. Also note that because of differences in data type behavior between SQL Server and Aurora MySQL, manual verification and strict testing are highly recommended.

For more information, see Data Types.

Action code	Action message
601	MySQL doesn't support including BL0B and TEXT columns in foreign keys.
706	AWS SCT replaced the unsupported %s data type.
707	AWS SCT can't convert the usage of a variable of the unsupported %s data type.

Action code	Action message
708	AWS SCT can't convert the usage of the unsupported %s data type.
775	Converted code might lose accuracy compared to the source code.
844	AWS SCT expanded fractional seconds support for `TIME`, `DATETIME2` , and `DATETIMEOFFSET` values with up to 6 digits of precision.
919	MySQL doesn't support the `DECIMAL` data type with scale greater than 30.

Collations

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

For more information, see Collations.

Action code	Action message
646	MySQL doesn't support the `COLLATE` clause.

Window Functions

Aurora MySQL version 5.7 doesn't support window functions. AWS SCT can't automatically convert window functions.

For workarounds using traditional SQL syntax, see Window Functions.

Action code	Action message
647	MySQL doesn't support the analytic form of the %s function.
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 ssupport `STDEV` functions with the `DISTINCT` clause.
755	MySQL doesn't support `STDEVP` functions with the `DISTINCT` clause.
756	MySQL doesn't support `VAR` functions with the `DISTINCT` clause.
757	MySQL doesn't support `VARP` functions with the `DISTINCT` clause.

PIVOT and UNPIVOT

Aurora MySQL version 5.7 doesn't support the `PIVOT` and `UNPIVOT` syntax. AWS SCT can't automatically convert the PIVOT and UNPIVOT clauses.

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

Action code	Action message
905	MySQL doesn't support `PIVOT` clauses for `SELECT` statements.
906	MySQL doesn't support `UNPIVOT` clauses for `SELECT` statements.

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` can't be automatically converted and require manual conversion.

For more information, see SQL Server TOP and FETCH and MySQL LIMIT.

Action code	Action message
604	MySQL doesn't support the `PERCENT` argument in `TOP` clauses. AWS SCT skips this argument in the converted code.

Action code	Action message
605	MySQL doesn't support the `WITH TIES` argument in `TOP` clauses. AWS SCT skips this argument in the converted code.
608	MySQL doesn't support the `PERCENT` argument in `TOP` clauses of `INSERT` statements.
612	MySQL doesn't support the `PERCENT` argument in `TOP` clauses of `UPDATE` statements.
621	MySQL doesn't support the `PERCENT` argument in TOP clauses. AWS SCT skips this argument in the converted code.
830	MySQL doesn't support `LIMIT` clauses with `IN`, `ALL`, `ANY`, or `SOME` subqueries.

Common Table Expressions

Aurora MySQL version 5.7 doesn't support common table expressions. AWS SCT can't automatically convert common table expressions.

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

Action code	Action message
611	MySQL doesn't support `WITH` queries in `UPDATE` statements.

Action code	Action message
619	MySQL doesn't support query definitions for common table expressions.
839	MySQL doesn't support query definitions for common table expressions.
840	AWS SCT can't convert updated common table expressions.

Cursors

AWS SCT automatically converts the most commonly used cursor operations. 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 aren't supported by Aurora MySQL, require manual conversions.

For more information, see Cursors.

Action code	Action message
618	MySQL doesn't support `CURRENT OF` clauses for DML queries that are in the body of a cursor loop.
624	MySQL doesn't support `CURRENT OF` clauses for DML queries that are in the body of a cursor loop.
625	MySQL doesn't support the `CURSOR` data type as a procedure argument.
637	MySQL doesn't support global cursors.

Action code	Action message
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.
692	MySQL doesn't support cursor variables.
700	AWS SCT can't convert the KEYSET option because MySQL doesn't support changing the membership and order of rows for cursors.
701	AWS SCT doesn't convert the FAST_FORWARD option because this is a default option for cursors in MySQL.
702	AWS SCT doesn't convert the READ_ONLY option because this is a default option for cursors in MySQL.
703	MySQL doesn't support the SCROLL_LOCKS option.

Action code	Action message
704	MySQL doesn't support the `OPTIMISTIC` option for cursors.
705	MySQL doesn't support the `TYPE_WARNING` option for cursors.
842	MySQL doesn't support the %s option in cursors.

Flow Control

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

For more information, see Flow Control.

Action code	Action message
628	MySQL doesn't support `GOTO` statements.
691	MySQL doesn't support the `WAITFOR TIME` feature.

Transaction Isolation

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

SCT automatically converts all these transaction isolation levels. 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 aren't supported by Aurora MySQL.

For more information, see Transactions.

Action code	Action message
629	MySQL doesn't support named transactions.
630	MySQL doesn't support `WITH MARK` options.
631	MySQL doesn't support distributed transacti ons.
632	MySQL doesn't support rolling back named transactions
633	MySQL doesn't support the `DELAYED_D URABILITY` option.
916	MySQL doesn't support the `SNAPSHOT` transaction isolation level.

Stored Procedures

Aurora MySQL stored procedures provide very similar functionality to SQL Server stored procedures. AWS SCT automatically converts SQL Server stored procedures. Manual conversion is required for procedures that use `RETURN` values and some less common `EXECUTE` options such as `RECOMPILE` and `RESULTS SETS`.

For more information, see Stored Procedures.

Action code	Action message
640	MySQL doesn't support `EXECUTE` statements with the `WITH RECOMPILE` option.
641	MySQL doesn't support `EXECUTE` statements with the `RESULT SETS UNDEFINED` option.
642	MySQL doesn't support `EXECUTE` statements with the `RESULT SETS NONE` option.
643	MySQL doesn't support `EXECUTE` statement s with the `RESULT SETS DEFINITION` option.
689	MySQL doesn't support `RETURN` statement s that are used to return values from a procedure.
695	MySQL doesn't support the call 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 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 required because Aurora MySQL triggers are ran once for each row, not once for each statement such as triggers in SQL Server.

For more information, see Triggers.

Action code	Action message
686	MySQL doesn't support triggers with the FOR STATEMENT clause.

GROUP BY

AWS SCT automatically converts the GROUP BY queries, except for CUBE and GROUPING SETS. You can create workarounds for these queries, but they require manual code changes.

For more information, see GROUP BY.

Action code	Action message
654	MySQL doesn't support the GROUP BY CUBE option.
655	MySQL doesn't support GROUP BY GROUPING SETS clauses.

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 aren't part of the primary key and the table-independent SEQUENCE objects.

For more information, see Identity and Sequences.

Action code	Action message
696	MySQL doesn't support identity columns with seed and increment.
697	MySQL doesn't support identity columns outside the primary key.
732	MySQL doesn't support identity columns in compound primary keys.
815	MySQL doesn't support sequences.
841	MySQL doesn't support numeric (x, 0) or decimal (x, 0) data types in columns with the `AUTO_INCREMENT` option. AWS SCT replaced this data type with a compatible data type.
920	MySQL doesn't support identity columns of the `DECIMAL` or `NUMERIC` data type with precision greater than 19.

Error Handling

The error handling paradigms in Aurora MySQL and SQL Server are significantly different; the former uses condition and handler objects. AWS SCT migrates the basic error handling constructs automatically. Due to the paradigm differences, we highly recommend that you perform strict inspection and validation of the migrated code. Manual conversions are required for `THROW` with variables and for built-in messages in SQL Server.

For more information, see Error Handling.

Action code	Action message
729	AWS SCT can't convert THROW operators with variables.
730	AWS SCT truncated the error code.
733	MySQL doesn't support PRINT procedures.
814	AWS SCT can't convert the RAISERROR operator with messages from the sys.messages view.
837	MySQL uses a different approach to handle errors compared to the source code.

Date and Time Functions

AWS SCT automatically converts the most commonly used date and time functions 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 information, see Date and Time Functions.

Action code	Action message
759	MySQL doesn't support DATEADD functions with the nanosecond date part.
760	MySQL doesn't support DATEDIFF functions with the week date part.

Action code	Action message
761	MySQL doesn't support DATEDIFF functions with the millisecond date part.
762	MySQL doesn't support DATEDIFF functions with the nanosecond date part.
763	MySQL doesn't support DATENAME functions with the millisecond date part.
764	MySQL doesn't support DATENAME functions with the nanosecond date part.
765	MySQL doesn't support DATENAME functions with the TZoffset date part.
767	MySQL doesn't support DATEPART functions with the nanosecond date part.
768	MySQL doesn't support DATEPART functions with the TZoffset date part.
773	AWS SCT can't convert arithmetic operations with 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 information, see User-Defined Functions.

Action code	Action message
777	AWS SCT can't emulate a table-valued function because the column from the current query is used as a function parameter.
822	MySQL doesn't support table-valued functions in views.

User-Defined Types

Aurora MySQL 5.7 doesn't support-user defined types and 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 information, see User-Defined Types.

Action code	Action message
690	MySQL doesn't support table types.

Synonyms

Aurora MySQL version 5.7 doesn't support synonyms. AWS SCT can't automatically convert synonyms.

For more information, see Synonyms.

Action code	Action message
792	MySQL doesn't support 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 subqueries, require manual conversion.

For more information, see JSON and XML.

Action code	Action message
817	AWS SCT can't convert `FOR XML` clauses with `EXPLICIT` mode specified.
818	AWS SCT can't convert correlated subqueries with `FOR XML` clauses.
843	AWS SCT can't convert `FOR XML` statements in functions.

Table Joins

AWS SCT automatically converts the most commonly used join types. These types include `INNER`, `OUTER`, and `CROSS` joins. `APPLY` joins, also known as `LATERAL` joins, aren't supported by Aurora MySQL and require manual conversion.

For more information, see Table JOIN.

Action code	Action message
831	MySQL doesn't support the CROSS APPLY and OUTER APPLY operators where the subquery references to the column of attachable table.

MERGE

Aurora MySQL version 5.7 doesn't support the MERGE statement. AWS SCT can't automatically convert MERGE statements. Manual conversion is straightforward in most cases.

For more information and potential workarounds, see MERGE.

Action code	Action message
832	MySQL doesn't support MERGE statements.

Query Hints

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. We recommend that you remove all hints before the start of migration testin. Then, selectively apply hints as a last resort if other means such as schema, index, and query optimizations have failed. Plan guides aren't supported by Aurora MySQL.

For more information, see Query Hints and Plan Guides.

Action code	Action message
610	MySQL doesn't support hints in `INSERT` statements. AWS SCT skips `WITH(Table_Hint_Limited)` options in the converted code.
617	MySQL doesn't support hints in `UPDATE` statements. AWS SCT skips `WITH(Table_Hint_Limited)` options in the converted code.
623	MySQL doesn't support hints in `DELETE` statements. AWS SCT skips `WITH(Table_Hint_Limited)` options in the converted code.
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. AWS SCT can't automatically convert full-text indexes.

For more information, see Full-Text Search.

Action code	Action message
687	MySQL doesn't support the `CONTAINS` predicate.

Action code	Action message
688	MySQL doesn't support the FREETEXT predicate.

Indexes

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

For more information, see Indexes.

Action code	Action message
602	MySQL has reached the limit of the internal InnoDB maximum key length.
681	MySQL doesn't support clustered indexes.
682	MySQL doesn't support the INCLUDE clause in indexes.
683	MySQL doesn't support the WHERE clause in indexes.
684	MySQL doesn't support the WITH clause in indexes.

Partitioning

Because Aurora MySQL stores each table in its own file, and because file management is performed by AWS and can't be modified, some of the physical aspects of partitioning in SQL Server don't 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 doesn't automatically convert table and index partitions. These items require manual conversion.

For more information, see Storage.

Action code	Action message
907	AWS SCT can't convert 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 Relational Database Service (Amazon RDS) provides guaranteed continuous backup with point-in-time restore up to 35 days. Therefore, AWS SCT doesn't automatically convert backups.

For more information, see Backup and Restore.

Action code	Action message
903	MySQL doesn't support functionality similar to SQL Server Backup.

SQL Server Database Mail

Aurora MySQL doesn't provide native support for sending mail from the database.

For more information and potential workarounds, see Database Mail.

Action code	Action message
900	MySQL doesn't support functionality similar to SQL Server Database Mail.

SQL Server Agent

Aurora MySQL doesn't provide functionality similar to SQL Server Agent as an external, cross-instance scheduler. However, Aurora MySQL provides 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't automatically convert Agent jobs and alerts.

For more information, see SQL Server Agent and MySQL Agent.

Action code	Action message
902	MySQL doesn't support functionality similar to SQL Server Agent.

Linked Servers

Aurora MySQL doesn't support remote data access from the database. Connectivity between schemas is trivial, but connectivity to other instances require a custom solution. AWS SCT can't automatically convert commands on linked servers.

For more information, see Linked Servers.

Action code	Action message
645	MySQL doesn't support running pass-through commands on linked servers.

Views

MySQL views are similar to views in SQL Server. However, there are slight differences between the two, mostly around indexing and triggers on views, and also in the query definition.

For more information, see Views.

Action code	Action message
779	AWS SCT can't convert `SELECT` statements that contain a subquery in the `FROM` clause.

AWS Database Migration Service

The AWS Database Migration Service (AWS 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. You can also use AWS DMS 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. The AWS Database Migration Service can also be used for continuous data replication with high availability.

For AWS DMS pricing, see Database Migration Service pricing.

For all supported sources for AWS DMS, see Sources for data migration.

For all supported targets for AWS DMS, see Targets for data migration.

Migration Tasks Performed by AWS DMS

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. You can start your migration 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 modern 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, Amazon S3, and Amazon DynamoDB.

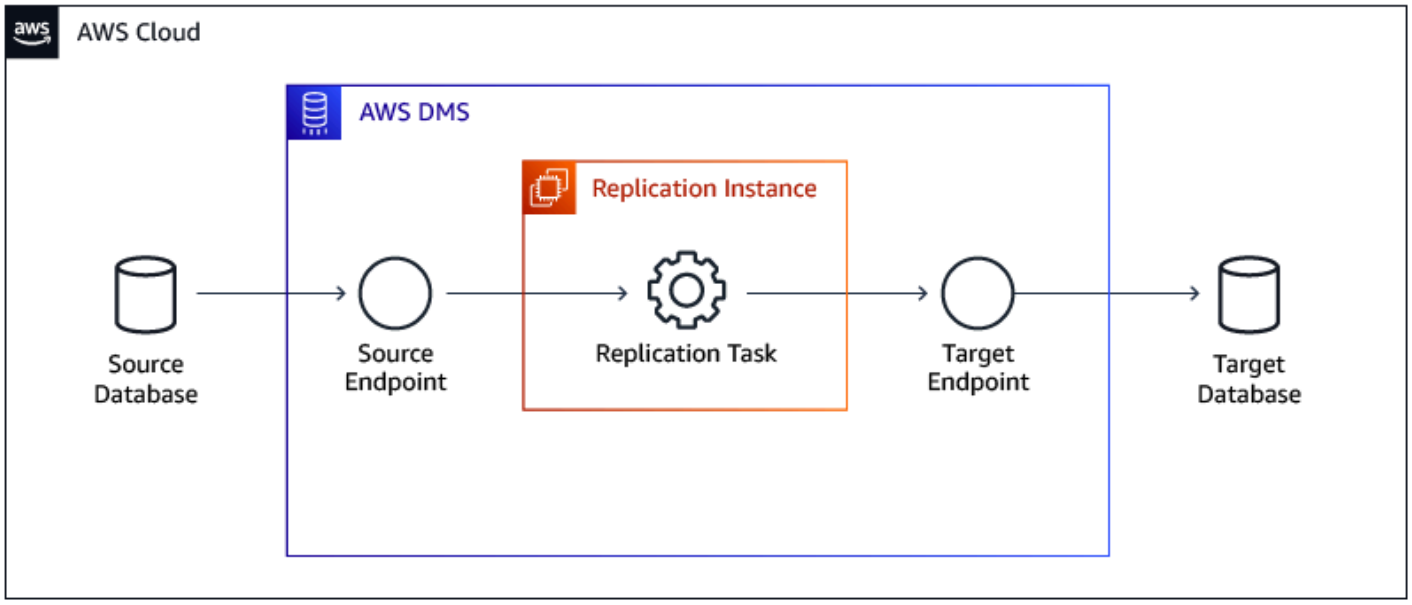
You can migrate from any of the supported data sources to any of the supported data targets. AWS DMS supports fully heterogeneous data migrations between the supported engines.

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 more information about AWS DMS, see What is Database Migration Service? and Best practices for Database Migration Service.

Amazon RDS on Outposts

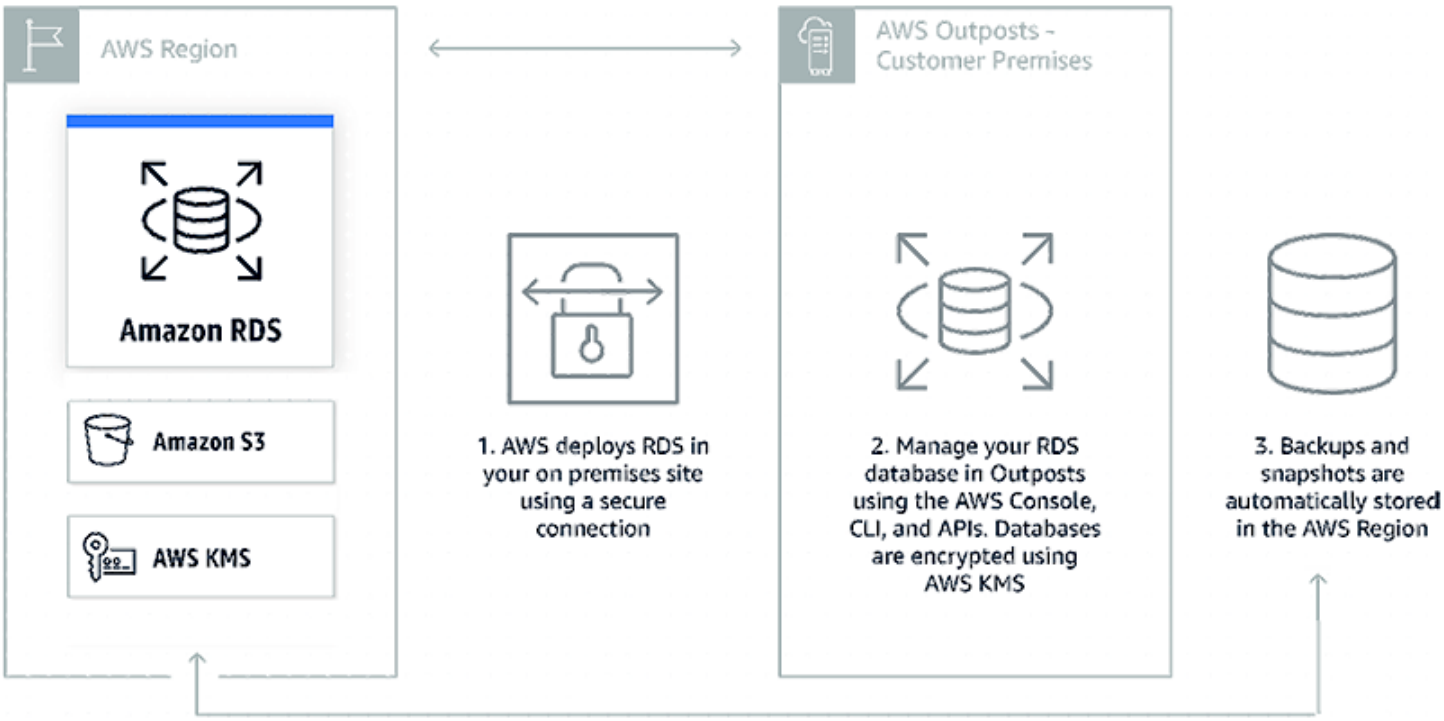
> **Note**
>
> This topic is related to Amazon Relational Database Service (Amazon RDS) and isn't supported with Amazon Aurora.

Amazon RDS on Outposts is a fully managed service that offers the same AWS infrastructure, AWS services, APIs, and tools to virtually any data center, co-location space, or on-premises facility for a truly consistent hybrid experience. Amazon RDS on Outposts is ideal for workloads that require low latency access to on-premises systems, local data processing, data residency, and migration of applications with local system inter-dependencies.

When you deploy Amazon RDS on Outposts, you can run Amazon RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications. Amazon RDS on Outposts also enables automatic backup to an AWS Region. You can manage Amazon RDS databases both in the cloud and on premises using the same AWS Management Console, APIs, and CLI. Amazon RDS on Outposts supports Microsoft SQL Server, MySQL, and PostgreSQL database engines, with support for additional database engines coming soon.

How It Works

Amazon RDS on Outposts lets you run Amazon RDS in your on-premises or co-location site. You can deploy and scale an Amazon RDS database instance in Outposts just as you do in the cloud, using the AWS console, APIs, or CLI. Amazon RDS databases in Outposts are encrypted at rest using AWS KMS keys. Amazon RDS automatically stores all automatic backups and manual snapshots in the AWS Region.



This option is helpful when you need to run Amazon RDS on premises for low latency workloads that need to be run in close proximity to your on-premises data and applications.

For more information, see AWS Outposts Family, Amazon RDS on Outposts, and Create Amazon RDS DB Instances on Outposts.

Amazon RDS Proxy

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon Relational Database Service (RDS) that makes applications more scalable, more resilient to database failures, and more secure.

Many applications, including those built on modern server-less architectures, can have many open connections to the database server, and may open and close database connections at a high rate,

exhausting database memory and compute resources. Amazon RDS Proxy allows applications to pool and share connections established with the database, improving database efficiency and application scalability. With Amazon RDS Proxy, fail-over times for Aurora and Amazon RDS databases are reduced by up to 66%. You can manage database credentials, authentication, and access through integration with AWS Secrets Manager and AWS Identity and Access Management (IAM).

You can turn on Amazon RDS Proxy for most applications with no code changes. You don't need to provision or manage any additional infrastructure. Pricing is simple and predictable: you pay for each vCPU of the database instance for which the proxy is enabled. Amazon RDS Proxy is now generally available for Aurora MySQL, Aurora PostgreSQL, Amazon RDS for MySQL, and Amazon RDS for PostgreSQL.

Amazon RDS Proxy Benefits

- **Improved application performance** — Amazon RDS proxy manages a connection pooling which helps with reducing the stress on database compute and memory resources that typically occurs when new connections are established and it is useful to efficiently support a large number and frequency of application connections.
- **Increase application availability** — By automatically connecting to a new database instance while preserving application connections Amazon RDS Proxy can reduce fail-over time by 66%.
- **Manage application security** — Amazon RDS Proxy also enables you to centrally manage database credentials using AWS Secrets Manager.
- **Fully managed** — Amazon RDS Proxy gives you the benefits of a database proxy without requiring additional burden of patching and managing your own proxy server.
- **Fully compatible with your database** — Amazon RDS Proxy is fully compatible with the protocols of supported database engines, so you can deploy Amazon RDS Proxy for your application without making changes to your application code.
- **Available and durable** — Amazon RDS Proxy is highly available and deployed over multiple Availability Zones (AZs) to protect you from infrastructure failure.

How Amazon RDS Proxy Works

For more information, see Amazon RDS Proxy for Scalable Serverless Applications and Amazon RDS Proxy.

Amazon Aurora Serverless v1

Amazon Aurora Serverless version 1 (v1) is an on-demand autoscaling configuration for Amazon Aurora. An Aurora Serverless DB cluster is a DB cluster that scales compute capacity up and down based on your application's needs. This contrasts with Aurora provisioned DB clusters, for which you manually manage capacity. Aurora Serverless v1 provides a relatively simple, cost-effective option for infrequent, intermittent, or unpredictable workloads. It is cost-effective because it automatically starts up, scales compute capacity to match your application's usage, and shuts down when it's not in use.

To learn more about pricing, see Serverless Pricing under MySQL-Compatible Edition or PostgreSQL-Compatible Edition on the Amazon Aurora pricing page.

Aurora Serverless v1 clusters have the same kind of high-capacity, distributed, and highly available storage volume that is used by provisioned DB clusters. The cluster volume for an Aurora Serverless v1 cluster is always encrypted. You can choose the encryption key, but you can't turn off encryption. That means that you can perform the same operations on an Aurora Serverless v1 that you can on encrypted snapshots. For more information, see Aurora Serverless v1 and snapshots.

Aurora Serverless v1 provides the following advantages:

- **Simpler than provisioned** — Aurora Serverless v1 removes much of the complexity of managing DB instances and capacity.
- **Scalable** — Aurora Serverless v1 seamlessly scales compute and memory capacity as needed, with no disruption to client connections.

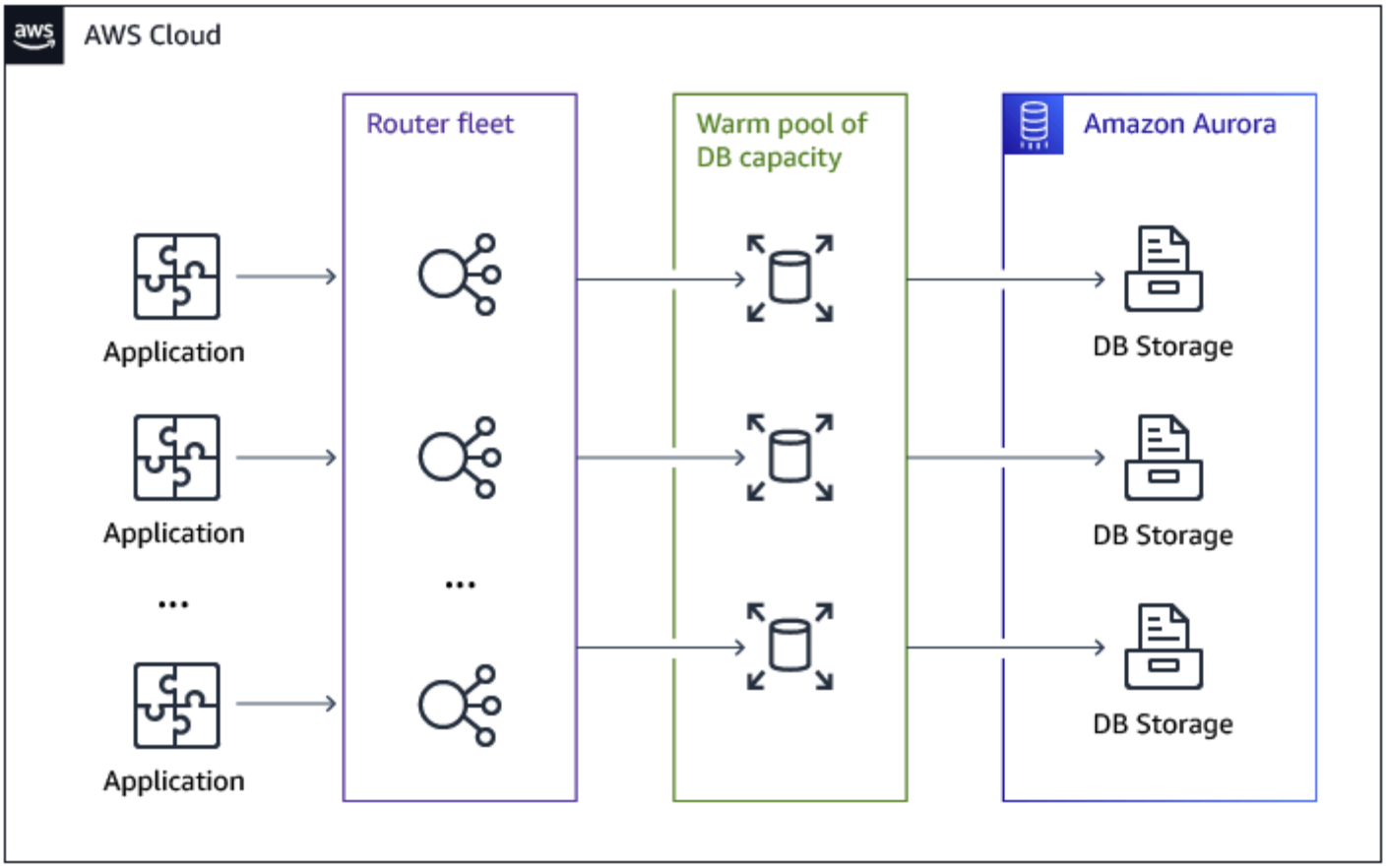
- **Cost-effective** — When you use Aurora Serverless v1, you pay only for the database resources that you consume, on a per-second basis.
- **Highly available storage** — Aurora Serverless v1 uses the same fault-tolerant, distributed storage system with six-way replication as Aurora to protect against data loss.

Aurora Serverless v1 is designed for the following use cases:

- **Infrequently used applications** — You have an application that is only used for a few minutes several times for each day or week, such as a low-volume blog site. With Aurora Serverless v1, you pay for only the database resources that you consume on a per-second basis.
- **New applications** — You're deploying a new application and you're unsure about the instance size you need. By using Aurora Serverless v1, you can create a database endpoint and have the database automatically scale to the capacity requirements of your application.
- **Variable workloads** — You're running a lightly used application, with peaks of 30 minutes to several hours a few times each day, or several times for each year. Examples are applications for human resources, budgeting, and operational reporting applications. With Aurora Serverless v1, you no longer need to provision for peak or average capacity.
- **Unpredictable workloads** — You're running daily workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. With Aurora Serverless v1, your database automatically scales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over.
- **Development and test databases** — Your developers use databases during work hours but don't need them on nights or weekends. With Aurora Serverless v1, your database automatically shuts down when it's not in use.
- **Multi-tenant applications** — With Aurora Serverless v1, you don't have to individually manage database capacity for each application in your fleet. Aurora Serverless v1 manages individual database capacity for you.

This process takes almost no time. Because the storage is shared between nodes Aurora can scale up or down in seconds for most workloads. The service currently has autoscaling thresholds of 1.5 minutes to scale up and 5 minutes to scale down. That means metrics must exceed the limits for 1.5 minutes to trigger a scale up or fall below the limits for 5 minutes to trigger a scale down. The cool-down period between scaling activities is 5 minutes to scale up and 15 minutes to scale down. Before scaling can happen the service has to find a "scaling point" which may take longer than anticipated if you have long-running transactions. Scaling operations are transparent to the

connected clients and applications since existing connections and session state are transferred to the new nodes. The only difference with pausing and resuming is a higher latency for the first connection, typically around 25 seconds. You can find more details in the documentation.



Amazon Aurora Serverless v2

Amazon Aurora Serverless v2 has been architected from the ground up to support serverless DB clusters that are instantly scalable. The Aurora Serverless v2 architecture rests on a lightweight foundation that's engineered to provide the security and isolation needed in multitenant serverless cloud environments. This foundation has very little overhead so it can respond quickly. It's also powerful enough to meet dramatic increases in processing demand.

When you create your Aurora Serverless v2 DB cluster, you define its capacity as a range between minimum and maximum number of Aurora capacity units (ACUs):

- **Minimum Aurora capacity units** — The smallest number of ACUs down to which your Aurora Serverless v2 DB cluster can scale.

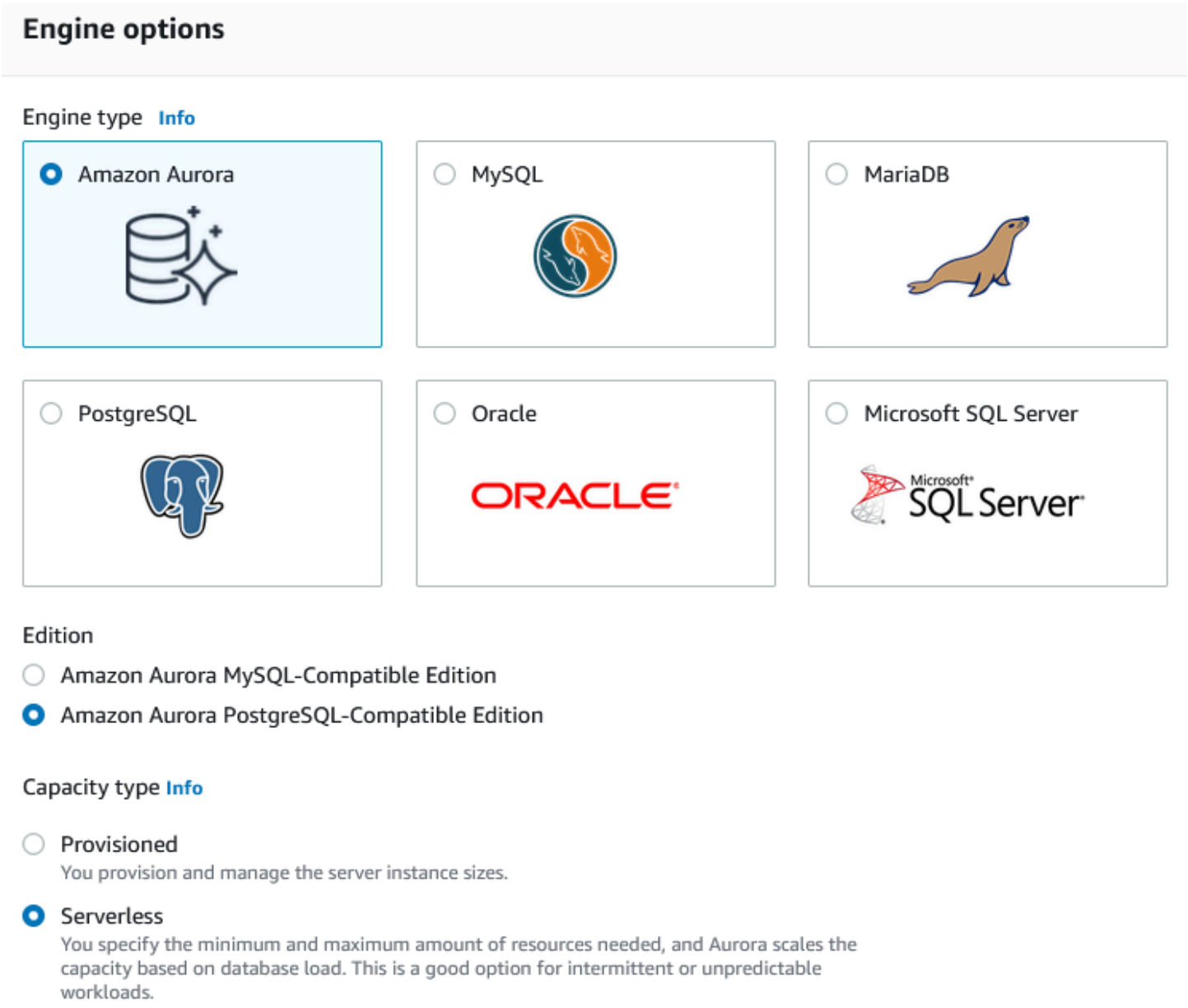
- **Maximum Aurora capacity units** — The largest number of ACUs up to which your Aurora Serverless v2 DB cluster can scale.

Each ACU provides 2 GiB (gibibytes) of memory (RAM) and associated virtual processor (vCPU) with networking. Unlike Aurora Serverless v1, which scales by doubling ACUs each time the DB cluster reaches a threshold, Aurora Serverless v2 can increase ACUs incrementally. When your workload demand begins to reach the current resource capacity, your Aurora Serverless v2 DB cluster scales the number of ACUs. Your cluster scales ACUs in the increments required to provide the best performance for the resources consumed.

How to Provision

Log in to your Management Console, choose **Amazon RDS** , and then choose **Create database**.

On **Engine options**, for **Engine versions**, choose **Show versions that support Serverless v2**.



Choose the capacity settings for your use case.

For more information, see Amazon Aurora Serverless, Aurora Serverless MySQL Generally Available, and Amazon Aurora PostgreSQL Serverless Now Generally Available.

Amazon Aurora Backtrack

We've all been there, you need to make a quick, seemingly simple fix to an important production database. You compose the query, give it a once-over, and let it run. Seconds later you realize that you forgot the `WHERE` clause, dropped the wrong table, or made another serious mistake, and

interrupt the query, but the damage has been done. You take a deep breath, whistle through your teeth, wish that reality came with an Undo option.

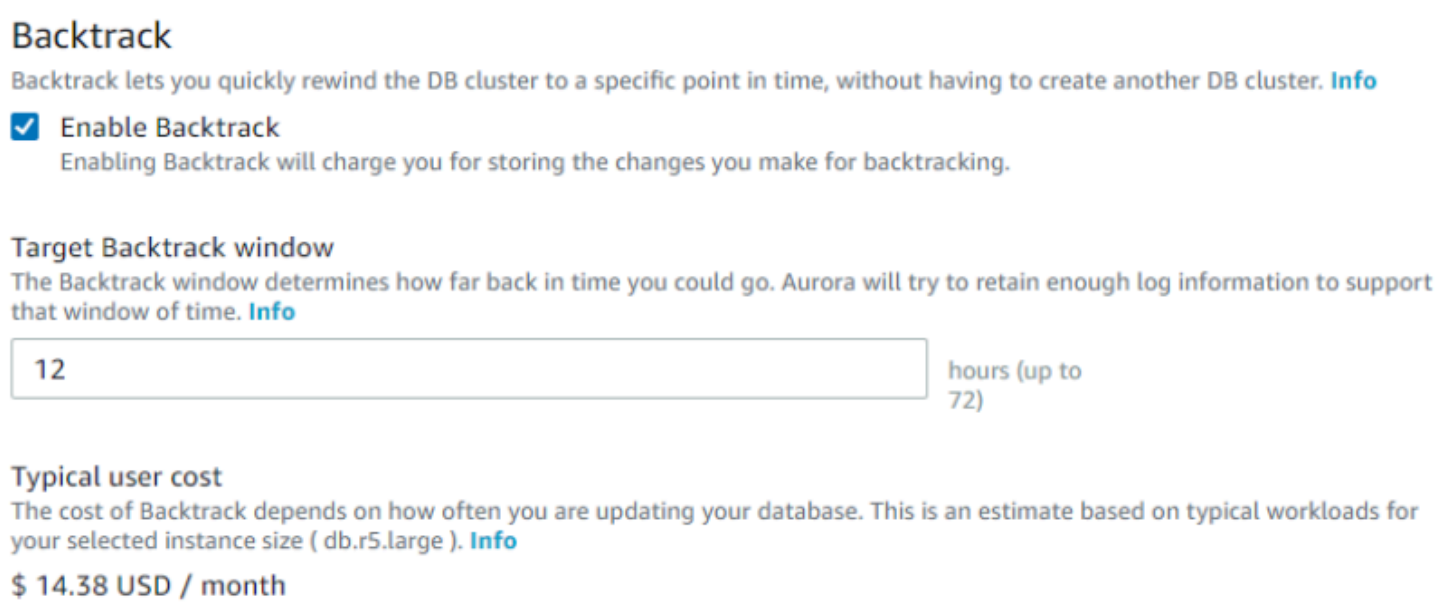
Backtracking rewinds the DB cluster to the time you specify. Backtracking isn't a replacement for backing up your DB cluster so that you can restore it to a point in time. However, backtracking provides the following advantages over traditional backup and restore:

- You can easily undo mistakes. If you mistakenly perform a destructive action, such as a `DELETE` without a `WHERE` clause, you can backtrack the DB cluster to a time before the destructive action with minimal interruption of service.
- You can backtrack a DB cluster quickly. Restoring a DB cluster to a point in time launches a new DB cluster and restores it from backup data or a DB cluster snapshot, which can take hours. Backtracking a DB cluster doesn't require a new DB cluster and rewinds the DB cluster in minutes.
- You can explore earlier data changes. You can repeatedly backtrack a DB cluster back and forth in time to help determine when a particular data change occurred. For example, you can backtrack a DB cluster three hours and then backtrack forward in time one hour. In this case, the backtrack time is two hours before the original time.

Amazon Aurora uses a distributed, log-structured storage system (read Design Considerations for High Throughput Cloud-Native Relational Databases to learn a lot more); each change to your database generates a new log record, identified by a Log Sequence Number (LSN). Enabling the backtrack feature provisions a FIFO buffer in the cluster for storage of LSNs. This allows for quick access and recovery times measured in seconds.

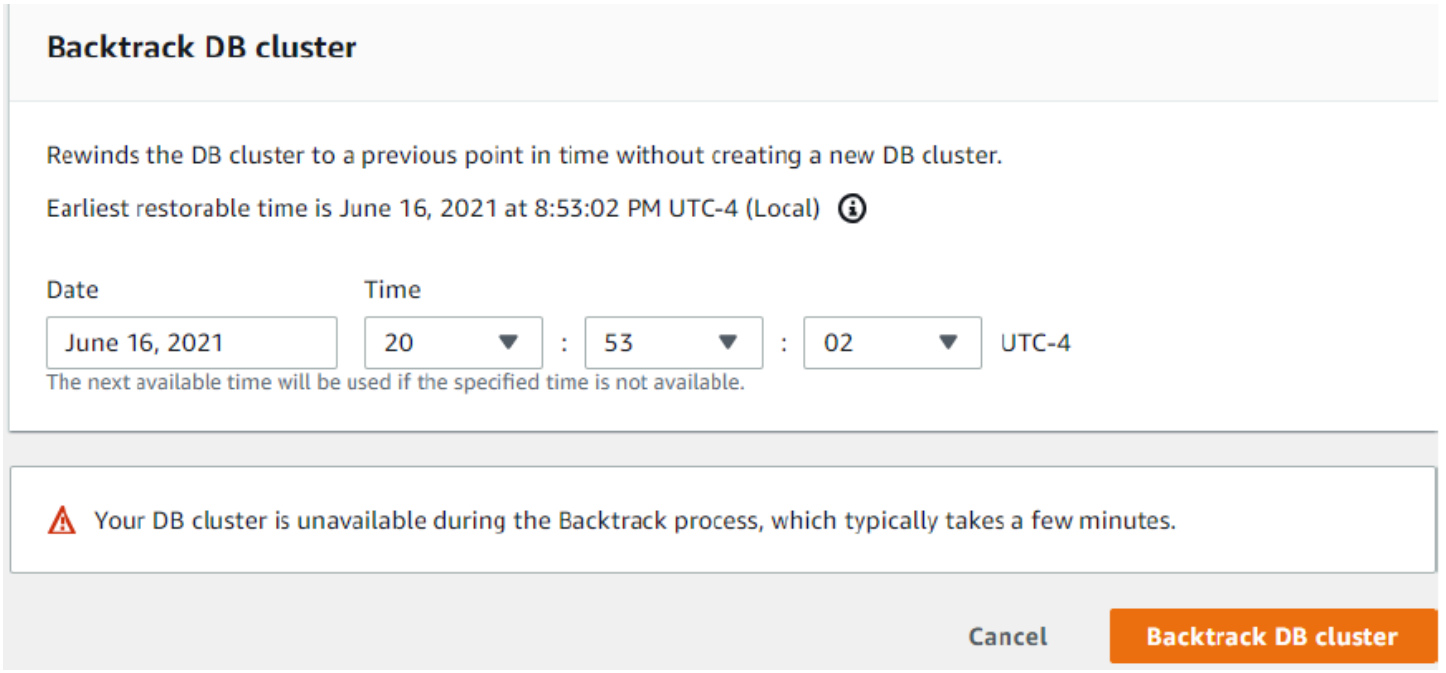
When you create a new Aurora MySQL DB cluster, backtracking is configured when you choose **Enable Backtrack** and specify a **Target Backtrack window** value that is greater than zero in the Backtrack section.

To create a DB cluster, follow the instructions in Creating an Amazon Aurora DB cluster. The following image shows the Backtrack section.



After a production error, you can simply pause your application, open up the Aurora Console, select the cluster, and choose **Backtrack DB cluster**.

Then you select **Backtrack** and choose the point in time just before your epic fail, and choose **Backtrack DB cluster**.



Then you wait for the rewind to take place, unpause your application and proceed as if nothing had happened. When you initiate a backtrack, Aurora will pause the database, close any open connections, drop uncommitted writes, and wait for the backtrack to complete. Then it will resume

normal operation and be able to accept requests. The instance state will be backtracking while the rewind is underway.

Backtrack Window

With backtracking, there is a target backtrack window and an actual backtrack window:

- The target backtrack window is the amount of time you want to be able to backtrack your DB cluster. When you enable backtracking, you specify a target backtrack window. For example, you might specify a target backtrack window of 24 hours if you want to be able to backtrack the DB cluster one day.
- The actual backtrack window is the actual amount of time you can backtrack your DB cluster, which can be smaller than the target backtrack window. The actual backtrack window is based on your workload and the storage available for storing information about database changes, called change records.

As you make updates to your Aurora DB cluster with backtracking enabled, you generate change records. Aurora retains change records for the target backtrack window, and you pay an hourly rate for storing them. Both the target backtrack window and the workload on your DB cluster determine the number of change records you store. The workload is the number of changes you make to your DB cluster in a given amount of time. If your workload is heavy, you store more change records in your backtrack window than you do if your workload is light.

You can think of your target backtrack window as the goal for the maximum amount of time you want to be able to backtrack your DB cluster. In most cases, you can backtrack the maximum amount of time that you specified. However, in some cases, the DB cluster can't store enough change records to backtrack the maximum amount of time, and your actual backtrack window is smaller than your target. Typically, the actual backtrack window is smaller than the target when you have extremely heavy workload on your DB cluster. When your actual backtrack window is smaller than your target, we send you a notification.

When backtracking is turned on for a DB cluster, and you delete a table stored in the DB cluster, Aurora keeps that table in the backtrack change records. It does this so that you can revert back to a time before you deleted the table. If you don't have enough space in your backtrack window to store the table, the table might be removed from the backtrack change records eventually.

Backtracking Limitations

The following limitations apply to backtracking:

- Backtracking an Aurora DB cluster is available in certain AWS Regions and for specific Aurora MySQL versions only. For more information, see Backtracking in Aurora.
- Backtracking is only available for DB clusters that were created with the Backtrack feature enabled. You can enable the Backtrack feature when you create a new DB cluster or restore a snapshot of a DB cluster. For DB clusters that were created with the Backtrack feature enabled, you can create a clone DB cluster with the Backtrack feature enabled. Currently, you can't perform backtracking on DB clusters that were created with the Backtrack feature turned off.
- The limit for a backtrack window is 72 hours.
- Backtracking affects the entire DB cluster. For example, you can't selectively backtrack a single table or a single data update.
- Backtracking isn't supported with binary log (binlog) replication. Cross-Region replication must be turned off before you can configure or use backtracking.
- You can't backtrack a database clone to a time before that database clone was created. However, you can use the original database to backtrack to a time before the clone was created. For more information about database cloning, see Cloning an Aurora DB cluster volume.
- Backtracking causes a brief DB instance disruption. You must stop or pause your applications before starting a backtrack operation to ensure that there are no new read or write requests. During the backtrack operation, Aurora pauses the database, closes any open connections, and drops any uncommitted reads and writes. It then waits for the backtrack operation to complete.
- Backtracking isn't supported for the following AWS Regions:
  - Africa (Cape Town)
  - China (Ningxia)
  - Asia Pacific (Hong Kong)
  - Europe (Milan)
  - Europe (Stockholm)
  - Middle East (Bahrain)
  - South America (São Paulo)
- You can't restore a cross-region snapshot of a backtrack-enabled cluster in an AWS Region that doesn't support backtracking.

- You can't use backtrack with Aurora multi-master clusters.
- If you perform an in-place upgrade for a backtrack-enabled cluster from Aurora MySQL version 1 to version 2, you can't backtrack to a point in time before the upgrade happened.

For more information, see: Amazon Aurora Backtrack — Turn Back Time.

Amazon Aurora Parallel Query

Amazon Aurora parallel query is a feature of the Amazon Aurora database that provides faster analytical queries over your current data, without having to copy the data into a separate system. It can speed up queries by up to two orders of magnitude, while maintaining high throughput for your core transactional workload.

While some databases can parallelize query processing across CPUs in one or a handful of servers, parallel query takes advantage of Aurora unique architecture to push down and parallelize query processing across thousands of CPUs in the Aurora storage layer. By offloading analytical query processing to the Aurora storage layer, parallel query reduces network, CPU, and buffer pool contention with the transactional workload.

Features

Accelerate Your Analytical Queries

In a traditional database, running analytical queries directly on the database means accepting slower query performance and risking a slowdown of your transactional workload, even when running light queries. Queries can run for several minutes to hours, depending on the size of the tables and database server instances. Queries are also slowed down by network latency, since the storage layer may have to transfer entire tables to the database server for processing.

With Amazon Aurora parallel query, query processing is pushed down to the Aurora storage layer. The query gains a large amount of computing power, and it needs to transfer far less data over the network. In the meantime, the Amazon Aurora database instance can continue serving transactions with much less interruption. This way, you can run transactional and analytical workloads alongside each other in the same Aurora database, while maintaining high performance.

Query on Fresh Data

Many analytical workloads require both fresh data and good query performance. For example, operational systems such as network monitoring, cyber-security or fraud detection rely on fresh,

real-time data from a transactional database, and can't wait for it to be extracted to a analytics system.

By running your queries in the same database that you use for transaction processing, without degrading transaction performance, Amazon Aurora parallel query enables smarter operational decisions with no additional software and no changes to your queries.

Benefits of Using Parallel Query

- Improved I/O performance, due to parallelizing physical read requests across multiple storage nodes.
- Reduced network traffic. Amazon Aurora doesn't transmit entire data pages from storage nodes to the head node and then filter out unnecessary rows and columns afterward. Instead, Aurora transmits compact tuples containing only the column values needed for the result set.
- Reduced CPU usage on the head node, due to pushing down function processing, row filtering, and column projection for the WHERE clause.
- Reduced memory pressure on the buffer pool. The pages processed by the parallel query aren't added to the buffer pool. This approach reduces the chance of a data-intensive scan evicting frequently used data from the buffer pool.
- Potentially reduced data duplication in your extract, transform, and load (ETL) pipeline, by making it practical to perform long-running analytic queries on existing data.

Important Notes

- **Table Formats** — The table row format must be `COMPACT`; partitioned tables aren't supported.
- **Data Types** — The `TEXT`, `BLOB`, and `GEOMETRY` data types aren't supported.
- **DDL** — The table can't have any pending fast online DDL operations.
- **Cost** — You can make use of parallel query at no extra charge. However, because it makes direct access to storage, there is a possibility that your IO cost will increase.

For more information, see Amazon Aurora Parallel Query.

ANSI SQL

Topics

Case Sensitivity Differences

Object name case sensitivity is different for SQL Server and Amazon Aurora MySQL-Compatible Edition (Aurora MySQL). SQL Server object names case sensitivity is being determined by the collection. Aurora MySQL names are case sensitive and can be adjusted based on the parameter mentioned following.

In Aurora MySQL, the case sensitivity is determined by the `lower_case_table_names` parameter value. In general, you can use one of the three possible values for this parameter. To avoid issues, we recommend that you use only the two following values for `lower_case_table_names`:

- 0 — names stored as given and comparisons are case-sensitive. You can choose this value for all Amazon Relational Database Service (Amazon RDS) for MySQL versions.
- 1 — names stored in lowercase and comparisons aren't case-sensitive. You can choose this value for Amazon RDS for MySQL version 5.6, version 5.7, and version 8.0.19 and higher 8.0 versions.

In Aurora MySQL version 2.10 and higher 2.x versions, make sure to reboot all reader instances after changing the `lower_case_table_names` setting and rebooting the writer instance. For details, see Rebooting an Aurora MySQL cluster (version 2.10 and higher).

In Aurora MySQL version 3, the value of the `lower_case_table_names` parameter is set permanently at the time when you create the cluster. If you use a nondefault value for this option, set up your Aurora MySQL version 3 custom parameter group before upgrading, and specify the parameter group during the snapshot restore operation that creates the version 3 cluster.

With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. For more information on the methods that you can use, see Major version upgrades.

We recommend that you don't changE the `lower_case_table_names` parameter for existing database instances. Doing so can cause inconsistencies with point-in-time recovery backups and read replica DB instances.

Read replicas should always use the same `lower_case_table_names` parameter value as the source DB instance.

By default, object names are being stored in lowercase for MySQL. In most cases, you'll want to use AWS Database Migration Service transformations to change schema, table, and column names to lowercase.

Examples

For example, to create a table named EMPLOYEES in uppercase in MySQL, you should use the following:

```
CREATE TABLE EMPLOYEES (
    EMP_ID NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL,
    AVG_SALARY NUMERIC NOT NULL);
```

The following command creates a table named employees in lowercase.

```
CREATE TABLE employees (
    EMP_ID NUMERIC PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL,
    AVG_SALARY NUMERIC NOT NULL);
```

MySQL will look for objects names in with the exact case sensitivity as written in the query.

You can turn off table name case sensitivity in MySQL by setting the parameter `lower_case_table_names` to 1. Column, index, stored routine, event names, and column aliases aren't case sensitive on either platform.

For more information, see Identifier Case Sensitivity in the *MySQL documentation*.

Constraints

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Constraints	Unsupported `CHECK`. Indexing requirements for `UNIQUE`.

SQL Server Usage

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

```
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.

You can assign multiple check constraints to a single 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't access other rows as part of the expression. In SQL Server, you can use user-defined functions in constraints to access other rows, tables, or even databases.

Unique Constraints

```
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 such as columns that uniquely identify each row in the relation or 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, in SQL Server, you can use 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

```
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

```
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 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't 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

The following example creates 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)
);
```

The following example creates 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)
);
```

The following example creates 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)
);
```

The following example creates 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 Unique Constraints and Check Constraints and Primary and Foreign Key Constraints in the *SQL Server documentation*.

MySQL Usage

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, you can use the `_rowid` alias 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 can't be configured as `NON CLUSTERED` like SQL Server. For more information, see Indexes.

Applications can reference a primary key using the `PRIMARY` alias. If a table has no primary key, which isn't 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 or nonclustered 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**
>
> MySQL doesn't support foreign key constraints for partitioned tables. For more information, see Storage.

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 nonunique 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, CASCADE, SET NULL, and NO ACTION. The default action is RESTRICT. 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 doesn't 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 don't exist. For example, if a `RESTRICT` action is defined and a child row has several parent rows, Aurora MySQL doesn't 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 turns off 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 don't raise syntax errors. However, they are ignored and aren't 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

- Aurora MySQL doesn't support check constraints. 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 doesn't exist, Aurora MySQL automatically creates one.

Examples

The following example creates a composite primary key.

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

The following example creates 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)
);
```

The following example creates 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.

Feature	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.

Feature	SQL Server	Aurora MySQL	Comments
Indexing of referenced columns	`PRIMARY KEY` or `UNIQUE`	Required	Aurora MySQL doesn't enforce uniqueness of referenced columns.
Cascade recursion	Not allowed, discovered at `CREATE` time	Not allowed, discovered at run time.	

For more information, see CREATE TABLE Statement, How MySQL Deals with Constraints, and FOREIGN KEY Constraints in the *MySQL documentation*.

Creating Tables

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Creating Tables	`IDENTITY` and `AUTO_INCREMENT` . Primary key is always clustered. `CREATE TEMPORARY TABLE` syntax. Unsupported `@table` variables.

SQL Server Usage

Tables in SQL Server are created using the `CREATE TABLE` statement and conform to the ANSI and 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 and ISO standards.

In its 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, 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 run scope where they were created and any sub-scope. Temporary tables are cleaned up by the server when the run 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 run 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 nonstandard `CREATE TABLE` syntax.

Creating a Table Based on an Existing Table or Query

In SQL Server, you can create 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>` doesn't 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 2008 R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

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 you migrate to Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) using AWS Schema Conversion Tool (AWS SCT), neither `ROWVERSION` nor `TIMESTAMP` are supported. AWS SCT raises the following error: `706 – Unsupported data type … of variable/column was replaced. Check the conversion result`.

To maintain this functionality, add customer logic, potentially in the form of a trigger.

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

The following example creates a basic table.

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

The following example creates 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)
);
```

The following example creates 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 CREATE TABLE (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

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 doesn't apply to Aurora MySQL. For more information, see Databases and Schemas.

In its most basic form, and very similar to SQL Server, you can use the `CREATE TABLE` statement in Aurora MySQL to define:

- Table name, containing security schema, and database.
- Column names.
- Column data types.

- Column and table constraints.
- Column default values.
- Primary, unique, and foreign keys.

Aurora MySQL Extensions

Aurora MySQL 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 similar to `SELECT INTO` in SQL Server. You can use this statement to create a new table and populate data in a single step. Unlike SQL Server, you

can combine standard column definitions and additional columns derived from the query in Aurora MySQL. This statement doesn't 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
```

For the preceding examples, the result looks as shown following.

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

Converting TIMESTAMP and ROWVERSION columns

> **Note**
>
> Aurora MySQL has a `TIMESTAMP` data type, which is a temporal type not to be confused with `TIMESTAMP` in SQL Server. For more information, see Data Types.

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

Consider the following 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, @Status 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 for each 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, see Triggers.

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
 Predicate>)]
(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 aren't 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 doesn't 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 `DBCC CHECKIDENT` in SQL Server.

In Aurora MySQL, you can add 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 can 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 FOREIGN KEY Constraint Differences in the *MySQL documentation*.

Examples

The following example creates a basic table.

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

The following example creates 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)
);
```

The following example creates 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

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

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 a new value.
Reseed auto generated value	`DBCC CHECKIDENT`	`ALTER TABLE`	For more informati on, see ALTER TABLE Statement.

Feature	SQL Server	Aurora MySQL	Comments
Index types	`CLUSTERED , NONCLUSTERED`	Implicit — primary keys use clustered indexes.	For more informati on, see Indexes.
Physical storage location	`ON <File Group>`	Not supported	Physical storage is managed by AWS.
Temporary tables	#TempTable	`CREATE TEMPORARY TABLE`	
Global temporary tables	`##GlobalT empTable`	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 Amazon ElastiCache for Redis. For more information, see Amazon ElastiCache for Redis.

For more information, see CREATE TABLE Statement in the *MySQL documentation*.

Common Table Expressions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Common Table Expressions	Rewrite non-recursive CTE to use views and derived tables. Redesign recursive CTE code.

SQL Server Usage

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 ANSI 99 standard. Recursive CTEs can reference themselves and re-run queries until the data set is exhausted, or the maximum number of iterations is exceeded.

Simplified CTE Syntax

```
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;
```

For the preceding example, the result looks as shown following.

```
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 recursive 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;
```

For the preceding example, the result looks as shown following.

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

For more information, see Recursive Queries Using Common Table Expressions in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) 5.7 doesn't support Common Table Expressions (CTE).

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports common table expressions both nonrecursive and recursive. Common table expressions enable use of named temporary result sets implemented by permitting a `WITH` clause preceding `SELECT` statements and certain other statements. As of MySQL 8.0.19, the recursive `SELECT` part of a recursive common table expression supports a `LIMIT` clause. `LIMIT` with `OFFSET` is also supported. For more information, see Recursive Common Table Expressions in the *MySQL documentation*.

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 isn't 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 shown following.

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;
```

For the preceding example, the result looks as shown following.

```
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 turned off 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 isn't 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 `EmpHierarchy` 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, see Stored Procedures and 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;
```

Run 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 WITH (Common Table Expressions) in the *MySQL documentation*.

Data Types

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Data Types	Minor syntax and handling differences. No special `UNICODE` data types.

SQL Server Usage

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 2008 R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

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, AWS SCT automatically converts them to the appropriate Amazon Aurora MySQL-Compatible Edition (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 Data types (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports the following data types:

Category	Data Types
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`
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 isn't set.

For more information, see Out-of-Range and Overflow Handling in the *MySQL documentation*.

Converting from TEXT, NTEXT, and IMAGE SQL Server Deprecated Data Types

The legacy SQL Server types for storing LOB data are deprecated as of SQL Server 2008 R2.

When you convert from these types to Aurora MySQL using the AWS Schema Conversion Tool (AWS SCT, they are converted as shown following:

SQL Server LOB Type	Converted to Aurora MySQL data type	Comments
TEXT	LONGTEXT	Make sure to choose the right collation. For more informati on, see Collations.
NTEXT	LONGTEXT	Make sure to choose the right collation. For more informati on, see Collations.
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 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 than SQL Server 8 KB limit. If the strings or binary data that you need to store don't exceed 32 KB, it may be more efficient to store these as non-LOB types in Aurora MySQL.

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. In SQL Server, literal bit notation uses the numerical digits 0 and 1. Aurora MySQL uses b'<value> or 0b<value> notations. For more information, see Bit-Value Type - BIT and Bit-Value Literals in the *MySQL documentation*.
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 Integer Types (Exact Value) in the *MySQL documentation*.
SMALLINT	SMALLINT	Compatible type. SQL Server supports only signed SMALLINT. Aurora MySQL

SQL Server Data Type	Convert to MySQL Data Type	Comments
		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 Integer Types (Exact Value) in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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 Integer Types (Exact Value) in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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. For more information, see Integer Types (Exact Value) in the *MySQL documentation*.
NUMERIC / DECIMAL	NUMERIC / DECIMAL	Compatible types. DECIMAL and NUMERIC are synonymous.
MONEY / SMALLMONEY	N/A	Aurora MySQL doesn't 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.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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 FLOAT and REAL in Aurora MySQL. In SQL Server, FLOAT(n), where n>24, uses 8 bytes. The Aurora MySQL DOUBLE PRECISION type always uses 8 bytes. Aurora MySQL also supports the nonstandard 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 Floating-Point Types (Approximate Value) in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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 convertin g from CHAR to VARCHAR, exercise caution because VARCHAR behaves differently than CHAR; trailing spaces are trimmed. For more information, see The CHAR and VARCHAR Types in the *MySQL documentation*.
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 The CHAR and VARCHAR Types in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
NCHAR	CHAR	Aurora MySQL doesn't 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 Unicode Character Sets in the *MySQL documentation*.
NVARCHAR	VARCHAR	Aurora MySQL doesn't 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 Unicode Character Sets in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
DATE	DATE	Compatible types. The range for SQL Server 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 doesn't support dates before 1000 AD. For more information, see Date and Time Functions. For more information, see The DATE, DATETIME, and TIMESTAMP Types in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
`TIME`	`TIME`	Compatible types. SQL Server supports explicit fractiona l seconds using the format `TIME(n)` where n is between 0 to 7. Aurora MySQL doesn't allow explicit fractional setting. Aurora MySQL supports up to 6 digits for microseco nd 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 than 24, or to represent a time interval, which can be greater than 24 hours and have negative values. For more information, see The TIME Type in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
SMALLDATETIME	DATETIME / TIMESTAMP	Aurora MySQL doesn't support SMALLDATETIME . Use DATETIME instead. Use SMALLDATETIME for storage space optimization where lower ranges and resolutions are required. For more information, see Date and Time Functions.
DATETIME	DATETIME	In SQL Server, the 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, see Date and Time Functions. For more information about DATETIME, see The DATE, DATETIME, and TIMESTAMP Types in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
DATETIME2	DATETIME	In SQL Server, the 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, see Date and Time Functions. For more information about DATETIME, see The DATE, DATETIME, and TIMESTAMP Types in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
DATETIMEOFFSET	TIMESTAMP	Aurora MySQL doesn't support full time zone awareness and managemen t functions. Use the time_zone system variable in conjunction with TIMESTAMP columns to achieve partial time zone awareness. For more information, see Server Options. In Aurora MySQL, TIMESTAMP isn't the same as in SQL Server. The latter is a synonym for ROW_VERSION . Aurora MySQL TIMESTAMP is equivalent to the DATETIME type with a smaller range. With Aurora MySQL DATETIME, you can use values between 1000-01-0 1 00:00:00 and 9999-12-31 23:59:59 . TIMESTAMP is limited to values between 1970-01-0 1 00:00:01 and 2038-01-19 03:14:07 . Aurora MySQL converts TIMESTAMP values from the current time zone to UTC for

SQL Server Data Type	Convert to MySQL Data Type	Comments
		storage and back from UTC to the current time zone for retrieval. For more information, see MySQL Server Time Zone Support in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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 explicit binary 0x notation. For more information, see The BINARY and VARBINARY Types and The binary Collation Compared to bin Collations in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
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 the 8,000 byte limit in SQL Server. Literal assignment for Aurora MySQL BINARY types use string literals, unlike SQL Server explicit binary 0x notation. For more information, see The BINARY and VARBINARY Types and The binary Collation Compared to bin Collations in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
TEXT / VARCHAR (MAX)	VARCHAR / TEXT / MEDIUMTEXT / LONGTEXT	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 list to determine the optimal Aurora MySQL data type: • For a string length of 2^16-1 bytes, use VARCHAR or TEXT. • For a string length of 2^24-1 bytes, use MEDIUMTEXT . • For a string length of 2^32-1 bytes, use LONGTEXT. For more information, see The BLOB and TEXT Types and Data Type Storage Requirements in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
NTEXT / NVARCHAR (MAX)	VARCHAR / TEXT / MEDIUMTEXT / LONGTEXT	Aurora MySQL doesn't require the use of specific data types for storing UNICODE data. Use the TEXT compatible data types listed earlier and define a UNICODE collation using the CHARACTER SET or COLLATE keywords. For more information, see Unicode Character Sets in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
IMAGE / VARBINARY (MAX)	VARBINARY / BLOB / MEDIUMBLOB / LONGBLOB	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 list to determine the optimal Aurora MySQL data type: • For a string length of 2^16-1 bytes, use VARBINARY or BLOB. • For a string length of 2^24-1 bytes, use MEDIUMBLOB . • For a string length of 2^32-1 bytes, use LONGBLOB. For more information, see The BLOB and TEXT Types, String Type Storage Requirements,

SQL Server Data Type	Convert to MySQL Data Type	Comments
		and The binary Collation Compared to bin Collations in the *MySQL documentation*.
CURSOR	CURSOR	Types are compatible, although in Aurora MySQL a cursor isn't really considered to be a type. For more information, see Cursors.
UNIQUEIDENTIFIER	N/A	Aurora MySQL doesn't support a native unique identifier type. Use `BINARY(16)` , which is the same base type used for the `UNIQUEIDENTIFIER` type in SQL Server. It generates values using the `UUID()` function, which is the equivalent of the `NEW_ID` function in SQL Server. UUID returns a Universal Unique Identifier generated according to RFC 4122. For more information, see A Universally Unique IDentifier (UUID) URN Namespace. For more information, see UUID() in the *MySQL documentation*.

SQL Server Data Type	Convert to MySQL Data Type	Comments
HIERARCHYID	N/A	Aurora MySQL doesn't support native hierarchy representation. Rewrite functionality with custom application code using one the common SQL hierarchi cal data representation approaches: • Adjacency list. • Nested set. • Closure table. • Materialized path. For more information, see Adjacency list and Nested set model.

SQL Server Data Type	Convert to MySQL Data Type	Comments
GEOMETRY	GEOMETRY	In SQL Server, the GEOMETRY type represents data in a Euclidean (flat) coordinat e 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 Spatial Data Types in the *MySQL documentation*.
TABLE	N/A	Aurora MySQL doesn't support a TABLE data type. For more information, see User-Defined Types.

SQL Server Data Type	Convert to MySQL Data Type	Comments
XML	N/A	Aurora MySQL doesn't support a native XML data type. However, it does provide full support for JSON data types, which SQL Server doesn't. Because 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 The JSON Data Type in the *MySQL documentation*.
ROW_VERSION	N/A	Aurora MySQL doesn't support a row version. Use triggers to update a dedicated column from a primary sequence value table.
SQL_VARIANT	N/A	Aurora MySQL doesn't support a hybrid, all-purpo se data type similar to SQL_VARIANT in SQL Server. Rewrite applications to use explicit types.

For more information, see Data Types in the *MySQL documentation*.

GROUP BY

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		GROUP BY	Basic syntax compatible. Advanced options such as `ALL`, `CUBE`, `GROUPING SETS` will require rewrites to use multiple queries with `UNION`.

SQL Server Usage

`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, and not ANSI-compliant syntax 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 rows with NULL values 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`.
- A super aggregated for all customers and all dates.

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
```

The additional four rows where the value for Customer is set to NULL, were added by CUBE. These rows provide super aggregates for every date for all customers that were not part of the ROLLUP results.

Legacy GROUP BY ALL

Use the Orders table from the preceding 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 last row 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 Aggregate Functions (Transact-SQL) and SELECT - GROUP BY- Transact-SQL in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports only the basic ANSI syntax for `GROUP BY` and doesn't support `GROUPING SETS` or the standard `GROUP BY CUBE` and `GROUP BY ROLLUP`. Aurora MySQL supports 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: `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 MySQL Handling of GROUP BY in the *MySQL documentation*.

Unlike SQL Server, in Aurora MySQL you can't use `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 can't 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 isn't 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 don't 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 run.

> **Note**
>
> To use non-aggregate columns in the HAVING, SELECT, and ORDER BY clauses, turn on the ONLY_FULL_GROUP_BY SQL mode.

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.

Because Aurora MySQL supports a wider range of aggregate functions, the migration shouldn't present major challenges. Some minor syntax changes, for example replacing `STDEVP` with `STDDEV_POP`, can be performed automatically by the AWS Schema Conversion Tool (AWS 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 doesn't support an equivalent function to `GROUPING_ID` in SQL Server.

Examples

Rewrite SQL Server WITH CUBE modifier for migration.

```
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 for each 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.

```
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 COUNT and COUNT_BIG in SQL Server.
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.
WITH ROLLUP	WITH ROLLUP	Compatible
WITH CUBE	N/A	Rewrite using UNION ALL.

SQL Server feature	Aurora MySQL feature	Comments
`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 to turn off the `ONLY_FULL_GROUP_BY` SQL mode. Functional dependencies are evaluated by the engine.

For more information, see MySQL Handling of GROUP BY in the *MySQL documentation*.

Table JOIN

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Table Joins	Basic syntax compatible. `FULL OUTER`, `APPLY`, and `ANSI SQL 89` outer joins will need to be rewritten.

SQL Server Usage

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 subquery, a `VALUES` row value constructor, or a table valued function that is evaluated for each row of `<Set A>` where the `<Set B>` query can reference columns from the current row in `<Set A>`. This functionality isn't possible with any type of standard `JOIN` operator.

There are two `APPLY` types:

- `<Set A> CROSS APPLY <Set B>` — Similar to a `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 2008 R2, 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. Consider the following 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. Consider the following 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. Consider the following 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 deprecated. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

> **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 FROM clause plus JOIN, APPLY, PIVOT (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 database engines.

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 aren't supported, but they can be easily worked around.

`CROSS APPLY` and `OUTER APPLY` aren't 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` clause	Supported	
`OUTER JOIN` with commas	Not supported	Requires T-SQL rewrite post SQL Server 2008 R2.
`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 JOIN Clause in the *MySQL documentation*.

Views

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Minor syntax and handling differences. Indexes, triggers, and temporary views aren't supported.

SQL Server Usage

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 don't 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 don't 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, see Overview of SQL Server Security in the *SQL Server documentation*.

View definitions are evaluated when they are created and aren't affected by subsequent changes to the underlying tables.

For example, a view that uses `SELECT *` doesn't 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 aren't permitted.
- If a view is created with the `CHECK OPTION`, rows being updated can't 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 run.
- **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 Views, Modify Data Through a View, and CREATE VIEW (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Similar to SQL Server, Amazon Aurora MySQL-Compatible Edition (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 isn't affected by subsequent changes to the underlying base tables.

Aurora MySQL views have the following restrictions:

- A view can't reference system variables or user-defined variables.

- When used within a stored procedure or function, the SELECT statement can't reference parameters or local variables.
- A view can't 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 can't reference TEMPORARY tables.
- TEMPORARY views aren't supported.
- Views don't 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 the 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 View Processing Algorithms in the *MySQL documentation*.
- 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 run 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 call of a stored function, privileges are checked for statements that run within the function based on the function's SQL SECURITY setting. For functions where the security is set to DEFINER, the function runs with the privileges of the

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

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 doesn't 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 isn't guaranteed.

Security context is explicit in Aurora MySQL, which isn't 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 aren't 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 the 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	Supported	N/A	You can create partitioned views in the same way as SQL Server, they won't

Feature	SQL Server	Aurora MySQL	Comments
			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	For more information, see Triggers.
Temporary views	`CREATE VIEW #View…`	N/A	
Refresh view definition	`sp_refreshview / ALTER VIEW`	`ALTER VIEW`	

For more information, see CREATE VIEW Statement, Restrictions on Views, and Updatable and Insertable Views in the *MySQL documentation*.

Window Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Window Functions	Rewrite window functions to use alternative SQL syntax.

SQL Server Usage

Window 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 database engines don't yet support the full ANSI specification.

Window 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 window functions:

Window function category	Examples
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 . For more information, see Identity and Sequences.

Syntax

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

Examples

Create and populate 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);
```

Use a window 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 window aggregate function to calculate the total quantity for each 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 SELECT - OVER Clause (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Aurora MySQL version 5.7 doesn't support Window functions.

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports window functions that for each row from a query perform a calculation using rows related to that row. These include functions such as `RANK()`, `LAG()`, and `NTILE()`. In addition, several existing aggregate functions now can be used as window functions, for example, `SUM()` and `AVG()`. For more information, see Window Functions in the *MySQL documentation*.

Migration Considerations

As a temporary workaround, rewrite the code to remove the use of Window 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 following examples for migrating Window 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 Window function syntax is ANSI compliant and will be compatible with SQL Server T-SQL syntax.

For more information, see Window Functions in the *MySQL documentation*.

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. The following example is a workaround for the window 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. The following example is a workaround for a partitioned Window 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. The following example is a workaround for the LEAD analytical function.

```
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 Window Function Descriptions in the *MySQL documentation*.

Temporary Tables

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

SQL Server temporary tables are stored in the `tempdb` system database. There are two types of temporary tables: local and global. They differ from each other in their names, their visibility, and their availability. Local temporary tables have a single number sign # as the first character of their names; they are visible only to the current connection for the user, and they are deleted when the user disconnects from the instance of SQL Server.

Global temporary tables have two number signs ## as the first characters of their names; they are visible to any user after they are created, and they are deleted when all users referencing the table disconnect from the instance of SQL Server.

```
CREATE TABLE #MyTempTable (col1 INT PRIMARY KEY);
```

For more information, see Tables and Temporary Tables in the *SQL Server documentation*.

MySQL Usage

In MySQL, the table structure (DDL) of temporary tables isn't stored in the database. When a session ends, the temporary table is dropped.

- **Session-Specific** — In MySQL, each session is required to create its own temporary tables. Each session can create its own private temporary tables using identical table names.
- **In SQL Server**, the default behavior when the `ON COMMIT` clause is omitted is `ON COMMIT DELETE ROWS`. In MySQL, the default is `ON COMMIT PRESERVE ROWS` and it can't be changed.

Note

In Amazon Relational Database Service (Amazon RDS) for MySQL 8.0.13, user-created temporary tables and internal temporary tables created by the optimizer are stored in session temporary tablespaces that are allocated to a session from a pool of temporary tablespaces. When a session disconnects its temporary tablespaces are truncated and released back to the pool. In previous releases temporary tables were created in the `ibtmp1` global temporary tablespace which did not return disk space to the operating system after temporary tables were dropped. The `innodb_temp_tablespaces_dir` variable defines the location where session temporary tablespaces are created.
The default location is the `#innodb_temp` directory in the data directory. The `INNODB_SESSION_TEMP_TABLESPACES` table provides metadata about session temporary tablespaces. The `ibtmp1` global temporary tablespace now stores rollback segments for changes made to user-created temporary tables.

Examples

```
CREATE TEMPORARY TABLE EMP_TEMP (
    EMP_ID INT PRIMARY KEY,
    EMP_FULL_NAME VARCHAR(60) NOT NULL,
    AVG_SALARY INT NOT NULL1;
```

Summary

Feature	SQL Server	Aurora MySQL
Semantic	Global temporary table	Temporary table
Create table	`CREATE GLOBAL TEMPORARY…`	`CREATE TEMPORARY…`
Accessible from multiple sessions	Yes	No
Temporary table DDL persist after session end or database	Yes	No (dropped at the end of the session)

Feature	SQL Server	Aurora MySQL
restart user-managed datafiles		
Create index support	Yes	Yes
Foreign key support	Yes	Yes
`ON COMMIT` default	`COMMIT DELETE ROWS`	`ON COMMIT PRESERVE ROWS`
`ON COMMIT PRESERVE ROWS`	Yes	Yes
`ON COMMIT DELETE ROWS`	Yes	Yes
Alter table support	Yes	Yes
Gather statistics	`dbms_stats.gather_table_stats`	`ANALYZE`
Oracle 12c `GLOBAL_TEMP_TABLE_STATS`	`dbms_stats.set_table_prefs`	`ANALYZE`

For more information, see CREATE TEMPORARY TABLE Statement in the *MySQL documentation*.

T-SQL

Topics

Collations

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Collations	UNICODE uses CHARACTER SET property instead of NCHAR or NVARCHAR data types.

SQL Server Usage

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 as shown following: `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 aren't compatible. A predicate or data modification that introduces a type conflict is resolved using predefined collation precedence rules. For more information, see Collation Precedence in the *SQL Server documentation*.

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 doesn't affect sorting. * SQL Server Collations provide backward compatibility with previous SQL Server versions. They aren't compatible with the windows collation rules for non-UNICODE data.

You can define collations 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't 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.

SQL Server 2019 adds support for UTF-8 for import and export encoding, and as database-level or column-level collation for string data. Support includes PolyBase external tables, and Always Encrypted (when not used with Enclaves). For more information, see Collation and Unicode Support in the *SQL Server documentation*.

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

The following example creates 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;
```

The following example creates 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 Collation and Unicode support in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports multiple character sets and a variety of collations that can be used for comparison. Similar to SQL Server, you can define collations at the server, database, and column level. Additionally, you can define collations at the table level in Aurora MySQL.

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.

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

At the database level, you can set a default character set and collation with the `CREATE DATABASE` and `ALTER DATABASE` statements. Consider the following 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, you can use the COLLATE function to explicitly declare a string's collation. In addition, a prefix to the string can be used to denote a specific character set. Consider the following 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 nonqualified 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 don't assume a specific setting and explicitly set the required character set and collation using the SET NAMES and SET CHARACTER SET statements.

For more information, see Connection Character Sets and Collations in the *MySQL documentation*.

Syntax

The following example creates a database-level collation.

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

The following example creates a table-level collation.

```
CREATE TABLE <Table Name>
(Column Specifications)
[DEFAULT] CHARACTER SET <Character Set>
[COLLATE <Collation>];
```

The following example creates a column collation.

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

The following example creates an expression collation.

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

Examples

The following walkthrough describes how to change the cluster character set and collation.

1. Log in to your Management Console, choose **Amazon RDS** , and then choose **Parameter groups**.
2. Choose **Create parameter group**.
3. For **Parameter group family**, choose **aurora-mysql5.7**.
4. For **Type**, choose **DB Cluster Parameter Group**.
5. For **Group name**, enter the identified for the DB parameter group.
6. Choose **Create**.
7. Choose the newly created group on the **Parameter groups** list.
8. For **Parameters**, enter **character_set_server** in the search box and choose **Edit parameters**.
9. Choose the server default character set.
10. Delete the search term and enter collation. Select the desired default server collation and choose **Preview changes**.
11. Check the values and choose **Close**, and then choose **Save changes**.
12. Return to the Management Console dashboard and choose **Create database**.

13For **Choose a database creation method**, choose **Easy create**.

14For **Engine type**, choose **Amazon Aurora** .

15Enter the instance size, cluster identifier and username. Choose **Create database**.

16Modify the created instance to change the **DB Parameter group**.

Summary

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

Feature	SQL Server	Aurora MySQL
Unicode support	UTF 16 using NCHAR and NVARCHAR data types	8 UNICODE character sets, using 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 Character Sets, Collations, Unicode in the *MySQL documentation*.

Cursors

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Cursors	Aurora MySQL supports only static, forward only, read-only cursors.

SQL Server Usage

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 invocation of a SQL statement can return a whole set or modify millions of rows.

Many developers are accustomed 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

The following example processes 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 SQL Server Cursors and Cursors (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 aren't updatable.
- **Nonscrollable** — Cursors can only be traversed in one direction and can't skip rows. The only supported cursor advance operation is `FETCH NEXT`.

In Aurora MySQL, cursor declarations appear before handler declarations and after variable and condition declarations.

Similar to SQL Server, you can declare cursors with the DECLARE CURSOR statement. To open a cursor, use the OPEN statement. To fetch a cursor, use the FETCH statement. You can close the cursor with the CLOSE statement.

> **Note**
>
> Aurora MySQL doesn't have a DEALLOCATE statement because you don't need it.

DECLARE Cursor

```
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 before, Aurora MySQL supports only FETCH NEXT. Make sure that the number of output variables specified in the FETCH statement matches the number of columns retrieved by the cursor.

Aurora MySQL cursors have additional characteristics:

- SELECT INTO isn't 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

```
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

```
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 isn't available, meaning the cursor has been exhausted, Aurora MySQL raises a no data condition with the SQLSTATE value set to 0200000.

To catch this condition, or the alternative NOT FOUND condition, 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

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

The CLOSE statement closes an open cursor. If the cursor with the specified name doesn't exist, Aurora MySQL raises an error. If a cursor isn't 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 examples use a cursor to iterate over source rows and merges into the `OrderItems` table.

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 and populate the `SourceTable` table.

```
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 information, see Stored Procedures.

```
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;
```

Run 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	
Declaration	`DECLARE CURSOR`	`DECLARE CURSOR`	No options for `DECLARE CURSOR` in Aurora MySQL.
Open	OPEN	OPEN	
Fetch	`FETCH NEXT | PRIOR | FIRST | LAST | ABSOLUTE | RELATIVE`	`FETCH NEXT`	
Close	CLOSE	CLOSE	
Deallocate	DEALLOCATE	N/A	Not required because the `CLOSE` statement deallocates the cursor
Cursor end condition	`@@FETCH_STATUS` system variable	Event Handler	Event handlers aren't specific to a cursor. For more

Feature	SQL Server	Aurora MySQL	Comments
			information, see Error Handling.

For more information, see Cursors in the *MySQL documentation*.

Date and Time Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Date and Time Functions	Time zone handling. Syntax differences.

SQL Server Usage

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 doesn't address time zone considerations and time zone aware functions. For more information, 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 date part of a specified date.	`SELECT MONTH(GETDATE()), YEAR(GETDATE())`	4, 2018	
DATEDIFF	Returns an integer value of date part boundaries that are crossed between two dates.	`SELECT DATEDIFF( DAY, GETDATE(), EOMONTH(GETDATE()))`	25	How many days are left until the end of the month.
DATEADD	Returns a datetime value that is calculated with an offset interval to the specified date part of a date.	`SELECT DATEADD(DAY, 25, GETDATE())`	2018-04-30 15:55:52.147	
CAST and CONVERT	Converts datetime values to and from string literals and to and from	`SELECT CAST(GETDATE() AS DATE)`	2018-04-05 20180405	Default date format. Style 112 (ISO) with no separators.

Function	Purpose	Example	Result	Comments
	other datetime formats.	`SELECT CONVERT(VARCHAR(20), GETDATE(), 112)`		

For more information, see Date and Time functions in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 runs, compared to `NOW`, which returns a constant time when the statement started running.

Function	Purpose	Example	Result	Comments
				Also, SET TIMESTAMP doesn't 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 the difference in days between two dates.	SELECT DATEDIFF(NOW(),'2018-05-01')	-25	DATEDIFF in Aurora MySQL is only for calculating difference in days. Use TIMESTAMPDIFF instead.
TIMESTAMPDIFF	Returns an integer value of the difference in date part between two dates.	SELECT TIMESTAMPDIFF(DAY, NOW(),'2018-05-01')	24	

Function	Purpose	Example	Result	Comments
DATE_ADD, DATE_SUB	Returns a datetime 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 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 separators.

Migration Considerations

The date and time handling paradigm in Aurora MySQL differs from SQL Server.

Be aware of the differences in data types, time zone awareness, and locale handling. For more information, see Data Types.

Summary

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

SQL Server function	Aurora MySQL function	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 runs, unlike NOW which returns a constant time when the statement started running. Also, SET TIMESTAMP doesn't 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

SQL Server function	Aurora MySQL function	Comments
		different and requires a rewrite.
CAST and CONVERT	DATE_FORMAT , TIME_FORMAT	Although Aurora MySQL supports both CAST and CONVERT, they aren't used for style conversion as in SQL Server. Use DATE_FORMAT and TIME_FORMAT .

For more information, see Date and Time Functions in the *MySQL documentation*.

String Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Differences with the UNICODE paradigm. For more information, see Collations. Syntax and option differences.

SQL Server Usage

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 expressio n (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 expressio ns 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 string from another string expressio n based on	`SELECT LEFT('abs ',2), SUBSTRING ('abcd',2 ,2)`	'ab', 'bc'	

Function	Purpose	Example	Result	Comments
	position and length.			
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('AB cd')	'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(3.141 5927,5,3)	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 concatena ted copies of another string expression.	SELECT REPLICATE ('abc', 3)	'abcabcabc'	

Function	Purpose	Example	Result	Comments
REPLACE	Replace all occurrences 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©	1 2	STRING_SPLIT is a table-valued function.
STRING_AGG	Return a string that consists of concatenated string values 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 aggregate function.

For more information, see String Functions (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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), don't 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 expressio n (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 expressio ns 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 string from another string expressio n based on position and length	SELECT LEFT('abs ',2), SUBSTRING ('abcd',2 ,2)	'ab', 'bc'	

Function	Purpose	Example	Result	Comments
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'	

Function	Purpose	Example	Result	Comments
REPEAT	Return a string that consists of zero or more concatena ted 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 doesn't handle ASCII and UNICODE types separately. Any string can be either UNICODE or ASCII, depending on its collation property. For more information, 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 doesn't support. 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. For more information, see Data Types.

SQL Server function	Aurora MySQL function	Comments
CHAR and NCHAR	CHAR	Unlike SQL Server, CHAR in Aurora MySQL accepts a list of values and constructs a concatenated string. For more information, see Data Types.
CHARINDEX and PATINDEX	LOCATE and POSITION	LOCATE and POSITION are synonymous but don't support wildcards as 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 be used to trim either leading or trailing characters. The syntax is shown following : `TRIM ([{BOTH | LEADING | TRAILING} [<Remove String>] FROM] <String>)`
STR	FORMAT	FORMAT doesn't support full precision and scale definitio n, 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 String Functions and Operators in the *MySQL documentation*.

Databases and Schemas

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Schema and database are synonymous.

SQL Server Usage

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), and 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 so on. You should 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 the metadata of database objects, 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

The preceding example uses default settings for the new database and schema.

For more information, see sys.databases (Transact-SQL), CREATE SCHEMA (Transact-SQL), and CREATE DATABASE in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports both the `CREATE SCHEMA` and `CREATE DATABASE` statements. However, in Aurora MySQL, these statements are synonymous.

Unlike SQL Server, Aurora MySQL doesn't 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 and schema in Aurora MySQL is managed as a separate set of physical files similar to an SQL Server database.

Aurora MySQL doesn't 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 information, 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 or schema for missing object qualifiers.

The syntax is identical to SQL Server:

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

After you run the USE command, 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 or schemas in Aurora MySQL.

```
SHOW DATABASES;
```

For the preceding example, the result looks as shown following.

```
database

information_schema
Demo
mysql
performance_schema
sys
```

Aurora MySQL also supports a CREATE DATABASE syntax reminder command.

```
SHOW CREATE DATABASE Demo;
```

For the preceding example, the result looks as shown following.

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

Examples

The following 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 or 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.

Current object architecture	Migrate to Aurora MySQL	Rewrites
Single database, objects in multiple schemas.	Single instance, multiple databases or schemas.	No identifier hierarchy rewrites needed. Code should be compatible with respect to the object hierarchy.
Multiple databases, all objects in the dbo schema.	Single instance, multiple databases or 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 applicati on level.

For more information, see CREATE DATABASE Statement in the *MySQL documentation*.

Transactions

Feature compatibi lity	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Transaction Isolation	Default isolation level is set to `REPEATABLE READ` . Default mechanism `CONSISTENT SNAPSHOT` is similar to `READ COMMITTED SNAPSHOT` isolation in SQL Server. Syntax

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			and option differences.

SQL Server Usage

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, transactions 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, for each ANSI definitions, shouldn't 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, and so on.
- **Isolation** — Concurrent run of transactions must result in a system state that would occur if transactions were run 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 or implicit transactions mode set to ON. Every statement is treated as a transaction on its own unless a transaction was explicitly defined. This behavior is different than other engines like Oracle where, by default, every DML requires an explicit `COMMIT` statement to be persisted.

Syntax

The following examples show the simplified syntax for the commands defining transaction boundaries.

Define the beginning of a transaction.

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

Commit 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).

Each level provides a different approach for managing the concurrent run of transactions. The main purpose of a transaction isolation level 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 a rollback, all data is restored to its previous state.
- **Read committed** — A transaction only sees data changes that were committed. Therefore, dirty reads aren't 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 doesn't permit transaction overwrites of another transaction's actions. Concurrent run 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 or 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 ran twice can return different values for the same rows.
- **Phantom or ghost reads** — Similar to a non-repeatable read, but it is related to new data created by another transaction. The same query ran 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.

Transaction isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Read uncommitted	Allowed	Allowed	Allowed

Transaction isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Read committed	Disallowed	Allowed	Allowed
Repeatable read	Disallowed	Disallowed	Allowed
Serializable	Disallowed	Disallowed	Disallowed

There are two common implementations for transaction isolation:

- **Pessimistic isolation or 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. You can use them concurrently.

For optimistic isolation, SQL Server introduced two additional isolation levels: read-committed snapshot and snapshot.

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

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

Examples

The following example runs 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 Transaction Isolation Levels (ODBC) and SET TRANSACTION ISOLATION LEVEL (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports the four transaction isolation levels specified in the SQL:1992 standard: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`.

The simplified syntax for setting transaction boundaries in Aurora MySQL is shown following:

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

Note

Setting the `GLOBAL` isolation level isn't supported in Aurora MySQL; only session scope can be changed. This behavior is similar to Oracle. Also, 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, explicitly change the default to `READ COMMITTED`.

The default isolation level for Aurora MySQL is `REPEATABLE READ`.

To set the transaction isolation level, you will need to set the `tx_isolation` parameter when using Aurora MySQL. For more information, see Server Options.

Note

Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports a new `innodb_deadlock_detect` dynamic variable. You can use this variable to turn off

the deadlock detection. On high concurrency systems deadlock detection can cause a slowdown when numerous threads wait for the same lock. At times it may be more efficient to turn off deadlock detection and rely on the `innodb_lock_wait_timeout` setting for transaction rollback when a deadlock occurs.

Starting from MySQL 8, InnoDB supports `NOWAIT` and `SKIP LOCKED` options with `SELECT ... FOR SHARE` and `SELECT ... FOR UPDATE` locking read statements. `NOWAIT` causes the statement to return immediately if a requested row is locked by another transaction.

`SKIP LOCKED` removes locked rows from the result set. `SELECT ... FOR SHARE` replaces `SELECT ... LOCK IN SHARE MODE` but `LOCK IN SHARE MODE` remains available for backward compatibility. The statements are equivalent. However, `FOR UPDATE` and `FOR SHARE` support `NOWAIT SKIP LOCKED` and `OF tbl_name` options. For more information, see SELECT Statement in the *MySQL documentation*.

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 isn't supported in Aurora MySQL. You can only change the session scope; 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, you can optionally specify a transaction intent. Setting a transaction to `READ ONLY` turns off 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` doesn't 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.

Use the following statement to commit work at the end of a transaction.

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

Use the following statement to rollback work at the end of a transaction.

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

One of the `ROLLBACK` options is `ROLLBACK TO SAVEPOINT<logical_name>`. This command will rollback all changes in current transaction up to the save point mentioned.

Create transaction save point during the transaction

```
SAVEPOINT <logical_name>
```

> **Note**
>
> If the current transaction has a save point with the same name, the old save point is deleted and a new one is set.

Aurora MySQL supports both auto commit and explicit commit modes. You can change mode using the `autocommit` system variable.

```
SET autocommit = {0 | 1}
```

Examples

The following example runs 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 Server. Evaluate application needs and set appropriately.
Initialize transaction syntax	`BEGIN TRAN` or `BEGIN TRANSACTION`	`START TRANSACTION`	Code rewrite is required from `BEGIN` to `START`. If using the shorthand `TRAN`, rewrite to `TRANSACTION` .

Transaction property	SQL Server	Aurora MySQL	Comments
Default isolation mechanism	Pessimistic lock based	Lock based for writes, consistent read for SELECT statements.	The Aurora MySQL default mode is similar to the READ COMMITTED SNAPSHOT isolation in SQL Server.
Commit transaction	COMMIT [WORK|TRAN|TRANSACTION]	COMMIT [WORK]	If you only use COMMIT or COMMIT WORK, no change is needed. Otherwise, rewrite TRAN and TRANSACTION to WORK.
Rollback transaction	ROLLBACK [WORK |[TRAN | TRANSACTION]	ROLLBACK [WORK]	If you only use ROLLBACK or ROLLBACK WORK, no change is needed. Otherwise, rewrite TRAN and TRANSACTION to WORK.
Set autocommit off or on	SET IMPLICIT_ TRANSACTIONS OFF | ON	SET autocommit = 0 | 1	For more information, 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 `READ COMMITTED SNAPSHOT` in SQL Server.
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 transacti on.
Transaction chaining	Not supported	Causes a new transaction to open immediately upon transaction completio n.	
Transaction release	Not supported	Causes the client session to disconnec t upon transaction completion.	

For more information, see Transaction Isolation Levels in the *MySQL documentation*.

DELETE and UPDATE FROM

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Rewrite to use subqueries.

SQL Server Usage

SQL Server supports an extension to the ANSI standard that allows using an additional `FROM` clause in `UPDATE` and `DELETE` statements.

You can use this additional `FROM` clause 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;
```

For the preceding examples, the result looks as shown following.

```
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;
```

For the preceding example, the result looks as shown following.

```
Customer  OrderDate
Jack      2018-03-24
Jack      2018-04-02
```

For more information, see UPDATE (Transact-SQL), DELETE (Transact-SQL), and FROM clause plus JOIN, APPLY, PIVOT (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't 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 subquery, such as in the following example, 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;
```

For the preceding example, the result looks as shown following.

```
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
IN (
    SELECT OrderID
    FROM OrderCorrections
);
```

```
SELECT *
FROM Orders;
```

For the preceding example, the result looks as shown following.

```
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 the 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 the WHERE clause to limit updates set.

For more information, see UPDATE Statement and DELETE Statement in the *MySQL documentation*.

Stored Procedures

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Stored Procedures	No support for table-valued parameters. Syntax and option differences.

SQL Server Usage

Stored procedures are encapsulated, persisted code modules you can run using the EXECUTE T-SQL statement. They may have multiple input and output 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.

In SQL Server, you can run stored procedures in any security context using the EXECUTE AS option. They can be explicitly recompiled for every run 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 you use 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, for example, RETURN -1. It's 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 ] }[;]
```

Creating and Running 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;
```

Run 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 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 Washer', 100);
```

Create a table-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 CREATE PROCEDURE (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) stored procedures provide similar functionality to SQL Server stored procedures.

As with SQL Server, Aurora MySQL supports security run 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 doesn't 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 `thelog_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 L00P cursor with a source table to replace table valued parameters.

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 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;
```

For the preceding example, the result looks as shown following.

```
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.

Feature	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

Feature	SQL Server	Aurora MySQL	Workaround
			the parameter declaration. Rewrite stored procedure parameter direction `OUTPUT` to `OUT` or `INOUT` for bidirectional parameters. `IN` is the parameter direction for both MySQL and SQL Server.

Feature	SQL Server	Aurora MySQL	Workaround
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, `OWNER`s can't be specified and must be explicitly named.
Encryption	Use the `WITH ENCRYPTION` option.	Not supported in Aurora MySQL.	

Feature	SQL Server	Aurora MySQL	Workaround
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, make sure that you rewrite the code for syntax compliance. Use OUT instead of OUTPUT. Use INOUT instead of OUT for bidirectional parameters.
Recompile	Use the 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 preceding example for a workaround.
INSERT… EXEC	Use the output of the stored procedure as input to an INSERT statement.	Not supported in Aurora MySQL.	Use tables to hold the data or pass string parameters formatted as CSV, XML, JSON (or any other convenient format) and then parse the parameters before the INSERT statement.

Feature	SQL Server	Aurora MySQL	Workaround
Additional restrictions	Use `BULK INSERT` to load data from text file.	The LOAD DATA statement isn't allowed in stored procedures.	
`RETURN` value	`RETURN <Integer Value>`	Not supported.	Use a standard `OUTPUT` parameter instead.

For more information, see Stored Procedures and Functions and CREATE PROCEDURE and CREATE FUNCTION Statements in the *MySQL documentation*.

Error Handling

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Error Handling	Different paradigm and syntax requires rewrite of error handling code.

SQL Server Usage

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.

Starting from SQL Server 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, the run 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
```

THROW

The THROW statement raises an exception and transfers run 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 preceding example. This behavior is in accordance with ANSI specifications stating that a constraint violation shouldn't 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 running session and forwards run 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 doesn't.
- 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>]])
```

Example

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 RAISERROR (Transact-SQL), TRY…CATCH (Transact-SQL), and THROW (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 metadata 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. You can reference this declared name 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 ran when a CONDITION arises. The handler object may be used to CONTINUE or EXIT the run.

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 an **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 aren't errors; they are used to represent successful runs 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 run 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 run 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 run result of an SQL statement.

The CURRENT keyword causes retrieval of the current diagnostics area. The STACKED keyword 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>
```

Example

Retrieve SQLSTATE and MESSAGE_TEXT from the diagnostic area for the last statement that you ran.

```
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 run, or to the SQL client. The SIGNAL statement enables explicitly defining the error's properties such as error number, SQLSTATE value, message, and so on.

The difference between SIGNAL and RESIGNAL is that RESIGNAL is used to pass on the error condition information available during the run 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 isn't 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 ran 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 can't 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 table 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

SQL Server error handling feature	Migrate to Aurora MySQL	Comments
		CATCH blocks, the handlers must be defined first, not later. Review the handler scope and handler choice sections.
THROW and RAISERROR	SIGNAL and RESIGNAL	Review the handler scope and handler choice sections.
THROW with variables.	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 condition handler run — always the next code block in statement order.	May be non-deterministic if multiple handlers have the same precedence and scope.	Review the handler scope and handler choice sections.

For more information, see The MySQL Diagnostics Area and Condition Handling in the *MySQL documentation*.

Flow Control

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Flow Control	Syntax and option differences, similar functionality.

SQL Server Usage

Although SQL is a mostly declarative language, it does support flow control commands, which provide run time dynamic changes in script run 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 run together.
- `RETURN` — Exit a server code module such as stored procedure, function, and so on and return control to the calling scope. You can use `RETURN <value>` to return an `INT` value to the calling scope.
- `BREAK` — Exit `WHILE` loop run.
- `THROW` — Raise errors and potentially return control to the calling stack.
- `CONTINUE` — Restart a `WHILE` loop.
- `TRY… CATCH` — Error handling. For more information, see Error Handling.
- `GOTO label` — Moves the run 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. For more information, see Cursors.

For more information, see Error Handling.

Examples

Create and populate 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);
```

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 Run Path

The following example demonstrates a solution for running different processes based on the number of items in an order.

Declare a cursor for looping through all OrderItems and calculating the total quantity for each 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;
```

For the preceding example, the result looks as shown following.

```
EXECUTING LogSmallOrder - 1 100
EXECUTING LogSmallOrder - 2 100
EXECUTING LogLargeOrder - 3 200
```

For more information, see Control-of-Flow in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) provides the following flow control constructs:

- `BEGIN... END` — Define boundaries for a block of commands that are ran together.
- `CASE` — Run 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 such as stored procedure, function, and so on, and return control to the calling scope.
- `LOOP` — Loop indefinitely.

- REPEAT… UNTIL — Loop until the predicate is true.
- RETURN — Terminate the run of the current scope and return to the calling scope.
- WHILE — Continue looping while the condition returns TRUE.
- SLEEP — Pause the run for a specified number of seconds.

Examples

Create and populate 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);
```

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 Run Path

The following example demonstrates a solution for running different processes 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 which you run as a batch script, Aurora MySQL variables can only be used in stored routines such as 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.

Feature	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.	Exit a stored function and return to caller.	For Aurora MySQL, RETURN is valid only

Feature	SQL Server	Aurora MySQL	Workaround
	Supported for both scripts and stored code such as procedures and functions.		in stored or user-defined functions. It isn't used in stored procedures, triggers, or events. Rewrite the T-SQL code using the LEAVE keyword. The RETURN statement can return a value in both products. However, LEAVE doesn't support return parameters. Rewrite the code to use output parameters. You can't RETURN in Aurora MySQL for scripts that aren't part of a stored routine.

Feature	SQL Server	Aurora MySQL	Workaround
BREAK	Exit the WHILE loop run.	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 beginning of the loop.
THROW	Raise errors and potentially return control to the calling stack.	Errors are handled by HANDLER objects.	For more informati on, see Error Handling.
TRY - CATCH	Error handling	Errors are handled by HANDLER objects.	For more informati on, see Error Handling.

Feature	SQL Server	Aurora MySQL	Workaround
GOTO	Move run to specified label.	Not supported.	Consider rewriting the flow logic using either CASE statements or nested stored procedures. You can use nested stored procedure s to circumvent this limitation by separating code sections and encap sulating them in sub-procedures. Use IF <conditio n> CALL <stored procedure> in place of GOTO.

Feature	SQL Server	Aurora MySQL	Workaround
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 isn't supported in Aurora MySQL. You can calculate the difference in seconds between the desired time and current time using date and rime functions and use the result to dynamically generate the SLEEP statement. Alternati vely, consider using CREATE EVENT with a predefined schedule.

Feature	SQL Server	Aurora MySQL	Workaround
`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.

Feature	SQL Server	Aurora MySQL	Workaround
`WHILE`	Continue running while condition is `TRUE`.	Continue running 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`. Aurora MySQL doesn't require a `BEGIN…END` block. Rewrite T-SQL code to use the Aurora MySQL keywords.

For more information, see Flow Control Statements in the *MySQL documentation*.

Full-Text Search

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Full-Text Search	Syntax and option differences, less comprehensive but simpler. Most common basic functionality is similar. Requires rewrite of administr

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			ation logic and queries.

SQL Server Usage

SQL Server supports an optional framework for running full-text search queries against character-based data in SQL Server tables using an integrated, in-process full-text engine, and the `fdhost.exe` filter daemon host process.

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, 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. For more information, see CREATE FULLTEXT INDEX (Transact-SQL) in the *SQL Server documentation*.

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, and so on.

Full-text catalogs are created 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. For more information, see CREATE FULLTEXT CATALOG (Transact-SQL) in the *SQL Server documentation*.

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**
>
> Don't 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 turn off 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 the 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 isn't 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 Full-Text Search in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 and a warning is issued.

Running a SHOW WARNINGS statement after creating a full-text index on a table that doesn't have this column generates a warning. Consider the following 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 doesn't 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 aren't automatically managed. Make sure that you 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 isn't 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, see Full-Text Stopwords in the *MySQL documentation*.

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 word *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, see Boolean Full-Text Searches in the *MySQL documentation*.

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 so on.

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, see Full-Text Searches with Query Expansion in the *MySQL documentation*.

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 Relational Database Service (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 highlighting, 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 Amazon RDS instances and the CloudSearch Service.

For more information, see Amazon 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 isn't 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 InnoDB Full-Text Indexes in the *MySQL documentation*.

SQL Server Graph Features

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Feature isn't supported. Migration will require implementing a workaround.

SQL Server Usage

SQL Server offers graph database capabilities to model many-to-many relationships. The graph relationships are integrated into Transact-SQL and receive the benefits of using SQL Server as the foundational database management system.

A graph database is a collection of nodes or vertices and edges or relationships. A node represents an entity (for example, a person or an organization) and an edge represents a relationship between the two nodes that it connects (for example, likes or friends). Both nodes and edges may have properties associated with them. Here are some features that make a graph database unique:

- Edges or relationships are first class entities in a graph database and can have attributes or properties associated with them.
- A single edge can flexibly connect multiple nodes in a graph database.
- You can express pattern matching and multi-hop navigation queries easily.
- You can express transitive closure and polymorphic queries easily.

A relational database can achieve anything a graph database can. However, a graph database makes it easier to express certain kinds of queries. Also, with specific optimizations, certain queries may perform better. Your decision to choose either a relational or graph database is based on following factors:

- Your application has hierarchical data. The `HierarchyID` data type can be used to implement hierarchies, but it has some limitations. For example, it doesn't allow you to store multiple parents for a node.
- Your application has complex many-to-many relationships; as application evolves, new relationships are added.
- You need to analyze interconnected data and relationships.

SQL Server 2017 adds new graph database capabilities for modeling graph many-to-many relationships. They include a new `CREATE TABLE` syntax for creating node and edge tables, and the keyword `MATCH` for queries.

For more information, see Graph processing with SQL Server and Azure SQL Database in the *SQL Server documentation*.

Consider the following `CREATE TABLE` example:

```
CREATE TABLE Person (ID INTEGER PRIMARY KEY, Name VARCHAR(100), Age INT) AS NODE;

CREATE TABLE friends (StartDate date) AS EDGE;
```

The new `MATCH` clause is introduced to support pattern matching and multi-hop navigation through the graph. The `MATCH` function uses ASCII-art style syntax for pattern matching. Consider the following example:

```
-- Find friends of John
SELECT Person2.Name
FROM Person Person1, Friends, Person Person2
WHERE MATCH(Person1-(Friends)->Person2)
AND Person1.Name = 'John';
```

SQL Server 2019 adds ability to define cascaded delete actions on an edge constraint in a graph database. Edge constraints enable users to add constraints to their edge tables, thereby enforcing specific semantics and also maintaining data integrity. For more information, see Edge constraints in the *SQL Server documentation*.

In SQL Server 2019, graph tables have support for table and index partitioning. For more information, see Partitioned tables and indexes in the *SQL Server documentation*.

MySQL Usage

Currently, MySQL doesn't provide native graph database features.

JSON and XML

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		XML and JSON	Minimal XML support, extensive JSON support. No XQUERY support, optionally convert to JSON.

SQL Server Usage

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"
}]
```

Its XML counterpart is show following:

```
<?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 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 or attribute values. An XML typed document may differ from the original text; whitespace 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.
- You can use an annotated schema (AXSD) 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

In SQL Server, you can create `PRIMARY` and `SECONDARY` XML indexes on columns with a native XML data type. You can create secondary indexes 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. In SQL Server, you can run queries directly against XML data and use them as expressions or sets in standard T-SQL statements. Consider the following example:

```
DECLARE @XMLVar XML = '<Root><Data>My XML Data</Data></Root>';
SELECT @XMLVar.query('/Root/Data');
```

For the preceding example, the result looks as shown following.

```
Result: <Data>My XML Data</Data>
```

JSON Data

SQL Server doesn't support a dedicated JSON data type. However, you can store JSON documents in an NVARCHAR column. For more information, 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 the 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

The following example creates a table with a native typed XML column.

```
CREATE TABLE MyTable
(
    XMLIdentifier INT NOT NULL PRIMARY KEY,
    XMLDocument XML NULL
);
```

The following example queries 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 JSON data in SQL Server and XML Data in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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.

MySQL 5.7.22 also added the JSON utility function `JSON_PRETTY()` which outputs an existing JSON value in an easy-to-read format; each JSON object member or array value is printed on a separate line and a child object or array is indented 2 spaces with respect to its parent. This function also works with a string that can be parsed as a JSON value. For more information, see JSON Utility Functions in the *MySQL documentation*.

MySQL 5.7.22 also added the JSON utility functions `JSON_STORAGE_SIZE()` and `JSON_STORAGE_FREE()`. `JSON_STORAGE_SIZE()` returns the storage space in bytes used for the binary representation of a JSON document prior to any partial update.

`JSON_STORAGE_FREE()` shows the amount of space freed after it has been partially updated using `JSON_SET()` or `JSON_REPLACE()`; this is greater than zero if the binary representation of

the new value is less than that of the previous value. Each of these functions also accepts a valid string representation of a JSON document.

For such a value `JSON_STORAGE_SIZE()` returns the space used by its binary representation following its conversion to a JSON document. For a variable containing the string representation of a JSON document `JSON_STORAGE_FREE()` returns zero. Either function produces an error if its (non-null) argument can't be parsed as a valid JSON document and NULL if the argument is NULL. For more information, see JSON Utility Functions in the *MySQL documentation*.

Note

Amazon Relational Database Service (Amazon RDS) for MySQL 8 added two JSON aggregation functions `JSON_ARRAYAGG()` and `JSON_OBJECTAGG()`. `JSON_ARRAYAGG()` takes a column or expression as its argument and aggregates the result as a single JSON array. The expression can evaluate to any MySQL data type; this doesn't have to be a JSON value. `JSON_OBJECTAGG()` takes two columns or expressions which it interprets as a key and a value; it returns the result as a single JSON object. For more information, see Aggregate Functions in the *MySQL documentation*.

Note

Amazon RDS for MySQL 8.0.17 adds two functions `JSON_SCHEMA_VALID()` and `JSON_SCHEMA_VALIDATION_REPORT()` for validating JSON documents. `JSON_SCHEMA_VALID()` returns `TRUE` or `1` if the document validates against the schema and `FALSE` or `0` if it doesn't. `JSON_SCHEMA_VALIDATION_REPORT()` returns a JSON document containing detailed information about the results of the validation.

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 doesn't return tags sub-tags contained within a matching tag or its content.

Consider the following example.

```
SELECT ExtractValue('<Root><Person>John</Person><Person>Jim</Person></Root>', '/Root/
Person');
```

```
Results: John Jim
```

You can use `UpdateXML` 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.

Consider the following example.

```
SELECT UpdateXML('<Root><Person>John</Person><Person>Jim</Person></Root>', '/Root',
 '<Person>Jack</Person>')
```

```
Results: <Person>Jack</Person>
```

> **Note**
>
> Aurora MySQL doesn't support MySQL `LOAD XML` syntax. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket in the *User Guide for Aurora*.

JSON Data Type

Aurora 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 isn't 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.

Consider the following 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, you can use spatial JSON functions for GeoJSON documents. For more information, see Spatial GeoJSON Functions in the *MySQL documentation*.

The JSON_ARRAY, JSON_OBJECT, and JSON_QUOTE functions return a JSON document from a list of values, a list of key-value pairs, or a JSON value respectively.

Consider the following example.

```
SELECT JSON_OBJECT('Person', 'John', 'Country', 'USA');
```

```
{"Person": "John", "Country": "USA"}
```

You can use The JSON_CONTAINS, JSON_CONTAINS_PATH, JSON_EXTRACT, JSON_KEYS, and JSON_SEARCH functions 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.

Consider the following 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.

Consider the following example.

```
SELECT JSON_ARRAY_INSERT('["Mary", "Paul", "Jim"]', '$[1]', 'Jack');
```

```
["Mary", "Jack", "Paul", "Jim"]
```

You can use `JSON_SEARCH` to find the location of an element value within a JSON document.

Consider the following example.

```
SELECT JSON_SEARCH('["Mary", "Paul", ["Jim", "Ryan"]]', 'one', 'Paul');
```

```
"$[1]"
```

JSON Indexes

JSON columns are effectively a `BINARY` family type, which can't 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 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
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 over 25 dedicated JSON functions. For more information, see JSON Function Reference in the *MySQL documentation*.
XML functions	`XQUERY` and `XPATH`, `OPEN_XML`, `FOR XML`	`ExtractValue` and `UpdateXML` .
XML and JSON indexes	Primary and secondary `PATH`, `VALUE`, and `PROPERTY` indexes	Requires adding always-generated (computed and persisted) columns with JSON expressions and indexing them explicitly. The optimizer can make use of JSON expressions only.

For more information, see XML Functions, The JSON Data Type, and JSON Functions in the *MySQL documentation*.

MERGE

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		MERGE	Rewrite to use `REPLACE` and `ON DUPLICATE KEY`, or individual constituent DML statements.

SQL Server Usage

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 doesn't exist and updating the existing row if the key value already exists.

You can 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. You can replicate all MERGE functionality 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 isn't 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 MERGE (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't 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 isn't possible to define custom MATCH conditions as with SQL Server 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 isn't 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 runs 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

REPLACE and INSERT … ON DUPLICATE KEY UPDATE don't provide a full functional replacement for MERGE in SQL Server. The key differences are:

- Key violation conditions are mandated by the primary key or unique constraints that exist on the target table. They can't 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 the preceding comment.
`WHEN NOT MATCHED BY SOURCE UPDATE`	`UPDATE Target SET <assignments> WHERE Key NOT IN (SELECT Key FROM Source)`	

SQL Server MERGE feature	Migrate to Aurora MySQL	Comments
`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 REPLACE Statement and INSERT … ON DUPLICATE KEY UPDATE Statement in the *MySQL documentation*.

PIVOT and UNPIVOT

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		PIVOT and UNPIVOT	Straightforward rewrite to use traditional SQL syntax.

SQL Server Usage

`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 isn't be pivoted and results in a single row for each 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 for each day. Days of month from 5 to 31 are 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;
```

For the preceding example, the result looks as shown following.

```
DayOfMonth                    1  2  3  4  /*...[31]*/
Number of Orders for Each Day  2  1  0  4
```

Note

The result set is now oriented in rows and columns. The first column is the description of the columns to follow.

PIVOT for number of orders for each day for each 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.

Because 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 real life, 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 for each date into individual rows for each 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 FROM - Using PIVOT and UNPIVOT in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't support the `PIVOT` and `UNPIVOT` relational operators.

Functionality of both operators can be rewritten to use standard SQL syntax, as shown in the following examples.

PIVOT Examples

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');
```

Create a simple PIVOT for the number of orders for each day. Days of month from 5 to 31 are 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;
```

For the preceding example, the result looks as shown following.

```
DayOfMonth                    1  2  3  4  /*...[31]*/
Number of Orders for Each Day  2  1  0  4
```

PIVOT for number of orders for each day for each 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 Examples

Create an populate the pivot-like `EmployeeSales` table. In real life, this is most likely a view or a set from an external source.

```
CREATE TABLE EmplyeeSales
(
    SaleDate DATE NOT NULL PRIMARY KEY,
    John INT,
    Kevin INT,
    Mary INT
);
```

```
INSERT INTO EmplyeeSales
VALUES
('20180101', 150, 0, 300),
('20180102', 0, 0, 0),
('20180103', 250, 50, 0),
('20180104', 500, 400, 100);
```

Unpivot employee sales for each date into individual rows for each 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 MySQL/Pivot table in the *MySQL documentation*.

Synonyms

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Synonyms	Use stored procedures and functions to abstract instance-wide objects.

SQL Server Usage

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.

You can create synonyms for the following objects:

- Assembly 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 server A exists on server B that points to server A.

For more information, see CREATE SYNONYM (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't support synonyms 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 aren't supported by Aurora MySQL.

For more information, see Linked Servers, Views, User-Defined Functions, and Stored Procedures.

SQL Server TOP and FETCH and MySQL LIMIT

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		TOP and FETCH	Syntax rewrite, very similar functionality. Convert PERCENT and TIES to subqueries.

SQL Server Usage

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 you use TOP with no ORDER BY clause, the query is non-deterministic and may return any rows up to the number specified by the TOP operator.

You can use TOP (n) 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 three 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 of 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 SELECT - ORDER BY Clause (Transact-SQL) and TOP (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports the non-ANSI compliant but popular with other database engines `LIMIT… OFFSET` operator for paging results sets.

The `LIMIT` clause limits the number of rows returned and doesn't 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.

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 AWS Schema Conversion Tool (AWS SCT except for the `WITH TIES` and `PERCENT` modifiers.

To replace the PERCENT option, first calculate how many rows the query returns and then calculate the fixed number of rows to be returned based on that number.

> **Note**
>
> Because 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, 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**
>
> Because 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.

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;
```

For the preceding example, the result looks as shown following.

```
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
```

For the preceding example, the result looks as shown following.

```
OrderID  Item            Quantity
3        M6 Locking Nut  300
3        M8 Washer       200
2        M8 Nut          100
```

```
1        M8 Bolt         100
```

Retrieve half of 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);
```

For the preceding example, the result looks as shown following.

```
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 the workaround.
`TOP (n) PERCENT`	Not supported	See examples for the workaround.
`OFFSET… FETCH`	`LIMIT… OFFSET`	

For more information, see SELECT Statement and LIMIT Query Optimization in the *MySQL documentation*.

Triggers

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Triggers	Only `FOR EACH ROW` processing. No DDL or `EVENT` triggers. `BEFORE` triggers replace `INSTEAD OF` triggers.

SQL Server Usage

Triggers are special type of stored procedure that run 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 Server doesn't support `FOR EACH ROW` triggers in which the trigger code is run once for each row of modified data.

Trigger Run

- `AFTER` triggers run after DML statements complete run.
- `INSTEAD OF` triggers run code in place of the original DML statement.

You can create `AFTER` triggers only on a table. You can create `INSTEAD OF` triggers on tables and views.

You can create only a single INSTEAD OF trigger 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 enables setting the first and last triggers to be run, but not the order of others.

Trigger Scope

SQL Server supports only statement level triggers. The trigger code runs only once for each 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 run.

SQL triggers always run within the transaction of the statement that triggered the run. 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 isn't run and, therefore, doesn't 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 the 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 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;
```

For the preceding example, the result looks as shown following.

```
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 can't 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 DML Triggers and DDL Triggers in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) provides Data manipulation Language (DML) triggers only.

MySQL supports `BEFORE` and `AFTER` triggers for `INSERT`, `UPDATE`, and `DELETE` with full control over trigger run 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 run once for each row, not once for each statement as with SQL Server.
- Aurora MySQL doesn't support DDL or system event triggers.
- Aurora MySQL supports BEFORE triggers; SQL Server doesn't support BEFORE triggers. l Aurora MySQL supports full run 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 turned on, MySQL SUPER privilege is required to run stored routines. However, you can run stored routines with binary logging enabled without SUPER privilege by setting `thelog_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 the `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 `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;
```

For the preceding example, the result looks as shown following.

```
InvoiceID  Customer  TotalAmount  DeleteDate      DeletedBy
3          James     677.22       20180224 13:02  George
```

> **Note**
>
> Additional code changes were required for this example because the `GETDATE()` function isn't supported by MySQL. For more information, see Date and Time Functions.

Summary

Feature	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.

Feature	SQL Server	Aurora MySQL	Workaround
			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	Make sure that you modify the trigger code to use `NEW` instead of `INSERTED`, and `OLD` instead of `DELETED`.

Feature	SQL Server	Aurora MySQL	Workaround
System event triggers	DDL, DCL and other event types	Not supported	
Trigger run phase	AFTER and INSTEAD OF	AFTER and BEFORE	For INSTEAD OF triggers, make sure that your modify the trigger code to remove the explicit run of the calling DML, which isn't 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 run completes .
Multi-trigger run order	Can only set first and last using sp_settriggerorder .	Can set any run order using PRECEDS and FOLLOWS.	Update the trigger code to reflect the desired run order.
Drop a trigger	DROP TRIGGER <trigger name>;	DROP TRIGGER <trigger name>;	Compatible syntax.

Feature	SQL Server	Aurora MySQL	Workaround
Modify trigger code	Use the `ALTER TRIGGER` statement.	Not supported	
Turn on and turn off 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 run. Modify the trigger code using conditional flow control (IF) to query the table and determine whether or not the trigger should run additional code or exit without performing any modifications to the database.
Triggers on views	`INSTEAD OF` triggers only	Not supported	

For more information, see Trigger Syntax and Examples and CREATE TRIGGER Statement in the *MySQL documentation*.

User-Defined Functions

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		User-Defined Functions	Scalar functions only, rewrite inline TVF

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			as views or derived tables, and multi-statement TVF as stored procedures.

SQL Server Usage

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 doesn't cover CLR code objects.

Function invocations can't 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 aren't 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 you run them with the same data. Non-deterministic functions may return different results each time they run. For example, a function that returns the current date or time.

SQL Server supports three types of T-SQL UDFs: scalar functions, table-valued functions, and multi-statement table-valued functions.

SQL Server 2019 adds scalar user-defined functions inlining. Inlining transforms functions into relational expressions and embeds them in the calling SQL query. This transformation improves the performance of workloads that take advantage of scalar UDFs. Scalar UDF inlining facilitates cost-based optimization of operations inside UDFs. The results are efficient, set-oriented, and parallel instead of inefficient, iterative, serial run plans. For more information, see Scalar UDF Inlining in the *SQL Server documentation*.

Scalar User-Defined Functions

Scalar UDFs accept zero or more parameters and return a scalar value. You can use scalar UDFs 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
```

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-Statement User-Defined Table-Valued Functions

Multi-statement table valued UDFs, like inline 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 aren't restricted to a single SELECT statement. They can consist of multiple statements including logic implemented with flow control, complex data processing, security checks, and so on.

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 CREATE FUNCTION (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports the creation of user-defined scalar functions only. There is no support for table-valued functions.

Unlike SQL Server, Aurora MySQL enables routines to read and write data using INSERT, UPDATE, and DELETE. It also allows DDL statements such as CREATE and DROP. Aurora MySQL doesn't

permit stored functions to contain explicit SQL transaction statements such as COMMIT and ROLLBACK.

In Aurora MySQL, you can explicitly 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 doesn't 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 doesn't 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 preceding options are advisory only. The server doesn't 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.
Multi-statement table-valued UDF	N/A	Use stored procedures to populate tables and read from the table directly.
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 CREATE PROCEDURE and CREATE FUNCTION Statements and CREATE FUNCTION Statement for Loadable Functions in the *MySQL documentation*.

User-Defined Types

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		User-Defined Types	Replace scalar UDT with base types. Rewrite stored

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			procedures that use table-type input parameters to use strings with CSV, XML, or JSON, or to process row-by-row. For more information, see Stored Procedures.

SQL Server Usage

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.

```
CREATE TYPE <type name> {
```

```
FROM <base type> [ NULL | NOT NULL ] | AS TABLE (<Table Definition>)}
```

Examples

User-defined types

Create a `ZipCodeScalar` user-defined type.

```
CREATE TYPE ZipCode
FROM CHAR(5)
NOT NULL
```

Use the `ZipCodetype` 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);
```

For the preceding example, the following error message appears. It indicates that NULL values for `ZipCodeare` aren't allowed.

```
Msg 515, Level 16, State 2, Line 78
Cannot insert the value NULL into column 'ZipCode', table 'tempdb.dbo.UserLocations';
column doesn't 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 CREATE TYPE (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) 5.7 doesn't support user defined types and user defined table valued parameters.

The current documentation doesn't 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 doesn't 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 aren't yet supported in Aurora MySQL. You must convert memory optimized tables to disk based tables.

Examples

Replacing a user-defined type

Replace the ZipCode user-defined type with a base type.

```
CREATE TABLE UserLocations
(
    UserID INT NOT NULL
    PRIMARY KEY,
    /*ZipCode*/ CHAR(5) NOT NULL
);
```

Replacing a table-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 information, 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;
```

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

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 Cursors in the *MySQL documentation*.

Identity and Sequences

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Identity and Sequences	MySQL doesn't support SEQUENCE objects. Rewrite IDENTITY to AUTO_INCREMENT . Last value is evaluated as

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			`MAX(Existing Value) + 1` on every restart.

SQL Server Usage

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 doesn't 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 shouldn't be confused with the `IDENTITY` property of a column.

You can use the `IDENTITY` function 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 created 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**
>
> You can use `NEWSEQUENTIALID` 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;
```

For the preceding example, the result looks as shown following.

```
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;
```

For the preceding example, the result looks as shown following.

```
Col1

9CC01320-C5AA-E811-8440-305B3A017068
```

For more information, see Sequence Numbers and CREATE TABLE (Transact-SQL) IDENTITY (Property) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports automatic sequence generation using the AUTO_INCREMENT column property, similar to the IDENTITY column property in SQL Server.

Aurora MySQL doesn't 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 not mention the relevant column's name in the insert command, in case the column was created with the NOT NULL definition then also inserting a NULL value into an AUTO_INCREMENT column will increment it. 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 doesn't support table-independent SEQUENCE objects, applications that rely on its properties must use a custom solution to meet their requirements.

In Aurora MySQL, you can use AUTO_INCREMENT instead of IDENTITY in SQL Server 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.

Make sure that your AUTO_INCREMENT columns are indexed and don't 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 isn't 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 preceding statement and assigns it to the in-memory autoincrement 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 IDENTITY columns in SQL Server, which by default don't 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;
```

For the preceding example, the result looks as shown following.

```
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;
```

For the preceding example, the result looks as shown following.

```
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%';
```

For the preceding example, the result looks as shown following.

```
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;
```

For the preceding example, the result looks as shown following.

```
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`	

Feature	SQL Server	Aurora MySQL	Comments
		`= <New Seed Value>`	
Column restrictions	Numeric	Numeric, indexed, and no `DEFAULT`	

Controlling seed and interval values	`CREATE/ALTER TABLE`	`auto_increment_increment`
		
		`auto_increment_offset`
			Aurora MySQL settings are global and can't be customized for each column as in SQL Server.

Sequence setting initialization	Maintained through service restarts	Re-initialized every service restart	For more information, see Sequence Value Initialization.
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 Using AUTO_INCREMENT, CREATE TABLE Statement, and AUTO_INCREMENT Handling in InnoDB in the *MySQL documentation*.

Managing Statistics

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Statistics contain only density information, and only for index key columns.

SQL Server Usage

Statistics objects in SQL Server are designed to support 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 doesn't 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 and causing queries to wait or to run asynchronously. When run asynchronously, the triggering run can't 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 Statistics, CREATE STATISTICS (Transact-SQL), and DBCC SHOW_STATISTICS (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 doesn't support independent statistics objects on columns that aren't part of an index.

Typically, administrators change the statistics management mode by setting the global parameter `innodb_stats_persistent = ON`. This option isn't 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` tables.

The following image shows an example of `mysql.innodb_table_stats` content.

database_name	table_name	last_update	n_rows	clustered_index_size	sum_of_other_index_sizes
mysql	aurora_s3_load_history	2021-06-16 01:44:29	0	1	0
mysql	columns_priv	2017-07-14 00:25:33	0	1	0
mysql	db	2017-07-14 00:25:33	0	1	1
mysql	event	2017-10-31 02:20:35	0	1	0
mysql	func	2017-07-14 00:25:33	0	1	0
mysql	host	2017-10-26 21:31:42	0	1	0
mysql	ndb_binlog_index	2017-07-13 18:08:32	0	1	0
mysql	proc	2021-06-16 01:44:40	79	97	0
mysql	procs_priv	2017-10-31 02:20:25	0	1	1
mysql	proxies_priv	2017-07-13 18:08:32	0	1	1
mysql	rds_configuration	2017-07-13 18:08:32	0	1	0
mysql	rds_global_status_history	2017-07-13 18:08:32	0	1	0
mysql	rds_global_status_history_old	2017-07-13 18:08:32	0	1	0
mysql	rds_heartbeat2	2017-07-13 18:08:32	0	1	0
mysql	rds_history	2021-06-16 01:44:30	1	1	0
mysql	rds_replication_status	2017-10-31 02:18:52	0	1	0
mysql	rds_sysinfo	2017-07-13 18:08:32	0	1	0

The following image shows an example of `mysql.innodb_index_stats` content.

```
database_name|table_name                 |index_name     |last_update        |stat_name   |stat_value|sample_size|stat_description
-------------|---------------------------|---------------|-------------------|------------|----------|-----------|-----------------------------------
mysql        |aurora_s3_load_history     |GEN_CLUST_INDEX|2021-06-16 01:44:29|n_diff_pfx01|         0|          1|DB_ROW_ID
mysql        |aurora_s3_load_history     |GEN_CLUST_INDEX|2021-06-16 01:44:29|n_leaf_pages|         1|           |Number of leaf pages in the index
mysql        |aurora_s3_load_history     |GEN_CLUST_INDEX|2021-06-16 01:44:29|size        |         1|           |Number of pages in the index
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx01|         0|          1|Host
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx02|         0|          1|Host,Db
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx03|         0|          1|Host,Db,User
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx04|         0|          1|Host,Db,User,Table_name
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx05|         0|          1|Host,Db,User,Table_name,Column_name
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|n_leaf_pages|         1|           |Number of leaf pages in the index
mysql        |columns_priv               |PRIMARY        |2017-07-14 00:25:33|size        |         1|           |Number of pages in the index
mysql        |db                         |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx01|         0|          1|Host
mysql        |db                         |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx02|         0|          1|Host,Db
mysql        |db                         |PRIMARY        |2017-07-14 00:25:33|n_diff_pfx03|         0|          1|Host,Db,User
mysql        |db                         |PRIMARY        |2017-07-14 00:25:33|n_leaf_pages|         1|           |Number of leaf pages in the index
mysql        |db                         |PRIMARY        |2017-07-14 00:25:33|size        |         1|           |Number of pages in the index
mysql        |db                         |User           |2017-07-14 00:25:33|n_diff_pfx01|         0|          1|User
mysql        |db                         |User           |2017-07-14 00:25:33|n_diff_pfx02|         0|          1|User,Host
mysql        |db                         |User           |2017-07-14 00:25:33|n_diff_pfx03|         0|          1|User,Host,Db
mysql        |db                         |User           |2017-07-14 00:25:33|n_leaf_pages|         1|           |Number of leaf pages in the index
mysql        |db                         |User           |2017-07-14 00:25:33|size        |         1|           |Number of pages in the index
mysql        |event                      |PRIMARY        |2017-10-31 02:20:35|n_diff_pfx01|         0|          1|db
```

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 replicas.

Use `ALTER TABLE ... ANALYZE PARTITION` to analyze one or more individual partitions. For more information, see Storage.

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8 adds new `INFORMATION_SCHEMA.INNODB_CACHED_INDEXES` table which reports the number of index pages cached in the InnoDB buffer pool for each index.

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 doesn't collect detailed key distribution histograms. This difference is critical for understanding run plans and troubleshooting performance issues, which aren't affected by individual values used by query parameters.

Statistics collection is managed at the table level. You can't manage individual statistics objects or individual columns. In most cases, that shouldn't 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.

Auto create statistics	`AUTO_CREATE_STATISTICS` database option	N/A	
Auto update statistics	`AUTO_UPDATE_STATISTICS` database 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 serve the same purpose.

Feature	SQL Server	Aurora MySQL	Comments
Non-persistent statistics	N/A	Use `STATS_PERSISTENT=0` table option	

For more information, see The INFORMATION_SCHEMA STATISTICS Table Configuring Persistent Optimizer Statistics Parameters, Configuring Optimizer Statistics for InnoDB, and Configuring Optimizer Statistics for InnoDB in the *MySQL documentation*.

Configuration

Topics

Upgrades

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
N/A	N/A	N/A	N/A

SQL Server Usage

As a database administrator, from time to time a database upgrade is required, it can be either for security fix, bugs fixes, compliance, or new database features.

The database upgrade approach can be planned to minimize the database downtime and risk. You can perform an upgrade in-place or migrate to a new installation.

Upgrade In-Place

With this approach, we are retaining the current hardware and OS version by adding the new SQL Server binaries on the same server and then upgrade the SQL Server instance.

Before upgrading the database engine, review the SQL Server release notes for the intended target release version for any limitations and known issues to help you plan the upgrade.

In general, these will be the steps to perform the upgrade:

Prerequisites steps

- Back up all SQL Server database files, so that it can be restored if required.

- Run the appropriate Database Console Commands (DBCC CHECKDB) on databases to be upgraded to make sure that they are in a consistent state.
- Ensure to allocate enough disk space for SQL Server components, in addition to user databases.
- Disable all startup stored procedures as stored procedures processed at startup time might block the upgrade process.
- Stop all applications, including all services that have SQL Server dependencies.

Steps for upgrade

- Install new software.
  - Fix issues raised.
  - Set if you prefer to have automatic updates or not.
  - Select products install to upgrade, this is the new binaries installation.
  - Monitor the progress of downloading, extracting, and installing the Setup files.
- Specify the instance of SQL Server to upgrade.
  - On the Select Features page, the features to upgrade will be preselected. The prerequisites for the selected features are displayed on the right-hand pane. SQL Server Setup will install the prerequisite that aren't already installed during the installation step described later in this procedure.
- Review upgrade plan before the actual upgrade.
- Monitor installation progress.

Post upgrade tasks

- Review summary log file for the installation and other important notes.
- Register your servers.

Migrate to a New Installation

This approach maintains the current environment while building a new SQL Server environment. This is usually done when migrating on a new hardware and with a new version of the operating system. In this approach migrate the system objects so that they are same as the existing environment, then migrate the user database either using backup and restore.

For more information, see Upgrade Database Engine in the *SQL Server documentation*.

MySQL Usage

After migrating your databases to Amazon Aurora MySQL-Compatible Edition (Aurora MySQL), you will still need to upgrade your database instance from time to time, for the same reasons you have done it in the past like new features, bugs and security fixes.

In a managed service like Amazon Relational Database Service (Amazon RDS), the upgrade process is much easier and simpler compare to the on-prem SQL Server process.

To determine the current Aurora MySQL version being used, you can use the following AWS CLI command:

```
aws rds describe-db-engine-versions --engine aurora-mysql --query '*[].[EngineVersion]'
 --output text --region your-AWS-Region
```

This can also be queried from the database, using the following queries:

```
SELECT AURORA_VERSION();
```

In an Aurora MySQL version number scheme, for example 2.08.1, the first digit represents the major version. Aurora MySQL version 1 is compatible with MySQL 5.6 and Aurora MySQL version 2 is compatible with MySQL 5.7. To find all Amazon Aurora and MySQL versions mapping, see Database engine updates for Amazon Aurora MySQL version 2.

AWS doesn't apply major version upgrades on Amazon Aurora automatically. Major version upgrades contains new features and functionality which often involves system table and other code changes. These changes may not be backward-compatible with previous versions of the database so applications testing is highly recommended.

Applying automatic minor upgrades can be set by configuring the Amazon RDS instance to allow it.

You can use the following AWS CLI command (Linux) to determine the current automatic upgrade minor versions.

```
aws rds describe-db-engine-versions --output=table --engine mysql --engine-version
 minor-version --region region
```

> **Note**
>
> If no results returned, there is no automatic minor version upgrade available and scheduled.

When enabled, the instance will be automatically upgraded during the scheduled maintenance window.

If you want to upgrade your cluster to a compatible cluster, you can do so by running an upgrade process on the cluster itself. This kind of upgrade is an in-place upgrade, in contrast to upgrades that you do by creating a new cluster. The upgrade is relatively fast because it doesn't require copying all your data to a new cluster volume. In place upgrade preserves the endpoints and set of DB instances for your cluster.

To verify application compatibility, performance and maintenance procedures for the upgraded cluster, you can perform a simulation of the upgrade by doing following

- Clone a cluster.
- Perform an in-place upgrade of the cloned cluster.
- Test applications, performance and so on, using the cloned cluster.
- Resolve any issues, adjust your upgrade plans to account for them.
- Once all the testing looks good, you can perform the in-place upgrade for your production cluster.

For major upgrades, this is the recommended:

- Check for open XA transactions by running the `XA RECOVER` statement. Commit or Rollback the XA transactions before starting the upgrade.
- Check for DDL statements by running a `SHOW PROCESSLIST` statement and looking for `CREATE`, `DROP`, `ALTER`, `RENAME`, and `TRUNCATE` statements in the output. Allow all DDLs to finish before starting the upgrade.
- Check for any uncommitted rows by querying the `INFORMATION_SCHEMA.INNODB_TRX` table. The table contains one row for each transaction. Let the transaction complete or shut down applications that are submitting these changes.

Aurora MySQL performs a major version upgrade in multiple steps. As each step begins, Aurora MySQL records an event. You can monitor the current status and events as they occur on the Events page in the Amazon RDS console.

Amazon Aurora performs a series of checks before beginning the upgrade process. If any issues are detected during these checks, resolve the issue identified in the event details and restart the upgrade process.

Aurora takes the cluster offline, performs a similar set of tests as in the previous step. If no new issues are identified, then Aurora moves with the next step. If any issues are detected during these checks, resolve the issue identified in the event details and restart the upgrade process again.

Aurora backups up the MySQL cluster by creating a snapshot of the cluster volume.

Aurora clones the cluster volume. If any issues are encountered during the upgrade, Aurora reverts to the original data from the cloned cluster volume and brings the cluster back online.

Aurora performs a clean shutdown and it rolls back any uncommitted transactions.

Aurora upgrades the engine version. It installs the binary for the new engine version and uses the writer DB instance to upgrade your data to new to MySQL compatible format. During this stage, Aurora modifies the system tables and performs other conversions that affect the data in your cluster volume.

The upgrade process is completed. Aurora records a final event to indicate that the upgrade process completed successfully. Now DB cluster is running the new major version.

Upgrade can be done through the AWS Console or AWS CLI.

Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to upgrade.
3. Choose **Modify**. The Modify DB cluster page appears.
4. For DB engine version, choose the new version.
5. Choose **Continue** and check the summary of modifications.

6. To apply the changes immediately, choose **Apply immediately**. Choosing this option can cause an outage in some cases. For more information, see Modifying an Amazon Aurora DB cluster.
7. On the confirmation page, review your changes. If they are correct, choose **Modify cluster** to save your changes. Choose **Back** to edit your changes or Cancel to cancel your changes.

AWS CLI

To upgrade the major version of an Aurora MySQL DB cluster, use the AWS CLI `modify-db-cluster` command with the following required parameters:

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
--db-cluster-identifier sample-cluster \
--engine aurora-mysql \
--engine-version 5.7.mysql_aurora.2.09.0 \
--allow-major-version-upgrade \
--apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
--db-cluster-identifier sample-cluster ^
--engine aurora-mysql ^
--engine-version 5.7.mysql_aurora.2.09.0 ^
--allow-major-version-upgrade ^
--apply-immediately
```

Summary

Phase	SQL Server Step	Aurora MySQL
Prerequisite	Perform an instance backup	Run Amazon RDS instance backup
Prerequisite	DBCC for consistent verification	N/A

Phase	SQL Server Step	Aurora MySQL
Prerequisite	Validate disk size and free space	N/A
Prerequisite	Disable all startup stored procedures (if applicable)	N/A
Prerequisite	Stop application and connection	N/A
Prerequisite	Install new software and fix prerequisites errors raised	Commit or rollback uncommitted transactions
Prerequisite	Select instances to upgrade	Select right Amazon RDS instance
Prerequisite	Review pre-upgrade summary	N/A
Runtime	Monitor upgrade progress	Can be reviewed from the console
Post-upgrade	Results	Can be reviewed from the console
Post-upgrade	Register server	N/A
Post-upgrade	Test applications again the new upgraded database	Same
Production deployment	Re-run all steps in a productio n environment	Same

For more information, see Upgrading Amazon Aurora MySQL DB clusters in the *User Guide for Aurora*.

Session Options

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	SET options are significantly different, except for transaction isolation control.

SQL Server Usage

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 the `exec sp_reset_connection` command for connection pooling.

Each session may have multiple run 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 run batch by a client application. Each run scope may contain additional sub-scopes. For example, scripts calling stored procedures or functions.

You can set the global session options, which all run scopes use by default, using the SET T-SQL command. Server code modules such as stored procedures and functions may have their own run 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 run 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, choose **Tools**, **Options**, **Query Execution**, **SQL Server**, **Advanced**.

Syntax

The following example includes categories and settings for the SET command:

```
SET
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
```

For more information, see SET Statements (Transact-SQL) in the *SQL Server documentation*.

SET ROWCOUNT for DML Deprecated Setting

The SET ROWCOUNT for DML statements has been deprecated as of SQL Server 2008.

Up to and including SQL Server 2008 R2, you could limit the number 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 isn't within an explicit transaction).

```
SET ROWCOUNT 5000;
WHILE @@ROWCOUNT > 0
BEGIN
    DELETE FROM MyTable
    WHERE ForDelete = 1;
```

```
END
```

Starting with SQL Server 2012, SET ROWCOUNT is ignored for INSERT, UPDATE and DELETE statements.

You can achieve the same functionality using TOP, which can be converted to LIMIT in Aurora MySQL. For example, you can rewrite the preceding example as shown following:

```
WHILE @@ROWCOUNT > 0
BEGIN
    DELETE TOP (5000)
    FROM MyTable
    WHERE ForDelete = 1;
END
```

AWS Schema Conversion Tool (AWS SCT automatically converts this example to Aurora MySQL.

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 run 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 SET Statements (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 don't exist in MySQL 5.7 standalone installations. These variables are prefixed with Amazon 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 doesn't 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't 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

The use of `SET ROWCOUNT` for DML operations is deprecated as of SQL Server 2008 R2. Code that uses the `SET ROWCOUNT` syntax can't be converted automatically. You can either rewrite to use `TOP` before running AWS SCT, or manually change it afterward.

The following example runs batch DELETE operations in SQL Server using TOP:

```
WHILE @@ROWCOUNT > 0
BEGIN
    DELETE TOP (5000)
    FROM MyTable
    WHERE ForDelete = 1;
END
```

You can rewrite the preceding example to use the LIMIT clause in Aurora MySQL.

```
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 operates different than DATEFIRST . You can use only Sunday and Monday as the start of the week. 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.
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 automatic

Category	SQL Server	Aurora MySQL	Comments
			ally. Syntax is compatible except the addition of the SESSION keyword.
Query run	IDENTITY_INSERT LANGUAGE QUOTED_ID ENTIFIER 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 isn't supported in Aurora MySQL. ANSI_QUOT ES is a value for the sql_mode parameter . Aurora MySQL doesn't add row count information to the errors collection.
Runtime stats	SHOWPLAN_ALL , TEXT, and XML STATISTICS IO , XML, PROFILE, and TIME	See Run Plans	
Miscellaneous	CONCAT_NU LL_YIELDS_NULL ROWCOUNT	N/A sql_selec t_limit	Aurora MySQL always returns NULL for any NULL concatena tion operation . sql_selec t_limit only affects SELECT statements unlike ROWCOUNT, which also affects all DML.

For more information, see Server System Variables in the *MySQL documentation*.

Database Options

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	SQL Server database options are inapplicable to Aurora MySQL.

SQL Server Usage

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.
- Turn on or turn off 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

Use the following syntax to set 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 ALTER DATABASE SET options (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

The concept of a database in Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) is different than SQL Server. In Aurora MySQL, a database is synonymous with a schema. Therefore, the notion of database options isn't 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.

Server Options

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use cluster and database parameter groups.

SQL Server Usage

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.
- Turn on and turn off 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 run `sp_configure`.

> **Note**
>
> Server audits are managed through the T-SQL commands `CREATE` and `ALTER SERVER AUDIT`.

Syntax

```
EXECUTE sp_configure <option>, <value>;
```

Examples

Limit server memory usage to 4 GB.

```
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;
```

View current values.

```
EXECUTE sp_configure
```

For more information, see Server Configuration Options (SQL Server) in the *SQL Server documentation*.

MySQL Usage

The concept of an database in Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) is different than SQL Server. For Aurora MySQL, the terms database and schema are synonymous. Therefore, the concept of database options does isn't applicable to Aurora MySQL.

The Aurora MySQL equivalent of SQL Server 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 standalone installations of MySQL, Amazon Aurora doesn't 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't 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 aren't 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 by
Cluster-level parameters Single cluster parameter group for each Amazon Aurora cluster.	Managed by cluster parameter groups. For example, `aurora_load_from_s3_role` , `default_password_lifetime` , `default_storage_engine` .
Database instance-level parameters	Managed by database parameter groups. For example, `autocommit` , `connect_timeout` , `innodb_change_buffer_max_size` .

Aurora MySQL Parameter Class	Controlled by
You can associate every instance in your Amazon Aurora cluster with a unique database parameter group.	

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:

1. Navigate to **Parameter group** in the Amazon RDS service of the AWS Console.
2. Choose **Create parameter group**.

> **Note**
>
> You can't edit the default parameter group. Create a custom parameter group to apply changes to your Amazon Aurora cluster and its database instances.

3. For **Parameter group family**, choose `aurora-mysql5.7`.
4. For **Type**, choose **DB Parameter Group**. Another option is to choose **Cluster Parameter Group** to modify cluster parameters.
5. Choose **Create**.

Modify a parameter group

The following walkthrough demonstrates how to modify an existing parameter group

1. Navigate to **Parameter group** in the Amazon RDS service of the AWS Console.
2. Choose the name of the parameter group to edit.
3. Choose **Edit parameters**.
4. Change parameter values and choose **Save changes**.

For more information, see Working with parameter groups in the *Amazon Relational Database Service User Guide* and Server System Variables in the *MySQL documentation*.

High Availability and Disaster Recovery

Topics

Backup and Restore

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Backup	Amazon RDS manages storage-level backups.

SQL Server Usage

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. You can then copy these backups 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 aren't supported.

When a database uses the `FULL` recovery model, transaction logs also need to be backed up. Use transaction logs to back 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 Recovery Models (SQL Server) in the *SQL Server documentation*.

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 don't 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 or 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't 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** don't 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 you can use to mirror and stripe backup devices. For more information, see Media Sets, Media Families, and Backup Sets in the *SQL Server documentation*.

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 running the compression algorithm. For more information, see Backup Compression in the *SQL Server documentation*.

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 isn't 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 that use the `SIMPLE` recovery model.

Syntax

The following code examples demonstrate the 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> );
```

The following code examples demonstrate the 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 = <Password>
| { 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 Backup Overview and Restore and Recovery Overview in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 don't 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 Relational Database Service (Amazon RDS).

> **Note**
>
> Starting from Amazon RDS 8.0.21, you can turn on and turn off redo logging using the `ALTER INSTANCE {ENABLE|DISABLE} INNODB REDO_LOG` syntax. This functionality is intended for loading data into a new MySQL instance. Disabling redo logging helps speed up data loading by avoiding redo log writes. The new `INNODB_REDO_LOG_ENABLE` privilege permits enabling and disabling redo logging. The

new `Innodb_redo_log_enabled` status variable permits monitoring redo logging status. For more information, see Disabling Redo Logging in the *MySQL documentation*.

Restoring Data

You can recover databases from Amazon Aurora 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 Amazon 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.

Restoring Database Backups from Amazon S3

You can now restore MySQL 5.7 backups stored on Amazon S3 to Amazon Aurora MySQL-Compatible Edition and Amazon RDS for MySQL.

If you are migrating a MySQL 5.5, 5.6, or 5.7 database to Amazon Aurora MySQL-Compatible Edition or Amazon RDS for MySQL, you can copy database backups to an Amazon S3 bucket and restore them for a faster migration. Both full and incremental backups of your database can be restored. Restoring backups can be considerably quicker than moving data using the mysqldump utility, which replays SQL statements to recreate the database.

For more information, see Restoring a backup into a MySQL DB instance in the *Amazon Relational Database Service User Guide*.

Backtracking an Aurora DB Cluster

With Amazon Aurora with MySQL compatibility, you can backtrack a DB cluster to a specific time, without restoring data from a backup.

Backtracking rewinds the DB cluster to the time you specify. Backtracking isn't a replacement for backing up your DB cluster so that you can restore it to a point in time. However, backtracking provides the following advantages over traditional backup and restore:

- You can easily undo mistakes. If you mistakenly perform a destructive action, such as a DELETE without a WHERE clause, you can backtrack the DB cluster to a time before the destructive action with minimal interruption of service.
- You can backtrack a DB cluster quickly. Restoring a DB cluster to a point in time launches a new DB cluster and restores it from backup data or a DB cluster snapshot, which can take hours. Backtracking a DB cluster doesn't require a new DB cluster and rewinds the DB cluster in minutes.
- You can explore earlier data changes. You can repeatedly backtrack a DB cluster back and forth in time to help determine when a particular data change occurred. For example, you can backtrack a DB cluster three hours and then backtrack forward in time one hour. In this case, the backtrack time is two hours before the original time.

For additional information, see Backtracking an Aurora DB cluster in the *User Guide for Aurora*.

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

You can copy and share database snapshots within the same AWS Region, across AWS Regions, and across AWS accounts. Snapshot sharing provides an authorized AWS account with access to 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 Amazon 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 aren't 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 can't turn off 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.

You can use the `SELECT ... INTO OUTFILE S3` statement to export data out of Aurora MySQL. Also, you can use the `LOAD DATA FROM S3` statement 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 shown following:

```
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 shown following:

```
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, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket in the *User Guide for Aurora*.

As you can see 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 Amazon S3

By default, each file created in an Amazon S3 bucket as a result of the export has a maximal size of 6 GB. 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 should be ran again. However, if some of the data has already been uploaded to the Amazon S3 bucket, it isn't deleted as part of the rollback and you can use a differential approach to upload only the remaining data.

> **Note**
>
> For exports larger than 25 GB, AWS recommends to split the `SELECT ... INTO OUTFILE S3` statement into multiple, smaller batches.

Metadata, such as table schema or file metadata, isn't uploaded by Aurora MySQL to Amazon S3.

Example — 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 Amazon RDS console.

1. Log in to your Management Console, choose **Amazon RDS** , and then choose **Databases**.
2. Choose the relevant DB identifier.
3. Verify the current automatic backup settings.
4. Select the database instance with the writer role and choose **Modify**.
5. Scroll down to the **Backup** section. Select **7 Days** from the list.
6. Choose **Continue**, review the summary, select if to use scheduled maintenance window or to apply immediate and choose **Modify DB instance**.

For more information, 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 Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket in the *User Guide for Aurora*.

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.
Differential Backups	`BACKUP DATABASE ... WITH DIFFERENTIAL`	N/A	
Database snapshots	`BACKUP DATABASE ... WITH COPY_ONLY`	Amazon RDS console or API	The terminology is inconsistent between SQL Server and Aurora MySQL. A

Feature	SQL Server	Aurora MySQL	Comments
			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 the Amazon RDS console or API	
Partial restore	`RESTORE DATABASE... FILE= ...	FILEGROUP = ...`	N/A	

Feature	SQL Server	Aurora MySQL	Comments
Export and import table data	DTS, SSIS, BCP, linked servers to files	`SELECT INTO ... OUTFILE S3` `LOAD DATA FROM S3`	

For more information, see Overview of backing up and restoring an Aurora DB cluster and Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket in the *User Guide for Aurora*.

High Availability Essentials

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Multi replica, scale out solution using Amazon Aurora clusters and Availability Zones.

SQL Server Usage

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.

SQL Server 2017 also adds new Availability Groups functionality which includes read-scale support without a cluster, Minimum Replica Commit Availability Groups setting, and Windows-Linux cross-OS migrations and testing.

SQL Server 2019 introduces support for creating Database Snapshots of databases. A database snapshot is a read-only, static view of a SQL Server database. The database snapshot is transactional consistent with the source database as of the moment of the snapshot's creation. Among other things, some benefits of the database snapshots with regard to high availability are:

- Snapshots can be used for reporting purposes.
- Maintaining historical data for report generation.
- Using a mirror database that you are maintaining for availability purposes to offload reporting.

For more information, see Database Snapshots in the *SQL Server documentation*.

SQL Server 2019 introduces secondary to primary connection redirection for Always On Availability Groups. It allows client application connections to be directed to the primary replica regardless of the target server specified in the connections string. The connection string can target a secondary replica. Using the right configuration of the availability group replica and the settings in the connection string, the connection can be automatically redirected to the primary replica.

For more information, see Secondary to primary replica read/write connection redirection in the *SQL Server documentation*.

Always On Failover Cluster Instances

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 hang up. 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 server if 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 Always On Failover Cluster Instances in the *SQL Server documentation*.

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 Windows Server Failover Cluster (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 don't 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 Always On availability groups: a high-availability and disaster-recovery solution in the *SQL Server documentation*.

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 Database Mirroring in the *SQL Server documentation*.

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 secondary databases for a single 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 for each secondary server. Unlike FCI and Always On Availability Groups, log shipping solutions don't 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 Configure Log Shippingin the *SQL Server documentation*.

The main characteristics of Log Shipping solutions are:

- Provides redundancy 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, see About Log Shipping in the *SQL Server documentation*.

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 Business continuity and database recovery in the *SQL Server documentation*.

MySQL Usage

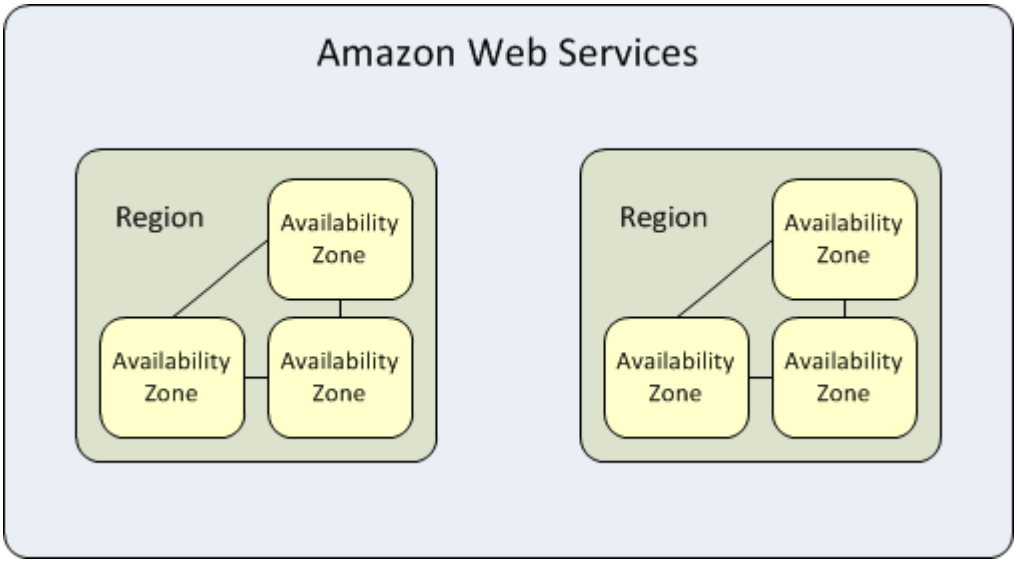
Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) is a fully managed Platform as a Service (PaaS) providing high availability capabilities. Amazon Relational Database Service (Amazon RDS) provides database and instance administration functionality for provisioning, patching, backup, recovery, failure 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 Relational Database Service (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 aren't 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 Resource locations in the *User Guide for Linux Instances*.

When you create a database instance, you can specify an availability zone or use the default **No preference** option. In this case, Amazon chooses the availability zone for you.

You can distribute Aurora MySQL instances across multiple availability zones. You can design applications 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.

You can use elastic IP addresses 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 Elastic IP addresses in the *User Guide for Linux Instances*.

An Availability Zone is represented by a region code followed by a letter identifier. For example, `us-east-1a`.

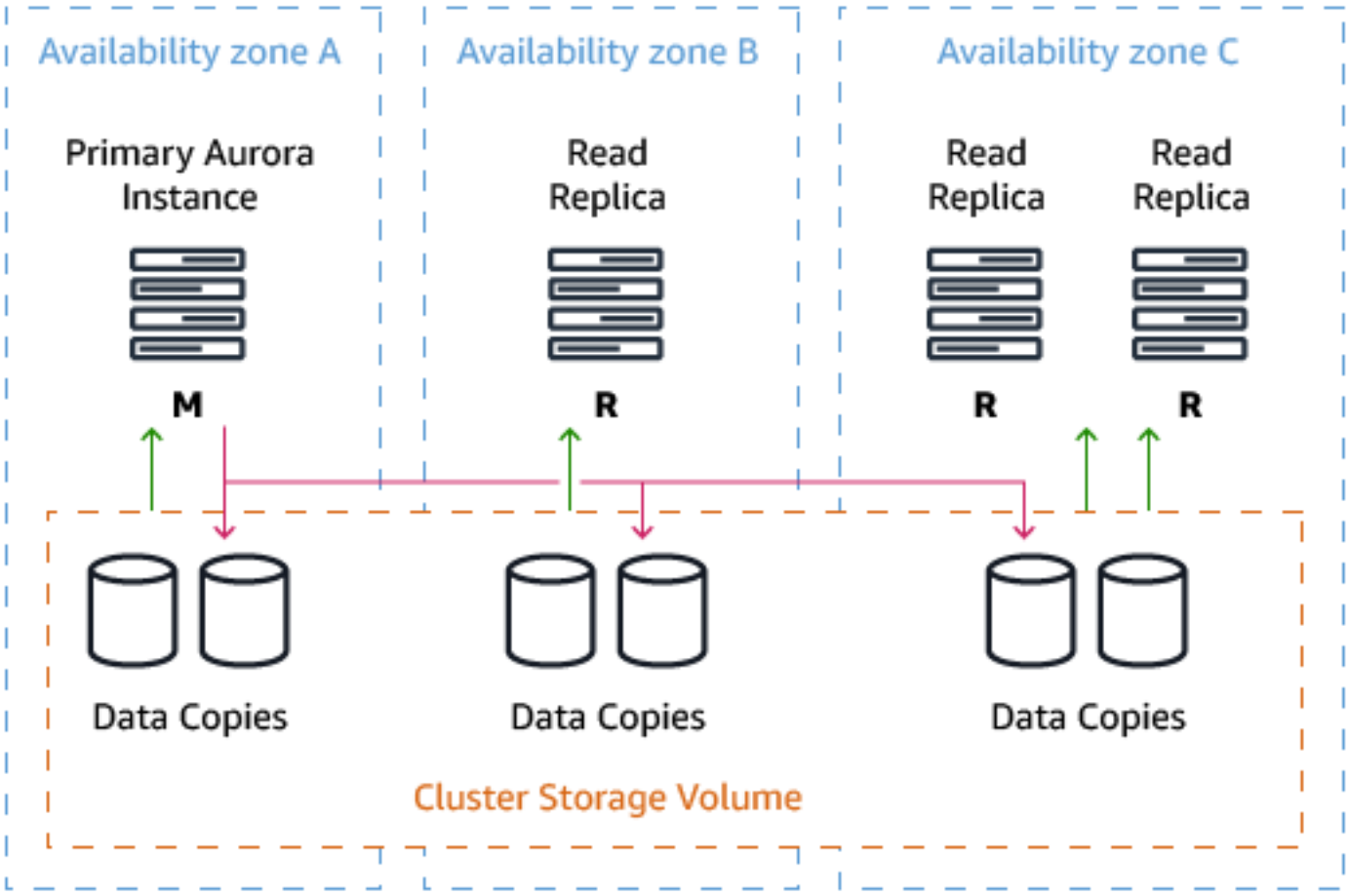
> **Note**
>
> To guarantee even resource distribution across Availability Zones for a region, Amazon RDS independently maps Availability Zones to identifiers for each account. For example, the Availability Zone us-east-1a for one account might not be in the same location as us-east-1a for another account. Users can't 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 Amazon 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 Amazon Aurora DB cluster has one, and only, one primary instance.
- An Amazon Aurora replica that supports read-only workloads. Every Aurora MySQL database cluster may contain from zero to 15 Amazon Aurora replicas in addition to the primary instance for a total maximum of 16 instances. Amazon Aurora Replicas enable scale-out of read operations by offloading reporting or other read-only processes to multiple replicas. Place Amazon 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 Amazon 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, Amazon Aurora automatically fails over server requests to a new primary instance. An 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. You can accomplish this configuration 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 Amazon RDS console 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 doesn't 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 in the preceding sections.

Amazon Aurora cluster volumes automatically grow to accommodate the growth in size of your databases. An Aurora cluster volume has a maximum size of 64 terabytes (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, see Fault tolerance for an Aurora DB cluster in the *User Guide for Aurora*.

Delayed Replication

> **Note**
>
> Amazon RDS for MySQL now supports delayed replication, which enables you to set a configurable time period for which a read replica lags behind the source database. In a standard MySQL replication configuration, there is minimal replication delay between the source and the replica. With delayed replication, you can introduce an intentional delay as a strategy for disaster recovery. A delay can be helpful when you want to recover from a human error. For example, if someone accidentally drops a table from your primary database, you can stop the replication just before the point at which the table was dropped and promote the replica to become a standalone instance. To assist with this process, Amazon RDS for MySQL now includes a stored procedure that will stop replication once a specified point in the binary log is reached. Refer to the blog post for more details.
> Configuring a read replica for delayed replication is done with stored procedure and can either be performed when the read replica is initially created or be specified for an existing read replica. Delayed replication is available for MySQL version 5.7.22 and later or MySQL 5.6.40 and later in all AWS Regions.
> For more information, see Working with MySQL read replicas in the *Amazon Relational Database Service User Guide*.

For more information, see Fault tolerance for an Aurora DB cluster in the *User Guide for Aurora*.

Examples

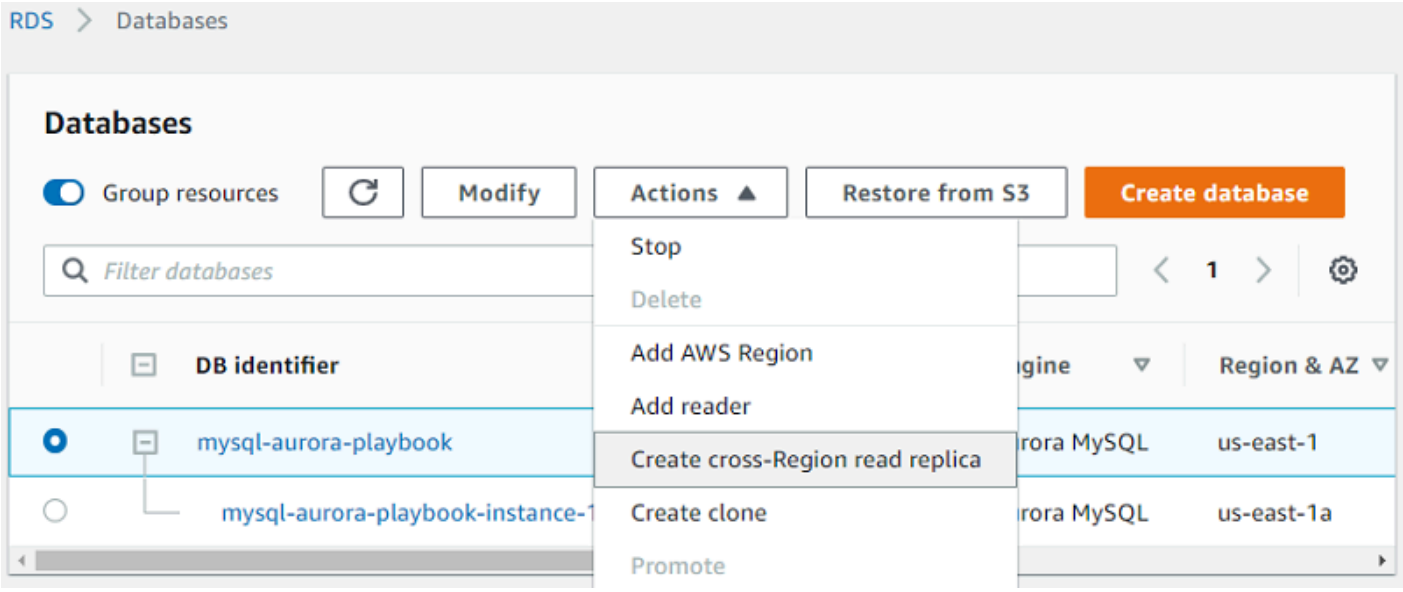
With Amazon RDS and Amazon Aurora for MySQL there are two options for additional reader instance.

Amazon RDS Read Instance Option	Description	Usage
Reader	Another reader instance in the same region	Better for lower costs and latency between instance
Cross-region read replica	Another reader instance in another region	Better when disaster recovery plan requires minimal distance between the primary and the standby instance

The following walkthroughs demonstrate how to create a cross-region read replica and a read replica in the same region.

To create a cross-region read replica

1. Log in to the AWS Console, and choose **RDS**.
2. Select the instance and choose **Instance actions**, **Create cross-region read replica**.

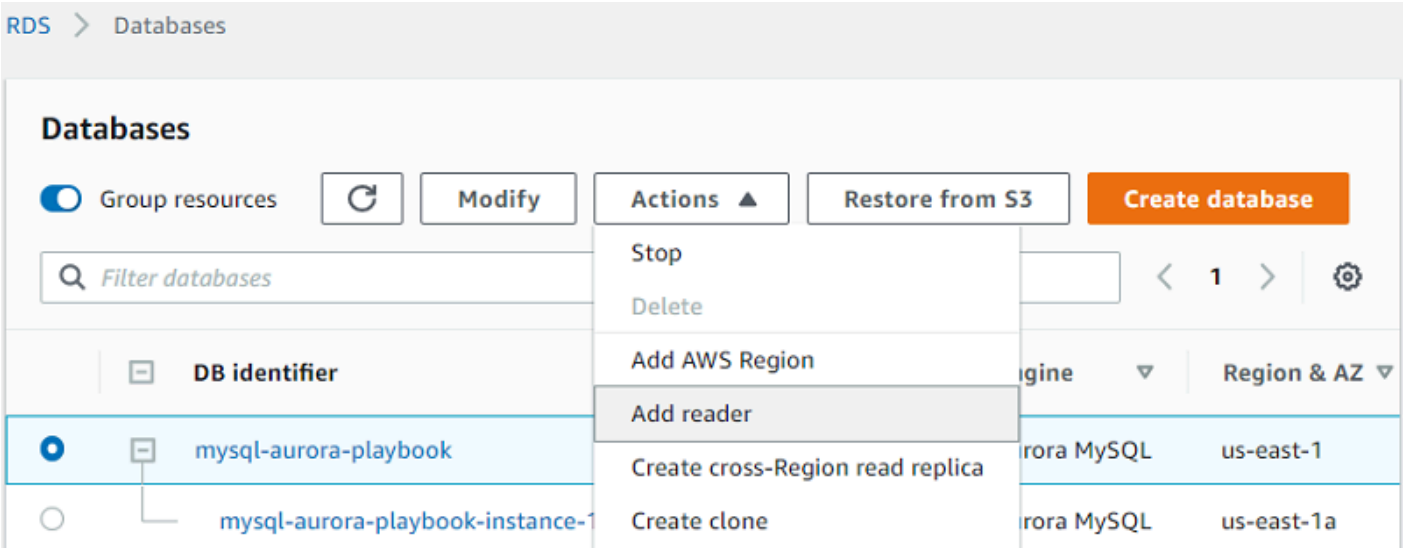


3. On the next page, enter all required details and choose **Create**.

After the replica is created, you can run read and write operations on the primary instance and read-only operations on the replica.

To create a read replica in the same region

1. Log in to the AWS Console, and choose **RDS**.
2. Select the instance and choose **Instance actions**, **Add reader**.



3. On the next page, enter all required details and choose **Create**.

After the replica is created, you can run read and write operations on the primary instance and read-only operations on the replica.

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 Availabil ity Groups	Amazon Aurora Replicas	
Log replication	Log Shipping	N/A	Not applicable. Aurora MySQL handles data replicati

Feature	SQL Server	Aurora MySQL	Comments
			on at the storage level.
Disk error protection	`RESTORE… PAGE=`	Automatically	
Maximum read-only replicas	8 + Primary	15 + Primary	
Failover address	Availability group listener	Cluster endpoint	
Read-only workloads	`READ INTENT` connection	Read endpoint	

For more information, see Amazon Aurora DB clusters in the *User Guide for Aurora* and Regions and Zones in the *User Guide for Linux Instances*.

Indexes

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Indexes	MySQL supports only clustered primary keys. MySQL doesn't support filtered indexes and included columns.

SQL Server Usage

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 1order 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: clustered Indexes and nonclustered 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 nonclustered.

You can create a clustered index 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

The following example creates a clustered index as part of table definition.

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

The following examples 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);
```

Nonclustered 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 nonclustered index contains pointers to the table rows; not the entire row as with a clustered index.

You can create up to 999 nonclustered indexes on a SQL Server table. The type of pointer used at the lead level of a nonclustered 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 nonclustered index updates when rows move around, or the clustered index key value changes.

Both clustered and nonclustered 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

The following example creates a unique nonclustered index as part of table definition.

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

The following examples create a unique nonclustered 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 nonclustered 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, nonclustered 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 to fetch the rest of the data not part of the index. If a query can get all the data it needs from the nonclustered index leaf level, that index is considered a covering index.

Examples

The following example creates a filtered index to exclude NULL values.

```
CREATE NONCLUSTERED INDEX IDX1
ON MyTable(Col2)
WHERE Col2 IS NOT NULL;
```

The following example creates 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

In SQL Server, you can create 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 aren't explicitly specified when data is inserted or updated. This feature is useful when a query filter predicates aren't 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. The format isn't consistent for all rows. Particularly, some rows include country code and some don't:

```
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 can't be used due to the preceding `%` wildcard:

```
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, you can use the following query to search for the customer based on the reverse string, This revers string places the wildcard 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 Clustered and nonclustered indexes described and CREATE INDEX (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports Balanced Tree (b-tree) indexes similar to SQL Server. However, the terminology, use, and options for these indexes are different.

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports invisible indexes. An invisible index isn't used by the optimizer at all but is otherwise maintained normally. Indexes are visible by default. Invisible indexes make it possible to test the effect of removing an index on query performance without making a destructive change that must be undone should the index turn out to be required. For more information, see Invisible Indexes in the *MySQL documentation*.

> **Note**
>
> Amazon RDS for MySQL 8 supports descending indexes: DESC in an index definition is no longer ignored but causes storage of key values in descending order. Previously, indexes could be scanned in reverse order but at a performance penalty. A descending index can be scanned in forward order which is more efficient. Descending indexes also make it possible for the optimizer to use multiple-column indexes when the most efficient scan order mixes ascending order for some columns and descending order for others. For more information, see Descending Indexes in the *MySQL documentation*.

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 aren't configurable; you can't 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 can't 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 don't enforce it. If a table doesn't 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

The following example creates a primary key index as part of the table definition.

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

The following example creates a primary key index for an existing table with no primary key.

```
ALTER TABLE MyTable ADD CONSTRAINT PRIMARY KEY (Col1);
```

Note

You don't need to explicitly name constraints in Aurora MySQL such as 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

The following example creates a unique b-tree index as part of the table definition.

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

The following example creates a non-unique multiple column index on an existing table.

```
CREATE INDEX IDX1 ON MyTable (Col1, Col2) USING BTREE;
```

Note

The `USING` clause isn't 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 can't exceed 64 KB for the entire table. For example, you can create a single generated column with an expression length of 64 KB or create 12 fields with a length of 5 KB 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).

Example

The following example creates 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-	Primary keys only.	

Index feature	SQL Server	Aurora MySQL	Comments
	unique, null or not null.		
Non clustered index supported for:	Table keys, composite or single column, unique and non-unique, null or not null.	Unique constraints, single column and multicolumn.	
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 for each index.	32.	16.	
Index Prefix.	N/A.	Optional for `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY` . Mandatory for `BLOB` and `TEXT`.	
Filtered Indexes.	Supported.	N/A.	
Included columns.	Supported.	N/A.	Add the required columns as index key columns instead of included.

Index feature	SQL Server	Aurora MySQL	Comments
Indexes on BLOBS	N/A.	Supported, limited by maximal index key size.	

For more information, see CREATE INDEX Statement, Column Indexes, and Multiple-Column Indexes in the *MySQL documentation*.

Management

Topics

SQL Server Agent and MySQL Agent

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		SQL Server Agent	For more information, see Alerting and Maintenance Plans.

SQL Server Usage

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 Server Agent jobs under the covers.

Maintenance plans, backups and alerting are covered in separate sections.

For more information, see SQL Server Agent in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 alerts.

Although Amazon Relational Database Service (Amazon RDS) doesn't currently provide an external scheduling agent like SQL Server Agent, CloudWatch Events provides the ability to specify a cron-like schedule to run Lambda functions. This approach requires writing custom code in C#, NodeJS, Java, or Python. Additionally, any task that runs longer than five minutes will not work due to the AWS 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 shouldn't 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 running by a dedicated thread, which can be seen in the process list. The global `event_scheduler` must be turned on explicitly from its 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)
```

Summary

For more information, see CREATE EVENT Statement and Event Scheduler Configuration in the *MySQL documentation*; Amazon CloudWatch and AWS Lambda.

Alerting

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use event notifications subscription with Amazon SNS. For more information, see Using Amazon RDS event notification and Amazon Simple Notification Service.

SQL Server Usage

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 runs 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:

- Run 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
 procedure',
    @job_name = N'Error 20 emergency response';
```

For more information, see Alerts in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't 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 (Amazon RDS) uses Amazon Simple Notification Service (Amazon 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 turn off 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 Amazon 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, make sure that you create the ARN using the Amazon SNS console or the Amazon SNS API.

Examples

The following walkthrough demonstrates how to create an event notification subscription:

1. Sign in to your AWS account, and choose **RDS**.
2. Choose **Events** on the left navigation pane. This screen that presents relevant Amazon RDS events occurs.
3. Choose **Event subscriptions** and then choose **Create event subscription**.
4. Enter the **Name of the subscription** and select a **Target of ARN** or **Email**. For email subscriptions, enter values for **Topic** name and **With these recipients**.
5. Select the event source, choose specific event categories to be monitored, and choose **Create**.
6. On the Amazon RDS dashboard, choose **Recent events**.

For more information, see Using Amazon RDS event notification in the *Amazon Relational Database Service User Guide*.

Database Mail

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		SQL Server Database Mail	Use AWS Lambda integration. For more information, see

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
			Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.

SQL Server Usage

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 added to the deny list.
- 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 runs, it inserts an row to the mail queue and records the email message.

The queue insert operation triggers the run 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 run 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 2008 R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

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 run 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 the `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 Database Mail in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't 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 send email from the database, is to use the AWS Lambda integration.

For more information, see AWS Lambda.

Examples

You can send emails from Aurora MySQL using AWS Lambda integration. For more information, see Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.

ETL

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use AWS Glue for ETL.

SQL Server Usage

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 and replaced with SQL Server Integration Services (SSIS), which was introduced with SQL Server 2005. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

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 SQL Server Integration Services in the *SQL Server documentation* and Data Transformation Services in the *Wikipedia*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) provides AWS 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 doesn't need to be concerned with infrastructure management.

AWS Glue key features include the following.

Integrated Data Catalog

The AWS Glue Data Catalog is a persistent meta-data 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. AWS Glue can automatically calculate statistics and register partitions to make queries more efficient. It maintains a comprehensive schema version history for tracking changes over time.

Automatic Schema Discovery

AWS 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 AWS Glue Data Catalog. You can schedule crawlers or run them on-demand. You can also trigger a crawler when an event occurs to keep meta-data current.

Code Generation

AWS 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. AWS Glue scripts can be generated in Scala or Python and are written for Apache Spark.

Developer Endpoints

When interactively developing Glue ETL code, AWS 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 AWS Glue GitHub repository. For more information, see this repository on *GitHub*.

Flexible Job Scheduler

AWS Glue jobs can be triggered for running 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 AWS Glue. Migration from SQL Server to Aurora MySQL requires rewriting ETL processes to use AWS 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 AWS Glue.

Examples

The following walkthrough describes how to create an AWS Glue job to upload a comma-separated values (CSV) file from Amazon 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 Amazon S3 bucket and create an AWS 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

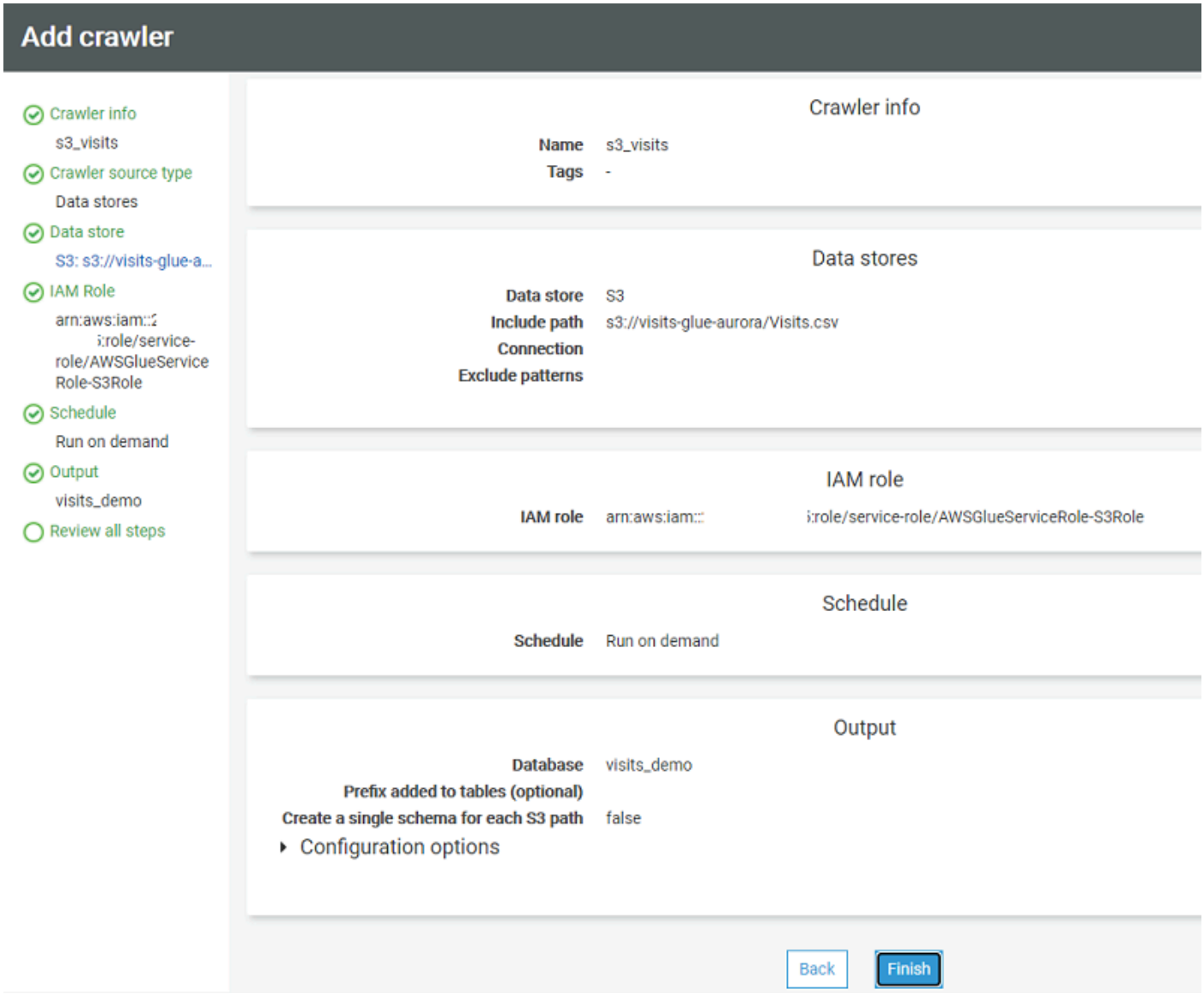
1. In the AWS console, choose **S3**, and then choose **Create bucket**.

> **Note**
>
> This walkthrough demonstrates how to create the buckets and upload the files manually, which is automated using the Amazon S3 API for production ETLs. Using the console to manually run all the settings will help you get familiar with the terminology, concepts, and workflow.

2. Enter a unique name for the bucket, select a region, and define the level of access.
3. Turn on versioning, add tags, turn on server-side encryption, and choose **Create bucket**.
4. On the Amazon S3 Management Console, choose the newly created bucket.
5. On the bucket page, choose **Upload**.
6. Choose **Add files**, select your CSV file, and choose **Upload**.

Step 2 — Add an Amazon Glue Crawler to Discover and Catalog the Visits File

1. In the AWS console, choose **AWS Glue**.
2. Choose **Tables**, and then choose **Add tables using a crawler**.
3. Enter the name of the crawler and choose **Next**.
4. On the **Specify crawler source type** page, leave the default values, and choose **Next**.
5. On the **Add a data store** page, specify a valid Amazon S3 path, and choose **Next**.
6. On the **Choose an IAM role** page, choose an existing IAM role, or create a new IAM role. Choose **Next**.
7. On the **Create a schedule for this crawler** page, choose **Run on demand**, and choose **Next**.
8. On the **Configure the crawler's output** page, choose a database for the crawler's output, enter an optional table prefix for easy reference, and choose **Next**.
9. Review the information that you provided and choose **Finish** to create the crawler.



Step 3 — Run the Amazon Glue Crawler

1. In the AWS console, choose **AWS Glue**, and then choose **Crawlers**.
2. Choose the crawler that you created on the previous step, and choose **Run crawler**.

After the crawler completes, the table should be discovered and recorded in the catalog in the table specified.

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.

If you don't want to add a crawler, you can add tables manually.

1. In the AWS console, choose **AWS Glue**.
2. Choose **Tables**, and then choose **Add table manually**.

Step 4 — Create an ETL Job to Copy the Visits Table to an Aurora MySQL Database

1. In the AWS console, choose **AWS Glue**.
2. Choose **Jobs (legacy)**, and then choose **Add job**.
3. 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 choose **Next**.
4. Select the data source for the job and choose **Next**.
5. On the **Choose a transform type** page, choose **Change schema**.
6. On the **Choose a data target** page, choose **Create tables in your data target**, use the JDBC Data store, and the `gluerds` connection type. Choose **Add connection**.
7. On the **Add connection** page, enter the access details for the Amazon Aurora Instance and choose **Add**.
8. Choose **Next** to display the column mapping between the source and target. Leave the default mapping and data types, and choose **Next**.
9. Review the job properties and choose **Save job and edit script**.
10. Review the generated script and make manual changes if 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.
11. Choose **Run job**.

12. In the AWS console, choose **AWS Glue**, and then choose **Jobs (legacy)**.
13. On the history tab, verify that the job status is set to **Succeeded**.
14. 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 AWS Glue Developer Guide and AWS Glue resources.

Viewing Server Logs

Feature compatibility	AWS SCT / AWS DMS automation level	AWS 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.

SQL Server Usage

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.

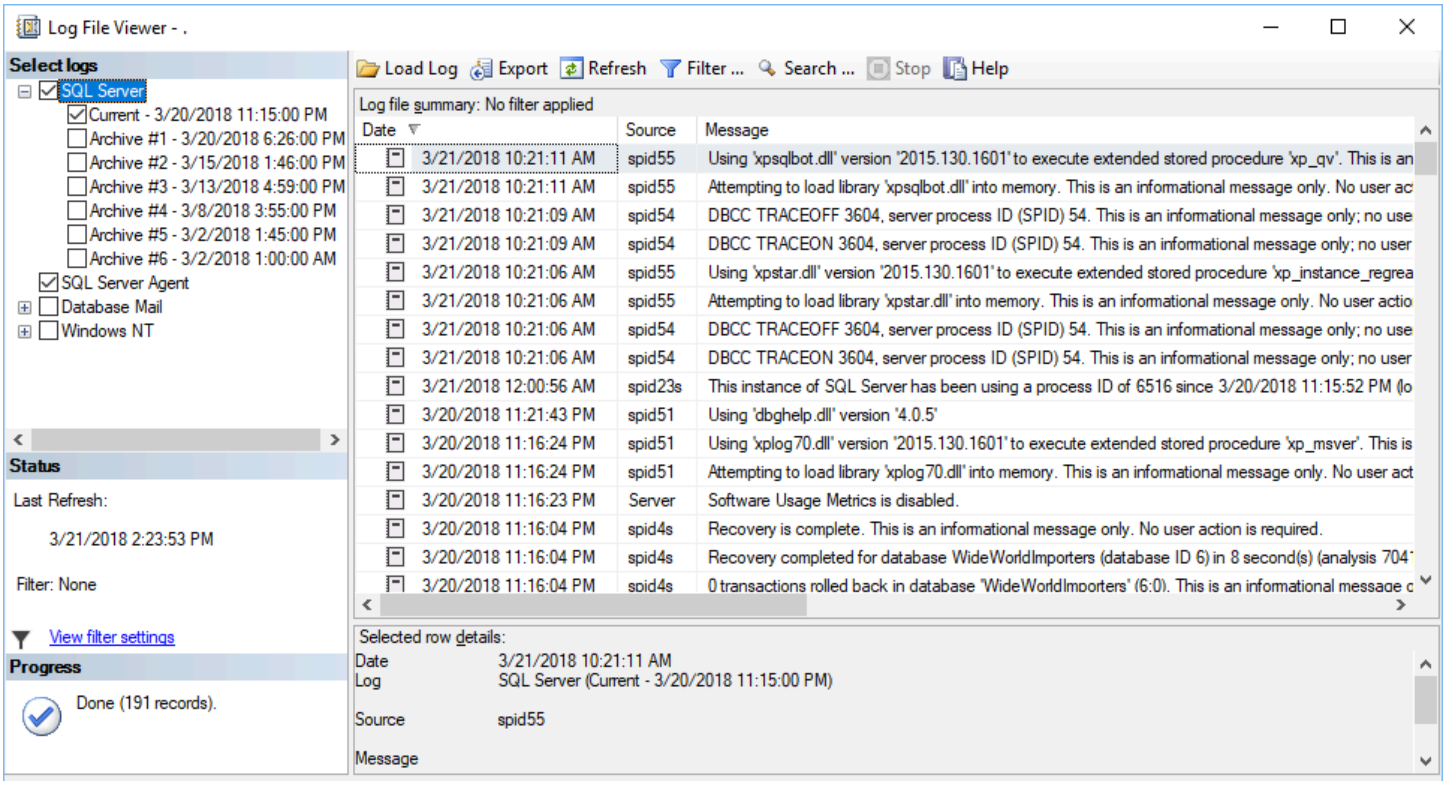
To view the log for SQL Server, SQL Server Agent, Database Mail, and Windows applications, open the SQL Server Management Studio Object Explorer pane, navigate to **Management**, **SQL Server Logs**, and choose the current log.

The following table identifies some common error codes database administrators typically look for in the error logs:

Error code	Error message
1105	Couldn’t 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 the typical log file viewer content:



For more information, see Monitoring the Error Logs in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 information, 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 `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.

The `mysql-error.log` file buffers are flushed every five minutes and are appended to the `filemysql-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.

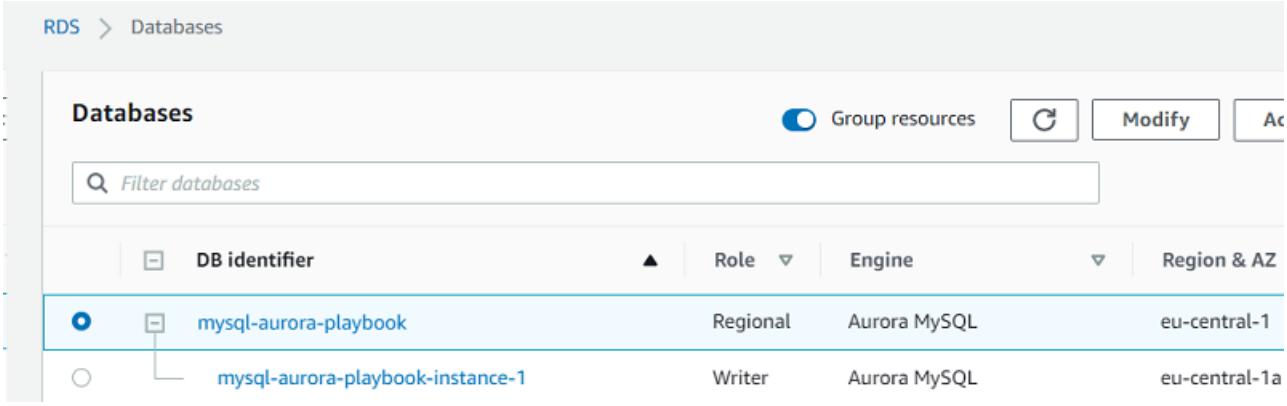
You can turn on and configure the Aurora MySQL Slow Query and general logs to write log entries to a file or a database table by setting the corresponding parameters in the database parameter group. The following list identifies he parameters that control the log options:

- `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 run 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 run time is less than the value of the `long_query_time` parameter.
- `log_output` — Specify one of the following options:
  - **TABLE** — Write general queries to the `mysql.general_log` table and slow queries to the `mysql.slow_log` table. This option is set by default.
  - **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 PostgreSQL error logs in the Amazon RDS console.

1. In the AWS console, choose **RDS**, and then choose **Databases**.
2. Choose the instance for which you want to view the error log.



3. Scroll down to the logs section and choose the log name. The log viewer displays the log content.

For more information, see MySQL database log files in the *Amazon Relational Database Service User Guide*.

Maintenance Plans

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use Amazon RDS for backups. Use SQL for table maintenance.

SQL Server Usage

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 run by SQL Server Agent jobs. You can run them manually or automatically at scheduled time intervals.

SQL Server provides a variety of pre-configured maintenance tasks. You can create custom tasks using TSQL 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.
- Running a T-SQL task.

Maintenance plans can include tasks for operator notifications and history or maintenance cleanup. They can also generate reports and output the contents to a text file or the maintenance plan tables in the `msdb` database.

You can create and manage maintenance plans 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, see SQL Server Agent and MySQL Agent.

Deprecated DBCC Index and Table Maintenance Commands

The DBCC DBREINDEX, INDEXDEFRAG, and SHOWCONTIG commands have been deprecated as of SQL Server 2008R2. For more information, see Deprecated Database Engine Features in SQL Server 2008 R2 in the *SQL Server documentation*.

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 turned off 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 Maintenance Plans in the *SQL Server documentation*.

MySQL Usage

Amazon Relational Database Service (Amazon RDS) performs automated database backups by creating storage volume snapshots that back up entire instances, not individual databases.

Amazon 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.

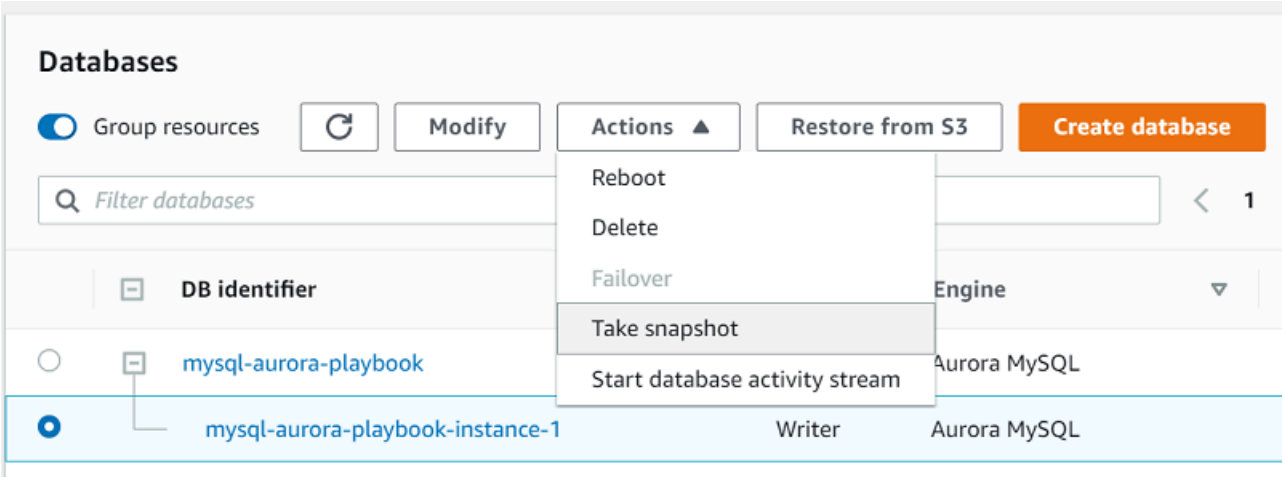
> **Note**
>
> The state of a database instance must be ACTIVE for automated backups to occur.

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 Amazon RDS console

1. In the AWS console, choose **RDS**, and then choose **Databases**.
2. Choose your Aurora PostgreSQL instance, and for **Instance actions** choose **Take snapshot**.



Restore a database from a snapshot

1. In the AWS console, choose **RDS**, and then choose **Snapshots**.

2. Choose the snapshot to restore, and for **Actions** choose **Restore snapshot**.

   This action creates a new instance.
3. Enter the required configuration options in the wizard for creating a new Amazon Aurora database instance. Choose **Restore DB Instance**.

You can also restore a database instance to a point-in-time. For more information, 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 to 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 doesn't 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 can't 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

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 don't 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 run 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`	
Decrease data file size by removing empty pages	`DBCC SHRINKDATABASE` / `DBCC SHRINKFILE`	Files are for each table; not for each 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 and snapshots	For more information, see Backup and Restore.

Task	SQL Server	Aurora MySQL	Comments
Run SQL Server Agent jobs for custom actions	`sp_start_job , scheduled`	Not supported	

For more information, see CHECK TABLE Statement in the *MySQL documentation* and Working with backups in the *Amazon Relational Database Service User Guide*.

Monitoring

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use Amazon CloudWatch service. For more information, see Monitoring metrics in an Amazon RDS instance in the *Amazon Relational Database Service User Guide*.

SQL Server Usage

Monitoring server performance and behavior is a critical aspect of maintaining service quality and includes ad-hoc data collection, ongoing data collection, root cause analysis, preventative actions, and reactive actions. SQL Server provides an array of interfaces to monitor and collect server data.

SQL Server 2017 introduces several new dynamic management views:

- `sys.dm_db_log_stats` exposes summary level attributes and information on transaction log files, helpful for monitoring transaction log health.

- `sys.dm_tran_version_store_space_usage` tracks version store usage for each database, useful for proactively planning `tempdb` sizing based on the version store usage for each database.
- `sys.dm_db_log_info` exposes VLF information to monitor, alert, and avert potential transaction log issues.
- `sys.dm_db_stats_histogram` is a new dynamic management view for examining statistics.
- `sys.dm_os_host_info` provides operating system information for both Windows and Linux.

SQL Server 2019 adds new configuration parameter, `LIGHTWEIGHT_QUERY_PROFILING`. It turns on or turns off the lightweight query profiling infrastructure. The lightweight query profiling infrastructure (LWP) provides query performance data more efficiently than standard profiling mechanisms and is enabled by default. For more information, see Query Profiling Infrastructure in the *SQL Server documentation*.

Windows Operating System Level Tools

You can use the Windows Scheduler to trigger run of script files such as CMD, PowerShell, and so on 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 SQL Server Operating System Related DMVs. For a full list of the DMVs, see SQL Server Operating System Related Dynamic Management Views (Transact-SQL) in the *SQL Server documentation*.

Performance counters exist for 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: Use SQL Server Objects in the *SQL Server documentation*.

SQL Server Extended Events

SQL Server 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=8MB);
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. For more information, see SQL Server Profiler in the *SQL Server documentation*.

SQL Server Management Studio

SQL Server Management Studio (SSMS) 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 run plans.
- **Query Live Statistics** displays query run progress in real time.
- **Replication Monitor** presents a publisher-focused view or distributor-focused view of all replication activity. For more information, see Overview of the Replication Monitor Interface in the *SQL Server documentation*.
- **Log Shipping Monitor** displays the status of any log shipping activity whose status is available from the server instance to which you are connected. For more information, see View the Log Shipping Report (SQL Server Management Studio) in the *SQL Server documentation*.
- **Standard Performance Reports** is set of reports that show the most important performance metrics such as change history, memory usage, activity, transactions, HA, and more.

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. The `sp_monitor` procedure 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, see System Dynamic Management Views in the *SQL Server documentation*.

Trace Flags

You can set trace flags to log events. For example, set trace flag 1204 to log deadlock information. For more information, see DBCC TRACEON - Trace Flags (Transact-SQL) in the *SQL Server documentation*.

SQL Server Query Store

Query Store is a database-level framework supporting automatic collection of queries, run plans, and run time statistics. This data is stored in system tables. You can use this data to diagnose performance issues, understand patterns, and understand trends. It can also be set to automatically revert plans when a performance regression is detected.

For more information, see Monitoring performance by using the Query Store in the *SQL Server documentation*.

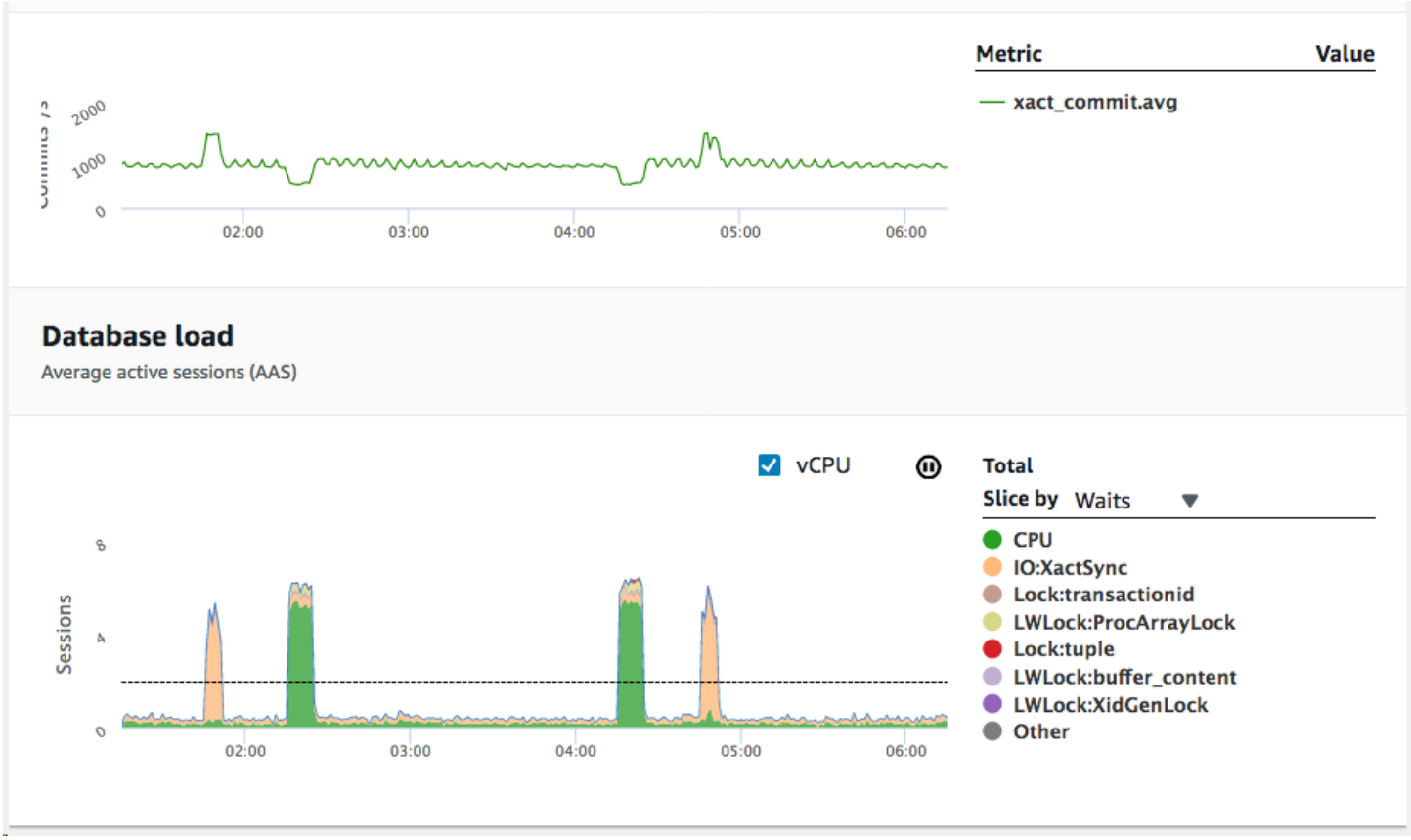
MySQL Usage

The native features for monitoring MySQL databases such as innodb logging and the performance schema are turned off for Aurora MySQL. Most third-party tools that rely on these features can't be used. Some vendors provide monitoring services specifically for Aurora MySQL.

However, Amazon RDS provides a very rich monitoring infrastructure for Aurora MySQL clusters and instances with the native Amazon CloudWatch service.

These services are improved frequently.

Amazon RDS Performance Insights, an advanced database performance monitoring feature that makes it easy to diagnose and solve performance challenges on Amazon RDS databases, now supports additional counter metrics on Amazon RDS for MySQL and Amazon Aurora MySQL-Compatible Edition (Aurora MySQL). With counter metrics, you can customize the Performance Insights dashboard to include up to 10 additional graphs that show a selection from dozens of operating system and database performance metrics. Counter metrics provide additional information that can be correlated with the database load chart to help identify performance issues and analyze performance. For more information, see Performance Insights.



To turn on Performance Insight for your instance, use the step-by-step walkthrough. For more information, see Turning Performance Insights on and off in the *Amazon Relational Database Service User Guide*.

When the Performance Schema is turned on for Aurora MySQL, Performance Insights provides more detailed information. For example, Performance Insights displays DB load categorized by detailed wait events. When Performance Schema is turned off, Performance Insights displays DB load categorized by the list state of the MySQL process.

The Performance Schema stores many useful metrics that will help you analyze and solve performance related issues.

You have the following options for enabling the Performance Schema:

- Allow Performance Insights to manage required parameters automatically. When you create an Aurora MySQL DB instance with Performance Insights enabled, Performance Schema is turned on automatically. In this case, Performance Insights automatically manages your parameters.

> **Note**
>
> In this scenario, Performance Insights changes schema-related parameters on the DB instance. These changes aren't visible in the parameter group associated with the DB instance. However, these changes are visible in the output of the `SHOW GLOBAL VARIABLES` command.

- Set the required parameters yourself. For Performance Insights to list wait events, you must set all parameters as shown in the following table.

Parameter name	Value
`performance_schema`	1 (the Source column has the value engine-default)
`performance-schema-consumer-events-waits-current`	ON
`performance-schema-instrument`	`wait/%=ON`
`performance-schema-consumer-global-instrumentation`	ON
`performance-schema-consumer-thread-instrumentation`	ON

For more information, see Server Options and Performance Schema Quick Start in the *MySQL documentation*, Monitoring metrics in an Amazon RDS instance and Monitoring OS metrics with Enhanced Monitoring in the *Amazon Relational Database Service User Guide*.

Resource Governor

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Use the resource limit for each user.

SQL Server Usage

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 you can change them in real time.

In SQL Server 2019 configurable value for the `REQUEST_MAX_MEMORY_GRANT_PERCENT` option of `CREATE WORKLOAD GROUP` and `ALTER WORKLOAD GROUP` has been changed from an integer to a float data type to allow more granular control of memory limits. For more information, see ALTER WORKLOAD GROUP (Transact-SQL) and CREATE WORKLOAD GROUP (Transact-SQL) in the *SQL Server documentation*.

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

Turn on 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 Resource Governor in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) doesn't support a server-wide, granular, resource-based, workload resource isolation and management capability similar to SQL Server 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 in hour an account is allowed to issue.
- The number of updates in hour an account is allowed to issue.
- The number of times in hour an account can establish a server connection.
- The total number of concurrent server connections allowed for the account.

For more information, 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 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 don't 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 the run; Aurora MySQL raises errors.

Example

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 a classifier function.	For each 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 runs.	Raises an error.	Application retry logic may need to be added.

For more information, see CREATE USER Resource-Limit Options and Setting Account Resource Limits in the *MySQL documentation*.

Linked Servers

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Linked Servers	Data transfer across schemas only, use a custom application solution to access remote instances.

SQL Server Usage

Linked servers enable the database engine to connect to external Object Linking and Embedding for databases (OLE-DB) sources. They are typically used to run 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:

- Reading external data for import or processing.
- Running distributed queries, data modifications, and transactions for enterprise-wide data sources.
- Querying heterogeneous data source using the familiar T-SQL API.

You can configure linked servers 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, and so on.

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**
>
> You can use SQL Server distributed queries 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.

You can manage linked servers using SQL Server Management Studio graphical user interface or T-SQL system stored procedures.

- `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.

You can access linked server data sources from T-SQL using a fully qualified, four-part naming scheme: `<Server Name>.<Database Name>.<Schema Name>.<Object Name>`.

Additionally, you can use the `OPENQUERY` row set function to explicitly invoke pass-through queries on the remote linked server. Also, you can use the `OPENROWSET` and `OPENDATASOURCE` row set functions for one-time 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 a folder.

```
EXEC sp_tables_ex MyTextLinkedServer;
```

Issue a SELECT query using a four-part name.

```
SELECT *
FROM MyTextLinkedServer...[FileName#text];
```

For more information, see sp_addlinkedserver (Transact-SQL) and Distributed Queries Stored Procedures (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition doesn't support remote data access.

Connectivity between schemas is trivial, connectivity to other instances will require an application custom solution.

Scripting

Feature compatibility	AWS SCT / AWS DMS automation level	AWS 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 AWS CLI.

SQL Server Usage

SQL Server supports T-SQL and XQuery scripting within multiple run frameworks such as SQL Server Agent, and stored procedures.

The `SQLCMD` command line utility can also be used to run 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 exposing the entire SQL Server Management Object Model (SMO) as PowerShell paths. Additionally, you can use `cmd` in SQL Server to run specific SQL Server commands.

> **Note**
>
> You can use `Invoke-Sqlcmd` to run scripts using the SQLCMD utility.

The `sqlps` utility launches the PowerShell scripting environment and automatically loads the SQL Server modules. You can launch `sqlps` from a command prompt or from the Object Explorer pane of SQL Server Management Studio. You can run one-time PowerShell commands and script files (for example, `.\SomeFolder\SomeScript.ps1`).

> **Note**
>
> SQL Server Agent supports running PowerShell scripts in job steps. For more information, see SQL Server Agent and MySQL Agent.

SQL Server also supports three types of direct database engine queries: T-SQL, XQuery, and the SQLCMD utility. You can call T-SQL and XQuery 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\SQLServerInstance" -Database
 "MyDB"
```

Get all rows from the `MyTable` table in the `MyDB` database.

```
PS C:\> Read-SqlTableData -ServerInstance MyServer\SQLServerInstance" -DatabaseName
 "MyDB" -TableName "MyTable"
```

For more information, see SQL Server PowerShell, Database Engine Scripting, and sqlcmd Utility in the *SQL Server documentation*.

MySQL Usage

As a Platform as a Service (PaaS), Aurora MySQL accepts connections from any compatible client, but you can't 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 run 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.

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 PostgreSQL and other relational databases. You can use Amazon RDS API to setup, operate, scale, backup, and perform many common administration tasks. The Amazon 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 Programmatic 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 PostgreSQL and other Amazon RDS services. To access the AWS Management Console, sign in to your AWS account, and choose **RDS**.

AWS Command Line Interface

The 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 can't use SQL Server cmdlets in PowerShell.

Amazon RDS Programmatic API

You can use the Amazon RDS API to automate management of database instances and other Amazon RDS objects.

For more information, see Actions, Data Types, Common Parameters, and Common Errors in the *Amazon Relational Database Service API Reference*.

Examples

The following walkthrough describes how to connect to an Aurora MySQL database instance using the MySQL utility.

1. Sign in to your AWS account, choose **RDS**, and then choose **Databases**.
2. Choose the MySQL database you want to connect to and copy the cluster endpoint address.

> **Note**
>
> You can also connect to individual database instances. For more information, see High Availability Essentials.

3. In the command shell, enter the following:

```
mysql -h <mysql-instance-endpoint-address> -P 3306 -u MasterUser
```

In the preceding example, the `-h` parameter is the endpoint Domain Name System (DNS) name of the Aurora MySQL database cluster.

In the preceding example, the `-P` parameter is the port number.

4. 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 MySQL Product Archives, MySQL Workbench 8.0.29, Command Line Interface, and Amazon Relational Database Service API Reference.

Performance Tuning

Topics

Run Plans

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Syntax differences. Completely different optimizer with different operators and rules.

SQL Server Usage

Run 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. Run plans provide critical data for troubleshooting query performance challenges.

SQL Server creates run plans for most queries and returns them to client applications as plain text or XML documents. SQL Server produces a run plan when a query run, but it can also generate estimated plans without running 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 run plan, use the `SET SHOWPLAN_XML`, `SHOWPLAN_ALL`, or `SHOWPLAN_TEXT` statements.

SQL Server 2017 introduces automatic tuning, which notifies users whenever a potential performance issue is detected and lets them apply corrective actions, or lets the database engine automatically fix performance problems. Automatic tuning SQL Server enables users to identify and fix performance issues caused by query run plan choice regressions. For more information, see Automatic tuning in the *SQL Server documentation*.

Examples

Show the estimated run plan for a query.

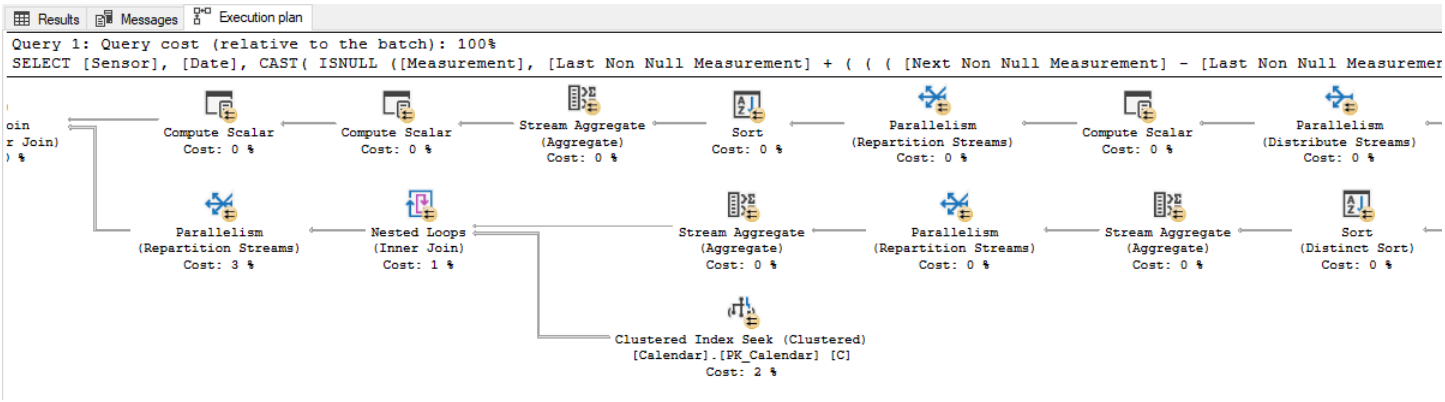
```
SET SHOWPLAN_XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET SHOWPLAN_XML OFF;
```

Actual run plans return after run of the query or batch of queries completes. Actual run plans include run-time statistics about resource usage and warnings. To request the actual run 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 run plan.

Show the actual run plan for a query.

```
SET STATISTICS XML ON;
SELECT *
FROM MyTable
WHERE SomeColumn = 3;
SET STATISTICS XML OFF;
```

The following example shows a partial graphical run plan from SQL Server Management Studio.



For more information, see Display and Save Execution Plans in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) provides the `EXPLAIN/DESCRIBE` statement to display run 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.

When you use `EXPLAIN` with a statement, MySQL returns the run plan generated by the query optimizer. MySQL explains how the statement will be processed including information about table joins and order. When you use `EXPLAIN` with the `FOR CONNECTION` option, it returns the run plan for the statement running 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 run plans.

Note

Amazon Relational Database Service (Amazon RDS) for MySQL implements a new form of the EXPLAIN statement. You can use EXPLAIN ANALYZE in MySQL 8.0.18. This statement provides expanded information about the run of SELECT statements in the TREE format for each iterator used in processing the query and making it possible to compare estimated cost with the actual cost of the query. This information includes startup cost total cost number of rows returned by this iterator and the number of run loops. In MySQL 8.0.21 and later this statement also supports a FORMAT=TREE specifier. TREE is the only supported format. For more information, see Obtaining Information with EXPLAIN ANALYZE in the *MySQL documentation*.

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 run plan for a statement.

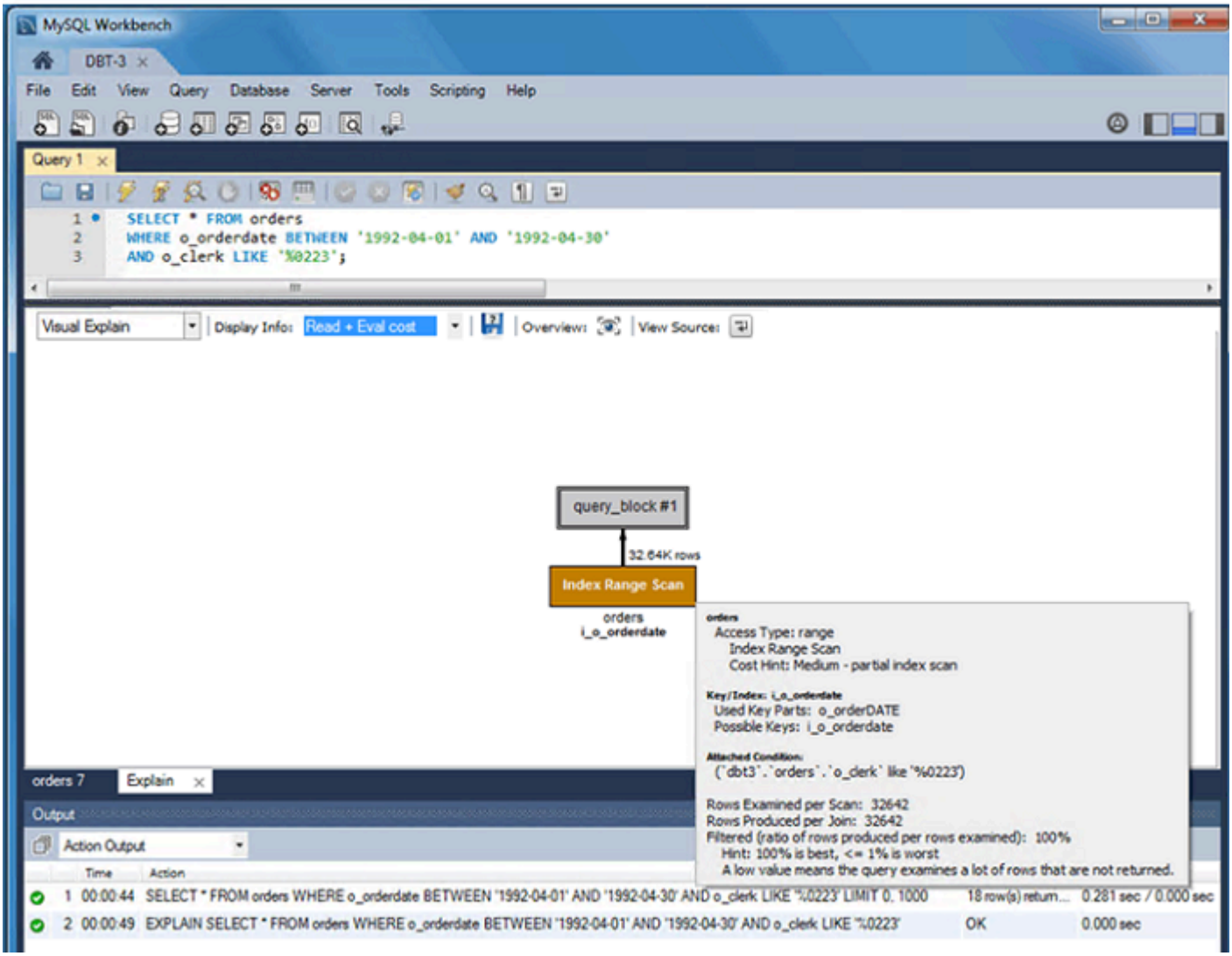
```
CREATE TABLE Employees
(
    EmployeeID INT NOT NULL PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    INDEX USING BTREE(Name)
);
```

```
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 run 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 information, see Query Hints and Plan Guides.

For more information, see EXPLAIN Statement in the *MySQL documentation*.

Query Hints and Plan Guides

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Query Hints	Difference.

SQL Server Usage

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 the run plan.

JOIN Hints

You can explicitly add `LOOP`, `HASH`, `MERGE`, and `REMOTE` hints 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 don't 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`, and `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 aren't an option if you have no control over the source code generating these queries. If an application uses one-time 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 run 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 run.

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 isn't 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

Use the following syntax to create 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>[[,...]])
});
```

Use the following syntax to 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 Hints and Plan Guides in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports two types of hints: optimizer hints and index hints. Unlike SQL Server, MySQL doesn't 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> allow list. Alternatively, indexes can be added to the deny list using the IGNORE keyword.

The FORCE INDEX hint is similar to USE INDEX (index_list), but with strong favor towards seek against scan. This hint is similar to the FORCESEEK hint in SQL Server, although the Aurora MySQL optimizer can choose a scan if other options aren't 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 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 subquery.

- 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>...
```

You can use the following optimizer hints in Aurora MySQL:

Hint name	Description	Applicable scopes
BKA, NO_BKA	Turns on or off Batched Key Access join processing	Query block, table
BNL, NO_BNL	Turns on or off Block Nested-Loop join processing	Query block, table
MAX_EXECUTION_TIME	Limits statement run time	Global
MRR, NO_MRR	Turns on or turns off multi-range read optimization	Table, index
NO_ICP	Turns off index condition push-down optimization	Table, index
NO_RANGE_OPTIMIZATION	Turns off range optimization	Table, index

Hint name	Description	Applicable scopes
QB_NAME	Assigns a logical name to a query block	Query block
SEMIJOIN, NO_SEMIJOIN	Turns on or off 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 as shown in the following example.

```
SELECT /*+ SEMIJOIN(@SubQuery1 FIRSTMATCH, LOOSESCAN) */ *
FROM Table1
WHERE Col1 IN (SELECT /*+ QB_NAME(SubQuery1) */ Col1
FROM t3);
```

Values for MAX_EXECUTION_TIME are measured in seconds and are always global for the entire query.

> **Note**
>
> This option doesn't exist in SQL Server where the run time limit is for the session scope.

For more information, see Controlling the Query Optimizer, Optimizer Hints, Index Hints, and Optimizing Subqueries, Derived Tables, and View References in the *MySQL documentation*.

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 allowed list or USE and disallowed list or IGNORE, as opposed to explicit index approach in SQL Server.

Index hints aren't mandatory instructions. Aurora MySQL has some room to choose alternatives if it can't use the hinted index. In SQL Server, forcing a non valid 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
Force a specific plan	Plan guides	N/A
Apply hints to a query at runtime	Plan guides	N/A

Feature	SQL Server	Aurora MySQL
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
Allowed list and disallowed 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 Controlling the Query Optimizer, Optimizer Hints, Index Hints, and Optimizing Subqueries, Derived Tables, and View References in the *MySQL documentation*.

Storage

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		Partitioning	More partition types in Aurora MySQL with more restrictions on partitioned tables.

SQL Server Usage

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 isn't 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 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 for each 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 that partition function uses 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 may be any data type that is a valid index column with less than 900 bytes for each 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

The following 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 Partitioned Tables and Indexes, CREATE TABLE (Transact-SQL), CREATE PARTITION SCHEME (Transact-SQL), and CREATE PARTITION FUNCTION (Transact-SQL) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 15,000 partitions in SQL Server, practical partitioning rarely contains more than a few hundred partitions.

> **Note**
>
> In Amazon Relational Database Service (Amazon RDS) for MySQL 8, `ADD PARTITION`, `DROP PARTITION`, `COALESCE PARTITION`, `REORGANIZE PARTITION`, and `REBUILD PARTITION` `ALTER TABLE` options are supported by native partitioning in-place APIs and may be used with `ALGORITHM={COPY|INPLACE}` and `LOCK` clauses. `DROP PARTITION` with `ALGORITHMM=INPLACE` deletes data stored in the partition and drops the partition. However, `DROP PARTITION` with `ALGORITHM=COPY` or `old_alter_table=ON` rebuilds

the partitioned table and attempts to move data from the dropped partition to another partition with a compatible `PARTITION ... VALUES` definition. Data that can't be moved to another partition is deleted.

The following sections describe the types of partitions supported by Aurora MySQL.

Range Partitioning

Range partitions are the equivalent of SQL Server `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't 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. Make sure that `<Column Expression>` returns 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 and List Columns Partitioning

Columns partitioning is a variant of both range and list partitioning. However, for columns partitioning, you can use multiple columns in partitioning keys. All column values are considered for matching to a particular partition.

Both range columns partitioning and list columns partitioning enable you to use non-integer values for defining 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 range or list partitioning, or 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 doesn't 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 can't 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, you 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't be a temporary table and the schema of both tables must be identical. The non-partitioned table can't have a foreign key being referenced, or referencing it. It is critical that all rows in the nonpartitioned 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 doesn't trigger the running of triggers on the partitioned table or the exchanged non-partitioned table.

> **Note**
>
> `AUTO_INCREMENT` columns in the exchanged table are reset when you run the `ALTER TABLE ... EXCHANGE PARTITION` statement. For more information, see Identity and Sequences.

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 or 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 can't be modified, some of the physical aspects of partitioning in SQL Server don't apply to Aurora MySQL. For example, the concept of file groups and assigning partitions to file groups.

Aurora MySQL doesn't support foreign keys partitioned tables. Neither the referencing table nor referenced table can use partitioning. Partitioned tables can't 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 can't be 1ENUM 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. In Aurora MySQL, 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 nonpartitioned 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.

Index 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 aren't partitioned, unless explicitly partitioned.	
Partition boundary direction.	LEFT or RIGHT.	RIGHT only.	Only determines to which partition the boundary value itself will go.

Index feature	SQL Server	Aurora MySQL	Comments
Dynamic range partition.	N/A — literal values must be explicitly set in partition function.		
Exchange partition.	Any partition to any partition.	Partition to table (nonpartitioned table).	Only partition to table, no partition to partition switch.
Partition function.	Abstract function object, independent of individual column.	Defined for each partitioned table.	
Partition scheme.	Abstract partition storage mapping object.	N/A	In Aurora MySQL, physical storage is managed by Amazon RDS.
Limitations on partitioned tables.	None — all tables are partitioned.	Extensive — no FK, no full text.	For more informati on, see Restrictions and Limitations on Partitioning.

For more information, see Overview of Partitioning in MySQL, Partition Management, and Partitioning Types in the *MySQL documentation*.

Security

Topics

Column Encryption

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Difference.

SQL Server Usage

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 aren't 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 doesn't cover Transparent Data Encryption (TDE) or AlwaysEncrypted end-to-end encryption.

Syntax

The following example includes the 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.

The following example creates a database master key.

```
USE MyDatabase;
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = '<MyPassword>';
```

The following examples 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;
```

The following example creates 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 Encrypt a Column of Data and Encryption Hierarchy in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 for 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 isn't supported in Aurora MySQL.

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports FIPS mode if compiled using OpenSSL and an OpenSSL library and FIPS Object Module are available at runtime. FIPS mode imposes conditions on cryptographic operations such as restrictions on acceptable encryption algorithms or requirements for longer key lengths. For more information, see FIPS Support in the *MySQL documentation*.

Syntax

The following example shows the 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>])
```

For more information, see AES_ENCRYPT in the *MySQL documentation*.

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 aren't encrypted unless you are using an SSL connection. One benefit of using AWS IAM is that database connections are encrypted with SSL by default.

Examples

The following examples demonstrate how to encrypt an employee Social Security Number.

The following example creates an employees table.

```
CREATE TABLE Employees
(
    EmployeeID INT NOT NULL PRIMARY KEY,
    SSN_Encrypted BINARY(32) NOT NULL
);
```

The following example inserts 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 Encryption and Compression Functions in the *MySQL documentation*.

Data Control Language

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	Difference.

SQL Server Usage

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.

For more information, see Users and Roles.

Syntax

The following examples show the 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 Permissions Hierarchy (Database Engine) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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. Administrators can grant permissions to multiple objects using wildcards.

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 isn't possible to REVOKE the permission for a single table. Instead, 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 the WITH GRANT OPTION clause in SQL Server. 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. For more information, 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.
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.

Permissions	Use to
EXECUTE	Enable the user to run 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 TABLES on 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 primary and secondary servers. Level: Global.
REPLICATION SLAVE	Enable replication replicas to read binary log events from the primary. 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 aren't 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
```

For the preceding example, the result looks as shown following.

```
SQL ERROR [1147][42000]: There is no such grant defined for user TestUser on host '%'
on table 'Invoices'
```

Grant the 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, see GRANT Statement in the *MySQL documentation*.

Transparent Data Encryption

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
	N/A	N/A	Enable encryption when creating the database instance.

SQL Server Usage

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 doesn't require application changes and is completely transparent to users. The storage engine encrypts and decrypts data on-the-fly. Data isn't 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 primary 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 Transparent data encryption (TDE) in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (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 Amazon Relational Database Service (Amazon 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 can't 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. For more information, see Copying a snapshot.

- Encrypted database instances can't be modified to turn off encryption.
- Encrypted Read Replicas must be encrypted with the same key as the source database instance.
- An unencrypted backup or snapshot can't 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 doesn't 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.

Table encryption can now be managed globally by defining and enforcing encryption defaults. The `default_table_encryption` variable defines an encryption default for newly created schemas and general tablespace. The encryption default for a schema can also be defined using the `DEFAULT ENCRYPTION` clause when creating a schema. By default a table inherits the encryption of the schema or general tablespace it is created in. Encryption defaults are enforced by enabling the `table_encryption_privilege_check` variable. The privilege check occurs when creating or altering a schema or general tablespace with an encryption setting that differs from the `default_table_encryption` setting or when creating or altering a table with an encryption setting that differs from the default schema encryption. The `TABLE_ENCRYPTION_ADMIN` privilege permits overriding default encryption settings when `table_encryption_privilege_check` is enabled. For more information, see Defining an Encryption Default for Schemas and General Tablespaces.

Creating an Encryption Key

To create your own key, browse to the Key Management Service (KMS) and choose **Customer managed keys** and create a new key.

1. Choose relevant options and choose **Next**.
2. Define alias as the name of the key and choose **Next**.
3. You can skip **Define Key Administrative Permissions** and choose **Next**.
4. On the next step make sure to assign the key to the relevant users who will need to interact with Amazon Aurora.
5. On the last step you will be able to see the ARN of the key and its account.
6. Choose **Finish** and now this key will be listed in under customer managed keys.

Now you will be able to set Master encryption key by using the ARN of the key that you have created or picking it from the list.

Proceed to finish and launch the instance.

As part of the database settings, you will be prompted to enable encryption and select a master key.

Encryption for an Amazon RDS DB instance can be enabled only during the instance creation.

You can select 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.

Users and Roles

Feature compatibility	AWS SCT / AWS DMS automation level	AWS 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.

SQL Server Usage

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 aren't 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 don't need to migrate these schemas.

Examples

The following example creates a login.

```
CREATE LOGIN MyLogin WITH PASSWORD = 'MyPassword'
```

The following example creates a database user for MyLogin.

```
USE MyDatabase; CREATE USER MyUser FOR LOGIN MyLogin;
```

The following example assigns MyLogin to a server role.

```
ALTER SERVER ROLE dbcreator ADD MEMBER 'MyLogin'
```

The following example assigns `MyUser` to the `db_datareader` role.

```
ALTER ROLE db_datareader ADD MEMBER 'MyUser';
```

For more information, see Database-level roles in the *SQL Server documentation*.

MySQL Usage

Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) supports only Users; Roles aren't 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. For more information, see Data Control Language.

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 don't 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.

Note

Amazon Relational Database Service (Amazon RDS) for MySQL 8 supports roles which are named collections of privileges. Roles can be created and dropped. Roles can have privileges granted to and revoked from them. Roles can be granted to and revoked from user accounts. The active applicable roles for an account can be selected from among those granted to the account and can be changed during sessions for that account. For more information, see Using Roles.

```
CREATE ROLE 'app_developer', 'app_read', 'app_write';
```

Note

Amazon RDS for MySQL 8 incorporates the concept of user account categories with system and regular users distinguished according to whether they have the SYSTEM_USER privilege. For more information, see Account Categories.

```
CREATE USER u1 IDENTIFIED BY 'password';

GRANT ALL ON *.* TO u1 WITH GRANT OPTION;
```

```
-- GRANT ALL includes SYSTEM_USER, so at this point

-- u1 can manipulate system or regular accounts
```

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

In Aurora MySQL, you can assign resource limitations to specific users, similar to SQL Server Resource Governor. For more information, see Resource Governor.

Examples

The following example creates a user, forces a password change, and imposes resource limits.

```
CREATE USER 'Dan'@'localhost'
IDENTIFIED WITH mysql_native_password BY 'Dan''sPassword'
WITH MAX_QUERIES_PER_HOUR 500
PASSWORD EXPIRE;
```

The following example creates 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 Management)
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`
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 What is IAM and IAM Identities (users, user groups, and roles).

Encrypted Connections

Feature compatibility	AWS SCT / AWS DMS automation level	AWS SCT action code index	Key differences
		N/A	N/A

SQL Server Usage

In SQL Server, you can encrypt data across communication channels. Encrypted connections are enabled for an instance of the SQL Server Database Engine and use SQL Server Configuration Manager to specify a certificate.

Make sure that the server has a certificate provisioned. To provision the certificate on the server, make sure to import it into Windows. The client machine must be set up to trust the certificate's root authority.

> **Note**
>
> Starting with SQL Server 2016 (13.x), Secure Sockets Layer (SSL) has been discontinued. Use Transport Layer Security (TLS) instead.

MySQL Usage

MySQL supports encrypted connections between clients and the server using the TLS (Transport Layer Security) protocol. TLS is sometimes referred to as SSL (Secure Sockets Layer) but MySQL doesn't actually use the SSL protocol for encrypted connections because its encryption is weak.

OpenSSL 1.1.1 supports the TLS v1.3 protocol for encrypted connections.

> **Note**
>
> Amazon Relational Database Service (Amazon RDS) for MySQL 8.0.16 and higher supports TLS v1.3 as well if both the server and client are compiled using OpenSSL 1.1.1 or higher.

For more information, see Encrypted Connection TLS Protocols and Ciphers in the *MySQL documentation*.

SQL Server 2018 Deprecated Features List

SQL Server 2018 deprecated feature	Section
TEXT, NTEXT, and IMAGE data types	Data Types
SET ROWCOUNT for DML	Session Options
TIMESTAMP syntax for CREATE TABLE	Creating Tables
DBCC DBREINDEX , INDEXDEFRAG , and SHOWCONTIG	Maintenance Plans
Old SQL Mail	Database Mail
IDENTITY seed, increment, non primary key, and compound	Identity and Sequences
Stored procedures RETURN values	Stored Procedures
GROUP BY ALL, Cube, and Compute By	GROUP BY
DTS	ETL
Old outer join syntax = and =	Table JOIN
'String Alias' = Expression	Migration Quick Tips
DEFAULT keyword for INSERT statements	Migration Quick Tips

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 isn't the same as SQL Server. A database in MySQL is synonymous with *schema*. For more information, see Databases and Schemas.
- You can't create explicit statistics objects in Aurora MySQL. Statistics are collected and maintained for indexes only.
- The equivalent of `CREATE DATABASE... AS SNAPSHOT OF...` in SQL Server resembles Amazon Aurora MySQL-Compatible Edition (Aurora MySQL) Database cloning. However, unlike SQL Server snapshots, which are read-only, Aurora MySQL cloned databases are updatable.
- In Aurora MySQL, *database snapshot* is equivalent to `BACKUP DATABASE... WITH COPY_ONLY` in SQL Server.
- 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 statistics, Aurora MySQL doesn't collect detailed key value distribution; it relies on selectivity only. When troubleshooting runtime, be aware that parameter values are insignificant to plan choices.

SQL

- Triggers work differently in Aurora MySQL. You can run triggers for each row. The syntax for inserted and deleted for each row is `new` and `old`. They always contain 0, or 1 row.
- You can't modify triggers in Aurora MySQL using the `ALTER` command. Drop and replace a trigger instead.

- Aurora MySQL doesn't support @@FETCH_STATUS system parameter for cursors. When you declare cursors in Aurora MySQL, create an explicit HANDLER object, which can set a variable based on the **row not found in cursor** event. For more information, see Stored Procedures.
- To run a stored procedure, use CALL instead of EXECUTE.
- To run a string as a query, use Aurora MySQL Prepared Statements instead of sp_executesql or EXECUTE (<String>).
- 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 doesn't require an explicit action in the trigger. To make changes to data affected by a trigger, you can UPDATE the new and old tables; the changes are persisted.
- Aurora MySQL doesn't support user defined types. Use base types instead and add column constraints as needed.
- The CASE keyword in Aurora MySQL isn't only a conditional expression as in SQL Server. Depending on the context where it appears, you can use CASE for flow control similar to IF <condition> BEGIN <Statement block> END ELSE BEGIN <statement block> END.
- In Aurora MySQL, terminate IF blocks with END IF. Also, terminate WHILE loops with END WHILE. The same rule applies to REPEAT — END REPEAT and LOOP — END LOOP.
- You can't deallocate cursors 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 READ COMMITTED in SQL Server. By default, it also uses consistent reads similar to READ COMMITTED SNAPSHOT in SQL Server.
- 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 doesn't 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 so on. 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.

- You can define collations at the server, database, and column level similar to SQL Server. You can also define collations at the table level.
- In SQL Server, you can use the `DELETE <Table Name>` syntax omitting the `FROM` keyword. This syntax isn't valid 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.
- In Aurora MySQL, you can use multiple rows with `NULL` for a UNIQUE constraint. In SQL Server, you can use only one row. 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 a major difference in the way sequences are maintained. SQL Server caches a set of values in memory and records the last allocation 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 you restart the service, the seed value to `AUTO_INCREMET` is reset to one increment interval larger than the largest existing value. Sequence position isn't maintained across service restarts.
- Parameter names in Aurora MySQL don't require a preceding "@". You can declare local variables such as `DECLARE MyParam1 INTEGER`.
- Parameters that use the @sign don't have to be declared first. You can assign a value directly, which implicitly declares the parameter. For example, `SET @MyParam = 'A'`.
- The local parameter scope isn't limited to an run scope. You can define or set a parameter in one statement, run 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 doesn't support the `MERGE` statement. Use the `REPLACE` statement and the `INSERT… ON DUPLICATE KEY UPDATE` statement as alternatives.
- In Aurora MySQL, you can't concatenate strings with the + operator. In Aurora MySQL, `'A' + 'B'` isn't a valid expression. Use the `CONCAT` function instead. For example, `CONCAT('A', 'B')`.
- Aurora MySQL doesn't 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 doesn't 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 isn't 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 LENGTH in T-SQL.
  - 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 information, see String Functions and Operators.
- Aurora MySQL Date and Time functions differ from SQL Server functions and can cause confusion during migration. Consider the following 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 isn't 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 the object name case.
- In Aurora MySQL, you can't declare variables 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. Make sure that you always use a semicolons at the end of statements.

- The syntax for `CREATE PROCEDURE` requires parenthesis after the procedure name, similar to user-defined functions in SQL Server. You can't use 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.