Amazon Aurora Migration Handbook

May 2017

© 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Notices

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

Contents

Abstract

This paper outlines the best practices for planning, executing, and troubleshooting database migrations from MySQL-compatible and non-MySQL-compatible database products to Amazon Aurora. It also teaches Amazon Aurora database administrators how to diagnose and troubleshoot common migration and replication errors.

Introduction

Migrations are among the most time-consuming tasks handled by database administrators. Although the task has become easier with the advent of managed migration services such as AWS Database Migration Service (AWS DMS), large-scale database migrations still require adequate planning and execution to meet strict compatibility and performance requirements.

This paper examines the following major contributors to the success of every database migration project:

- Factors that justify the migration to Amazon Aurora, such as compatibility, performance, cost, and high availability and durability
- Best practices for choosing the optimal migration method
- Best practices for planning and executing a migration
- Migration troubleshooting hints

Database Migration Considerations

This section discusses important considerations that apply to most database migration projects. For an extended discussion of related topics, see the Amazon Web Services (AWS) whitepaper Migrating Your Databases to Amazon Aurora.[1]

Features and Compatibility

Amazon Aurora is a relational database service that is wire-compatible with MySQL 5.6. This means that most of the drivers, connectors, and tools that you currently use with MySQL can be used with Amazon Aurora with little or no change.

Certain MySQL features, such as the MyISAM storage engine for non-temporary tables, are not available with Amazon Aurora. Although most applications are not affected by such limitations, you should consult the Amazon Aurora feature overview to identify and address any compatibility concerns before migrating.

For more details, see Aurora on Amazon RDS in the *Amazon Relational Database Service (Amazon RDS) User Guide*.[2]

Performance

Performance is often the key motivation behind database migrations. However, deploying your database on Amazon Aurora can be beneficial even if your applications don't have performance issues. For example, Amazon Aurora scalability features can greatly reduce the amount of engineering effort that is required to prepare your database platform for future traffic growth.

You should include benchmarks and performance evaluations in every migration project.

Cost

Amazon Aurora provides consistent high performance together with the security, availability, and reliability of a commercial database at one-tenth the cost.

Amazon Aurora can even be more cost-efficient than open source databases because its high scalability helps you reduce the number of database instances that are required to handle the same workload.

For more details, see the Amazon RDS for Aurora Pricing page.[3]

Availability and Durability

High availability and disaster recovery are important considerations for databases. Your application may already have very strict recovery time objective (RTO) and recovery point objective (RPO) requirements. Amazon Aurora can help you meet or exceed your availability goals.

For more information about durability and availability features in Amazon Aurora, see Aurora on Amazon RDS in the *Amazon RDS User Guide*.[4]

Planning and Testing a Database Migration

After you determine that Amazon Aurora is the right fit for your application, the next step is to decide on a migration approach and create a database migration plan. Here are the suggested high-level steps:

1. Review the available migration techniques described in this document, and choose one that satisfies your requirements.
2. Prepare a migration plan in the form of a step-by-step checklist. A checklist ensures that all migration steps are executed in the correct order and that the migration process flow can be controlled (e.g., suspended or resumed) without the risk of important steps being missed.
3. Prepare a "shadow" checklist with rollback procedures. Ideally, you should be able to roll the migration back to a known, consistent state from any point in the migration checklist.
4. Use the checklist to perform a test migration, and take note of the time required to complete each step. If any missing steps are identified, add them to the checklist. If any issues are identified during the test migration, address them and rerun the test migration.
5. Test all rollback procedures. If any rollback procedure has not been tested successfully, assume that it will not work.
6. After you complete the test migration and become fully comfortable with the migration plan, execute the migration.

Homogeneous Migrations

Amazon Aurora was designed as a drop-in replacement for MySQL 5.6. It offers a wide range of options for homogeneous migrations (e.g., migrations from MySQL and MySQL-compatible databases).

Summary of Available Migration Methods

This section lists common migration sources and the migration methods available to them, in order of preference. Detailed descriptions, step-by-step instructions, and tips for advanced migration scenarios are available in subsequent sections.

Method \ Source	Amazon RDS for MySQL	Self-Managed MySQL-Compatible Database	Non-MySQL-Compatible Database
Amazon RDS Snapshot Migration	Supported (preferred)	Not supported	Not supported
Percona XtraBackup	Not supported	Supported (preferred)	Not supported
Self-Managed Export/Import	Supported	Supported	Supported
AWS Database Migration Service	Supported	Supported	Supported (preferred)

Figure 1: Common migration sources and migration methods for Amazon Aurora

Amazon RDS Snapshot Migration

Compatible sources:

- Amazon RDS for MySQL 5.6
- Amazon RDS for MySQL 5.1 and 5.5 (after upgrading to RDS for MySQL 5.6)

Feature highlights:

- Managed point-and-click service available through the AWS Management Console
- Best migration speed and ease of use of all migration methods
- Can be used with binary log replication for near-zero migration downtime

For details, see Migrating Data from a MySQL DB Instance to an Amazon Aurora DB Cluster in the *Amazon RDS User Guide*.[5]

Percona XtraBackup

Compatible sources:

- On-premises or self-managed MySQL 5.5 and 5.6

Feature highlights:

- Managed backup ingestion from Percona XtraBackup files stored in an Amazon Simple Storage Service (Amazon S3) bucket
- High performance
- Can be used with binary log replication for near-zero migration downtime

For details, see Migrating Data from MySQL by Using an Amazon S3 Bucket in the *Amazon RDS User Guide*.[6]

Self-Managed Export/Import

Compatible sources:

- MySQL and MySQL-compatible databases such as MySQL, MariaDB, or Percona Server, including managed servers such as Amazon RDS for MySQL or MariaDB
- Non-MySQL-compatible databases

Highlights for MySQL-compatible sources:

- Schemas can be migrated as-is without conversion
- Data migration can be performed manually using existing, well-documented command-line utilities
- Can be used with binary log replication for near-zero migration downtime
- Migration performance depends on tooling choices and operator experience

Highlights for non-MySQL-compatible sources:

- Requires manual schema conversion from source database format into MySQL-compatible format.
- Data migration can be performed manually using a universal data format such as comma-separated values (CSV).
- Change data capture (CDC) replication might be possible with third-party tools for near-zero migration downtime.

AWS Database Migration Service

Compatible sources:

- MySQL-compatible and non-MySQL-compatible databases

Feature highlights:

- Supports heterogeneous and homogenous migrations.
- Managed point-and-click data migration service available through the AWS Management Console.
- Schemas must be migrated separately.
- Supports CDC replication for near-zero migration downtime.

For details, see What Is AWS Database Migration Service? in the *AWS DMS User Guide*.[7]

Migrating from Amazon RDS for MySQL

If you are migrating from an RDS MySQL 5.6 database (DB) instance, the recommended approach is to use the snapshot migration feature.

Snapshot migration is a fully managed, point-and-click feature that is available through the AWS Management Console. You can use it to migrate an RDS MySQL 5.6 DB instance snapshot into a new Aurora DB cluster. It is the fastest and easiest to use of all the migration methods described in this document.

For more information about the snapshot migration feature, see Migrating Data to an Amazon Aurora DB Cluster in the *Amazon RDS User Guide*.[8]

This section provides ideas for projects that use the snapshot migration feature. The list-style layout in our example instructions can help you prepare your own migration checklist.

The naming conventions used in this section are as follows:

- **Source RDS DB instance** refers to the RDS MySQL 5.6 DB instance that you are migrating from.
- **Target Aurora DB cluster** refers to the Aurora DB cluster that you are migrating to.

Migrating with Downtime

When migration downtime is acceptable, you can use the following high-level procedure to migrate an RDS MySQL 5.6 DB instance to Amazon Aurora:

1. Stop all write activity against the source RDS DB instance. Database downtime begins here.
2. Take a snapshot of the source RDS DB instance.
3. Wait until the snapshot shows as "Available" in the AWS Management Console.
4. Use the AWS Management Console to migrate the snapshot to a new Aurora DB cluster. For instructions, see Migrating Data to an Amazon Aurora DB Cluster in the *Amazon RDS User Guide*.
5. Wait until the snapshot migration finishes and the target Aurora DB cluster enters the "Available" state. The time to migrate a snapshot primarily depends on the size of the database. You can determine it ahead of the production migration by running a test migration.
6. Configure applications to connect to the newly created target Aurora DB cluster instead of the source RDS DB instance.
7. Resume write activity against the target Aurora DB cluster. Database downtime ends here.

Migrating with Near-Zero Downtime

If prolonged migration downtime is not acceptable, you can perform a near-zero downtime migration through a combination of snapshot migration and binary log replication.

Perform the high-level procedure as follows:

1. On the source RDS DB instance, ensure that automated backups are enabled.
2. Create a Read Replica of the source RDS DB instance.
3. After you create the Read Replica, manually stop replication and obtain binary log coordinates.
4. Take a snapshot of the Read Replica.
5. Use the AWS Management Console to migrate the Read Replica snapshot to a new Aurora DB cluster.
6. Wait until snapshot migration finishes and the target Aurora DB cluster enters the "Available" state.
7. On the target Aurora DB cluster, configure binary log replication from the source RDS DB instance using the binary log coordinates that you obtained in step 3.
8. Wait for the replication to catch up, that is, for the replication lag to reach zero.
9. Begin cut-over by stopping all write activity against the source RDS DB instance. Application downtime begins here.
10. Verify that there is no outstanding replication lag, and then configure applications to connect to the newly created target Aurora DB cluster instead of the source RDS DB instance.
11. Complete cut-over by resuming write activity. Application downtime ends here.
12. Terminate replication between the source RDS DB instance and the target Aurora DB cluster.

For a detailed description of this procedure, see Replication Between Aurora and MySQL or Between Aurora and Another Aurora DB Cluster in the *Amazon RDS User Guide*.[9]

If you don't want to set up replication manually, you can also create an Aurora Read Replica from a source RDS MySQL 5.6 DB instance by using the RDS Management Console.

The RDS automation does the following:

1. Creates a snapshot of the source RDS DB instance.
2. Migrates the snapshot to a new Aurora DB cluster.
3. Establishes binary log replication between the source RDS DB instance and the target Aurora DB cluster.

After replication is established, you can complete the cut-over steps as described previously.

Migrating from Amazon RDS for MySQL Engine Versions Other Than 5.6

Direct snapshot migration is only supported for RDS MySQL 5.6 DB instance snapshots. You can migrate RDS MySQL DB instances that are running other engine versions by using the following procedures.

RDS for MySQL 5.1 and 5.5

Follow these steps to migrate RDS MySQL 5.1 or 5.5 DB instances to Amazon Aurora:

1. Upgrade the RDS MySQL 5.1 or 5.5 DB instance to MySQL 5.6.
    - You can upgrade RDS MySQL 5.5 DB instances directly to MySQL 5.6.
    - You must upgrade RDS MySQL 5.1 DB instances to MySQL 5.5 first, and then to MySQL 5.6.
2. After you upgrade the instance to MySQL 5.6, test your applications against the upgraded database, and address any compatibility or performance concerns.
3. After your application passes the compatibility and performance tests against MySQL 5.6, migrate the RDS MySQL 5.6 DB instance to Amazon Aurora. Depending on your requirements, choose the Migrating with Downtime or Migrating with Near-Zero Downtime procedures described earlier.

For more information about upgrading RDS MySQL engine versions, see Upgrading the MySQL DB Engine in the *Amazon RDS User Guide*.[10]

RDS for MySQL 5.7

For migrations from RDS MySQL 5.7 DB instances, the snapshot migration approach is not supported because the database engine version can't be downgraded to MySQL 5.6.

In this case, we recommend a manual dump-and-import procedure for migrating MySQL-compatible databases, described later in this whitepaper. Such a procedure may be slower than snapshot migration, but you can still perform it with near-zero downtime using binary log replication.

Migrating from MySQL-Compatible Databases

Moving to Amazon Aurora is still a relatively simple process if you are migrating from an RDS MariaDB instance, an RDS MySQL 5.7 DB instance, or a self-managed MySQL-compatible database such as MySQL, MariaDB, or Percona Server running on Amazon Elastic Compute Cloud (Amazon EC2) or on-premises.

There are many techniques you can use to migrate your MySQL-compatible database workload to Amazon Aurora. This section describes various migration options to help you choose the most optimal solution for your use case.

Percona XtraBackup

Amazon Aurora supports migration from Percona XtraBackup files that are stored in an Amazon S3 bucket. Migrating from binary backup files can be significantly faster than migrating from logical schema and data dumps using tools like **mysqldump**. Logical imports work by executing SQL commands to re-create the schema and data from your source database, which involves considerable processing overhead. By comparison, you can use a more efficient binary ingestion method to ingest Percona XtraBackup files.

This migration method is compatible with source servers using MySQL versions 5.5 and 5.6. Migrating from Percona XtraBackup files involves three steps:

1. Use the **innobackupex** tool to create a backup of the source database.

2. Upload backup files to an Amazon S3 bucket.
3. Restore backup files through the AWS Management Console.

For details and step-by-step instructions, see Migrating Data from MySQL by Using an Amazon S3 Bucket in the *Amazon RDS User Guide*.

Self-Managed Export/Import

You can use a variety of export/import tools to migrate your data and schema to Amazon Aurora. The tools can be described as "MySQL native" because they are either part of a MySQL project or were designed specifically for MySQL-compatible databases.

Examples of native migration tools include the following:

- MySQL utilities such as mysqldump, mysqlimport, and mysql command-line client.[11] [12] [13]
- Third-party utilities such as **mydumper** and **myloader**. For details, see this mydumper project page.[14]
- Built-in MySQL commands such as `SELECT INTO OUTFILE` and `LOAD DATA INFILE`.

Native tools are a great option for power users or database administrators who want to maintain full control over the migration process. Self-managed migrations involve more steps and are typically slower than RDS snapshot or Percona XtraBackup migrations, but they offer the best compatibility and flexibility.

For an in-depth discussion of the best practices for self-managed migrations, see the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.[15]

You can execute a self-managed migration with downtime (without replication) or with near-zero downtime (with binary log replication).

Self-Managed Migration with Downtime

The high-level procedure for migrating to Amazon Aurora from a MySQL-compatible database is as follows:

1. Stop all write activity against the source database. Application downtime begins here.
2. Perform a schema and data dump from the source database.
3. Import the dump into the target Aurora DB cluster.
4. Configure applications to connect to the newly created target Aurora DB cluster instead of the source database.
5. Resume write activity. Application downtime ends here.

For an in-depth discussion of performance best practices for self-managed migrations, see the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.

Self-Managed Migration with Near-Zero Downtime

The following is the high-level procedure for near-zero downtime migration into Amazon Aurora from a MySQL-compatible database:

1. On the source database, enable binary logging and ensure that binary log files are retained for at least the amount of time that is required to complete the remaining migration steps.
2. Perform a schema and data export from the source database. Make sure that the export metadata contains binary log coordinates that are required to establish replication at a later time.
3. Import the dump into the target Aurora DB cluster.
4. On the target Aurora DB cluster, configure binary log replication from the source database using the binary log coordinates that you obtained in step 2.
5. Wait for the replication to catch up, that is, for the replication lag to reach zero.
6. Stop all write activity against the source database instance. Application downtime begins here.
7. Double-check that there is no outstanding replication lag. Then configure applications to connect to the newly created target Aurora DB cluster instead of the source database.
8. Resume write activity. Application downtime ends here.

9. Terminate replication between the source database and the target Aurora DB cluster.

For an in-depth discussion of performance best practices of self-managed migrations, see the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.

AWS Database Migration Service

AWS DMS is a managed database migration service that is available through the AWS Management Console. It can perform a range of tasks, from simple migrations with downtime to near-zero downtime migrations using CDC replication.

AWS DMS may be the preferred option if your source database can't be migrated using methods described previously, such as the RDS MySQL 5.6 DB snapshot migration, Percona XtraBackup migration, or native export/import tools.

AWS DMS might also be advantageous if your migration project requires advanced data transformations such as the following:

- Remapping schema or table names
- Advanced data filtering
- Migrating and replicating multiple database servers into a single Aurora DB cluster

Compared to the migration methods described previously, AWS DMS carries certain limitations:

- It does not migrate secondary schema objects such as indexes, foreign key definitions, triggers, or stored procedures. Such objects must be migrated or created manually prior to data migration.
- The CDC replication uses plain SQL statements to apply data changes in the target database. Therefore, it might be slower and more resource-intensive than the native binary log replication in MySQL.

For step-by-step instructions on how to migrate your database using AWS DMS, see the AWS whitepaper Migrating Your Databases to Amazon Aurora.

Heterogeneous Migrations

If you are migrating a non-MySQL-compatible database to Amazon Aurora, several options can help you complete the project quickly and easily.

A heterogeneous migration project can be split into two phases:

1. **Schema migration** to review and convert the source schema objects (e.g., tables, procedures, and triggers) into a MySQL-compatible representation.
2. **Data migration** to populate the newly created schema with data contained in the source database. Optionally, you can use a CDC replication for near-zero downtime migration.

Schema Migration

You must convert database objects such as tables, views, functions, and stored procedures to a MySQL 5.6-compatible format before you can use them with Amazon Aurora.

This section describes two main options for converting schema objects. Whichever migration method you choose, always make sure that the converted objects are not only compatible with Aurora but also follow MySQL's best practices for schema design.

AWS Schema Conversion Tool

The AWS Schema Conversion Tool (AWS SCT) can greatly reduce the engineering effort associated with migrations from Oracle, Microsoft SQL Server, and PostgreSQL. AWS SCT can automatically convert the source database schema and a majority of the custom code, including views, stored procedures, and functions, to a format compatible with Amazon Aurora. Any code that can't be automatically converted is clearly marked so that it can be processed manually.

For more information, see the AWS Schema Conversion Tool User Guide.[16]

For step-by-step instructions on how to convert a non-MySQL-compatible schema using the AWS Schema Conversion Tool, see the AWS whitepaper Migrating Your Databases to Amazon Aurora.

Manual Schema Migration

If your source database is not Oracle, SQL Server, or PostgreSQL, you can either manually rewrite your database object definitions or use available third-party tools to migrate schema to a format compatible with Amazon Aurora.

Many applications use data access layers that abstract schema design from business application code. In such cases, you can consider redesigning your schema objects specifically for Amazon Aurora and adapting the data access layer to the new schema. This might require a greater upfront engineering effort, but it allows the new schema to incorporate all the best practices for performance and scalability.

Data Migration

After the database objects are successfully converted and migrated to Amazon Aurora, it's time to migrate the data itself.

The task of moving data from a non-MySQL-compatible database to Amazon Aurora is best done using AWS DMS. AWS DMS supports initial data migration as well as CDC replication. After the migration task starts, AWS DMS manages all the complexities of the process, including data type transformations, compression, and parallel data transfer. The CDC functionality automatically replicates any changes that are made to the source database during the migration process.

For more information, see the AWS Database Migration Service User Guide.

For step-by-step instructions on how to migrate data from a non-MySQL-compatible database into an Amazon Aurora cluster using AWS DMS, see the AWS whitepaper Migrating Your Databases to Amazon Aurora.

Example Migration Scenarios

There are several approaches for performing both self-managed homogeneous migration and heterogeneous migrations.

Self-Managed Homogeneous Migrations

This section provides examples of migration scenarios from self-managed MySQL-compatible databases to Amazon Aurora.

For an in-depth discussion of homogeneous migration best practices, see the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.

Note that if you are migrating from an Amazon RDS MySQL DB instance, you can use the RDS snapshot migration feature instead of doing a self-managed migration. See the Migrating from Amazon RDS for MySQL section for more details.

Migrating Using Percona XtraBackup

One option for migrating data from MySQL to Amazon Aurora is to use the Percona XtraBackup utility.

Approach

This scenario uses the Percona XtraBackup utility to take a binary backup of the source MySQL database. The backup files are then uploaded to an Amazon S3 bucket and restored into a new Amazon Aurora DB cluster.

When to Use

You can adopt this approach for small- to large-scale migrations when the following conditions are met:

- The source database is a MySQL 5.5 or 5.6 database.
- You have administrative, system-level access to the source database.
- You are migrating database servers in a 1-to-1 fashion: one source MySQL server becomes one new Aurora DB cluster.

When to Consider Other Options

This approach is not currently supported in the following scenarios:

- Migrating into existing Aurora DB clusters.

- Migrating multiple source MySQL servers into a single Aurora DB cluster.

Examples

For a step-by-step example, see Migrating Data from MySQL by Using an Amazon S3 Bucket in the *Amazon RDS User Guide*.

One-Step Migration Using mysqldump

Another migration option uses the mysqldump utility to migrate data from MySQL to Amazon Aurora.

Approach

This scenario uses the **mysqldump** utility to export schema and data definitions from the source server and import them into the target Aurora DB cluster in a single step without creating any intermediate dump files.

When to Use

You can adopt this approach for many small-scale migrations when the following conditions are met:

- The data set is very small (up to 1-2 GB).
- The network connection between source and target databases is fast and stable.
- Migration performance is not critically important, and the cost of re-trying the migration is very low.
- There is no need to do any intermediate schema or data transformations.

When to Consider Other Options

This approach might not be an optimal choice if any of the following conditions are true:

- You are migrating from an RDS MySQL DB instance or a self-managed MySQL 5.5 or 5.6 database. In that case, you might get better results with snapshot migration or Percona XtraBackup, respectively. For more

details, see the Migrating from Amazon RDS for MySQL and Percona XtraBackup sections.

- It is impossible to establish a network connection from a single client instance to source and target databases due to network architecture or security considerations.
- The network connection between source and target databases is unstable or very slow.
- The data set is larger than 10 GB.
- Migration performance is critically important.
- An intermediate dump file is required in order to perform schema or data manipulations before you can import the schema/data.

Notes

For the sake of simplicity, this scenario assumes the following:

1. Migration commands are executed from a client instance running a Linux operating system.
2. The source server is a self-managed MySQL database (e.g., running on Amazon EC2 or on-premises) that is configured to allow connections from the client instance.
3. The target Aurora DB cluster already exists and is configured to allow connections from the client instance. If you don't yet have an Aurora DB cluster, review the step-by-step cluster launch instructions in the *Amazon RDS User Guide*.[17]
4. Export from the source database is performed using a privileged, "super-user" MySQL account. For simplicity, this scenario assumes that the user holds all permissions available in MySQL.
5. Import into Amazon Aurora is performed using the Aurora master user account, that is, the account whose name and password were specified during the cluster launch process.

Examples

The following command, when filled with the source and target server and user information, migrates data and all objects in the named schema(s) between the source and target servers.

```
mysqldump --host=<source_server_address> \
--user=<source_user> \
--password=<source_user_password> \
--databases <schema(s)> \
--single-transaction \
--compress | mysql --host=<target_cluster_endpoint> \
--user=<target_user> \
--password=<target_user_password>
```

Descriptions of the options and option values for the **mysqldump** command are as follows:

- *<source_server_address>*: DNS name or IP address of the source server.
- *<source_user>*: MySQL user account name on the source server.
- *<source_user_password>*: MySQL user account password on the source server.
- *<schema(s)>*: One or more schema names.
- *<target_cluster_endpoint>*: Cluster DNS endpoint of the target Aurora cluster.
- *<target_user>*: Aurora master user name.
- *<target_user_password>*: Aurora master user password.
- *--single-transaction*: Enforces a consistent dump from the source database. Can be skipped if the source database is not receiving any write traffic.
- *--compress*: Enables network data compression.

See the mysqldump documentation for more details.

Example:

```
mysqldump --host=source-mysql.example.com \
--user=mysql_admin_user \
--password=mysql_user_password \
--databases schema1 \
--single-transaction \
--compress | mysql --host=aurora.cluster-xxx.xx.amazonaws.com \
```

```
--user=aurora_master_user \
--password=aurora_user_password
```

Note that this migration approach requires application downtime while the dump and import are in progress. You can avoid application downtime by extending the scenario with MySQL binary log replication. See the Self-Managed Migration with Near-Zero Downtime section for more details.

Migration Using mysqldump with Error Troubleshooting

This scenario is a simple two-step migration with an intermediate dump file. It also demonstrates the discovery, troubleshooting, and resolution of an example import error.

Approach

The scenario uses a **mysqldump** utility to dump schema and data definitions from the source server into a compressed SQL-format dump file. The dump file is then transferred to an Amazon EC2 instance located in the same AWS Region and Availability Zone as the target Aurora DB cluster. After the dump file is transferred, it is imported into Amazon Aurora.

When to Use

You can adopt this approach for many small-scale migrations when the following conditions are met:

- The data set is small (up to a few gigabytes).
- Migration performance is not critically important.
- You want to decouple the dump and import migration stages for better performance and greater control over the migration process.
- An intermediate dump file is required in order to perform schema or data manipulations before the dump can be imported.

When to Consider Other Options

This approach is not an optimal choice if any of the following conditions are true:

- You are migrating from an RDS MySQL DB instance or a self-managed MySQL 5.5 or 5.6 database. In that case, you might get better results with

snapshot migration or Percona XtraBackup, respectively. See the Migrating from Amazon RDS for MySQL and Percona XtraBackup sections for more details.

- The data set is a few gigabytes or larger.
- Migration performance is critically important.

Notes

In order to simplify the demonstration, this scenario assumes the following:

1. Migration commands are executed from client instances running a Linux operating system:
    - **Client instance A** located in the source server's network
    - **Client instance B** located in the same Amazon Virtual Private Cloud (VPC), Availability Zone, and Subnet as the target Aurora DB cluster
2. The source server is a self-managed MySQL database (e.g., running on Amazon EC2 or on-premises) that is configured to allow connections from client instance A.
3. The target Aurora DB cluster already exists and is configured to allow connections from client instance B. If you don't yet have an Aurora DB cluster, review the step-by-step cluster launch instructions in the *Amazon RDS User Guide*.
4. Communication is allowed between client instance A and client instance B.
5. Export from the source database is performed using a privileged, "super-user" MySQL account. For simplicity, the scenario assumes that the user holds all permissions available in MySQL.
6. Import into Amazon Aurora is performed using the master user account, that is, the account whose name and password were specified during the cluster launch process.

Note that this migration approach requires application downtime while the dump and import are in progress. You can avoid application downtime by extending the scenario with MySQL binary log replication. For more details, see the Self-Managed Migration with Near-Zero Downtime section.

Examples

The following command is executed on client instance A and dumps data and all objects in the named schema(s) into an SQL-format dump file.

```
mysqldump --host=<source_server_address> \
--user=<source_user> \
--password=<source_user_password> \
--databases <schema(s)> \
--single-transaction > myschema_dump.sql
```

Descriptions of the options and option values for the **mysqldump** command are as follows:

- *<source_server_address>*: DNS name or IP address of the source server.
- *<source_user>*: MySQL user account name on the source server.
- *<source_user_password>*: MySQL user account password on the source server.
- *<schema(s)>*: One or more schema names.
- *<target_cluster_endpoint>*: Cluster DNS endpoint of the target Aurora cluster.
- *<target_user>*: Aurora master user name.
- *<target_user_password>*: Aurora master user password.
- *--single-transaction*: Enforces a consistent dump from the source database. Can be skipped if the source database is not receiving any write traffic.

See mysqldump in the *MySQL 5.6 Reference Manual* for more details.

Here is an example for schema `myschema` dumped from MySQL server running on IP "11.22.33.44".

```
admin@clientA:~$ mysqldump --host=11.22.33.44 --user=root \
--password=pAssw0rd --databases myschema \
--single-transaction > myschema_dump.sql
```

Check the size of the resulting SQL dump file.

```
admin@clientA:~$ ls -sh myschema_dump.sql
717M myschema_dump.sql
```

The file is over 700 MB, so it's worth compressing before you transfer it to client B (this example uses the **gzip** compression tool).

```
admin@clientA:~$ gzip myschema_dump.sql
admin@clientB:~$ ls -sh myschema_dump.sql.gz
42M myschema_dump.sql.gz
```

Note that you can also create a compressed SQL file in a single step by passing **mysqldump** output through the compression tool.

```
admin@clientA:~$ mysqldump --host=11.22.33.44 \
--user=root --password=pAssw0rd --databases myschema \
--single-transaction |gzip > myschema_dump.sql.gz
```

The file is now transferred to client instance B located close to the target Aurora DB cluster. You can use any available file transfer method (e.g., FTP or Amazon S3). This example uses Secure Copy (SCP) with SSH private key authentication.

```
admin@clientA:~$ scp -i ssh-key.pem myschema_dump.sql.gz \
<clientB_ssh_user>@<clientB_address>:/home/ec2-user/
```

After connecting to client instance B, the file is uncompressed and imported into an Aurora DB cluster. For convenience, you can use a **pv** command-line utility that provides a progress indicator throughout the import.

```
admin@clientB:~$ gunzip myschema_dump.sql.gz
admin@clientB:~$ pv myschema_dump.sql | mysql \
--host=<cluster_endpoint> \
--user=master --password=pAssw0rd
```

```
125MiB 0:00:53 [2.34MiB/s] [===========>                  ] 17%
ETA 0:04:10
```

Unfortunately, just as the import is about to finish, an error message appears.

```
716MiB 0:05:06 [2.34MiB/s]
[==================================================>] 100%
ERROR 1227 (42000) at line 884: Access denied; you need (at
least one of) the SUPER privilege(s) for this operation
```

The message indicates that a statement stored in the dump file has failed because of insufficient privileges. Namely, the statement requires a `SUPER` privilege. This assertion is correct: Amazon Aurora is a managed database service and does not provide the `SUPER` privilege.

The message doesn't show the text of the statement that failed. Fortunately, the message contains a line number (`ERROR … at line 884`), which is the location in the dump file that you should examine.

The following example shows lines 884 through 886 from the dump file.

```
admin@clientB:~$ sed -n 884,886p myschema_dump.sql
/*!50001 CREATE ALGORITHM=UNDEFINED */
/*!50013 DEFINER=`someuser`@`%` SQL SECURITY DEFINER */
/*!50001 VIEW `v1` AS select `myschema`.`t1`.`id` AS
`id`,`myschema`.`t1`.`s1` AS `s1` from `t1` */;
```

The problematic statement is `CREATE VIEW`. Upon closer investigation, you can identify the root cause of the issue as follows:

- The `CREATE VIEW` statement uses a `DEFINER` clause. This clause carries information about the MySQL user account that originally created the view on the source MySQL database.
- The user account that is specified in the `DEFINER` clause (`someuser@%`) is different from the user account that was used to perform the import (`master`).

- In MySQL, only users with the `SUPER` privilege are allowed to specify names other than their own in `DEFINER` clauses. If a user does not have the `SUPER` privilege, only that user's name can be used in `DEFINER` clauses, or the user can skip the `DEFINER` clause altogether.
- The same limitation applies to other stored objects such as functions, triggers, and procedures. For more information, see CREATE VIEW Syntax in the *MySQL 5.6 Reference Manual*.[18]

There are two ways to resolve the issue, depending on whether you are keeping the original DEFINER clauses.

If you are keeping the original DEFINER clauses for imported database objects:

1. Remove all `CREATE` statements with `DEFINER` clauses from the dump file and rerun the import.
2. Make sure that all accounts that are referenced in the `DEFINER` clauses already exist in the target database. Create any accounts that are missing.
3. For each account that is mentioned as `DEFINER` for any database object:
    - Log in to the target database using the user account in question.
    - For all objects that define this user as the `DEFINER`, re-create the objects by running `CREATE` statements manually.

If you are not keeping the original DEFINER clauses:

For each `CREATE` statement found in the dump file:

1. If the statement uses a `DEFINER` clause, remove the clause, but leave the remaining portion of the `CREATE` statement unmodified.
2. Rerun the import.

This demonstration assumed that it is acceptable to remove original object definers. However, you don't want to browse and modify the dump file manually

because it is fairly large and could contain dozens of `CREATE` statements with `DEFINER` clauses. Instead of manipulating the dump file using a text editor, you can use a Perl script to conveniently remove all `DEFINER` clauses at once.

```
admin@clientB:~$ perl -pe 's/\sDEFINER=`[^`]+`@`[^`]+`//' \
< myschema_dump.sql > myschema_dump_nodefiners.sql
```

Perform one additional check to confirm that the `DEFINER` clause is gone.

```
admin@clientB:~$ sed -n 884,886p myschema_dump_nodefiners.sql
/*!50001 CREATE ALGORITHM=UNDEFINED */
/*!50013 SQL SECURITY DEFINER */
/*!50001 VIEW `v1` AS select `myschema`.`t1`.`id` AS
`id`,`myschema`.`t1`.`s1` AS `s1` from `t1` */;
```

Now you can import the modified file. Note that you don't have to remove any database objects that were already imported from the original dump file. The dump file contains `DROP` statements that remove any existing objects prior to re-creating them.

```
admin@clientB:~$ pv myschema_dump.sql | mysql --
host=<cluster_endpoint> \
--user=master --password=pAssw0rd

716MiB 0:05:07 [2.33MiB/s]
[=========================================>] 100%
```

Success! The file was imported without further issues.

Flat-File Migration Using Files in CSV Format

This scenario demonstrates a schema and data migration using flat-file dumps, that is, dumps that do not encapsulate data in SQL statements. Many database administrators prefer to use flat files over SQL-format files for the following reasons:

- Lack of SQL encapsulation results in smaller dump files and reduces processing overhead during import.
- Flat-file dumps are easier to process using OS-level tools; they are also easier to manage (e.g., split or combine).
- Flat-file formats are compatible with a wide range of database engines, both SQL and NoSQL.

Approach

The scenario uses a hybrid migration approach:

- Use the **mysqldump** utility to create a schema-only dump in SQL format. The dump describes the structure of schema objects (e.g., tables, views, and functions) but does not contain data.
- Use `SELECT INTO OUTFILE` SQL commands to create data-only dumps in CSV format. The dumps are created in a one-file-per-table fashion and contain table data only (no schema definitions).

The import phase can be executed in two ways:

- **Traditional approach.** Transfer all dump files to an Amazon EC2 instance located in the same AWS Region and Availability Zone as the target Aurora DB cluster. After transferring the dump files, you can import them into Amazon Aurora using the **mysql** command line client and `LOAD DATA LOCAL INFILE` SQL commands for SQL-format schema dumps and the flat-file data dumps, respectively.

  This is the approach that is demonstrated later in this section.
- **Alternative approach.** Transfer the SQL-format schema dumps to an Amazon EC2 client instance, and import them using the **mysql** command-line client. You can transfer the flat-file data dumps to an Amazon S3 bucket and then import them into Amazon Aurora using `LOAD DATA FROM S3` SQL commands.

For more information, including an example of loading data from Amazon S3, see Loading Data into a DB Cluster from Text Files in an Amazon S3 Bucket in the *Amazon RDS User Guide*.[19]

When to Use

You can adopt this approach for most migration projects where performance and flexibility are important:

- You can dump small data sets and import them one table at a time. You can also run multiple `SELECT INTO OUTFILE` and `LOAD DATA INFILE` operations in parallel for best performance.
- Data that is stored in flat-file dumps is not encapsulated in database-specific SQL statements. Therefore, it can be handled and processed easily by the systems participating in the data exchange.

When to Consider Other Options

You might choose not to use this approach if any of the following conditions are true:

- You are migrating from an RDS MySQL DB instance or a self-managed MySQL 5.5 or 5.6 database. In that case, you might get better results with snapshot migration or Percona XtraBackup, respectively. See the Migrating from Amazon RDS for MySQL and Percona XtraBackup sections for more details.
- The data set is very small and does not require a high-performance migration approach.
- You want the migration process to be as simple as possible and you don't require any of the performance and flexibility benefits listed earlier.

Notes

To simplify the demonstration, this scenario assumes the following:

1. Migration commands are executed from client instances running a Linux operating system:
    - **Client instance A** is located in the source server's network
    - **Client instance B** is located in the same Amazon VPC, Availability Zone, and Subnet as the target Aurora DB cluster

2. The source server is a self-managed MySQL database (e.g., running on Amazon EC2 or on-premises) configured to allow connections from client instance A.
3. The target Aurora DB cluster already exists and is configured to allow connections from client instance B. If you don't have an Aurora DB cluster yet, review the step-by-step cluster launch instructions in the *Amazon RDS User Guide*.
4. Communication is allowed between both client instances.
5. Export from the source database is performed using a privileged, "super user" MySQL account. For simplicity, this scenario assumes that the user holds all permissions available in MySQL.
6. Import into Amazon Aurora is performed using the master user account, that is, the account whose name and password were specified during the cluster launch process.

Note that this migration approach requires application downtime while the dump and import are in progress. You can avoid application downtime by extending the scenario with MySQL binary log replication. See the Self-Managed Migration with Near-Zero Downtime section for more details.

Examples

In this scenario, you migrate a MySQL schema named `myschema`. The first step of the migration is to create a schema-only dump of all objects.

```
mysqldump --host=<source_server_address> \
--user=<source_user> \
--password=<source_user_password> \
--databases <schema(s)> \
--single-transaction \
--no-data > myschema_dump.sql
```

Descriptions of the options and option values for the **mysqldump** command are as follows:

- *<source_server_address>*: DNS name or IP address of the source server.
- *<source_user>*: MySQL user account name on the source server.

- *<source_user_password>*: MySQL user account password on the source server.
- *<schema(s)>*: One or more schema names.
- *<target_cluster_endpoint>*: Cluster DNS endpoint of the target Aurora cluster.
- *<target_user>*: Aurora master user name.
- *<target_user_password>*: Aurora master user password.
- *--single-transaction*: Enforces a consistent dump from the source database. Can be skipped if the source database is not receiving any write traffic.
- *--no-data*: Creates a schema-only dump without row data.

For more details, see mysqldump in the *MySQL 5.6 Reference Manual*.

Example:

```
admin@clientA:~$ mysqldump --host=11.22.33.44 --user=root \
--password=pAssw0rd --databases myschema \
--single-transaction --no-data > myschema_dump_schema_only.sql
```

After you complete the schema-only dump, you can obtain data dumps for each table. After logging in to the source MySQL server, use the `SELECT INTO OUTFILE` statement to dump each table's data into a separate CSV file.

```
admin@clientA:~$ mysql --host=11.22.33.44 --user=root --
password=pAssw0rd

mysql> show tables from myschema;
+--------------------+
| Tables_in_myschema |
+--------------------+
| t1                 |
| t2                 |
| t3                 |
| t4                 |
```

```
+--------------------+
4 rows in set (0.00 sec)

mysql> SELECT * INTO OUTFILE
       '/home/admin/dump/myschema_dump_t1.csv'
       FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
       LINES TERMINATED BY '\n'
       FROM myschema.t1;
Query OK, 4194304 rows affected (2.35 sec)

(repeat for all remaining tables)
```

For more information about `SELECT INTO` statement syntax, see SELECT ... INTO Syntax in the *MySQL 5.6 Reference Manual*.[20]

After you complete all dump operations, the `/home/admin/dump` directory contains five files: one schema-only dump and four data dumps, one per table.

```
admin@clientA:~/dump$ ls -sh1
total 685M
4.0K myschema dump schema only.sql
172M myschema_dump_t1.csv
172M myschema_dump_t2.csv
172M myschema_dump_t3.csv
172M myschema_dump_t4.csv
```

Next, you compress and transfer the files to client instance B located in the same AWS Region and Availability Zone as the target Aurora DB cluster. You can use any file transfer method available to you (e.g., FTP or Amazon S3). This example uses SCP with SSH private key authentication.

```
admin@clientA:~/dump$ gzip myschema_dump_*.csv
admin@clientA:~/dump$ scp -i ssh-key.pem myschema_dump_* \
<clientB_ssh_user>@<clientB_address>:/home/ec2-user/
```

After transferring all the files, you can decompress them and import the schema and data. Import the schema dump first because all relevant tables must exist before any data can be inserted into them.

```
admin@clientB:~/dump$ gunzip myschema_dump_*.csv.gz
admin@clientB:~$ mysql --host=<cluster_endpoint> --user=master \
--password=pAssw0rd < myschema_dump_schema_only.sql
```

With the schema objects created, the next step is to connect to the Aurora DB cluster endpoint and import the data files.

Note the following:

- The **mysql** client invocation includes a `--local-infile` parameter, which is required to enable support for `LOAD DATA LOCAL INFILE` commands.
- Before importing data from dump files, use a `SET` command to disable foreign key constraint checks for the duration of the database session. Disabling foreign key checks not only improves import performance, but it also lets you import data files in arbitrary order.

```
admin@clientB:~$ mysql --local-infile --host=<cluster_endpoint>
\
--user=master --password=pAssw0rd

mysql> SET foreign_key_checks = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> LOAD DATA LOCAL INFILE '/home/ec2-
user/myschema_dump_t1.csv'
    -> INTO TABLE myschema.t1
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n';
Query OK, 4194304 rows affected (1 min 2.66 sec)
Records: 4194304  Deleted: 0  Skipped: 0  Warnings: 0

(repeat for all remaining CSV files)

mysql> SET foreign_key_checks = 1;
Query OK, 0 rows affected (0.00 sec)
```

That's it—you have imported the schema and data dumps into the Aurora DB cluster.

Useful Tips

- This example involved a single-threaded data dump and a single-threaded data import, but you can easily parallelize the operations for greater migration performance. Just run multiple `SELECT INTO OUTFILE` or `LOAD DATA LOCAL INFILE` commands in parallel, one command per table. There is a limit to how many parallel operations you can run before the performance stabilizes. A good rule of thumb is to use one thread per server CPU core (for dumps) and one thread per two CPU cores (for imports).
- Note that each `LOAD DATA LOCAL INFILE` operation uses a single database transaction to process the entire input file. For very large files, this might degrade the performance and stability of the import process. It is good practice to split very large dump files into smaller chunks (e.g., 1 GB) and import them sequentially. Chunked imports have another useful side effect, namely, they can be easily suspended and resumed.
- You might not need to use `SELECT INTO OUTFILE` commands to produce flat-file dumps if your export tool offers similar functionality. For more information, see Dumping Data in Delimited-Text Format with mysqldump in the *MySQL 5.6 Reference Manual*.
- You might not need to use `LOAD DATA LOCAL INFILE` commands to import flat-file dumps into Amazon Aurora if your import tool offers similar functionality. For more information, see mysqlimport — A Data Import Program in the *MySQL 5.6 Reference Manual*.

You can find more tips and best practices for self-managed migrations in the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.

Multi-Threaded Migration Using **mydumper** and **myloader**

Mydumper and **myloader** are popular open source MySQL export/import tools designed to address performance issues associated with the legacy **mysqldump** program. They operate on SQL-format dumps and offer advanced features such as the following:

- Dumping and loading data using multiple parallel threads
- Creating dump files in a file-per-table fashion
- Creating chunked dumps in a multiple-files-per-table fashion

- Dumping data and metadata into separate files for easier parsing and management
- Configurable transaction size during import
- Ability to schedule dumps in regular intervals

For more details, see the MySQL Data Dumper project page.

Approach

The scenario uses the **mydumper** and **myloader** tools to perform a multi-threaded schema and data migration without the need to manually invoke any SQL commands or design custom migration scripts.

The migration is performed in two steps:

1. Use the **mydumper** tool to create a schema and data dump using multiple parallel threads.
2. Use the **myloader** tool to process the dump files and import them into an Aurora DB cluster, also in multi-threaded fashion.

Note that **mydumper** and **myloader** might not be readily available in the package repository of your Linux/Unix distribution. For your convenience, the scenario also shows how to build the tools from source code.

When to Use

You can adopt this approach in most migration projects:

- The utilities are easy to use and enable database users to perform multi-threaded dumps and imports without the need to develop custom migration scripts.
- Both tools are highly flexible and have reasonable configuration defaults. You can adjust the default configuration to satisfy the requirements of both small- and large-scale migrations.

When to Consider Other Options

You might decide not to use this approach if any of the following conditions are true:

- You are migrating from an RDS MySQL DB instance or a self-managed MySQL 5.5 or 5.6 database. In that case, you might get better results with snapshot migration or Percona XtraBackup, respectively. See the Migrating from Amazon RDS for MySQL and Percona XtraBackup sections for more details.
- You can't use third-party software because of operating system limitations.
- Your data transformation processes require intermediate dump files in a flat-file format and not an SQL format.

Notes

To simplify the demonstration, this scenario assumes the following:

1. You execute the migration commands from client instances running a Linux operating system:
   a. **Client instance A** is located in the source server's network
   b. **Client instance B** is located in the same Amazon VPC, Availability Zone, and Subnet as the target Aurora cluster
2. The source server is a self-managed MySQL database (e.g., running on Amazon EC2 or on-premises) configured to allow connections from client instance A.
3. The target Aurora DB cluster already exists and is configured to allow connections from client instance B. If you don't have an Aurora DB cluster yet, review the step-by-step cluster launch instructions in the *Amazon RDS User Guide*.
4. Communication is allowed between both client instances.
5. You perform the export from the source database using a privileged, "super user" MySQL account. For simplicity, the example assumes that the user holds all permissions available in MySQL.

6. You perform the import into Amazon Aurora using the master user account, that is, the account whose name and password were specified during the cluster launch process.
7. The Amazon Linux 2016.03.3 operating system is used to demonstrate the configuration and compilation steps for **mydumper** and **myloader**.

Note that this migration approach requires application downtime while the dump and import are in progress. You can avoid application downtime by extending the scenario with MySQL binary log replication. See the Self-Managed Migration with Near-Zero Downtime section for more details.

Examples (Preparing Tools)

The first step is to obtain and build the **mydumper** and **myloader** tools. See the MySQL Data Dumper project page for up-to-date download links and to ensure that tools are prepared on both client instances.

The utilities depend on several packages that you should install first.

```
[ec2-user@clientA ~]$ sudo yum install glib2-devel mysql56 \
mysql56-devel zlib-devel pcre-devel openssl-devel g++ gcc-c++
cmake
```

The next steps involve creating a directory to hold the program sources, and then fetching and unpacking the source archive.

```
[ec2-user@clientA ~]$ mkdir mydumper
[ec2-user@clientA ~]$ cd mydumper/

[ec2-user@clientA mydumper]$ wget
https://launchpad.net/mydumper/0.9/0.9.1/+download/mydumper-
0.9.1.tar.gz

2016-06-29 21:39:03 (153 KB/s) - 'mydumper-0.9.1.tar.gz' saved
[44463/44463]

[ec2-user@clientA mydumper]$ tar zxf mydumper-0.9.1.tar.gz
```

```
[ec2-user@clientA mydumper]$ cd mydumper-0.9.1
```

Next, you build the binary executables.

```
[ec2-user@clientA mydumper-0.9.1]$ cmake .
(…)
 [ec2-user@clientA mydumper-0.9.1]$ make
Scanning dependencies of target mydumper
[ 25%] Building C object CMakeFiles/mydumper.dir/mydumper.c.o
[ 50%] Building C object
CMakeFiles/mydumper.dir/server_detect.c.o
[ 75%] Building C object
CMakeFiles/mydumper.dir/g unix signal.c.o
Linking C executable mydumper
[ 75%] Built target mydumper
Scanning dependencies of target myloader
[100%] Building C object CMakeFiles/myloader.dir/myloader.c.o
Linking C executable myloader
[100%] Built target myloader
```

Optionally, you can move the binaries to a location defined in the operating system `$PATH` so that they can be executed more conveniently.

```
[ec2-user@clientA mydumper-0.9.1]$ sudo mv mydumper
/usr/local/bin/mydumper
[ec2-user@clientA mydumper-0.9.1]$ sudo mv myloader
/usr/local/bin/myloader
```

As a final step, confirm that both utilities are available in the system.

```
[ec2-user@clientA ~]$ mydumper -V
mydumper 0.9.1, built against MySQL 5.6.31

[ec2-user@clientA ~]$ myloader -V
myloader 0.9.1, built against MySQL 5.6.31
```

Examples (Migration)

After completing the preparation steps, you can perform the migration.

The **mydumper** command uses the following basic syntax.

```
mydumper -h <source_server_address> -u <source_user> \
-p <source_user_password> -B <source_schema> \
-t <thread_count> -o <output_directory>
```

Descriptions of the parameter values are as follows:

- <*source_server_address*>: DNS name or IP address of the source server
- <*source_user*>: MySQL user account name on the source server
- <*source_user_password*>: MySQL user account password on the source server
- <*source_schema*>: Name of the schema to dump
- <*thread_count*>: Number of parallel threads used to dump the data
- <*output_directory*>: Name of the directory where dump files should be placed

Note that **mydumper** is a highly customizable data dumping tool. For a complete list of supported parameters and their default values, use the built-in help.

```
mydumper --help
```

The example dump is executed as follows.

```
[ec2-user@clientA ~]$ mydumper -h 11.22.33.44 -u root \
-p pAssw0rd -B myschema -t 4 -o myschema_dump/
```

The operation results in the following files being created in the dump directory.

```
[ec2-user@clientA ~]$ ls -sh1 myschema_dump/
total 733M
4.0K metadata
4.0K myschema-schema-create.sql
4.0K myschema.t1-schema.sql
184M myschema.t1.sql
4.0K myschema.t2-schema.sql
184M myschema.t2.sql
4.0K myschema.t3-schema.sql
184M myschema.t3.sql
4.0K myschema.t4-schema.sql
184M myschema.t4.sql
```

The directory contains a collection of metadata files in addition to schema and data dumps. You don't have to manipulate these files directly. It's enough that the directory structure is understood by the **myloader** tool.

Compress the entire directory and transfer it to client instance B.

```
[ec2-user@clientA ~]$ tar czf myschema_dump.tar.gz myschema_dump
[ec2-user@clientA ~]$ scp -i ssh-key.pem myschema_dump.tar.gz \
<clientB_ssh_user>@<clientB_address>:/home/ec2-user/
```

When the transfer is complete, connect to client instance B and double-check that the **myloader** utility is available.

```
[ec2-user@clientB ~]$ myloader -V
myloader 0.9.1, built against MySQL 5.6.31
```

Now you can unpack the dump and import it. The syntax used for the **myloader** command is very similar to what you already used for **mydumper**. The only difference is the `--d` (source directory) parameter replacing the `--o` (target directory) parameter.

```
[ec2-user@clientB ~]$ tar zxf myschema_dump.tar.gz
[ec2-user@clientB ~]$ myloader -h <cluster_dns_endpoint> \
```

```
-u master -p pAssw0rd -B myschema -t 4 -d myschema_dump/
```

Useful Tips

- The concurrency level (thread count) does not have to be the same for export and import operations. A good rule of thumb is to use one thread per server CPU core (for dumps) and one thread per two CPU cores (for imports).
- The schema and data dumps produced by **mydumper** use an SQL format and are compatible with MySQL 5.6. Although you will typically use the pair of **mydumper** and **myloader** tools together for best results, technically you can import the dump files from **myloader** by using any other MySQL-compatible client tool.

You can find more tips and best practices for self-managed migrations in the AWS whitepaper Best Practices for Migrating MySQL Databases to Amazon Aurora.

Heterogeneous Migrations

For detailed, step-by-step instructions on how to migrate schema and data from a non-MySQL-compatible database into an Aurora DB cluster using AWS SCT and AWS DMS, see the AWS whitepaper Migrating Your Databases to Amazon Aurora.

Troubleshooting

The following sections provide examples of common issues and error messages to help you troubleshoot self-managed migrations and binary log replication.

Troubleshooting Self-Managed Migrations

This section describes common issues you might encounter during self-managed dump/import operations. Note that although the error messages and troubleshooting suggestions are discussed in the context of Amazon Aurora, they remain valid for any MySQL-compatible database.

For a complete reference of MySQL server and client-side error messages, see the following pages in the *MySQL 5.6 Reference Manual*:

- Server Error Codes and Messages[21]
- Client Error Codes and Messages[22]

Error 2005: Unknown MySQL server host

The following is an example of this error message.

```
Got error: 2005: Unknown MySQL server host 'aurora.xyz.us-west-
2.rds.amazonaws.com' (0) when trying to connect
```

Description

This error message appears when the MySQL client program can't find the specified MySQL server name. Most common causes include DNS resolution issues and typographical errors in the DNS name.

Troubleshooting

- Check the DNS name for typographical errors. For best results, try copying the DNS name from the AWS Management Console instead of entering it manually.
- Check whether the DNS name correctly resolves to an IP address using operating system tools such as **host**, **dig**, and **nslookup**. For accuracy, test DNS resolution from the same client instance that experiences database connectivity issues.
- If DNS resolution fails from one client host, repeat the test from other clients. Ideally, try to use clients that are located in the same network and also clients located in other networks. You can also use online DNS resolution services for verification. This can help you rule out client-specific and network provider–specific DNS resolution issues.
- If DNS resolution fails from all clients regardless of their network location, and you are certain that the DNS name doesn't contain typographical errors, there might be issues with the DNS endpoint itself.

If it's an AWS-managed endpoint such as an Aurora DB cluster endpoint, you can contact AWS Support for further assistance.

- If DNS resolution consistently fails on some clients, but not all, the issue is likely not specific to the particular DNS endpoint but instead to the client's operating system or network DNS configuration.

Error 2003: Can't connect to MySQL server

The following is an example of this error message.

```
Got error: 2003: Can't connect to MySQL server on
'aurora.xyz.us-west-2.rds.amazonaws.com' (110) when trying to
connect
```

Description

This error message appears when the server DNS name or IP address is correct but the server can't be contacted. Common causes include the following:

- Network communication between the client instance and the server endpoint is not possible because of the configuration of security groups, network access control lists (ACLs), routing tables, or any self-managed network or firewalling solutions.
- Connection attempt to the Aurora DB instance is made from outside the instance's VPC, but the instance does not have the Publicly Accessible option enabled. For details, see Working with an Amazon RDS DB Instance in a VPC in the *Amazon RDS User Guide*.[23]
- The server is using a custom port number such as 3307 instead of 3306, but the MySQL client program was not instructed to use the custom port (e.g., the `--port` parameter for the **mysqldump** command).

Troubleshooting

For Amazon Aurora:

- Verify that the instance is in the "Available" state.

- Verify that the network and security configuration of the Aurora DB instance allows for connections from the client host. For details, see Working with an Amazon RDS DB Instance in a VPC in the *Amazon RDS User Guide*.
- Verify that the client program uses the correct database port number.
- Use a network route-tracking tool such as **traceroute** to verify if and where the network traffic is dropped. Note that Amazon Aurora does not respond to ICMP echo requests. You must use a route-tracing tool that supports TCP probes, such as **traceroute** for Linux and **tracetcp** for Windows.

For self-managed servers:

- Verify that the server daemon is running and in a state that allows client connections (e.g., not in recovery or shutdown mode).
- Verify that the server and its underlying network components are configured to accept traffic from the client host. For example, some database servers might be configured to accept traffic only from localhost, or traffic might be dropped on your local firewall.

Error 1045: Access denied for user when trying to connect

The following are examples of this error message.

```
Got error: 1045: Access denied for user 'user'@'host' (using
password: YES) when trying to connect

Got error: 1045: Access denied for user 'user'@'host' (using
password: NO) when trying to connect
```

Description

This error appears when you try to connect using an incorrect password or no password.

Troubleshooting

If the error message includes "`using password: NO`", it indicates that the client program is not sending the password to the server. Check the client program syntax and make sure that the password is included in the parameter

list. For example, the following **mysql** command syntax is incorrect because there must not be a space between "`--p`" and the password.

```
mysql -h 127.0.0.1 -u master -p pAssw0rd
```

If the error message includes "`using password: YES`", check the supplied password. If you are confident that the password you supplied is correct, you can first test the connection from another MySQL client tool to rule out client-specific issues. If the password doesn't work with any client, you can choose to reset the user password:

- To reset the master user password in Amazon Aurora, use the "Modify" action in RDS Management Console or the modify-db-cluster method in the AWS Command Line Interface (CLI).[24]
- To reset the master user password in a self-managed MySQL server, follow the instructions in the MySQL 5.6 Reference Manual.[25]
- To reset a non-master user password in Amazon Aurora or a self-managed MySQL server, log in to the database as the master user and use the `SET PASSWORD` SQL statement.[26]

Error 1045: Access denied for user

The following is an example of this error message.

```
ERROR 1045 (28000) at line 85: Access denied for user
'user'@'host' (using password: YES)
```

Description

This error appears when the database user doesn't have sufficient privileges to execute a given SQL statement. The error message reports a line number in the SQL script (or dump file) where the error occurred.

Troubleshooting

- Find the command in the SQL script or dump file at the given line number.

- Check that the database user has permissions to execute all SQL operations referenced on that line.

For a sample migration scenario that involves the troubleshooting and resolution of an "`access denied`" error, see the Migration Using mysqldump with Error Troubleshooting section.

Error 1227: Access denied; you need (at least one of) the SUPER privilege(s)

The following is an example of this error message.

```
ERROR 1227 (42000) at line 1: Access denied; you need (at least
one of) the SUPER privilege(s) for this operation
```

Description

This error appears when a user tries to execute an operation that is protected by the `SUPER` privilege, but the user doesn't have that privilege. The error message reports a line number in the SQL script (or dump file) where the error occurred.

Troubleshooting

To identify the SQL statement that failed, find the contents of the given line in the SQL script or dump file.

If the SQL statement fails against a self-managed MySQL server, consult a database administrator who is responsible for the server's security management and configuration. The database administrator can resolve the issue by granting the `SUPER` privilege or by executing the privileged actions on behalf of the unprivileged user.

Note that Amazon Aurora does not provide the `SUPER` privilege. If a SQL command fails against Amazon Aurora due to the lack of `SUPER` privilege:

- Identify the problematic SQL statement as described earlier.

- Determine whether the statement can be safely skipped or modified so that it no longer requires super-user privileges.

For a sample migration scenario that involves the troubleshooting and resolution of an "`access denied`" error, see the Migration Using mysqldump with Error Troubleshooting section.

Error 1064: You have an error in your SQL syntax

The following is an example of this error message.

```
ERROR 1064 (42000) at line 662: You have an error in your SQL
syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near 'inset into
myschema.test values (1)' at line 1
```

Description

This error appears when an SQL operation can't be processed by the server due to incorrect or unrecognized syntax. The error message reports a line number in the SQL script (or dump file) where the error occurred, as well as the failing statement or part thereof.

The following conditions can cause the error:

- The statement was obtained from a different MySQL server version and the target server doesn't understand the statement syntax.
- The statement text is corrupted or contains typographical errors (e.g., missing characters in SQL keywords, missing spaces, or delimiters).

Troubleshooting

To identify the SQL statement that failed, find the contents of the given line in the SQL script or dump file, and verify that the statement is syntactically correct. If you are using a graphical SQL client, it might already contain syntax-checking features, or you can use an online SQL syntax checker.

If the statement was produced by a MySQL-compatible server (e.g., as part of a dump file) and appears to be syntactically correct, try to execute it against two servers:

- The Aurora DB cluster you are migrating to.
- Another server running the same database version as the source server that produced the statement in the first place.

  If the statement succeeds on the source-compatible server but not on the target Aurora DB cluster, it might have configuration or version compatibility issues.

If a MySQL 5.6-compatible statement consistently returns syntax errors when executed against Amazon Aurora, contact AWS Premium Support for guidance.[27]

Error 1049 or 1146: Unknown database or Table doesn't exist

The following are examples of this error message.

```
ERROR 1049 (42000) at line 1: Unknown database 'myscscd'

ERROR 1146 (42S02) at line 1: Table 'myschema.test' doesn't
exist
```

Description

This error can appear in the following two situations:

- When you try to access a table that doesn't exist.
- When you try to access a table in a schema that doesn't exist.

The following conditions can cause the error:

- The table and/or schema do not exist, that is, they were not created before you tried to access them.
- The table or schema name is incorrect.

The error message reports a line number in the SQL script (or dump file) where the error occurred, and also the table or schema name that can't be found. Note that although an "`Unknown database`" message indicates a missing schema, a "`Table doesn't exist`" message indicates that either the table or its schema is missing.

Troubleshooting

- Check that the table or schema that is referenced in the error message exists on the server.
- Check that the source and target servers use the same configuration parameters for table name case sensitivity. For details, see the MySQL 5.6 Reference Manual.[28]

Error 2020: Got packet bigger than 'max_allowed_packet' bytes when dumping table

The following is an example of this error message.

```
Error 2020: Got packet bigger than 'max_allowed_packet' bytes
when dumping table `bigtable`
```

Description

This error can appear when you are dumping a table that contains individual column values that are larger than the value that is configured for the `max_allowed_packet` parameter. The parameter applies to both the server and client configurations.

For example, if your server and MySQL client program have the `max_allowed_packet` parameter value set to 4194304 bytes (4 MB), and you try to dump data from a table that contains column values larger than 4 MB, the dump might fail with this error.

Troubleshooting

- Determine the maximum length of column values that are expected to be present in your database.
- Set the `max_allowed_packet` parameter on server and client side to a value larger than the longest column value found in the database.

You can use the DB Parameter Groups to modify server-side configuration parameters on Amazon Aurora. For details, see Working with DB Parameter Groups in the *Amazon RDS User Guide*.[29]

To change client-side configuration parameters, refer to the documentation of your MySQL client program. For example, the **mysqldump** program offers a `--max-allowed-packet` parameter for this purpose.

Error 2006: MySQL server has gone away

The following is an example of this error message.

```
ERROR 2006 (HY000) at line 38: MySQL server has gone away
```

Description

This error appears when the connection to the server is lost unexpectedly. The range of possible causes is very wide. In fact, the MySQL Reference Manual contains an entire page dedicated to this particular error.

Troubleshooting

See the MySQL 5.6 Reference Manual for details and troubleshooting tips.[30]

No space left on device, table is full, incorrect key file for table

The following is an example of this error message.

```
Error 28: no space left on device
Error 1114: The table is full
Error 126: Incorrect key file for table
Error 1034: Incorrect key file for table
```

Description

This error can appear when the server runs out of temporary storage space.

The following conditions can cause the error:

- A Data Definition Language (DDL) statement such as `ALTER TABLE` is executed against a very large table.
- A `SELECT` or `INSERT INTO … SELECT` statement requires a large temporary table for internal processing.

Each Amazon Aurora DB instance uses an Amazon EC2 instance-store volume to hold non-permanent data such as logs and temporary tables that the database engine creates during query processing. The size of the temporary storage volume depends on the Aurora DB instance class.

If an Aurora DB instance class doesn't provide enough space to accommodate a given SQL operation, the operation might fail with one of these error messages.

For more information about the size of instance-store volumes that are available to each instance class, see Instance Store Volumes in the *Amazon EC2 User Guide*.[31]

You can use the "Free Local Storage" Amazon CloudWatch metric to monitor the amount of local temporary storage that is available to each of your Amazon Aurora instances. For details, see Monitoring an Amazon Aurora DB Cluster in the *Amazon RDS User Guide*.[32]

Troubleshooting

Try the following if the error occurs during an `ALTER TABLE` operation:

- Determine the size of the table that is being altered. Depending on the nature of the change, the database might need an amount of temporary storage equal to or bigger than the size of the table.
- If possible, scale to a larger instance class that offers a bigger temporary storage volume.
- If you are altering multiple tables, try performing one alter at a time.
- When the table is too large to be altered regardless of the instance size, consider alternative approaches that don't require temporary tables, such as the **pt-online-schema-change** tool from Percona Toolkit. Make sure that you evaluate and test the tool thoroughly before using it in your production environment.

If the error occurs during a different type of SQL operation (e.g., a long-running `SELECT`), try splitting the operation into smaller chunks. This not only improves the stability of this particular statement, but it also increases its scalability as your data set grows.

Troubleshooting Binary Log Replication

> This section applies to migrations from MySQL-compatible servers only.

Binary log ("binlog") replication is a MySQL technology that enables changes introduced on a MySQL-compatible database server ("master") to be applied to one or more MySQL-compatible servers ("slaves"). As a MySQL 5.6-compatible database, Amazon Aurora can participate in replication setups with other MySQL 5.6-compatible databases.

Binary log replication is a popular technique for reducing migration downtime because it enables the source server to continue accepting traffic during the migration process. Downtime is only required for the final cut-over, that is, the moment when database traffic is shifted from the source to the target server.

See the following sections for details on how binary log replication helps reduce migration downtime:

- Migrating with Downtime
- Migrating with Near-Zero Downtime

When you are migrating into Amazon Aurora, an Aurora DB cluster serves as the replication target ("slave"), receiving and applying changes from another MySQL-compatible server. This section is intended to help you address the most common questions and issues related to operating a binary log replication slave.

Binary Log Terms and Concepts

> This section discusses binary log replication from external MySQL-compatible servers. Note that binary logs are not used for replication between database instances within an Aurora cluster.

The binary log is an ordered and sequential record of changes (or "events") that occur on a replication master. With transactional MySQL storage engines, such as InnoDB, the binary log records are written at commit time so that they refer only to changes that were successfully applied to the database. After the binary log is written, it becomes available to replication slaves. For details, see The Binary Log in the *MySQL 5.6 Reference Manual*.[33]

A replication slave uses two types of internal threads to obtain and apply change vectors. The **slave I/O thread** is responsible for connecting to the replication master, downloading binary log files, and storing them on the replication slave. A binary log that is downloaded from the replication master and stored on the replication slave is called a **relay log**. The binary log content is downloaded as-is (the slave does not modify log records before storing them locally). So the "binary log" and "relay log" naming convention refers to the location of the log files rather than their content.

The **slave SQL thread** reads records from the relay log and applies them to the database. The log records can be in the form of literal SQL statements (e.g., `UPDATE table SET ...`) or non-SQL change vectors (e.g., "*find row X in table T, change values to Y*").

Each binary log record is marked with a timestamp of when the change was executed on the master. **Replication lag** is the difference between the current

time on the slave and the timestamp of the replication event that is currently being applied on the slave. You can monitor the replication lag using the `AuroraBinlogReplicaLag` CloudWatch metric.[34]

Binary log replication is not **idempotent**, which means that a change record can be applied only once with the same result. Change records cannot be applied multiple times without potentially causing data consistency issues.

If the slave I/O thread is unable to download binary log data from the master, or if the SQL thread encounters a binary log record that can't be successfully applied, the thread might stop with an error. The database administrator can diagnose and resolve the error and restart the replication. If the replication error can't be corrected, the replication slave must be re-created from scratch.

In Amazon Aurora, you can also manually start or stop binary log replication. For details, see mysql.rds_stop_replication in the *Amazon RDS User Guide*.[35]

For a complete discussion of binary log replication, see Replication in the *MySQL 5.6 Reference Manual*.[36]

Obtaining Binary Log Replication Status

You can connect to the Aurora cluster and use the `SHOW SLAVE STATUS` SQL command to obtain replication status. If the result is empty, the Aurora DB cluster is not configured to replicate from a binary log master.

Some of the most important fields from the output of the `SHOW SLAVE STATUS` command are shown and discussed in this section. For full documentation, see SHOW SLAVE STATUS Syntax in the *MySQL 5.6 Reference Manual*.[37]

```
mysql> SHOW SLAVE STATUS\G
...
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
...
        Relay_Master_Log_File: master-bin.000002
          Exec_Master_Log_Pos: 1307
...
        Seconds_Behind_Master: 0
...
                Last_IO_Errno: 0
                Last_IO_Error:
```

```
               Last_SQL_Errno: 0
               Last_SQL_Error:
...
               Slave_IO_State: Waiting for master to send event
      Slave_SQL_Running_State: Slave has read all relay log;
waiting for the slave I/O thread to update it
```

Descriptions of the fields are as follows:

- *Slave_IO_Running*: Reports "Yes" if the slave I/O thread is running. A value of "Connecting" indicates that the I/O thread is trying to connect to the master. A value of "No" indicates that the thread is stopped or down due to an error.
- *Slave_SQL_Running*: Reports "Yes" if the slave SQL thread is running. A value of "No" indicates that the thread is stopped or down due to an error.
- *Relay_Master_Log_File*: The name of the master binary log file that contains the most recently executed event.
- *Exec_Master_Log_Pos*: The position (byte offset) in *Relay_Master_Log_File* to which the slave SQL thread has read and executed. If this value changes in subsequent `SHOW SLAVE STATUS` outputs, replication is progressing normally.
- *Seconds_Behind_Master*: The replication lag in seconds. A value of "NULL" indicates that the slave SQL thread is not running, or that the slave SQL thread has already applied all relay logs and the slave I/O thread is not running.
- *Last_*_Error*: Contains the error codes and messages, if any, for the I/O and SQL replication threads. These values are the primary source of information when you diagnose replication issues.
- *Slave_IO_State, Slave_SQL_Running_State*: Verbose description of the current I/O and SQL thread state. See the *MySQL 5.6 Reference Manual* for the discussion of possible I/O thread states and SQL thread states.[38] [39]

Common Replication Issues

> This section assumes a scenario in which an Amazon Aurora DB cluster serves as a replication slave of a MySQL-compatible replication master.

Replication Is Slow

On a typical database server, many client sessions execute and introduce changes in parallel. After the change vectors are written to the binary log and are transferred to the replication slave, a MySQL-5.6 compatible replication slave would typically use only a single SQL thread to apply the changes, one change at a time.

If the replication master experiences a heavy load, the single slave SQL thread might not be able to apply changes quickly enough. As a result, the slave might develop replication lag. For the purposes of database migration, replication lag is an issue if it grows continuously or if it falls too slowly. For example, if the initial replication lag is 48 hours and it falls by 2 hours per day, it would take 24 days for the slave to fully catch up.

Consider the following techniques for improving replication performance:

- Disable all unnecessary features on the replication slave. Features such as query logging or binary logging might introduce overhead that impacts slave SQL thread processing.
- If the replication master uses a `STATEMENT` or `MIXED` binary log format, try using a `ROW` format instead. This might improve replication performance for some types of statements. For details, see Binary Logging Formats in the *MySQL Reference Manual*.[40]
- If the replication workload consists of a large number of very small transactions, you might configure the replication slave to use a more relaxed `innodb_flush_log_at_trx_commit` setting for transaction log flushing.[41] Parameter values of "0" or "2" are not recommended in production environments, but they can be introduced temporarily on the slave to speed up replication.
- If the replication workload consists of changes to multiple schemas, you might configure the slave for multi-threaded replication.[42] This configuration enables multiple slave SQL threads and allows for a better

use of the slave's CPU and I/O resources. Note that the SQL threads don't use any conflict resolution mechanisms. This feature is not safe to use if individual transactions make changes in more than one schema.

Replication Appears Stuck

Speaking broadly, binary log replication can be in one of the following three states:

- Replication is stopped.
- Replication can't move forward due to errors encountered by slave I/O or SQL threads.
- Replication is moving forward at a certain speed, given the type of workload and master/slave configuration.

In some cases, the relatively slow progress made by replication threads might create an impression that replication is "stuck". Fortunately, you can confirm replication status using these steps:

1. Confirm that the slave I/O and SQL threads don't report errors.
2. Obtain replication status multiple times and check the value of *Exec_Master_Log_Pos*. If the value is increasing, replication is progressing normally.

See the Obtaining Binary Log Replication Status section for more details.

Note that metrics such as the *Exec_Master_Log_Pos* and the replication lag (*Seconds_Behind_Master*) are updated after executing a replication event. If the replication event is very time-consuming (e.g., an `UPDATE` statement that takes several minutes or hours), the metrics might not change until that event finishes. When you obtain replication status with the `SHOW SLAVE STATUS` command, allow sufficient time between command executions in case the slave SQL thread is processing a time-consuming event.

Last but not least, you must have a way to confirm whether the slave SQL thread is in the middle of a time-consuming event. The following example explains how to do it.

Example

First, obtain the replication status (output reduced for brevity).

```
mysql> show slave status\G
```

```
*************************** 1. row ***************************
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
...
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
...
        Relay_Master_Log_File: mysql-bin-changelog.062779
          Exec_Master_Log_Pos: 120
        Seconds_Behind_Master: 31
...
      Slave_SQL_Running_State: updating
```

The output shows the following:

- Per *Slave_*_Running* and *Last_*_Error*, slave threads are both running and are not reporting any errors.
- The slave is currently processing the binary log file `062779` at position `120`.
- The slave SQL thread status is "`updating`", which suggests it's currently executing an `UPDATE` statement. You might observe a different thread state, such as "`inserting`" or "`Reading event from the relay log`", depending on the type and logging format of the replication event that is currently being executed.
- Replication lag is 31 seconds.

Obtain the status again.

```
mysql> show slave status\G
*************************** 1. row ***************************
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
...
                Last_IO_Errno: 0
                Last_IO_Error:
```

```
                Last_SQL_Errno: 0
                Last_SQL_Error:
...
         Relay_Master_Log_File: mysql-bin-changelog.062779
           Exec_Master_Log_Pos: 120
         Seconds_Behind_Master: 34
...
       Slave_SQL_Running_State: updating
```

This output shows the following:

- Replication threads still do not report errors.
- Master log file number and position remain unchanged.
- Slave SQL thread status still reports "`updating`".
- Replication lag is growing (was 31 seconds; now is 34 seconds).

At this point, you might very well assume that the replication thread is "stuck" because it's not reporting progress as far as the status variables are concerned.

Continue the investigation by obtaining the list of server processes, among which is the slave SQL thread.

```
mysql> show processlist;
+-----+-------------+---------+------+---------------------------------+-----------------------+
| Id  | User        | Command | Time | State                           | Info                  |
+-----+-------------+---------+------+---------------------------------+-----------------------+
...
| 232 | system user | Connect | 4425 | Waiting for master to send event | NULL                  |
| 233 | system user | Connect |  239 | updating                        | update t1 set s1 = 'a' |
...
```

This output shows the following:

- Thread ID 232 is the slave I/O thread responsible for downloading binary logs from the master.

- Thread ID 233 (highlighted in yellow) is the slave SQL thread with a current state of "`updating`".
- The SQL thread is currently executing an SQL statement "`update t1 set s1 = 'a'`". Depending on the type and logging format of the replication event, you might not see a literal SQL statement here.

You have already gathered a lot of information, but you still don't have proof that the statement in question is not "stuck". You can use the [InnoDB Standard Monitor] (`SHOW ENGINE INNODB STATUS` command) to learn more about the internal state of the statement.[43] The command output contains the following information under the `TRANSACTIONS` section.

```
mysql> show innodb engine status\G
...
---TRANSACTION 2146412, ACTIVE 41 sec updating or deleting
mysql tables in use 1, locked 1
28097 lock struct(s), heap size 5158440, 5057125 row lock(s), undo log entries 5038395
MySQL thread id 233, OS thread handle 0x2afef04d6700, query id 31495 updating
update t1 set s1 = 'a'
...
```

This output shows the following:

- You can tell that you're looking at the correct thread/transaction because it has the same MySQL thread ID as the one you observed earlier in the process list (highlighted in yellow).
- The transaction has been active for 41 seconds. So far it acquired approximately 5 million row locks and made approximately 5 million changes (per "`undo log entries`").

Invoke the command again and compare the outputs.

```
mysql> show innodb engine status\G
...
---TRANSACTION 2146412, ACTIVE 45 sec updating or deleting
mysql tables in use 1, locked 1
30102 lock struct(s), heap size 5158440, 5541045 row lock(s), undo log entries 5513375
MySQL thread id 233, OS thread handle 0x2afef04d6700, query id 31495 updating
update t1 set s1 = 'a'
```

```
...3
```

The final observation is that the number of row locks and changes is growing, roughly by half a million within four seconds. It ultimately proves that the slave SQL thread is, in fact, not stuck, but is busy applying a large change.

I/O State: Waiting for the slave SQL thread to free enough relay log space

The following are the symptoms of this I/O thread state.

```
mysql> show slave status \G
*************************** 1. row ***************************
Slave_IO_State: Waiting for the slave SQL thread to free enough
relay log space
...
```

Description

When working as a replication slave, Amazon Aurora places a limit on how much binary log data can be pre-fetched from the replication master. The limit helps avoid unnecessary growth of the auto-scaled cluster storage volume in case the master contains gigabytes or terabytes of binary log data. When the combined size of the relay logs exceeds the limit, the slave I/O thread pauses until some relay logs are processed and removed by the slave SQL thread.

Troubleshooting

This I/O thread state is normal. It does not slow replication down and does not require intervention.

I/O Error: Error connecting to master

The following is an example of this error message.

```
mysql> SHOW SLAVE STATUS\G;
*************************** 1. row ***************************
       Slave_IO_State: Connecting to master
     Slave_IO_Running: Connecting
```

```
...
        Last_IO_Errno: 2003
        Last_IO_Error: error connecting to master
'USER@HOST:PORT' - retry-time: 60  retries: 86400
               Last_SQL_Errno: 0
```

Description

This issue might occur if the replication slave can't connect to the replication master. If the *Last_IO_Errno* and *Last_IO_Error* don't report errors, but the I/O thread continuously remains in "`Connecting`" state, check the slave's error log for replication-related records that contain the keyword `ERROR`.

Troubleshooting

- Verify that the network and security configuration of the replication master allows for connections from the slave.
- Verify that the slave is trying to connect to the correct port number. The default port number for MySQL is 3306, but your replication master might be configured differently.
- You might test your settings by trying a manual MySQL connection against the master, from a client instance in the same security/network domain as the Aurora cluster. Replication uses the regular MySQL connection protocol, so the host, port, user, and password that you configured for replication should also work with any MySQL client.

SQL State: Invalidating query cache entries (table)

The following shows the symptoms of this SQL thread state.

```
mysql> show slave status \G
*************************** 1. row ***************************
Slave_SQL_Running_State: invalidating query cache entries
(table)
...
```

Description

The Query Cache is an in-memory buffer for storing result sets from `SELECT` queries. When data changes are introduced through replication, the relevant query cache entries must be invalidated. For more information, see The MySQL Query Cache in the *MySQL 5.6 Reference Manual*.[44]

Troubleshooting

You can safely ignore this slave SQL thread state. In Amazon Aurora, the Query Cache has been reworked for better performance and scalability. The Query Cache invalidations are no longer expected to cause performance issues.

Note that due to Aurora internals, you might observe this thread state even if the Query Cache is disabled on your Aurora DB instances. This is expected and does not require an intervention.

SQL Error 1236: Could not find first log file name in binary log index file

The following are examples of this error message.

```
mysql> show slave status \G
*************************** 1. row ***************************
      Master_Log_File: mysql-bin.000289
...
     Slave_IO_Running: No
        Last_IO_Errno: 1236
        Last_IO_Error: Got fatal error 1236 from master when
reading data from binary log: 'Could not find first log file
name in binary log index file'
...
```

Description

This message indicates that the binary log file that the replication slave intends to download is not available on the replication master. For example, the file might have been deleted before the slave managed to download it.

Troubleshooting

Connect to the replication master and use the `SHOW BINARY LOGS` command to confirm whether the desired binary log file is available.

If the log file you want does not appear in the output from `SHOW BINARY LOGS`, determine the next closest available binary log by looking at the numeric suffix in the file name. If the difference between the desired file's suffix and next closest file's suffix is greater than 1, you can't resume replication without risking serious data consistency issues. You should re-create the replication slave from scratch and make sure that the replication master is configured to retain binary logs for as long as the slave requires it.

If the difference between the desired file's suffix and next closest file's suffix is equal to 1, you might be able to resume replication by skipping to the next binary log file. For details, see mysql.rds_next_master_log in the *Amazon RDS User Guide*.[45] There is still a risk of data consistency issues, so you should monitor the slave carefully from that point on.

SQL Error 1236: Client requested master to start replication from impossible position

The following are examples of this error message.

```
mysql> show slave status \G
*************************** 1. row ***************************
      Master_Log_File: mysql-bin.012345
...
     Slave_IO_Running: No
        Last_IO_Errno: 1236
        Last_IO_Error: Got fatal error 1236 from master when
reading data from binary log: 'Client requested master to start
replication from impossible position; the first event 'mysql-
bin-changelog.013406' at 1219393, the last event read from
'/rdsdbdata/log/binlog/mysql-bin-changelog.012345' at 4, the
last byte read from '/rdsdbdata/log/binlog/mysql-bin-
changelog.012345' at 4.'
...
```

Description

This message indicates that the slave tries to replicate from an invalid position within a binary log file. This issue can occur if the replication master crashes or restarts without closing the binary log file cleanly.

Troubleshooting

You might be able to resume replication by skipping to the next binary log file. For more information, see mysql.rds_next_master_log in the *Amazon RDS User Guide*.

SQL Errors 1062: Duplicate entry

The following are examples of these error messages.

```
mysql> show slave status \G
*************************** 1. row ***************************
...
 Slave_SQL_Running: No
    Last_SQL_Errno: 1062
    Last_SQL_Error: Error 'Duplicate entry 'key_value' for key
'key_name'' on query. (...) Default database: ''. Query: 'INSERT
INTO table_name ...
...
```

Description

These messages indicate that the slave SQL thread tried to insert a row that violates an existing `PRIMARY` or `UNIQUE` constraint on the table. A common cause of this issue is that the replication slave receives conflicting changes from multiple sources. For example, there might be users or applications that are connecting to the replication slave and introducing changes that conflict with replication events.

Troubleshooting

- Confirm that the row in question does indeed exist in the table. You can use the table name, key name, and key value mentioned in the error message to construct a `SELECT` query that checks for the existence of the row.

- Review your replication slave access rules and make sure that users and applications are not allowed to introduce changes that interfere with replication traffic. You can use General Logging or Advanced Auditing to pinpoint the source of offending statements.[46] [47]
- Resolve the data inconsistency (e.g., by removing the duplicate row), and then resume replication.
- If you can't resolve the data inconsistency, you can skip the error.[48] Note that this introduces data drift between master and slave, which might result in issues such as queries returning incorrect results and more frequent replication breakdowns.
- If you can't resolve the data inconsistency and you don't accept the risks associated with skipping replication errors, you should re-create the slave from scratch.

SQL Error 1032: Can't find record

The following is an example of this error message.

```
mysql> show slave status \G
*************************** 1. row ***************************
...
 Slave_SQL_Running: No
    Last_SQL_Errno: 1032
    Last_SQL_Error: Could not execute Delete_rows event on table
test.t; Can't find record in 'table_name', Error_code: 1032;
handler error HA_ERR_KEY_NOT_FOUND; the event's master log
mysql-bin.012345, end_log_pos 34567
```

Description

These messages indicate that the slave SQL thread tried to modify or delete a row that does not exist in the table. A common cause of this issue is that the replication slave receives conflicting changes from multiple sources. For example, there might be users or applications that are connecting to the replication slave and are introducing changes that conflict with replication events.

Troubleshooting

- Review your replication slave access rules, and make sure that users and applications are not allowed to introduce changes that interfere with replication traffic. You can use General Logging or Advanced Auditing to pinpoint the source of offending statements.
- If you know the correct values that should be stored in the missing row, you might resolve the data inconsistency by manually creating the row on the replication slave. After you do that, resume replication.
- If you can't resolve the data inconsistency, you can skip the error. Note that this introduces data drift between master and slave, which might result in issues such as queries returning incorrect results and more frequent replication breakdowns.
- If you can't resolve the data inconsistency and you don't accept the risks associated with skipping replication errors, you should re-create the slave from scratch.

Conclusion

Multiple factors contribute to a successful database migration:

- The choice of the database product.
- A migration approach (e.g., methods, tools) that meets performance and uptime requirements.
- Well-defined migration procedures that enable database administrators to prepare, test, and complete all migration steps with confidence.
- The ability to identify, diagnose, and deal with issues with little or no interruption to the migration process.

We hope that the guidance provided in this document will help you introduce meaningful improvements in all of these areas, and that it will ultimately contribute to creating a better overall experience for your database migrations into Amazon Aurora.

Contributors

The following individual contributed to this document:

- Szymon Komendera, Database Engineer, Amazon Web Services

Further Reading

For additional information, see the following:

- Aurora on Amazon RDS User Guide[49]
- Migrating Your Databases to Amazon Aurora AWS whitepaper[50]
- Best Practices for Migrating MySQL Databases to Amazon Aurora AWS whitepaper[51]

Notes

1 https://d0.awsstatic.com/whitepapers/RDS/Migrating%20your%20databases%20to%20Amazon%20Aurora.pdf

2 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Aurora.html

3 http://aws.amazon.com/rds/aurora/pricing/

4 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Aurora.html#Aurora.Overview.Reliability

5 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Migrate.RDSMySQL.html

6 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Migrate.MySQL.html#Aurora.Migrate.MySQL.S3

7 http://docs.aws.amazon.com/dms/latest/userguide/Welcome.html

8 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Migrate.html#USER_ImportAurora

9 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Overview.Replication.MySQLReplication.html

10 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_UpgradeDBInstance.MySQL.html

11 https://dev.mysql.com/doc/refman/5.6/en/mysqldump.html

12 http://dev.mysql.com/doc/refman/5.6/en/mysqlimport.html

13 https://dev.mysql.com/doc/refman/5.6/en/mysql.html

14 https://github.com/maxbube/mydumper

15 https://d0.awsstatic.com/whitepapers/RDS/Best-Practices-for-Migrating-MySQL-Databases-to-Amazon-Aurora.pdf

16 http://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/Welcome.html

17 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.CreateInstance.html

18 http://dev.mysql.com/doc/refman/5.6/en/create-view.html

19 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.LoadFromS3.html

20 http://dev.mysql.com/doc/refman/5.6/en/select-into.html

21 http://dev.mysql.com/doc/refman/5.6/en/error-messages-server.html

22 http://dev.mysql.com/doc/refman/5.6/en/error-messages-client.html

23 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_VPC.WorkingWithRDSInstanceinaVPC.html

24 http://docs.aws.amazon.com/cli/latest/reference/rds/modify-db-cluster.html

[25] http://dev.mysql.com/doc/refman/5.6/en/resetting-permissions.html

[26] http://dev.mysql.com/doc/refman/5.6/en/set-password.html

[27] https://aws.amazon.com/premiumsupport/

[28] https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_lower_case_table_names

[29] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithParamGroups.html

[30] http://dev.mysql.com/doc/refman/5.6/en/gone-away.html

[31] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html#instance-store-volumes

[32] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Monitoring.html

[33] http://dev.mysql.com/doc/refman/5.6/en/binary-log.html

[34] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Monitoring.html#Aurora.Monitoring.Metrics

[35] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/mysql_rds_stop_replication.html

[36] http://dev.mysql.com/doc/refman/5.6/en/replication.html

[37] https://dev.mysql.com/doc/refman/5.6/en/show-slave-status.html

[38] https://dev.mysql.com/doc/refman/5.6/en/slave-io-thread-states.html

[39] https://dev.mysql.com/doc/refman/5.6/en/slave-sql-thread-states.html

[40] http://dev.mysql.com/doc/refman/5.6/en/binary-log-formats.html

[41] http://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit

[42] https://dev.mysql.com/doc/refman/5.6/en/replication-options-slave.html#sysvar_slave_parallel_workers

[43] http://dev.mysql.com/doc/refman/5.6/en/innodb-standard-monitor.html

44 http://dev.mysql.com/doc/refman/5.6/en/query-cache.html

45 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/mysql_rds_next_master_log.html

46 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_LogAccess.Concepts.MySQL.html#USER_LogAccess.MySQL.Generallog

47 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Auditing.html

48 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/mysql_rds_skip_repl_error.html

49 http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Aurora.html

50 https://d0.awsstatic.com/whitepapers/RDS/Migrating%20your%20databases%20to%20Amazon%20Aurora.pdf

51 https://d0.awsstatic.com/whitepapers/RDS/Best-Practices-for-Migrating-MySQL-Databases-to-Amazon-Aurora.pdf