Amazon Aurora Migration Handbook

July 2020

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. 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

Introduction 5
Database Migration Considerations 6
Migration Phases 7
Features and Compatibility 7
Performance 8
Cost 9
Availability and Durability 9
Planning and Testing a Database Migration 11
Homogeneous Migrations 11
Summary of Available Migration Methods 12
Migrating Large Databases to Amazon Aurora 15
Partition and Shard Consolidation on Amazon Aurora 16
MySQL and MySQL compatible Migration Options at a Glance 17
Migrating from Amazon RDS for MySQL 18
Migrating from MySQL-Compatible Databases 23
Heterogeneous Migrations 26
Schema Migration 27
Data Migration 28
Example Migration Scenarios 28
Self-Managed Homogeneous Migrations 28
Multi-Threaded Migration Using mydumper and myloader 39
Heterogeneous Migrations 45
Testing and Cutover 46
Migration Testing 46

Cutover 47
Troubleshooting 49
Troubleshooting MySQL Specific Issues 49
Conclusion 54
Contributors 55
Further Reading 56
Document Revisions 56

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

For decades, traditional relational databases have been the primary choice for data storage and persistence. These database systems continue to rely on monolithic architectures and were not designed to take advantage of cloud infrastructure. These monolithic architectures present many challenges, particularly in areas such as cost, flexibility, and availability. In order to address these challenges, AWS redesigned relational database for the cloud infrastructure and introduced Amazon Aurora.

Amazon Aurora is a MySQL-compatible relational database engine that combines the speed, availability, and security of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. Aurora provides up to five times better performance than MySQL and comparable performance of high-end commercial databases. Amazon Aurora is priced at one-tenth the cost of commercial engines.

Amazon Aurora is available through the Amazon Relational Database Service (Amazon RDS) platform. Like other Amazon RDS databases, Aurora is a fully managed database service. With the Amazon RDS platform, most database management tasks such as hardware provisioning, software patching, setup, configuration, monitoring, and backup are completely automated.

Amazon Aurora is built for mission-critical workloads and is highly available by default. An Aurora database cluster spans multiple Availability Zones (AZs) in a region, providing out-of-the-box durability and fault tolerance to your data across physical data centers. An Availability Zone is composed of one or more highly available data centers operated by Amazon. AZs are isolated from each other and are connected through low-latency links. Each segment of your database volume is replicated six times across these AZs.

Aurora cluster volumes automatically grow as the amount of data in your database increases with no performance or availability impact, so there is no need for estimating and provisioning large amount of database storage ahead of time. An Aurora cluster volume can grow to a maximum size of 64 terabytes (TB). You are only charged for the space that you use in an Aurora cluster volume.

Aurora's automated backup capability supports point-in-time recovery of your data, enabling you to restore your database to any second during your retention period, up to the last five minutes. Automated backups are stored in Amazon Simple Storage Service (Amazon S3), which is designed for 99.999999999% durability. Amazon Aurora backups are automatic, incremental, and continuous and have no impact on database performance.

For applications that need read-only replicas, you can create up to 15 Aurora Replicas per Aurora database with very low replica lag. These replicas share the same underlying storage as the source instance, lowering costs and avoiding the need to perform writes at the replica nodes.

Amazon Aurora is highly secure and allows you to encrypt your databases using keys that you create and control through AWS Key Management Service (AWS KMS). On a database instance running with Amazon Aurora encryption, data stored at rest in the underlying storage is encrypted, as are the automated backups, snapshots, and replicas in the same cluster. Amazon Aurora uses SSL (AES-256) to secure data in transit.

For a complete list of Aurora features, see Amazon Aurora. Given the rich feature set and cost effectiveness of Amazon Aurora, it is increasingly viewed as the go-to database for mission-critical applications.

Database Migration Considerations

A database represents a critical component in the architecture of most applications. Migrating the database to a new platform is a significant event in an application’s lifecycle and may have an impact on application functionality, performance, and reliability. You should take a few important considerations into account before embarking on your first migration project to Amazon Aurora.

Migrations are among the most time-consuming and critical tasks handled by database administrators. Although the task has become easier with the advent of managed

migration services such as AWS Database Migration Service, large-scale database migrations still require adequate planning and execution to meet strict compatibility and performance requirements.

Migration Phases

Because database migrations tend to be complex, we advocate taking a phased, iterative approach.

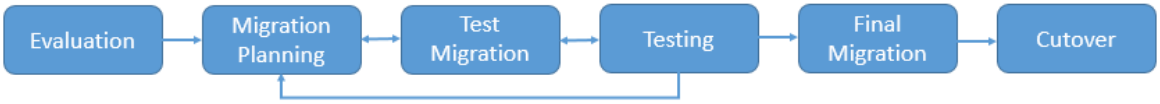

Figure 1 - Migration phases

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

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.

Features and Compatibility

Although most applications can be architected to work with many relational database engines, you should make sure that your application works with Amazon Aurora. Amazon Aurora is designed to be wire-compatible with MySQL 5.5,5.6,5.7 and 8.0 Therefore, most of the code, applications, drivers, and tools that are used today with MySQL databases can be used with Aurora with little or no change.

However, certain MySQL features, like the MyISAM storage engine, are not available with Amazon Aurora. Also, due to the managed nature of the Aurora service, SSH

access to database nodes is restricted, which may affect your ability to install third-party tools or plugins on the database host.

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

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. Therefore, many successful database migration projects start with performance evaluations of the new database platform. Although the RDS Aurora Performance Assessment Benchmarking paper gives you a decent idea of overall database performance, these benchmarks do not emulate the data access patterns of your applications. For more useful results, test the database performance for time-sensitive workloads by running your queries (or subset of your queries) on the new platform directly.

Consider these strategies:

- If your current database is MySQL, migrate to Amazon Aurora with downtime and performance test your database with a test or staging version of your application or by replaying the production workload.
- If you are on a non-MySQL compliant engine, you can selectively copy the busiest tables to Amazon Aurora and test your queries for those tables. This gives you a good starting point. Of course, testing after complete data migration will provide a full picture of real-world performance of your application on the new platform.

Amazon Aurora delivers comparable performance with commercial engines and significant improvement over MySQL performance. It does this by tightly integrating the database engine with an SSD-based virtualized storage layer designed for database workloads. This reduces writes to the storage system, minimizes lock contention, and

eliminates delays created by database process threads. Our tests with SysBench on r3.8xlarge instances show that Amazon Aurora delivers over 585,000 reads per second and 107,000 writes per second, five times higher than MySQL running the same benchmark on the same hardware.

One area where Amazon Aurora significantly improves upon traditional MySQL is highly concurrent workloads. In order to maximize your workload's throughput on Amazon Aurora, we recommend architecting your applications to drive a large number of concurrent queries.

Cost

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

Owning and running databases come with associated costs. Before planning a database migration, an analysis of the total cost of ownership (TCO) of the new database platform is imperative. Migration to a new database platform should ideally lower the total cost of ownership while providing your applications with similar or better features. If you are running an open source database engine (MySQL, Postgres), your costs are largely related to hardware, server management, and database management activities. However, if you are running a commercial database engine (Oracle, SQL Server, DB2 etc.), a significant portion of your cost is database licensing.

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.

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 by having the following components:

1. **Read replicas**: Increase read throughput to support high-volume application requests by creating up to 15 database Aurora replicas. Amazon Aurora Replicas share the same underlying storage as the source instance, lowering costs and avoiding the need to perform writes at the replica nodes. This frees up more processing power to serve read requests and reduces the replica lag time, often down to single digit milliseconds. Aurora provides a reader endpoint so the application can connect without having to keep track of replicas as they are added and removed. Aurora also supports auto-scaling, where it automatically adds and removes replicas in response to changes in performance metrics that you specify. Aurora supports cross-region read replicas. Cross-region replicas provide fast local reads to your users, and each region can have an additional 15 Aurora replicas to further scale local reads
2. **Global Database**: You can choose between Global Database, which provides the best replication performance, and traditional binlog-based replication. You can also set up your own binlog replication with external MySQL databases. Amazon Aurora Global Database is designed for globally distributed applications, allowing a single Amazon Aurora database to span multiple AWS regions. It replicates your data with no impact on database performance, enables fast local reads with low latency in each region, and provides disaster recovery from region-wide outages.
3. **Multi-AZ**: Aurora stores copies of the data in a DB cluster across multiple Availability Zones in a single AWS Region, regardless of whether the instances in the DB cluster span multiple Availability Zones. For more information on Aurora, see Managing an Amazon Aurora DB Cluster. When data is written to the primary DB instance, Aurora synchronously replicates the data across Availability Zones to six storage nodes associated with your cluster volume. Doing so provides data redundancy, eliminates I/O freezes, and minimizes latency spikes during system backups. Running a DB instance with high availability can enhance availability during planned system maintenance, and help protect your databases against failure and Availability Zone disruption

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

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)

Common method is widely adopted is built aurora read replica asynchronized with source master RDS or self-managed MySQL databases.

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*.

Percona XtraBackup

Compatible sources and limitations:

- On-premises or self-managed MySQL 5.6 in EC2 can be migrated zero downtime migration
- You can't restore into an existing RDS instance using this method
- The total size is limited to 6 TB
- User accounts, functions, and stored procedures are not imported automatically.

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*.

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

DMS Migration

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*.

> For a heterogeneous migration, where you are migrating from a database engine other than MySQL to a MySQL database, AWS DMS is almost always the best migration tool to use. But for homogeneous migration, where you are migrating from a MySQL database to a MySQL database, native tools can be more effective.

Using Any MySQL Compatible Database as a Source for AWS DMS:

Before you begin to work with a MySQL database as a source for AWS DMS, make sure that you the following prerequisites. These prerequisites apply to either self-managed or Amazon managed sources.

You must have an account for AWS DMS that has the *Replication Admin Role*. The role needs the following privileges:

- **Replication Client:** This privilege is required for change data capture (CDC) tasks only. In other words, full-load-only tasks don't require this privilege
- **Replication Slave:** This privilege is required for change data capture (CDC) tasks only. In other words, full-load-only tasks don't require this privilege
- **Super:** This privilege is required only in MySQL versions before 5.6.6

DMS 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.

Migrating Large Databases to Amazon Aurora

Migration of large datasets presents unique challenges in every database migration project. Many successful large database migration projects use a combination of the following strategies:

- **Migration with continuous replication:** Large databases typically have extended downtime requirements while moving data from source to target. To reduce the downtime, you can first load baseline data from source to target and then enable replication (using MySQL native tools, AWS DMS, or third-party tools) for changes to catch up.
- **Copy static tables first:** If your database relies on large static tables with reference data, you may migrate these large tables to the target database before migrating your active dataset. You can leverage AWS DMS to copy tables selectively or export and import these tables manually.
- **Multiphase migration:** Migration of large database with thousands of tables can be broken down into multiple phases. For example, you may move a set of tables with no cross joins queries every weekend until the source database is fully migrated to the target database. Note that in order to achieve this, you need to make changes in your application to connect to two databases simultaneously while your dataset is on two distinct nodes. Although this is not a common migration pattern, this is an option nonetheless.
- **Database cleanup:** Many large databases contain data and tables that remain unused. In many cases, developers and DBAs keep backup copies of tables in the same database, or they just simply forget to drop unused tables. Whatever the reason, a database migration project provides an opportunity to clean up the existing database before the migration. If some tables are not being used, you might either drop them or archive them to another database. You might also delete old data from large tables or archive that data to flat files.

Partition and Shard Consolidation on Amazon Aurora

If you are running multiple shards or functional partitions of your database to achieve high performance, you have an opportunity to consolidate these partitions or shards on a single Aurora database. A single Amazon Aurora instance can scale up to 64 TB, supports thousands of tables, and supports a significantly higher number of reads and writes than a standard MySQL database. Consolidating these partitions on a single Aurora instance not only reduces the total cost of ownership and simplify database management, but it also significantly improves performance of cross-partition queries.

- **Functional partitions:** Functional partitioning means dedicating different nodes to different tasks. For example, in an e-commerce application, you might have one database node serving product catalog data, and another database node capturing and processing orders. As a result, these partitions usually have distinct, nonoverlapping schemas.
    - **Consolidation strategy:** Migrate each functional partition as a distinct schema to your target Aurora instance. If your source database is MySQL compliant, use native MySQL tools to migrate the schema and then use AWS DMS to migrate the data, either one time or continuously using replication. If your source database is non-MySQL complaint, use AWS Schema Conversion Tool to migrate the schemas to Aurora and use AWS DMS for one-time load or continuous replication.
- **Data shards:** If you have the same schema with distinct sets of data across multiple nodes, you are leveraging database sharding. For example, a high-traffic blogging service may shard user activity and data across multiple database shards while keeping the same table schema.

- **Consolidation strategy:** Since all shards share the same database schema, you only need to create the target schema once. If you are using a MySQL-compliant database, use native tools to migrate the database schema to Aurora. If you are using a non-MySQL database, use AWS Schema Conversion Tool to migrate the database schema to Aurora. Once the database schema has been migrated, it is best to stop writes to the database shards and use native tools or an AWS DMS one-time data load to migrate an individual shard to Aurora. If writes to the application cannot be stopped for an extended period, you might still use AWS DMS with replication but only after proper planning and testing.

MySQL and MySQL compatible Migration Options at a Glance

Source Database Type	Migration with Downtime	Near-zero Downtime Migration
Amazon RDS MySQL	Option 1: RDS snapshot migration Option 2: Manual migration using native tools* Option 3: Schema migration using native tools and data load using AWS DMS	Option 1: Migration using native tools + binlog replication Option 2: RDS snapshot migration + binlog replication Option 3: Schema migration using native tools + AWS DMS for data movement
MySQL Amazon EC2 or on-premises	Option 1: Schema migration with native tools + AWS DMS for data load	Option 1: Schema migration using native tools + AWS DMS to move data
Oracle/SQL server	Option 1: AWS Schema Conversion Tool + AWS DMS (recommended) Option 2: Manual or third-party tool for schema conversion + manual or third-party data load in target	Option 1: AWS Schema Conversion Tool + AWS DMS (recommended) Option 2: Manual or third-party tool for schema conversion.

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*.

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.

Estimating Space Requirements for Snapshot Migration

When you migrate a snapshot of a MySQL DB instance to an Aurora DB cluster, Aurora uses an Amazon Elastic Block Store (Amazon EBS) volume to format the data from the snapshot before migrating it. There are some cases where additional space is needed to format the data for migration. The two features that can potentially cause space issues during migration are MyISAM tables and using the ROW_FORMAT=COMPRESSED option. If you are not using either of these features in your source database, then you can skip this section because you should not have space issues. During migration, MyISAM tables are converted to InnoDB and any compressed tables are uncompressed. Consequently, there must be adequate room for the additional copies of any such tables.

The size of the migration volume is based on the allocated size of the source MySQL database that the snapshot was made from. Therefore, if you have MyISAM or compressed tables that make up a small percentage of the overall database size and there is available space in the original database, then migration should succeed without encountering any space issues. However, if the original database would not have enough room to store a copy of converted MyISAM tables as well as another (uncompressed) copy of compressed tables, then the migration volume will not be big enough. In this situation, you would need to modify the source Amazon RDS MySQL

database to increase the database size allocation to make room for the additional copies of these tables, take a new snapshot of the database, and then migrate the new snapshot.

When migrating data into your DB cluster, observe the following guidelines and limitations:

- Although Amazon Aurora supports up to 64 TB of storage, the process of migrating a snapshot into an Aurora DB cluster is limited by the size of the Amazon EBS volume of the snapshot, and therefore is limited to a maximum size of 6 TB.

Non MyISAM tables in the source database can be up to 6 TB in size. However, due to additional space requirements during conversion, make sure that none of the MyISAM and compressed tables being migrated from your MySQL DB instance exceed 3 TB in size. For more information, see Migrating Data from an Amazon RDS MySQL DB Instance to an Amazon Aurora MySQL DB Cluster.

You might want to modify your database schema (convert MyISAM tables to InnoDB and remove `ROW_FORMAT=COMPRESSED`) prior to migrating it into Amazon Aurora. This can be helpful in the following cases:

- You want to speed up the migration process.
- You are unsure of how much space you need to provision.
- You have attempted to migrate your data and the migration has failed due to a lack of provisioned space.

Make sure that you are not making these changes in your production Amazon RDS MySQL database but rather on a database instance that was restored from your production snapshot. For more details on doing this, see Reducing the Amount of Space Required to Migrate Data into Amazon Aurora in the *Amazon RDS User Guide*.

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*.

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*.

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 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:

1. MySQL utilities such as mysqldump, mysqlimport, and mysql command- line client.
2. Third-party utilities such as **mydumper** and **myloader**. For details, see this mydumper project page.
3. 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.

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 Database Migration Service 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 Database Migration Service 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 Database Migration Service 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 DMS CDC replication uses plain SQL statements from binlog to apply data changes in the target database. Therefore, it might be slower and more resource-intensive than the native master/slave 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, Sybase, DB2, Azure SQL Database, Terradata, Greenplum, Vertica, Cassandra and PostgreSQL etc. 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.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 in the scope of SCT compatible databases, 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:** 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. For more information about using Percona Xtrabackup utility, see Migrating Data from an External MySQL Database, in the *Amazon RDS User Guide*.

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 an External MySQL Database, 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: 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.

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 Migrating Data from MySQL by Using an Amazon S3 Bucket, in the *Amazon RDS User Guide*.

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 5.6 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*.

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.

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:** 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: 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 verify 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. Prior to running migration, we suggest you to review Proof of Concept with Aurora to

understand the volume of data and representative of your production environment as a blueprint.

Testing and Cutover

Once the schema and data have been successfully migrated from the source database to Amazon Aurora, you are now ready to perform end-to-end testing of your migration process. The testing approach should be refined after each test migration, and the final migration plan should include a test plan that ensures adequate testing of the migrated database.

Migration Testing

Test Category	Purpose
Basic acceptance tests	These pre-cutover tests should be automatically executed upon completion of the data migration process. Their primary purpose is to verify whether the data migration was successful. Following are some common outputs from these tests: • Total number of items processed • Total number of items imported • Total number of items skipped • Total number of warnings • Total number of errors If any of these totals reported by the tests deviate from the expected values, then it means the migration was not successful, and the issues need to be resolved before moving to the next step in the process or the next round of testing.

Test Category	Purpose
Functional tests	These post-cutover tests exercise the functionality of the application(s) using Aurora for data storage. They include a combination of automated and manual tests. The primary purpose of the functional tests is to identify problems in the application caused by the migration of the data to Aurora.
Nonfunctional tests	These post-cutover tests assess the nonfunctional characteristics of the application, such as performance under varying levels of load.
User acceptance tests	These post-cutover tests should be executed by the end users of the application once the final data migration and cutover is complete. The purpose of these tests is for the end users to decide if the application is sufficiently usable to meet its primary function in the organization.

Cutover

Once you have completed the final migration and testing, it is time to point your application to the Amazon Aurora database. This phase of migration is known as cutover. If the planning and testing phase has been executed properly, cutover should not lead to unexpected issues.

Pre-cutover Actions

- Choose a cutover window: Identify a block of time when you can accomplish cutover to the new database with minimum disruption to the business. Normally you would select a low activity period for the database (typically nights and/or weekends).

- Make sure changes are caught up: If a near-zero downtime migration approach was used to replicate database changes from the source to the target database, make sure that all database changes are caught up and your target database is not significantly lagging behind the source database.
- Prepare scripts to make the application configuration changes: In order to accomplish the cutover, you need to modify database connection details in your application configuration files. Large and complex applications may require updates to connection details in multiple places. Make sure you have the necessary scripts ready to update the connection configuration quickly and reliably.
- Stop the application: Stop the application processes on the source database and put the source database in read-only mode so that no further writes can be made to the source database. If the source database changes aren't fully caught up with the target database, wait for some time while these changes are fully propagated to the target database.
- Execute pre-cutover tests: Run automated pre-cutover tests to make sure that the data migration was successful.

Cutover

- Execute cutover: If pre-cutover checks were completed successfully, you can now point your application to Amazon Aurora. Execute scripts created in the pre-cutover phase to change the application configuration to point to the new Aurora database.
- Start your application: At this point, you may start your application. If you have an ability to stop users from accessing the application while the application is running, exercise that option until you have executed your post-cutover checks.

Post-cutover Checks

- Execute post-cutover tests: Execute predefined automated or manual test cases to make sure your application works as expected with the new database. It's a good strategy to start testing read-only functionality of the database first before executing tests that write to the database.

Enable user access and closely monitor: If your test cases were executed successfully, you may give user access to the application to complete the migration process. Both application and database should be closely monitored at this time.

Troubleshooting

The following sections provide examples of common issues and error messages to help you troubleshoot heterogenous DMS migrations.

Troubleshooting MySQL Specific Issues

The following issues are specific to using AWS DMS with MySQL databases.

Topics

- CDC Task Failing for Amazon RDS DB Instance Endpoint Because Binary Logging Disabled
- Connections to a target MySQL instance are disconnected during a task
- Adding Autocommit to a MySQL-compatible Endpoint
- Disable Foreign Keys on a Target MySQL-compatible Endpoint
- Characters Replaced with Question Mark
- "Bad event" Log Entries
- Change Data Capture with MySQL 5.5
- Increasing Binary Log Retention for Amazon RDS DB Instances
- Log Message: Some changes from the source database had no impact when applied to the target database.
- Error: Identifier too long
- Error: Unsupported Character Set Causes Field Data Conversion to Fail
- Error: Codepage 1252 to UTF8 [120112] A field data conversion failed

CDC Task Failing for Amazon RDS DB Instance Endpoint Because Binary Logging Disabled

This issue occurs with Amazon RDS DB instances because automated backups are disabled. Enable automatic backups by setting the backup retention period to a non-zero value.

Connections to a target MySQL instance are disconnected during a task

If you have a task with LOBs that is getting disconnected from a MySQL target with the following type of errors in the task log, you might need to adjust some of your task settings.

```
[TARGET_LOAD ]E: RetCode: SQL_ERROR SqlState: 08S01 NativeError:
2013 Message: [MySQL][ODBC 5.3(w) Driver][mysqld-5.7.16-log]Lost
connection
to MySQL server during query [122502] ODBC general error.
```

To solve the issue where a task is being disconnected from a MySQL target, do the following:

- Check that you have your database variable `max_allowed_packet` set large enough to hold your largest LOB.
- Check that you have the following variables set to have a large timeout value. We suggest you use a value of at least 5 minutes for each of these variables.
    - `net_read_timeout`
    - `net_write_timeout`
    - `wait_timeout`
    - `interactive_timeout`

Adding Autocommit to a MySQL-compatible Endpoint

To add autocommit to a target MySQL-compatible endpoint, use the following procedure:

1. Sign in to the AWS Management Console and select **DMS**.
2. Select **Endpoints**.
3. Select the MySQL-compatible target endpoint that you want to add autocommit to.
4. Select **Modify**.
5. Select **Advanced**, and then add the following code to the **Extra connection attributes** text box:

```
Initstmt= SET AUTOCOMMIT=1
```

6. Choose **Modify**.

Disable Foreign Keys on a Target MySQL-compatible Endpoint

You can disable foreign key checks on MySQL by adding the following to the **Extra Connection Attributes** in the **Advanced** section of the target MySQL, Amazon Aurora with MySQL compatibility, or MariaDB endpoint.

To disable foreign keys on a target MySQL-compatible endpoint, use the following procedure:

1. Sign in to the AWS Management Console and select **DMS**.
2. Select **Endpoints**.
3. Select the MySQL, Aurora MySQL, or MariaDB target endpoint that you want to disable foreign keys.
4. Select **Modify**.
5. Select **Advanced**, and then add the following code to the **Extra connection attributes** text box:

```
Initstmt=SET FOREIGN_KEY_CHECKS=0
```

6. Choose **Modify**.

Characters Replaced with Question Mark

The most common situation that causes this issue is when the source endpoint characters have been encoded by a character set that AWS DMS doesn't support. For example, AWS DMS engine versions prior to version 3.1.1 don't support the UTF8MB4 character set.

Bad event Log Entries

Bad event entries in the migration logs usually indicate that an unsupported DDL operation was attempted on the source database endpoint. Unsupported DDL operations cause an event that the replication instance cannot skip so a bad event is logged. To fix this issue, restart the task from the beginning, which will reload the tables and will start capturing changes at a point after the unsupported DDL operation was issued.

Change Data Capture with MySQL 5.5

AWS DMS change data capture (CDC) for Amazon RDS MySQL-compatible databases requires full image row-based binary logging, which is not supported in MySQL version 5.5 or lower. To use AWS DMS CDC, you must up upgrade your Amazon RDS DB instance to MySQL version 5.6.

Increasing Binary Log Retention for Amazon RDS DB Instances

AWS DMS requires the retention of binary log files for change data capture. To increase log retention on an Amazon RDS DB instance, use the following procedure. The following example increases the binary log retention to 24 hours.

```
call mysql.rds_set_configuration('binlog retention hours', 24);
```

Log Message: Some changes from the source database had no impact when applied to the target database.

When AWS DMS updates a MySQL database column's value to its existing value, a message of `zero rows affected` is returned from MySQL. This behavior is unlike other database engines such as Oracle and SQL Server that perform an update of one row, even when the replacing value is the same as the current one.

Error: Identifier too long

The following error occurs when an identifier is too long:

```
TARGET_LOAD E: RetCode: SQL_ERROR SqlState: HY000 NativeError:
1059 Message: MySQLhttp://ODBC 5.3(w) Driverhttp://mysqld-
5.6.10Identifier
name '<name>' is too long 122502 ODBC general error.
(ar_odbc_stmt.c:4054)
```

When AWS DMS is set to create the tables and primary keys in the target database, it currently does not use the same names for the Primary Keys that were used in the source database. Instead, AWS DMS creates the Primary Key name based on the tables name. When the table name is long, the auto-generated identifier created can be longer than the allowed limits for MySQL. The solve this issue, currently, pre-create the tables and Primary Keys in the target database and use a task with the task setting **Target table preparation mode** set to **Do nothing** or **Truncate** to populate the target tables.

Error: Unsupported Character Set Causes Field Data Conversion to Fail

The following error occurs when an unsupported character set causes a field data conversion to fail:

```
"[SOURCE_CAPTURE ]E: Column '<column name>' uses an unsupported character
set [120112]
A field data conversion failed. (mysql_endpoint_capture.c:2154)
```

This error often occurs because of tables or databases using UTF8MB4 encoding. AWS DMS engine versions prior to 3.1.1 don't support the UTF8MB4 character set. In addition, check your database's parameters related to connections. The following command can be used to see these parameters:

```
SHOW VARIABLES LIKE '%char%';
```

Error: Codepage 1252 to UTF8 [120112] A field data conversion failed

The following error can occur during a migration if you have non codepage-1252 characters in the source MySQL database.

```
[SOURCE_CAPTURE ]E: Error converting column 'column_xyz' in table
'table_xyz with codepage 1252 to UTF8 [120112] A field data conversion
failed.
(mysql_endpoint_capture.c:2248)
```

As a workaround, you can use the `CharsetMapping` extra connection attribute with your source MySQL endpoint to specify character set mapping. You might need to restart the AWS DMS migration task from the beginning if you add this extra connection attribute.

For example, the following extra connection attribute could be used for a MySQL source endpoint where the source character set is `utf8` or `latin1`. 65001 is the UTF8 code page identifier.

```
CharsetMapping=utf8,65001
CharsetMapping=latin1,65001
```

Conclusion

Amazon Aurora is a high performance, highly available, and enterprise-grade database built for the cloud. Leveraging Amazon Aurora can result in better performance and greater availability than other open-source databases and lower costs than most commercial grade databases. This paper proposes strategies for identifying the best method to migrate databases to Amazon Aurora and details the procedures for planning

and executing those migrations. In particular, AWS Database Migration Service (AWS DMS) as well as the AWS Schema Conversion Tool are the recommended tools for heterogeneous migration scenarios. These powerful tools can greatly reduce the cost and complexity of database migrations.

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

Contributors to this document include:

- Bala Mugunthan, Sr. Partner Solution Architect, Amazon Web Services
- Ashar Abbas, Database Specialty Architect
- Sijie Han, SA Manager, Amazon Web Services
- Szymon Komendera, Database Engineer, Amazon Web Services

Further Reading

For additional information, see:

- Aurora on Amazon RDS User Guide
- Migrating Your Databases to Amazon Aurora AWS whitepaper
- Best Practices for Migrating MySQL Databases to Amazon Aurora AWS whitepaper

Document Revisions

Date	Description
July 2020	Added information for the large databases migrations on Amazon Aurora and functional partition and data shard consolidation strategies are discussed in homogenous migration sections. Multi-threaded migration using **mydumper** and **myloader** open source tools are introduced. Overall basic acceptance testing, functional test, non-functional test, and user acceptance tests are explained in the testing phase and pre, cutover and post cut-overs phase scenarios are further explained.
September 2019	First publication