Amazon RDS for Aurora Export/Import Performance Best Practices

Szymon Komendera
Database Support Operations, Amazon Web Services

Contents

This document discusses some of the important factors affecting the performance of export/import operations for Amazon Relational Database Service (Amazon RDS) for Amazon Aurora. Although these performance topics discuss Amazon RDS for Aurora, many of the ideas presented also apply to the MySQL Community Edition found in RDS MySQL and self-managed MySQL installations.

Note

This document is not an exhaustive list of export/import performance topics . We provide it to cover basic concepts and technical background, using which export/import solutions can be built.

Basic Performance Concepts for Export/Import Operations

To achieve optimal performance for export/import operations, we recommend that you take a certain set of approaches. The most important points are highlighted directly following, with more detailed discussion later in the document.

Client-Side Considerations

Generally, you should perform a dump or import from an RDS DB instance launched in the same location as the database server:

- For on-premises servers, the client machine should be in the same on-premises network.
- For either RDS or Amazon Elastic Compute Cloud (Amazon EC2) server instances, the client instance should exist in the same Amazon Virtual Private Cloud (Amazon VPC) and Availability Zone (AZ) as the server. In case of EC2-Classic (that is, non-VPC) servers, the client should be located in the same AWS region and AZ.

In order to follow the preceding recommendations during migrations between distant databases, it may be necessary to use two client machines:

- One in the source network, so that it's close to the server you're migrating from.
- Another in the target network, so that it's close to the server you're migrating to.

In such a case, you can move dump files between client machines using file transfer protocols or Amazon Simple Storage Service (Amazon S3). To further reduce the total migration time, you can compress files prior to transferring them.

Regardless of its location, the client machine should have adequate CPU, I/O, and network capacity to perform the operation. Although the definition of adequate varies between use cases, the general recommendations are as follows:

- If the dump or export involves processing of data, for example real-time compression or decompression, choose an instance class with at least one CPU core per dump or import thread.
- Ensure that there's enough network throughput available to the client instance by choosing an instance class that supports enhanced networking. For more information, see http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.
- Ensure that the client's storage layer provides the expected read/write performance. For example, if you expect to dump or load data at 100 MB/second, the instance and its underlying Amazon Elastic Block Store (Amazon EBS) volumes must provide at least 100 MB/second (800 Mbps) of throughput.

For best performance on Linux instances, we recommend that you enable the receive packet steering (RPS) and receive flow steering (RFS) features.

To enable RPS, use the following code:

```
sudo sh -c 'for x in /sys/class/net/eth0/queues/rx-*; do echo ffffffff > $x/rps_cpus;
done'
sudo sh -c "echo 4096 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt"
sudo sh -c "echo 4096 > /sys/class/net/eth0/queues/rx-1/rps_flow_cnt
```

To enable RFS, use the following code:

```
sudo sh -c "echo 32768 > /proc/sys/net/core/rps_sock_flow_entries"
```

For more information, see Client-Side Considerations in the Other Considerations for Export/Import Operations topic following.

Server-Side Considerations

To dump or ingest data at optimal speed, the DB instance must have enough I/O and CPU capacity.

In traditional databases, I/O performance is often the ultimate bottleneck during exports and imports. Amazon RDS for Aurora addresses this challenge by providing a custom, distributed storage layer designed to provide low latency and high throughput under multithreaded workloads. That said, you don't have to choose between storage types or provision storage specifically for export/import purposes.

For CPU power, we recommend one CPU core per thread for exports and two CPU cores per thread for imports. If you've chosen an instance class with enough CPU cores to handle your export or import, the instance should already offer adequate network performance.

For more information, see Server-Side Considerations in the Other Considerations for Export/Import Operations topic following.

Tooling and Procedures

Recommendations for tooling and procedures generally fall in the following categories.

Whenever possible, export and import operations should be performed in multithreaded fashion. On modern systems equipped with multicore CPUs and distributed storage, this technique ensures that available resources are consumed more efficiently.

Export/import procedures should be designed to avoid unnecessary overhead. The following table provides some solutions for typical bottlenecks.

Import technique	**Potential bottleneck**	**Solution**	**Rule of thumb**
Single-row INSERT statements	SQL processing overhead	Use multirow statements or flat files	Import 1 MB of data per statement
Single- or multirow statements with a small transaction size (for example, INSERT statements in auto-commit mode)	Transactional overhead	Increase transaction size	Use 1000 statements per transaction
Single-threaded import (one table at a time)	Under-utilization of server resources	Import multiple tables in parallel	Use 8 concurrent threads (1 thread = 1 table)

Moreover, if data is being exported from a live, production database, a balance has to be found between the performance of production queries and that of the export itself. In other words, we recommend doing export operations carefully so as not to degrade the performance of production queries.

For more information, see Tooling in the Other Considerations for Export/Import Operations topic and the Procedure Optimizations and Challenges for Export/Import Operations topic following.

Other Considerations for Export/Import Operations

Certain other client-side and server-side considerations apply for export/import operations, as described following.

Client-Side Considerations

Both client location and client specifications can affect export and import processes.

Client Location

The location of the client machine is a critical factor affecting the performance of batch exports/imports, benchmarks, and day-to-day operations alike. A remote client can experience network latency ranging from dozens to hundreds of milliseconds. This latency adds unnecessary overhead to every database operation and results in substantial performance degradation.

We strongly recommend that all types of database activities be performed from an EC2 instance located in the same VPC and AZ as the database server. For EC2-Classic (non-VPC) servers, the client should be located in the same AWS region and AZ.

The reason we recommend that you locate VPC resources not only in the same AWS region but also in the same VPC is that cross-VPC traffic is treated as public and uses publicly routable IP addresses. As such, this traffic must travel through a public network segment, which makes the network path longer and increases communication latency.

Client Specifications

It is a common assumption that the specifications of client machines have little or no impact on batch exports/imports. Although the server side uses most resources, it is still important to remember the following:

- On small client instances, multithreaded exports and imports can become CPU-bound, especially if data is compressed or decompressed on the fly (for example, when a data stream is piped through a compression tool).
- Multithreaded dumps can consume substantial network and I/O bandwidth. This network and bandwidth consumption is particularly important when working with remote clients (which we recommend against using) or instances with very small storage volumes that cannot sustain high I/O throughputs.

All operating systems provide diagnostic tools that can help detect CPU, network, and I/O bottlenecks. When you investigate performance issues, we recommend using these tools and ruling out client-side problems before digging deeper into server configuration.

Server-Side Considerations

Even a highly optimized export/import tool can't work miracles for a database server that has inadequate horsepower to dump or ingest data at reasonable speed. Of all server characteristics, the

following are most important: storage performance, CPU power, and network throughput. Let's discuss them in the context of Amazon RDS for Aurora:

1. **Storage performance:** Aurora uses a custom, distributed storage layer designed to provide low latency and high throughput under multithreaded workloads. That said, you don't have to choose between storage types or provision storage specifically for export/import purposes.
2. **CPU power:** Multithreaded export/import operations can become CPU bound when executed against smaller instance types. We recommend using an instance class with one CPU core per thread for exports and two CPU cores per thread for imports.
3. **Network throughput:** Because Amazon RDS for Aurora does not use Amazon EBS storage, it is not constrained by the bandwidth of dedicated EBS network links that are so important for other RDS engines. As a general rule, if you've chosen an instance class with enough CPU cores to handle your export or import (as discussed previously), the instance should already offer adequate network performance.

What if the export/import operation requires much more CPU or network capacity than day-to-day operations? Thanks to point-and-click scaling feature of RDS, you can temporarily overprovision your instance and then scale it back down when additional capacity is no longer required.

Tooling

Using mysqldump

The usual export approach involves exporting individual tables or entire schemas using the mysqldump command-line tool. The tool produces dumps in the form of SQL files containing data definition language (DDL) and data manipulation language (DML) statements, which carry information about data structures and actual data rows, respectively. The mysqldump tool can also produce flat-file dumps, but this feature is not compatible with RDS instances due to privilege restrictions.

A typical mysqldump output contains the following main types of statements:

- Metadata statements responsible for setting server configuration values for the duration of the import.
- CREATE TABLE statements to create relevant table structures before data can be inserted.
- INSERT statements to populate tables with data. Each INSERT typically contains data from multiple rows, but the dataset for each table is essentially represented as a series of INSERT statements.

The mysqldump-based approach introduces multiple issues related to performance:

- When used with RDS instances, mysqldump cannot produce dumps suitable for parallel loading. The exception is when you manually use multiple mysqldump operations to dump individual tables into separate files.
- When used with RDS instances, mysqldump only supports SQL-format dumps and doesn't support flat file dumps. With SQL-format dumps, each INSERT statement run against the target server incurs some execution overhead. As a consequence, SQL-format files generally import slower than flat files.

- By default, files produced by mysqldump don't include any transaction control statements. Consequently, you have very little control over the size of database transactions used to ingest data. This lack of control can lead to poor performance. Here are a couple of examples:
    - With the default MySQL configuration (that is, auto-commit enabled), each individual INSERT statement runs inside its own transaction, which causes noticeable execution overhead.
    - With auto-commit mode disabled, each table will be populated using one massive transaction. This approach can lead to side-effects such as unnecessary tablespace bloat and long rollback times if an import operation is interrupted.

Files produced by mysqldump can be imported using the following methods, both supported in RDS:

- For SQL format dumps: the mysql command-line tool.
- For flat-file dumps: the mysqlimport client tool or LOAD DATA LOCAL INFILE SQL statement.

Using SELECT INTO OUTFILE

As opposed to SQL-format dumps that contain row data encapsulated in SQL statements, flat file dumps come with very little overhead. The only control characters are delimiters used to separate individual rows and columns. Files in comma-separated value (CSV) format are a good example of the flat-file approach.

Two tools are most commonly used to produce flat file dumps:

- The MySQL SELECT ... INTO OUTFILE statement, which dumps table contents (but not table structure) to a file located in the server's local file system.
- Mysqldump with the **--tab** parameter, which also dumps table contents to a file (by calling SELECT INTO OUTFILE internally) but in addition creates metadata files with relevant CREATE TABLE statements.

Note that due to privilege restrictions, these methods are not supported in Amazon RDS. However, they can be used against self-managed EC2 or on-premises servers to produce dump files that you can import into RDS.

Imports based on flat files have the following advantages:

- The lack of SQL encapsulation results in smaller dump files and reduces execution overhead during import.
- Flat files are typically created in file-per-table fashion, meaning that they can be easily imported in parallel.

Flat files have their disadvantages too, such as the need to import each file using a single, massive transaction unless files and import operations are split manually. This approach can cause unnecessary tablespace bloat and long rollback times if an import operation is interrupted.

Flat files can be imported into RDS instances using the MySQL LOAD DATA INFILE statement or the mysqlimport tool. The latter method, which uses LOAD DATA INFILE internally, supports loading multiple files in parallel. By default, both methods will attempt to load dump files from the server's local file system. You can change this behavior using the LOCAL keyword for LOAD DATA INFILE or the **--local** parameter for mysqlimport. For more details, see the LOAD DATA INFILE and mysqlimport documentation.

Using Third-Party Tools and Alternatives

Mydumper and myloader are two popular, open-source MySQL export/import tools written by Domas Mituzas, Andrew Hutchins, and Mark Leith. These tools operate on SQL-format dumps and are designed to address numerous performance issues found in the legacy mysqldump program. They offer features such as the following:

- Dumping and reloading data in multiple threads
- Dumping tables into separate files
- Creating multiple data chunks (files) per table
- Dumping data and metadata into separate files for easy parsing and management
- Configurable transaction size during import
- The ability to perform dumps in regular intervals

Third-party tools are not the only way to dump and import data efficiently. Given enough effort and coding skill, you can solve issues associated with native SQL or flat file dumps:

- In some cases, you can work around the single-threaded nature of legacy tools by manually dumping multiple groups of tables in parallel (using multiple invocations of the export tool). You can use a similar approach to perform a multithreaded import.
- You can control transaction size by manually splitting large dump files into smaller chunks.

Procedure Optimizations and Challenges for Export/Import Operations

Following are some ways to optimize export/import operations and to deal with some of the challenges.

Disabling Foreign Key Constraints and Unique Key Checks During Import

Disabling foreign constraints is another optimization you can use to speed up the import process. This feature is widely supported even in older tools like mysqldump and doesn't require manual changes.

Unique key checks can be disabled by invoking the following statement prior to loading data:

```
SET unique_checks=0;
```

Note that the above statement disables unique key checks during the lifetime of the database session (connection). When using multiple sessions, make sure to execute the statement in each session.

Choosing the Right Number of Threads for Multithreaded Operations

As a rule of thumb, we recommend using one thread per server CPU core for exports and one thread per two CPU cores for imports. In other words, an instance class should provide one CPU core per thread for exports and two CPU cores per thread for imports.

Importing Data from Legacy MyISAM Sources into InnoDB

We recommend that prior to migrating data, you review your dump files and ensure that each table contains an explicit PRIMARY key or UNIQUE NOT NULL key definition. For more information on this, see https://dev.mysql.com/doc/refman/5.6/en/innodb-index-types.html in the MySQL documentation.

Unlike in MyISAM, the InnoDB engine stores data in what is called a "clustered index." To create that index, InnoDB uses one of the following:

- The table's PRIMARY key.
  or
- The first UNIQUE key with NOT NULL columns, if a PRIMARY key is not defined.
  or
- An internal integer column, if neither PRIMARY nor UNIQUE NOT NULL key is defined. This column is hidden from database users and cannot be referenced in SQL queries (for example, to identify rows in WHERE clauses).

When importing tables from sources other than InnoDB, it's a good practice (though not a strict requirement) to ensure that PRIMARY or UNIQUE NOT NULL key definitions are present in each CREATE TABLE statement. This approach will prevent InnoDB from creating a hidden, internal clustered index.

Importing Individual Large Tables

Data distribution skew is a common characteristic of OLTP databases. In other words, a relatively small number of tables might store the vast majority of the dataset. In this situation, write operations (including imports) can be difficult to parallelize due to MySQL locking and logging internals. Fortunately, due to improvements introduced in Amazon Aurora, you can use multithreaded imports efficiently even in single-table scenarios.

In such scenarios, you can split large tables into chunks and import them in parallel. For more information, see Splitting Input Data into Chunks, following.

If single-table imports don't reach the expected performance even after spitting them into chunks, you can partition large tables into smaller slices. You can do this through built-in InnoDB partitioning or manually by creating multiple tables, each containing a slice of the original dataset. This solution allows for higher concurrency during imports. However, it's not a silver bullet—you can sacrifice performance in the long run. To take this approach without degrading performance over time, do the following:

- Choose your partitioning scheme carefully, with query and application requirements in mind. If you choose partitioning keys incorrectly, severe performance degradation can result.
- Numerous engine features are not supported for tables partitioned using built-in InnoDB partitioning. These include (but aren't limited to): foreign key constraints, FULLTEXT indexes, and spatial data types. Note that these limitations come from MySQL itself and not from the Aurora implementation.

Before partitioning, we recommend performing extensive research into the long-term implications.

Importing Multiple Large Tables

If a dataset is spread evenly across multiple tables and thus is not affected by distribution skew, it is relatively straightforward to parallelize export/import operations. You can simply split the problem of exporting/importing the entire database into subproblems of exporting/importing individual tables.

The task might seem trivial, but it's important to keep these basic recommendations in mind:

- Import multiple tables in parallel, but use only a single thread per table.
- Avoid using single-row INSERT statements.
- Avoid using small transactions, but also don't let each transaction grow too large. For big tables, split each table into 1 GB chunks and import one chunk per transaction.

For more information, see Tooling and Procedures under Basic Performance Concepts for Export/Import Operations, preceding.

Splitting Input Data into Chunks

Importing multiple tables of comparable size is an easy task because each table is typically imported from a single data chunk for small tables, or a contiguous sequence of data chunks for larger tables. In this scenario, you achieve parallelism by importing multiple tables at a time with one thread per table.

The situation changes significantly when importing individual very large tables, in which case we cannot use the aforementioned technique—in some situations, because we only have a single, huge table to import. As a workaround, split tables into multiple chunks so you can import the chunks in parallel. To produce chunks for the database to ingest them at optimal speed, follow these guidelines:

- When importing into partitioned tables, import partitions in parallel with one thread per partition. If individual partitions are too large to use a single data chunk per partition, use one of the solutions following.
- For tables with an auto-incremented PRIMARY key:
    - If PRIMARY key values will be provided in the dump, it is a good practice not to split data in a random fashion. Instead, use range-based partitioning so that each chunk contains a contiguous range of primary key values. (For example, if our table has a PRIMARY key column called `id` and we want each data chunk to contain a contiguous range of values, we can sort the data by `id`, then slice the sorted set into chunks.) This approach reduces locking contention during import and allows it to proceed more quickly.
    - If PRIMARY key values will not be provided in the dump, the engine will generate them automatically for each inserted row. In this case, the data doesn't have to be chunked in any particular way and you can choose the method that's easiest for you to implement.
- If the table does not use an auto-incremented PRIMARY key, data should be split so that each chunk contains a contiguous range of clustered index values, as described preceding. For more information about clustered indexes, see https://dev.mysql.com/doc/refman/5.6/en/innodb-index-types.html in the MySQL documentation.

For more information about importing non-InnoDB tables into Aurora, see Importing Data from Legacy MyISAM Sources into InnoDB.

Reducing the Impact of Long-Running Dump Operations

In most cases, dumps are performed from active database servers that are still part of a production environment. In these situations, a long-running dump can cause elevated latency for production queries, such as the following:

- The dump consumes I/O and network resources that would otherwise be used to handle production workload.
- If the server handles substantial amounts of write load, long dumps performed in single-transaction mode can result in excessive undo logging. This situation can lead to InnoDB tablespace bloat and further performance issues.

Here are some hints that can help you reduce the impact of dump operations on production servers:

- If the server has read replicas, consider performing dumps from the replica instead of the main server.
- If the server is covered by regular backup procedures:
    - Use backup data as input data for the import process (if the backup format allows for that),
    - Use the backup to provision a temporary database, then perform a logical dump from the temporary database.
- If neither replicas nor backups are available:
    - Perform dumps during off-peak hours, when write activity is minimal.
    - Reduce the export concurrency level until database performance reaches acceptable levels.