Best Practices for Migrating MySQL Databases to Amazon Aurora

October 2016

This paper has been archived

For the latest technical content, see the AWS Whitepapers & Guides page:

https://aws.amazon.com/whitepapers

Notices

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

Contents

Abstract

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

Target Audience

The target audience of this paper includes:

- Database and system administrators performing migrations from MySQL-compatible databases into Aurora, where AWS-managed migration tools cannot be used
- Software developers working on bulk data import tools for MySQL-compatible databases

Introduction

Migrations are among the most time-consuming tasks handled by database administrators (DBAs). Although the task becomes easier with the advent of managed migration services such as the AWS Database Migration Service (AWS DMS), many large-scale database migrations still require a custom approach due to performance, manageability, and compatibility requirements.

The total time required to export data from the source repository and import it into the target database is one of the most important factors determining the success of all migration projects. This paper discusses the following major contributors to migration performance:

- Client and server performance characteristics
- The choice of migration tools; without the right tools, even the most powerful client and server machines cannot reach their full potential
- Optimized migration procedures to fully utilize the available client/server resources and leverage performance-optimized tooling

Basic Performance Considerations

The following are basic considerations for client and server performance. Tooling and procedure optimizations are described in more detail in “Tools and Procedures," later in this document.

Client Location

Perform export/import operations from a client machine that is launched in the same location as the database server:

- For on-premises database servers, the client machine should be in the same on-premises network.
- For Amazon RDS or Amazon Elastic Compute Cloud (Amazon EC2) database instances, the client instance should exist in the same Amazon Virtual Private Cloud (Amazon VPC) and Availability Zone as the server.

For EC2-Classic (non-VPC) servers, the client should be located in the same AWS Region and Availability Zone.

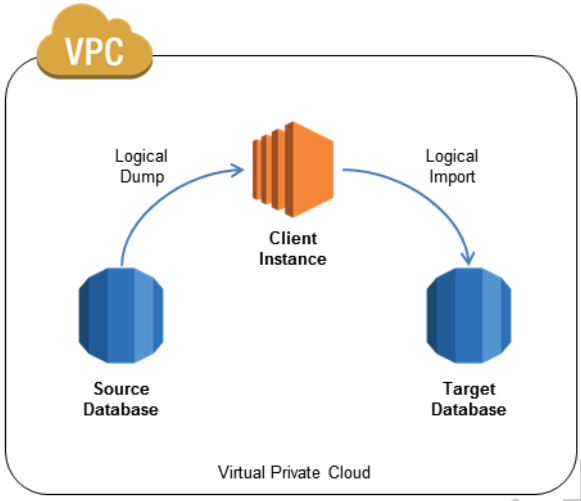

Figure 1: Logical migration between AWS Cloud databases

To follow the preceding recommendations during migrations between distant databases, you might need 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 this case, you can move dump files between client machines using file transfer protocols (such as FTP or SFTP) or upload them to Amazon Simple Storage Service (Amazon S3). To further reduce the total migration time, you can compress files prior to transferring them.

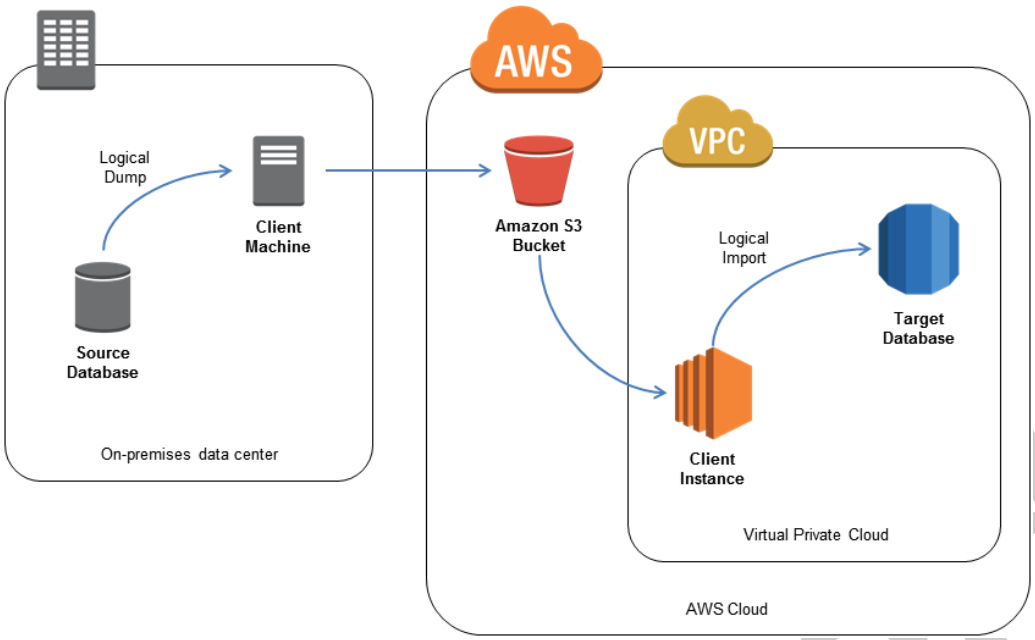

Figure 2: Data flow in a self-managed migration from on-premises to an AWS Cloud database

Client Capacity

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

- If the export or import involves real-time processing of data, for example, compression or decompression, choose an instance class with at least one CPU core per export/import thread.
- Ensure that there is enough network bandwidth available to the client instance. We recommend using instance types that support enhanced networking. For more information, see the Enhanced Networking section in the *Amazon EC2 User Guide*.[1]
- Ensure that the client's storage layer provides the expected read/write capacity. For example, if you expect to dump data at 100 megabytes per second, the instance and its underlying Amazon Elastic Block Store

(Amazon EBS) volume must be capable of sustaining at least 100 MB/s (800 Mbps) of I/O throughput.

Client Configuration

For best performance on Linux client 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"
```

Server Capacity

To dump or ingest data at optimal speed, the database server should have enough I/O and CPU capacity.

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

We recommend using Aurora for instances with 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/import, the instance should already offer adequate network bandwidth.

For more information, see "Server Topics," later in this document.

Tools and Procedures

Whenever possible, perform export and import operations in multithreaded fashion. On modern systems equipped with multicore CPUs and distributed storage, this approach ensures that all available client/server resources are consumed efficiently. Engineer export/import procedures to avoid unnecessary overhead.

The following table lists common export/import performance challenges and provides sample solutions. You can use it do drive your tooling and procedure choices.

Import Technique	Challenge	Potential Solution	Examples
Single-row INSERT statements	Storage and SQL processing overhead	Use multi-row SQL statements Use non-SQL format (e.g., CSV flat files)	Import 1 MB of data per statement Use a set of flat files (chunks), 1 GB each
Single-row or multi-row statements with small transaction size	Transactional overhead, each statement is committed separately	Increase transaction size	Commit once per 1,000 statements
Flat file imports with very large transaction size	Undo management overhead	Reduce transaction size	Commit once per 1 GB of data imported
Single-threaded export/import	Under-utilization of server resources, only one table is accessed at a time	Export/import multiple tables in parallel	Export from or load into 8 tables in parallel

If you are exporting data from an active production database, you have to find a balance between the performance of production queries and that of the export itself. Execute export operations carefully so that you don't compromise the performance of the production workload.

This information is discussed in more detail in the following section.

Advanced Performance Concepts

Client Topics

Contrary to the popular opinion that total migration time depends exclusively on server performance, data migrations can often be constrained by client-side factors. It is important that you identify, understand, and finally address client-side bottlenecks; otherwise, you may not achieve the goal of reaching optimal import/export performance.

Client Location

The location of the client machine is an important factor affecting data migrations, performance benchmarks, and day-to-day database operations alike. Remote clients can experience network latency ranging from dozens to hundreds of milliseconds. Communication latency introduces unnecessary overhead to every database operation and can result in substantial performance degradation.

The performance impact of network latency is particularly visible during single-threaded operations involving large amounts of short database statements. With all statements executed on a single thread, the statement throughput becomes the inverse of network latency, yielding very low overall performance.

We strongly recommend that you perform all types of database activities from an Amazon EC2 instance located in the same VPC and Availability Zone as the database server. For EC2-Classic (non-VPC) servers, the client should be located in the same AWS Region and Availability Zone.

The reason we recommend that you launch client instances not only in the same AWS Region but also in the same VPC is that cross-VPC traffic is treated as public and thus uses publicly routable IP addresses. Because the traffic must travel through a public network segment, the network path becomes longer, resulting in higher communication latency.

Client Capacity

It is a common misconception that the specifications of client machines have little or no impact on export/import operations. Although it is often true that resource utilization is higher on the server side, 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, e.g., when the data stream is piped through a compression tool like **gzip**.
- Multithreaded data migrations can consume substantial network and I/O bandwidth. Choose the instance class and size and type of the underlying Amazon EBS storage volume carefully. For more information, see the Amazon EBS Volume Performance section in the *Amazon EC2 User Guide*.[2]

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

Server Topics

Server-side storage performance, CPU power, and network throughput are among the most important server characteristics affecting batch export/import operations. Aurora supports point-and-click instance scaling that enables you to modify the compute and network capacity of your database cluster for the duration of the batch operations.

Storage Performance

Aurora leverages a purpose-built, distributed storage layer designed to provide low latency and high throughput under multithreaded workloads. You don't need to choose between storage volume types or provision storage specifically for export/import purposes.

CPU Power

Multithreaded exports/imports can become CPU bound when executed against smaller instance types. We recommend using a server instance class with one CPU core per thread for exports and two CPU cores per thread for imports. CPU capacity can be consumed efficiently only if the export/import is realized in multithreaded fashion. Using an instance type with more CPU cores is unlikely to improve performance dump or import that is executed in a single thread.

Network Throughput

Aurora does not use Amazon EBS volumes for storage. As a result, it is not constrained by the bandwidth of EBS network links or throughput limits of the EBS volumes.

However, the theoretical peak I/O throughput of Aurora instances still depends on the instance class. As a rule of thumb, if you choose an instance class with enough CPU cores to handle the export/import (as discussed earlier), the instance should already offer adequate network performance.

Temporary Scaling

In many cases, export/import tasks can require significantly more compute capacity than day-to-day database operations. Thanks to the point-and-click compute scaling feature of Amazon RDS for MySQL and Aurora, you can temporarily overprovision your instance and then scale it back down when you no longer need the additional capacity.

> **Note:** Due to the benefits of the Aurora custom storage layer, storage scaling is not needed before, during, or after exporting/importing data.

Tools

With client and server machines located close to each other and sized adequately, let's look at the different methods and tools you can use to actually move the data.

Percona XtraBackup

Aurora supports migration from Percona XtraBackup files stored in Amazon S3. Migrating from backup files can be significantly faster than migrating from logical schema and data dumps using tools such as **mysqldump**. Logical imports work by executing SQL commands to recreate the schema and data from your source database, which carries considerable processing overhead. However, Percona XtraBackup files can be ingested directly into an Aurora storage volume, which removes the additional SQL execution cost.

A migration from Percona XtraBackup files involves three main steps:

1. Using the **innobackupex** tool to create a backup of the source database.
2. Copying the backup to Amazon S3.
3. Restoring the backup through the AWS RDS console.

You can use this migration method for source servers using MySQL versions 5.5 and 5.6.

For more information and step-by-step instructions for migrating from Percona XtraBackup files, see the *Amazon Relational Database Service User Guide*.[3]

mysqldump

The **mysqldump** tool is perhaps the most popular export/import tool for MySQL-compatible database engines. The tool produces dumps in the form of SQL files that contain data definition language (DDL), data control language (DCL), and data manipulation language (DML) statements. The statements carry information about data structures, data access rules, and the actual data, respectively.

In the context of this whitepaper, two types of statements are of interest:

- 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 certain issues related to performance:

- When used against managed database servers, such as Amazon RDS instances, the tool's functionality is limited. Due to privilege restrictions, it cannot dump data in multiple threads or produce flat-file dumps suitable for parallel loading.
- The SQL files do not include any transaction control statements by default. Consequently, you have very little control over the size of database transactions used to import data. This lack of control can lead to poor performance, for example:
  - With auto-commit mode enabled (default), each individual INSERT statement runs inside its own transaction. The database must COMMIT frequently, which increases the overall execution overhead.
  - With auto-commit mode disabled, each table is populated using one massive transaction. The approach removes COMMIT overhead but leads to side effects such as tablespace bloat and long rollback times if the import operation is interrupted.

Note: Work is in progress to provide a modern replacement for the legacy **mysqldump** tool. The new tool, called **mysqlpump**, is expected to check most of the boxes on MySQL DBA's performance checklist. For more information, see the *MySQL Reference Manual*.[4]

Flat Files

As opposed to SQL-format dumps that contain data encapsulated in SQL statements, flat-file dumps come with very little overhead. The only control

characters are the delimiters used to separate individual rows and columns. Files in comma-separated value (CSV) or tab-separated value (TSV) format are both examples of the flat-file approach.

Flat files are most commonly produced using:

- The `SELECT … INTO OUTFILE` statement, which dumps table contents (but not table structure) into a file located in the server's local file system.
- **mysqldump** command with the `--tab` parameter, which also dumps table contents to a file and creates the relevant metadata files with `CREATE TABLE` statements. The command uses `SELECT … INTO OUTFILE` internally, so it also creates dump files on the server's local file system.

> **Note:** Due to privilege restrictions, you cannot use the methods mentioned previously with managed database servers such as Amazon RDS. However, you can import flat files dumped from self-managed servers into managed instances with no issues.

Flat files have two major benefits:

- The lack of SQL encapsulation results in much smaller dump files and removes SQL processing overhead during import.
- Flat files are always created in file-per-table fashion, which makes it easy to import them in parallel.

Flat files also have their disadvantages. For example, the server would use a single transaction to import data from each dump file. To have more control over the size of import transactions, you need to manually split very large dump files into chunks, and then import one chunk at a time.

Third-Party Tools and Alternative Solutions

The **mydumper** and **myloader** tools are two popular, open-source MySQL export/import tools designed to address performance issues that are associated

with the legacy **mysqldump** program. They operate on SQL-format dumps and offer advanced features such as:

- Dumping and loading data in multiple threads
- Creating dump files in file-per-table fashion
- Creating chunked dumps, that is, multiple files per table
- Dumping data and metadata into separate files
- Ability to configure transaction size during import
- Ability to schedule dumps in regular intervals

For more information about **mydumper** and **myloader**, see the project home page.[5]

Efficient exports and imports are possible even without the help of third-party tools. With enough effort, you can solve issues associated with SQL-format or flat file dumps manually, as follows:

- Solve single-threaded mode of operations in legacy tools by running multiple instances of the tool in parallel. However, this does not allow you to create consistent database-wide dumps without temporarily suspending database writes.
- Control transaction size by manually splitting large dump files into smaller chunks.

Procedure Optimizations

This section describes ways that you can handle some of the common export/import challenges.

Choosing the Right Number of Threads for Multithreaded Operations

As mentioned earlier, a rule of thumb is to use one thread per server CPU core for exports and one thread per two CPU cores for imports. For example, you should use 16 concurrent threads to dump data from a 16-core db.r3.4xlarge instance and 8 concurrent threads to import data into the same instance type.

Exporting and Importing Multiple Large Tables

If the dataset is spread fairly evenly across multiple tables, export/import operations are relatively easy to parallelize. To achieve optimal performance, follow these guidelines:

- Perform export and import operations using multiple parallel threads. To achieve this, use a modern export tool such as **mydumper**, described in "Third-Party Tools and Alternative Solutions."
- Never use single-row INSERT statements for batch imports. Instead, use multi-row INSERT statements or import data from flat files.
- Avoid using small transactions, but also don't let each transaction become too heavy. As a rule of thumb, split large dumps into 500-MB chunks and import one chunk per transaction.

Exporting and Importing Individual Large Tables

In many databases, data is not distributed equally across tables. It is not uncommon for the majority of the data set to be stored in just a few tables or even a single table. In this case, the common approach of one export/import thread per table can result in suboptimal performance. This is because the total export/import time depends on the slowest thread, which is the thread that is processing the largest table. To mitigate this, you must leverage multithreading at the table level.

The following ideas can help you achieve better performance in similar situations.

Large Table Approach for Exports

On the source server, you can perform a multithreaded dump of table data using a custom export script or a modern third-party export tool, such as **mydumper**, described in "Third-Party Tools and Alternative Solutions."

When using custom scripts, you can leverage multithreading by exporting multiple ranges (segments) of rows in parallel. For best results, you can produce segments by dumping ranges of values in an indexed table column, preferably the primary key. For performance reasons, you should not produce segments using pagination (`LIMIT … OFFSET` clause).

When using **mydumper**, know that the tool uses multiple threads across multiple tables, but it does not parallelize operations against individual tables. To use multiple threads per table, you must explicitly provide the `--rows` parameter when invoking the **mydumper** tool, as follows.

```
--rows : Split table into chunks of this many rows, default unlimited
```

You can choose the parameter value so that the total size of each chunk doesn't exceed 100 MB. For example, if the average row length in the table is 1 KB, you can choose a chunk size of 100,000 rows for the total chunk size of ~100 MB.

Large Table Approach for Imports

Once the dump is completed, you can import it into the target server using custom scripts or the **myloader** tool.

> **Note**: Both **mydumper** and **myloader** default to using four parallel threads, which may not be enough to achieve optimal performance on Aurora db.r3.2xlarge instances or larger. You can change the default level of parallelism using the `--threads` parameter.

Splitting Dump Files into Chunks

You can import data from flat files using a single data chunk (for small tables) or a contiguous sequence of data chunks (for larger tables).

Use the following guidelines to decide how to split table dumps into multiple chunks:

- Avoid generating very small chunks (<1 MB) so that you can avoid protocol and transactional overhead. Alternatively, very large chunks can put unnecessary pressure on server resources without bringing performance benefits. As a rule of thumb, you might use a 500-MB chunk size for large batch imports.
- For partitioned InnoDB tables, use one chunk per partition and don't mix data from different partitions in one chunk. If individual partitions are very large, split partition data further using one of the following solutions.
- For tables or table partitions with an autoincremented PRIMARY key:
    - If PRIMARY key values are provided in the dump, it is good practice not to split data in a random fashion. Instead, use range-based splitting so that each chunk contains monotonically increasing primary key values. For example, if a table has a PRIMARY key column called *id*, data can be sorted by *id* in ascending order and then sliced into chunks. This approach reduces page fragmentation and lock contention during import.
    - If PRIMARY key values are not provided in the dump, the engine generates them automatically for each inserted row. In such cases, you don't need to chunk the data in any particular way and you can choose the method that's easiest for you to implement.
- If the table or table partition has a PRIMARY or NOT NULL UNIQUE key that is not autoincremented, split the data so that each chunk contains monotonically increasing key values for that PRIMARY or NOT NULL UNIQUE KEY, as described previously.

- If the table does not have a PRIMARY or NOT NULL UNIQUE key, the engine creates an implicit, internal clustered index and fills it with monotonically increasing values, regardless of how the input data is split.

For more information about InnoDB index types, see the *MySQL Reference Manual*.[6]

Avoiding Secondary Index Maintenance Overhead

CREATE TABLE statements found in a typical SQL-format dump include the definition of the table primary key and all secondary keys. Consequently, the cost of secondary index management has to be paid for every row inserted during the import. You can observe the index management cost as a gradual decrease in import performance as the table grows.

The negative effects of index management overhead are particularly visible if the table is large or if there are multiple secondary indexes defined on it. In extreme cases, importing data into a table with secondary indexes can be several times slower than importing the same data into a table with no secondary indexes.

Unfortunately, none of the tools mentioned in this document support built-in secondary index optimization. You can, however, implement the optimization using this simple technique:

- Modify the dump files so that CREATE TABLE statements do not include secondary key or foreign key definitions.
- Import data.
- Recreate secondary and foreign keys using ALTER TABLE statements or third-party online schema manipulation tools such as “pt-online-schema-change” from Percona Toolkit. When using ALTER TABLE:
    - Avoid using separate ALTER TABLE statements for each index. Instead, use one ALTER TABLE statement per table to recreate all indexes for that table in a single operation.

    - You may run multiple ALTER TABLE statements in parallel (one per table) to reduce the total time required to process all tables.

ALTER TABLE operations can consume a significant amount of temporary storage space, depending on the table size and the number and type of indexes defined on the table. Aurora instances use local (per-instance) temporary storage volumes. If you observe that ALTER TABLE operations on large tables are failing to complete, it can be due to lack of free space on the instance's temporary volume. If this occurs, you can apply one of the following solutions:

- Scale the Aurora instance to a larger type.
- If altering multiple tables in parallel, reduce the number of ALTER statements running concurrently or try running only one ALTER at a time.
- Consider using a third-party online schema manipulation tool, such as **pt-online-schema-change** from Percona Toolkit.

To learn more about monitoring the local temporary storage on Aurora instances, see the *Amazon Relational Database Service User Guide*.[7]

Reducing the Impact of Long-Running Data Dumps

Data dumps are often performed from active database servers that are part of a mission-critical production environment. If severe performance impact of a massive dump is not acceptable in your environment, consider one of the following ideas:

- If the source server has replicas, you can dump data from one of the replicas.
- If the source server is covered by regular backup procedures:
    - Use backup data as input for the import process if backup format allows for that.

  - If backup format is not suitable for direct importing into the target database, use the backup to provision a temporary database and dump data from it.
- If neither replicas nor backups are available:
  - Perform dumps during off-peak hours, when production traffic is at its lowest.
  - Reduce the concurrency of dump operations so that the server has enough spare capacity to handle production traffic.

Conclusion

This paper discussed important factors affecting the performance of self-managed export/import operations in Amazon Relational Database Service (Amazon RDS) for MySQL and Amazon Aurora:

- The location and sizing of client and server machines
- The ability to consume client and server resources efficiently, which is mostly achieved through multithreading
- The ability to identify and avoid unnecessary overhead at all stages of the migration process

We hope that the ideas and observations we provide will contribute to creating a better overall experience for data migrations in your MySQL-compatible database environments.

Contributors

The following individuals and organizations contributed to this document:

- Szymon Komendera, Database Engineer, Amazon Web Services

Notes

[1] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[2] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSPerformance.html

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

[4] https://dev.mysql.com/doc/refman/5.7/en/mysqlpump.html

[5] https://launchpad.net/mydumper/

[6] https://dev.mysql.com/doc/refman/5.6/en/innodb-index-types.html

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