Sharding with Amazon Relational Database Service

by Lei Zeng | on 20 MAR 2019 | in Amazon RDS, Database | Permalink | Comments | Share

Sharding, also known as *horizontal partitioning*, is a popular scale-out approach for relational databases. Amazon Relational Database Service (Amazon RDS) is a managed relational database service that provides great features to make sharding easy to use in the cloud. In this post, I describe how to use Amazon RDS to implement a sharded database architecture to achieve high scalability, high availability, and fault tolerance for data storage. I discuss considerations for schema design and monitoring metrics when deploying Amazon RDS as a database shard. I also outline the challenges for resharding and highlight the push-button scale-up and scale-out solutions in Amazon RDS.

Sharded database architecture

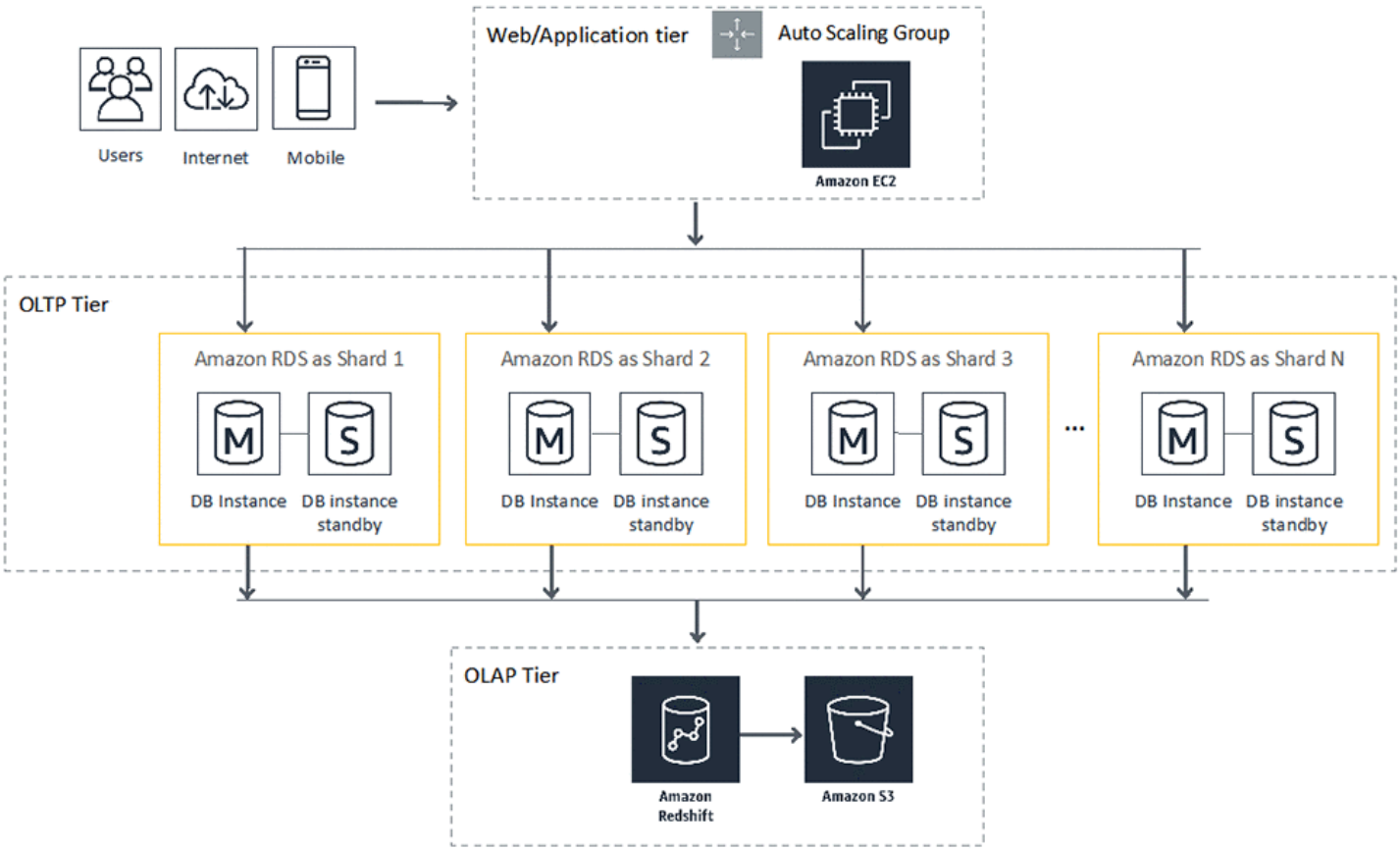
Sharding is a technique that splits data into smaller subsets and distributes them across a number of physically separated database servers. Each server is referred to as a database *shard*. All database shards usually have the same type of hardware, database engine, and data structure to generate a similar level of performance. However, they have no knowledge of each other, which is the key characteristic that differentiates sharding from other scale-out approaches such as database clustering or replication.

The share-nothing model offers the sharded database architecture unique strengths in scalability and fault tolerance. There is no need to manage communications and contentions among database members. The complexities and overhead involved in doing so don't exist. If one database shard has a hardware issue or goes through failover, no other shards are impacted because a single point of failure or slowdown is physically isolated. Sharding has the potential to take advantage of as many database servers as it wants, provided that there is very little latency coming from a piece of data mapping and routing logic residing at the application tier.

However, the share-nothing model also introduces an unavoidable drawback of sharding: The data spreading out on different database shards is separated. The query to read or join data from multiple database shards must be specially engineered. It typically incurs a higher latency than its peer that runs on only one shard. The inability to offer a consistent, global image of all data limits the sharded database architecture in playing an active role in the online analytic processing (OLAP) environment, where data analytic functions are usually performed on the whole dataset.

In an online transaction processing (OLTP) environment, where the high volume of writes or transactions can go beyond the capacity of a single database, and scalability is of concern, sharding is always worth pursuing. With the advent of Amazon RDS, database setup and operations have been automated to a large extent. This makes working with a sharded database architecture a much easier task. Amazon RDS offers a set of database engines, including Amazon RDS for MySQL, MariaDB, PostgreSQL, Oracle, SQL Server, and Amazon Aurora. You can use any one of these as the building block for a database shard in the sharded database architecture.

Let's take a look at an example of a sharded database architecture that is built with Amazon RDS. In the context of the AWS Cloud computing environment, its position in the data flow path has several characteristics (illustrated in the following diagram).

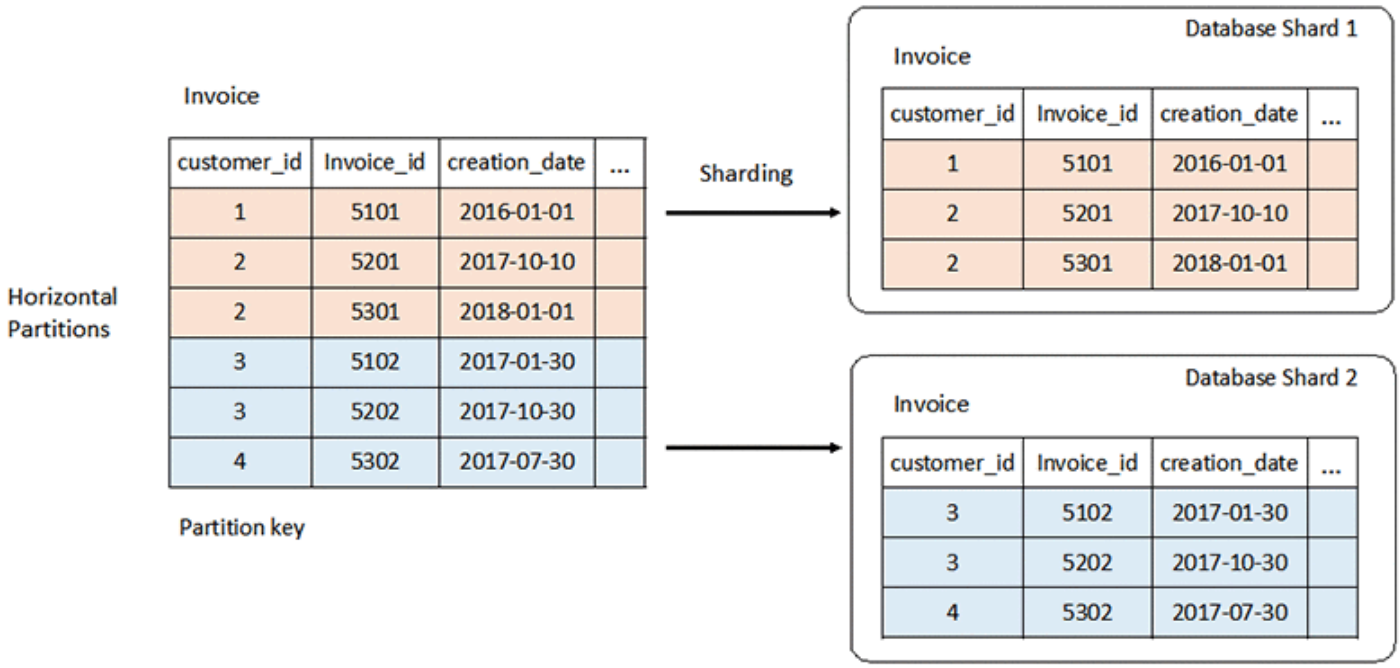


- Data is entered into the system through web applications that are hosted on a group of Amazon EC2 instances with the Auto Scaling feature.
- Data storage is layered where the OLTP environment is separated from the OLAP environment to meet different business and ownership requirements.
- The OLTP environment uses database sharding. It consists of a group of databases built with Amazon RDS for high scalability to meet the growing demand for write throughput. Each database shard is built for high availability using a standalone database deployed with the Multi-AZ feature.**Note:** If you choose an Aurora DB cluster to build a database shard, you can also achieve high availability by configuring a read replica with the primary instance.
- Data is pulled out of the OLTP environment into the OLAP environment based on a schedule. You can use Amazon Redshift to accommodate data from all database shards and construct a time-consistent, global dataset for data analytics functions. Aggregated data result sets are further pushed to Amazon S3 for data sharing, other analytics services, and long-term storage.

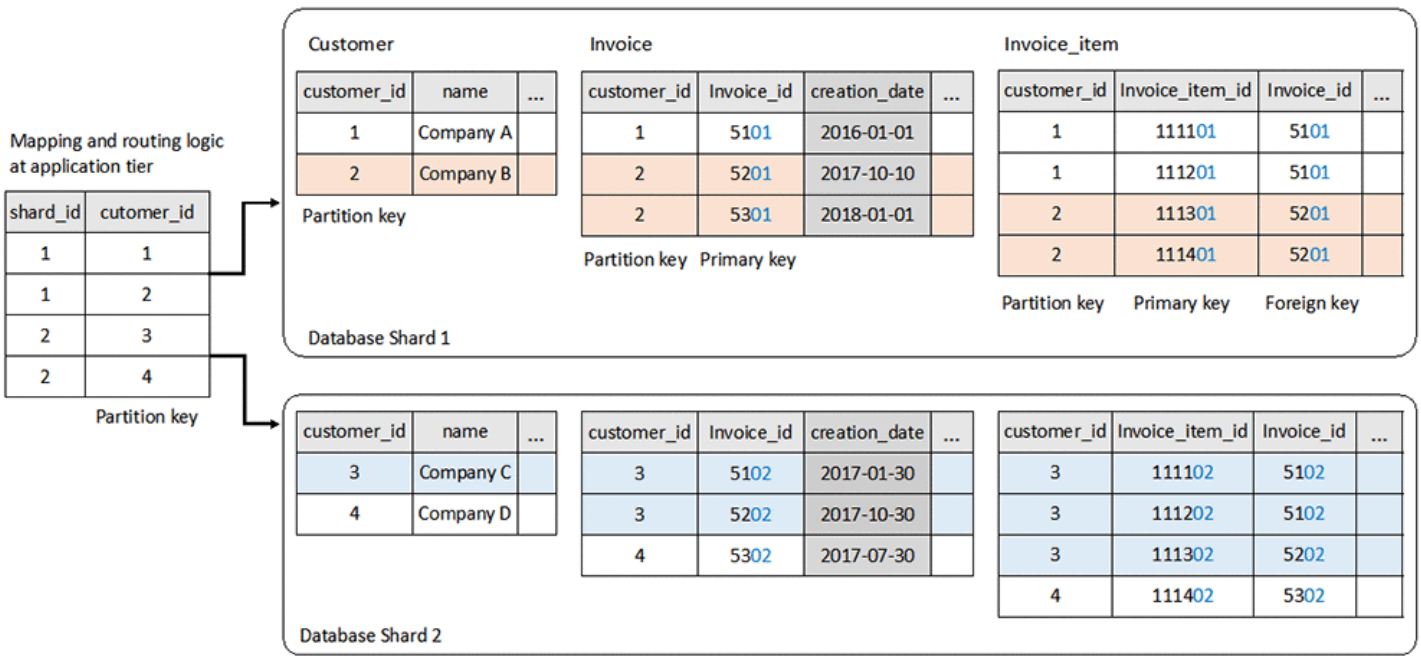
Data partitioning and schema design

The prerequisite to implementing a sharded database architecture is to partition data horizontally and distribute data partitions across database shards. You can use various strategies to partition a table, such as list partitioning, range partitioning, or hash partitioning. You can allow each database shard to accommodate one or more table partitions in the format of separate tables.

The following diagram is an example of horizontal partitioning in the `Invoice` table with `customer_id` as the partition key.



When multiple tables are involved and bound by foreign key relationships, you can achieve horizontal partitioning by using the same partition key on all the tables involved. The data that spans across tables but belongs to one partition key is distributed to one database shard. The following diagram shows an example of horizontal partitioning in a set of tables.



The basic design techniques used are as follows:

- Each database shard contains a *partition key mapping table*, `customer`, to store partition keys that reside on that shard. Applications read this table from all shards to construct the data-mapping logic that maps a partition key to a database shard. Sometimes applications can use a pre-defined algorithm to determine which database shard a partition key is located in, and so this table could be omitted.

- The partition key, `customer_id`, is added to all other tables as a *data isolation point*. It has direct influence on data and workload distribution to different database shards. Queries intended for a single partition include the partition key so that the data routing logic at the applications tier can use it to map to a database shard and route the queries accordingly.
- Primary keys have unique key values across all database shards to avoid key collision when the data is migrated from one shard to another or when data is merged in the OLAP environment. For example, the primary key of the Invoice table, `invoice_id`, uses interleaved numbers and has the last two digits reserved for the `shard_id`. Another common practice is to use a GUID or UUID as primary keys.
- The column with the timestamp data type can be defined in tables as the *data consistency point*. It acts as the criteria to merge data from all database shards into the global view when necessary. For example, when data is pulled into the OLAP environment, the `creation_date` of the Invoice table is used as a filter in the pulling query to specify a consistent time range for all shards.

A well-designed shard database architecture allows the data and the workload to be evenly distributed across all database shards. Queries that land on different shards are able to reach an expected level of performance consistently. The increasing number of data partitions on a shard would raise a database engine's resource consumption, building up contention spots and bottlenecks even on large-scale hardware. To decide how many data partitions per shard to use, you can usually strike a balance between the commitment to optimize query performance and the goal to consolidate, to get better resource use for cost-cutting.

When deploying Amazon RDS as a database shard, you must also consider the type of database engine, DB instance class, and RDS storage. To help you easily manage database configurations, Amazon RDS provides a DB parameter group. It contains the desired set of configuration values that can be applied to all database shards consistently.

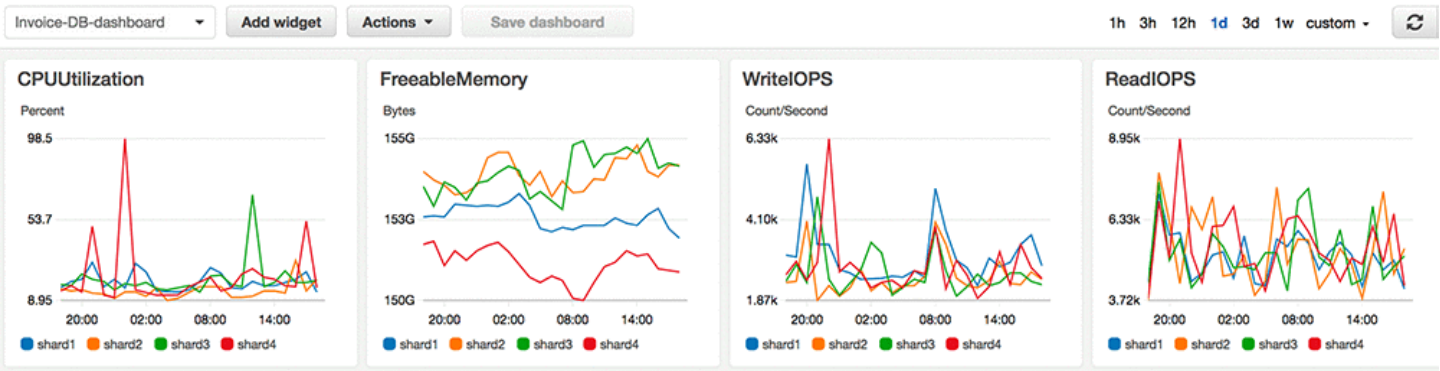
Monitoring for scalability

Monitoring metrics from one shard (such as system resource usage or database throughput) are considered more meaningful in the context of a global picture where you can compare one shard with others to verify whether there is a hot spot in the system. It is also beneficial to set up an appropriate retention period for monitoring data. You can then use historical information to analyze trends and plan capacity to help the system adapt to changes.

As a managed service, Amazon RDS automatically collects monitoring data and publishes it to Amazon CloudWatch. CloudWatch provides a unified view of metrics at the database and system level. Some metrics are generic to all databases, whereas others are specific to a certain database engine. For a complete list of metrics, see the documentation for Amazon RDS and Amazon Aurora.

I always recommend metrics that monitor overall system resource usage, such as `CPUUtilization`, `FreeableMemory`, `ReadIOPS`, `WriteIOPS`, and `FreeStorageSpace`. These metrics are indicators of whether the resource usage on a database shard is within capacity and how much room remains for growth. System resource consumption is an important factor to justify that a sharded database architecture either needs to be further scaled or consolidated otherwise.

The following is an example CloudWatch dashboard that offers great visibility into the sharded database architecture. It brings real-time and historical metrics data from all database shards together into one graph.



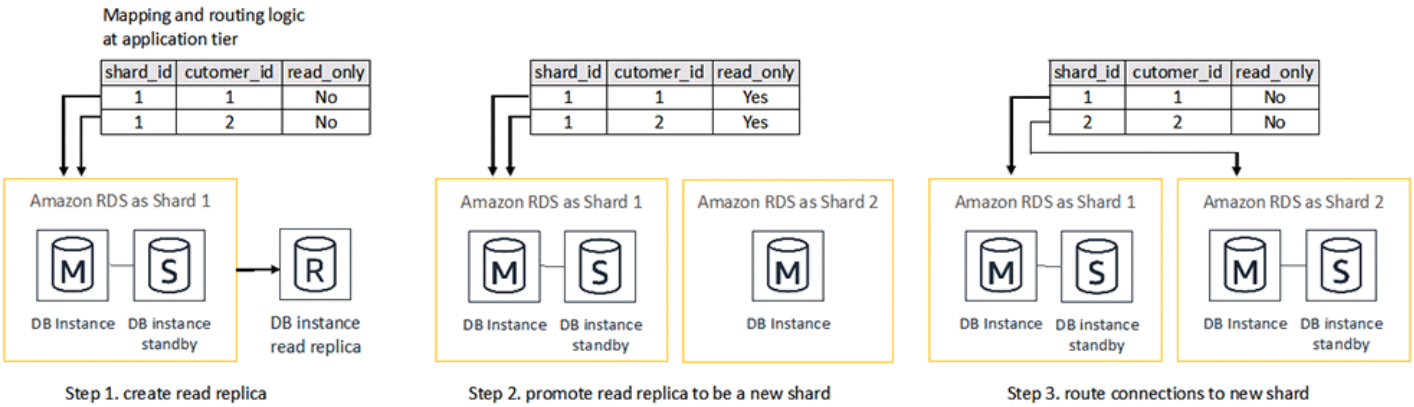
When a database shard has high system resource usage and requires more horsepower, it can be either scale-up or scale-out. Amazon RDS provides a push-button scale-up option. You can use it to modify a DB instance class to larger hardware with more CPU and RAM, or modify DB instance storage for more storage space and increased IOPS capacity. The choices of available instance classes might vary for different database engines or specific database versions. The AWS Management Console is a good place to check that.

If you choose Amazon RDS for MySQL or PostgreSQL to build a database shard, there is another scale up option: migrate to an Amazon Aurora DB cluster. An Aurora DB cluster can consist of more than one database instance on top of clustered storage volumes to deliver higher performance and IOPS capacity than an Amazon RDS DB instance. Amazon RDS provides a push-button option to create an Aurora read replica. This option sets up the replication process to migrate data from an Amazon RDS DB instance for MySQL or PostgreSQL to an Aurora read replica. When the data migration is complete, the Aurora read replica can be promoted to be a standalone Aurora DB cluster.

Resharding

The scale-out option for a database shard is known as *resharding*, meaning *sharding again*. In a broad sense, resharding can also refer to all the procedures that intend to adjust the number of shards in a sharded database architecture. These procedures include adding a new shard, splitting one shard into multiple shards, or merging multiple shards into one. Despite their different format, those procedures essentially need to perform the same type of operation: to migrate existing data from one shard to another shard. In a live production database where data is constantly being accessed and changed, data migration with minimum downtime is always a challenge for resharding.

Amazon RDS has made a great effort to make resharding easier! In Amazon RDS for MySQL, MariaDB, or PostgreSQL, Amazon RDS provides a push-button scale-out option—read replicas—to split one standalone database into multiple new ones. The following diagram shows an example workflow of resharding that uses a read replica as a data replication technique to migrate data between databases.



Step 1. create read replica

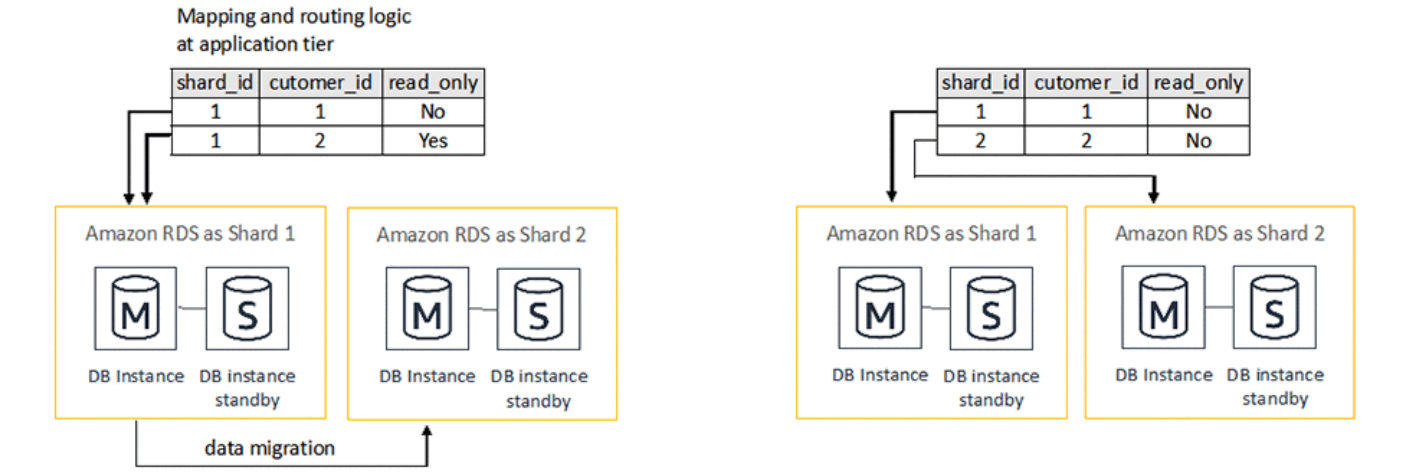
Step 2. promote read replica to be a new shard

Step 3. route connections to new shard

1. The read replica is created to replicate data from the master database continuously.
2. The master database holds off write activities so that the read replica can sync up and be promoted to be a new standalone database. During this time, the mapping and routing logic at the application tier updates the status of multiple data partitions on the master database to be read-only.
3. The data mapping and routing logic is modified to route connections to the new database.

If you choose an Amazon Aurora DB cluster to build a database shard, its read replicas share the storage volumes with the primary instance. Therefore, they cannot be used to replicate data between Aurora clusters. Instead, Amazon RDS provides the clone database feature to create a new Amazon Aurora DB cluster with the same data as the source database. For an Aurora MySQL DB cluster, there is also an option to set up MySQL replication to replicate its data to another new Aurora MySQL DB cluster.

In addition to using existing database technologies to replicate data, resharding can also take advantage of other tools to migrate data between database shards, including AWS Database Migration Service (AWS DMS) or user-defined data extract processes. The following diagram is an example of a tool-based resharding workflow that migrates one data partition at a time.



Step 1. migrate a data partition to new shard and turn on read only mode for sync up

Step 2. turn off read only mode and route connections to new shard

1. The data migration tool is set up to replicate a data partition from one database shard to another. The mapping and routing logic at the application tier updates the status of the data partition to be read-only. The data migration tool can then sync up the data between the two database shards.

2. After the data is in sync, the data mapping and routing logic at the application tier updates the mapping and status of the data partition so that it goes live on the new database shard.

The tool-based resharding method has flexibility because each data partition can be migrated individually. While one data partition is being migrated, it is brought into read-only mode, and applications can still read its data. Other data applications remain accessible for both read and write, and the availability of the overall sharded database architecture can be guaranteed.

The database-based resharding method places the downtime of write activities on multiple data partitions on a database shard when it comes to spin off a new database. After the new database is live, it contains all the data partitions replicated from the original database even though only some of them will be used on it. Therefore you will need to clean up the duplicated data.

Summary

In this post, you read about sharding as an approach for relational databases to achieve high scalability. You learned about an example use case that uses Amazon RDS to implement a sharded database architecture in the cloud, and how it fits into the data storage and analytics workflow. You also learned about various features that RDS provides to manage, monitor, scale up, and scale out database shards in a highly automated manner.

I hope this post gives you a better understanding of sharding and how easy it is to use in the AWS Cloud computing environment. If you have any questions, please feel free to leave a comment below.

About the Author

Lei Zeng is a Senior Database Engineer at AWS.

TAGS: Amazon RDS, sharding

Comments