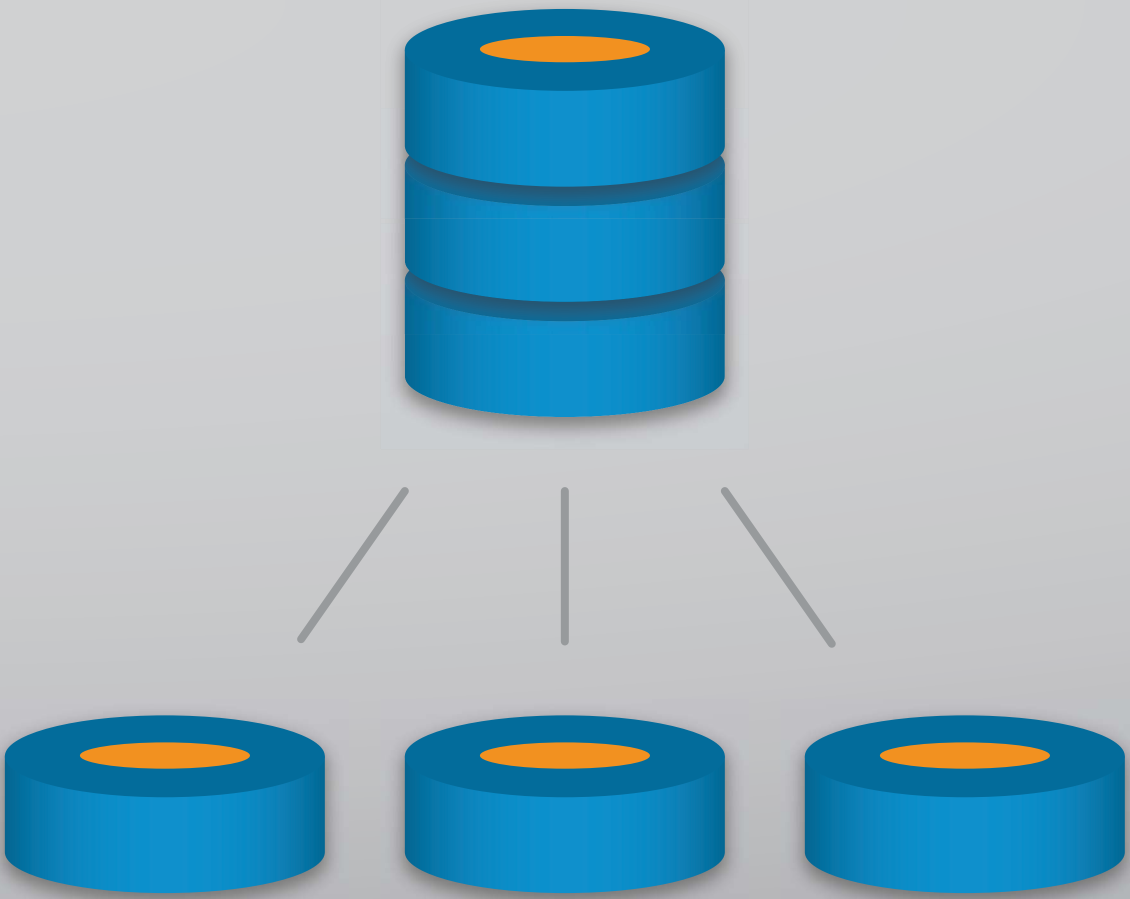
severalnines

Database Sharding with MySQL Fabric



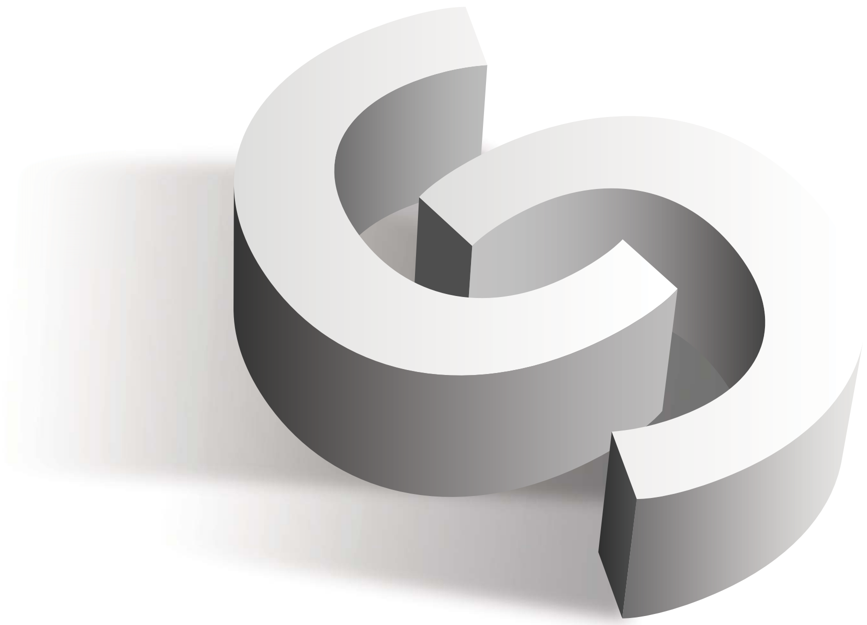


Table of Contents

Why Sharding?

Database systems with large data sets or high throughput applications can challenge the capacity of a single database server. High query rates can exhaust CPU capacity, I/O resources, RAM or even network bandwidth.

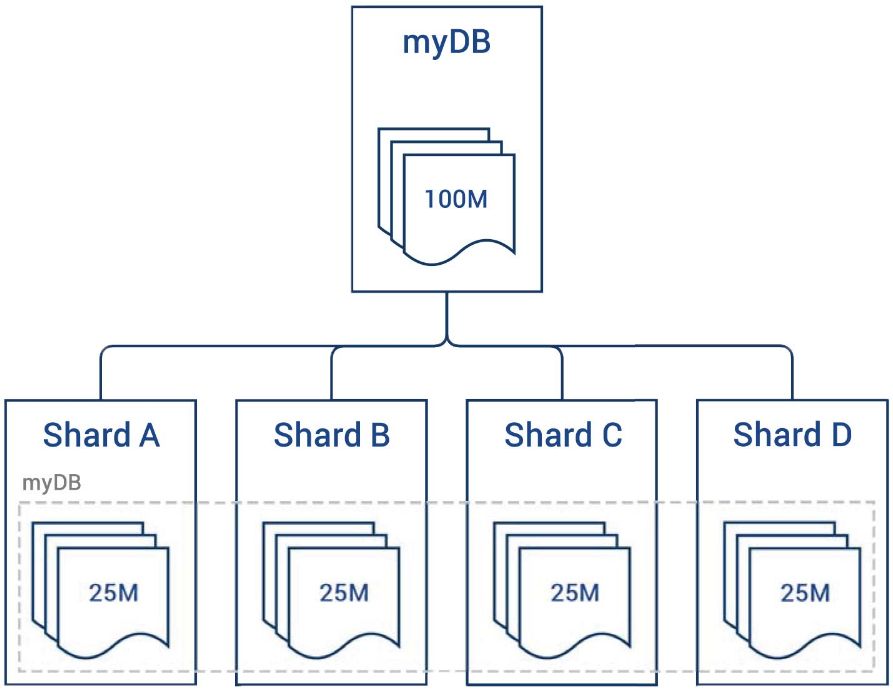
Horizontal scaling is often the only way to scale out your infrastructure. You can upgrade to more powerful hardware, but there is a limit on how much load a single host can handle. You may be able to purchase the most expensive and the fastest CPU or storage on the market, but it still may not be enough to handle your workload. The only feasible way to scale beyond the constraints of a single host is to utilize multiple hosts working together as a part of a cluster or connected using replication.

Horizontal scaling has its limits too, though. When it comes to scaling reads, it is very efficient - just add a node and you can utilize additional processing power. With writes, things are completely different. Consider a MySQL replication setup. Historically, MySQL replication used a single thread to process writes - in a multi-user, highly concurrent environment, this was a serious limitation. This has changed recently. In MySQL 5.6, multiple schemas could be replicated in parallel. In MySQL 5.7, after addition of a 'logical clock' scheduler, it became possible for a single-schema workload to benefit from the parallelization of multi-threaded replication. Galera Cluster for MySQL also allows for multi-threaded replication by utilizing multiple workers to apply writesets. Still, even with those enhancements, you can get just some incremental improvement in the write throughput - it is not the solution to the problem.

One solution would be to split our data across multiple servers using some kind of a pattern and, in that way, to split writes across multiple MySQL hosts. This is sharding.

How does Sharding work?

The idea is really simple - if my database server cannot handle the amount of writes, let's split the data somehow and store one part, generating part of the write traffic, on one database host and the other part on another host. In that way, each host will have to handle half of the writes which should be well within their hardware limits. We can further split the data and distribute it on more servers if our write workload grows.

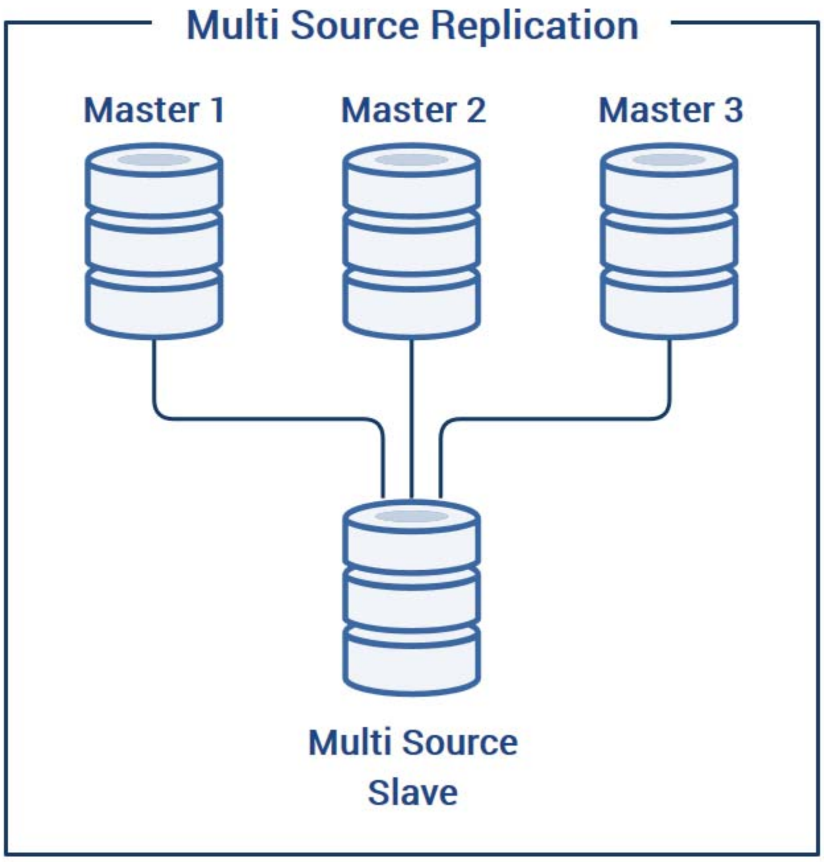


The actual implementation is more complex as there are numerous issues you need to solve before you can implement sharding. How will you split the data? How will you find a correct shard for a query? What you are going to do when one of your shards grows in size and traffic, and outgrows the hardware? How will you scale your environment - preshard it or maybe add new shards when the need arises?

The first, very important question that you need to answer is - how are you going to split your data?

2.1. Functional sharding

Let's imagine your application is built out of multiple modules, or microservices if we want to be fashionable. Assume it's a large online store with a backend of several warehouses. Such site may contain a module to handle warehouse logistics - check the availability of an item, track shipment from a warehouse to a customer. Another module may be an online store - a website with a presentation of available goods. Yet another module would be a transaction module - collect and store credit cards, handle transaction processing and so on. Maybe the online store has a large, buzzing forum where customers share opinions on goods, discuss support issues etc. You may start your voyage in the world of shards by using a separate database per module. This will allow you to gain some breathing space and plan for next steps. On the other hand, the next step may not be necessary at all if each shard can comfortably handle its workload. Of course, there are downsides of such setup - you cannot easily query data across modules (shards) - you have to execute separate queries to separate databases and then combine together resultsets. Unfortunately, this is a typical limitation of a sharded system and there's not much you can do about it. Recently, with MySQL 5.7, multi-source replication has become possible - this may become a method to aggregate data from multiple shards and query it.



The issue you'd have to solve when utilizing this method is, well, the very same issue which forced you to use sharding - limited write capacity. In short, if you aggregate all shards into a single slave, it is highly improbable the slave will keep up with replication. On the other hand, maybe it is enough to aggregate only a couple of tables from each shard and query them on a single slave. Another problem with the functional sharding is that, at the end, some of the modules in your application may still outgrow your hardware - maybe you can further split these modules but if that is not possible, then you need to consider a different sharding strategy.

2.2. Expression-based sharding

Another method of splitting the data across shards would be to use some kind of expression or function/algorithm to help us decide where the data should be located. Let's imagine you have a database with one large table that is commonly accessed and written to. For example, assume a social media site and our largest table contains data about users and their activities. This table uses some kind of id column as primary key - we need to split it somehow and one of the ways would be to apply an expression to the ID value. A very popular choice is to use a modulo function - if we want to generate

128 shards, we can just apply expression of ‘id % 128’ and this would calculate the shard number where a given row should be stored. Another method include making use of a date range, e.g., all user activity in year 2015 is stored in one database, activity in year 2016 is stored in a separate database). Yet another one would be to distribute data based on a list of attributes, e.g., all users from a specific country end up in the same shard.

This approach has both pros and cons. It’s really nice that you can easily locate the shard for any given row - no need to do any complex operations or queries, just evaluate the expression used for sharding using a value of a given ID (i.e. calculate the modulo and see which shard the row belongs to) and you are all set. This works both ways - not only when you retrieve the data but also when you store it. The main limitation is that, once you deploy your shards, it may not be easy to add more of them - in our example we could add more shards by increasing the value in our modulo expression, but it would seriously affect calculations on where data is stored currently. The only feasible way is to completely redistribute data across shards, but this would be a time-consuming process.

2.3. Metadata-based sharding

As we discussed above, both functional sharding and expression-based sharding have limitations when it comes to scaling out in terms of number of shards. There’s still one more method which gives you more flexibility in managing shards - a metadata-based sharding. The idea is very simple - instead of using some kind of hard-coded algorithm, let’s just write down where a given row is located: row of id=1 - shard 1, row with id=2 - shard 5. Finally, let’s build a database to keep this metadata.

This approach has a huge benefit - you can store any row in any shard. You can also easily add new shards to the mix - just set them up and start to store data on them. You can also easily migrate data between shards - nothing stops you from copying data between shards and then making an adjustment in the metadata. In reality it’s more complex than it sounds as you have to make sure you move all the data so some kind of data locking is required. For example, to copy data between shards, you’d have to do an initial copy of the data across shards, lock access to the part of the data which is migrated, make a final sync and, finally, change an entry in the metadata database and unlock the data.

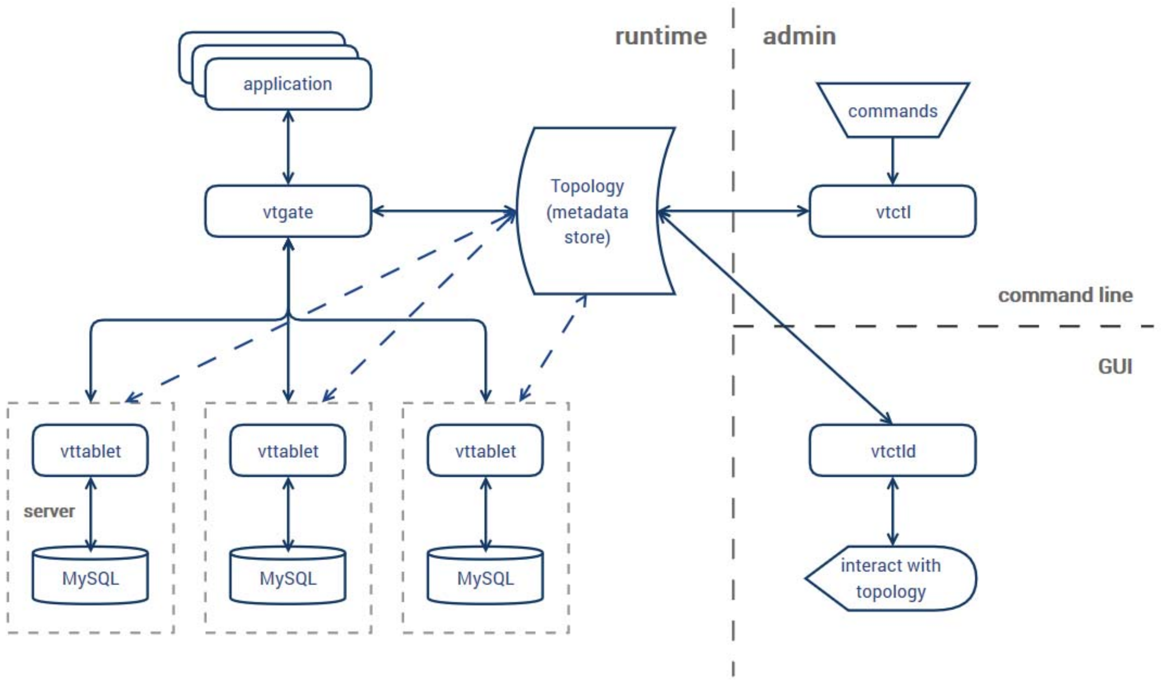
Another issue is the metadata itself - if there’s no algorithm to locate a shard for a given row, you have to query the metadata database and check where you should look for a particular row. If the metadata database become unavailable, your whole application won’t be able to operate. This makes the availability of the metadata crucial - it has to be rock-solid so your application can reach it and check where the data is. High availability is a one challenge. Scaling the metadata database is another one - again, scaling reads can be done by e.g., adding slaves to a replication setup. If the write capacity is limited, you may have to shard the metadata database as well.

Sharding solutions

We've discussed several approaches to sharding. The most flexible one, sharding using metadata, is also the most complex one to implement. You need to design the meta-database, and build high availability not only for your application data but also for the metadata. On top of that, you need to design your application so it will be aware of the complex database infrastructure beneath - it has to query metadata first and then it has to be directed to a correct shard to read or write data. You will also have to build a set of tools to manage and maintain the metadata. Migrating data requires caution so it has to be done carefully. You also have to make sure that any operations on the production databases are mirrored in the metadata - have you taken a slave out of rotation? This should be reflected in the metadata. Have you added a new slave to a shard? You have to modify the metadata and add that host. As you can imagine, lot of time and effort has to be put into developing and maintaining scripts and tools to manage such setup. It begs the question - is there some external solution to design, deploy and manage a sharded environment? In this chapter, we will cover a couple of solutions which are available on the market and which may help you to build a scalable, sharded infrastructure.

3.1. Vitess

Vitess is a tool built to help manage sharded environments. It was developed to help scale out databases at Youtube. In short, it is a solution based on metadata - by default, it uses range sharding but it is also possible to implement a custom sharding schema. Topology data is stored and maintained in a service like Zookeeper or etcd. Application access data using a lightweight proxy, named 'vtgate' in Vitess' nomenclature. Vtgate connects to the metadata store and checks the data distribution - this allows it to route queries to correct shards - 'tablets'.



3.1.1. Tablets

A tablet is a pair of vttablet and mysqld processes - basically, it's a MySQL installation. A tablet can have couple of roles - just like a MySQL host. A tablet can be a master - which means that that particular MySQL acts as a master for its particular shard. A replica is another role - such tablets act more or less as slaves. They serve traffic and they can be promoted to a master role at any time. There are more roles than that, though. Rdonly is a tablet which acts as a slave but it cannot be promoted to a master. Such role fits great with tablets which are intended to handle some intense, heavy duty work. It can be used as a backup server, or dedicated for some CPU-intensive processes like analytical queries. A backup tablet is a tablet which has replication stopped and it's "frozen" data-wise. Most likely a consistent backup is being taken from it and it will resume its replication and return to its original type as soon as the backup operation completes. Another type is restore - it means that the tablet has been started without any data and it is currently in the process of having its data restored. Once it completes, the tablet will resume replication and will become either rdonly or replica. Finally, a worker - this is a tablet which has been reserved by one of Vitess background processes. Once this process completes, the tablet will return to its original state.

3.1.2. How sharding works in Vitess?

To understand how sharding works in Vitess, we need to introduce some new terminology. We've already mentioned tablets, which may have multiple roles. Usually you have a master and several replicas - for high availability. This set of tablets form a shard. A shard, on the other hand, is a partition of a keyspace. Keyspace is a collection of tables, more or less similar to MySQL's schema. You can have one or more keyspaces - just like schemas in MySQL. If you do not use shards, a keyspace will be located on a single set of tablets - a master and several slaves.

Vitess supports range sharding - the keyspace is divided into two or more partitions, each partition covering a range of data. To find ranges, Vitess has to use a column of some kind to calculate them - currently supported data types are BIGINT UNSIGNED and VARBINARY. This works very well with id's which usually use unsigned integer format.

When the first two shards are added in Vitess, relevant ranges are calculated so that data is split more or less in half - we will not go into details of how this is implemented. What's important is that you'll end up with two ranges - let's say id's 0-500 and 501 - maximum value. Each of those ranges can be split in half to create another two shards. You may see a potential limitation here - it is not possible to group different, not adjacent ranges together. Therefore sometimes, you'll end up with more shards that you actually need. This should not be a problem.

Day-to-day operations may require changes in shard structure - most often you will end up having split shards because they would outgrow the hardware performance-wise, but sometimes the workload may reduce and you may want to merge some of the shards. Of course, Vitess supports those operations within the limits of the range-based sharding - you can combine adjacent shards, you can split a shard into two or more parts.

While range-based sharding is the default option in Vitess, it is also possible to implement a custom sharding schema using Vitess. It is more complex though and requires additional logic implemented on the application side. In short, when trying to implement custom sharding in Vitess, you should treat it more like a set of MySQL hosts. What you basically have to do is to create shards, name them however you want and then use one of the keyspaces (which is, at the minimum, a set of tablets - master and couple replicas) as a lookup keyspace in which you'll store data about where a given row is located. So, at the end you will end up with at least two keyspaces. One which contain application data and a lookup keyspace (which also can be sharded, using Vitess' range sharding). The lookup keyspace would be queried by the application to retrieve the shard name where a given row is located - then the application can direct queries to the correct shard.

Unfortunately, using custom sharding makes impossible to benefit from the additional tooling Vitess provides - there's no option for automated resharding nor there is a support for custom sharding in Vitess' API.

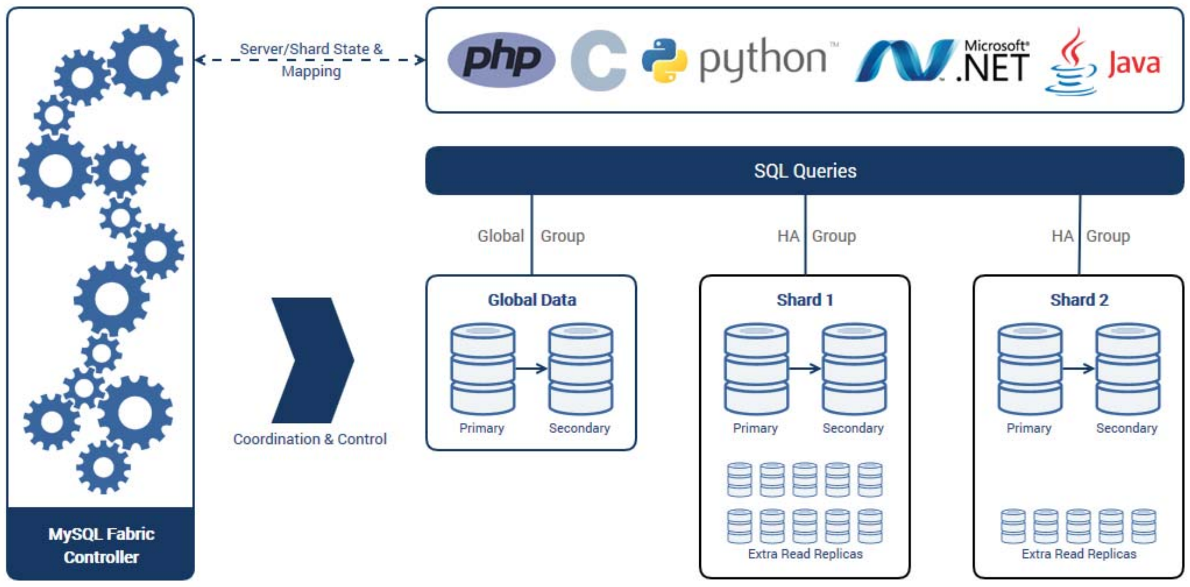
3.1.3. Migration into Vitess cluster

Once you deploy a Vitess cluster, you need to migrate into it from your production MySQL infrastructure. Unfortunately, this is tricky. Obviously, you could dump and then reload the data but such process takes a long time and requires downtime - which makes it not suitable for majority of cases.

The only feasible method of migrating into a Vitess cluster would be to setup replication between your production system and new Vitess cluster. Unfortunately, such operation may not be the easiest, especially if your current environment is a complex one. Additional issue may be the fact that, to connect to Vitess, you may need to modify your application to use different libraries than what you already using to connect to MySQL. There's work in progress to integrate libraries with several database drivers - Go (database/sql) and Python (DB API) should work fine, Java (JDBC) and PHP (PDO) are work in progress, but if your application uses a different language, it may not be that easy to migrate into Vitess.

3.2. MySQL Fabric

In 2014, Oracle introduced a new set of tools for MySQL, called "MySQL Fabric". Historically, there was no official tool which would allow users to build highly available topologies, including sharded setups. The idea behind Fabric is to provide an "official" tooling for building such setups. It provides a framework and tools to manage groups of highly-available MySQL instances. It supports implementation of HA setups and scaling through sharding.



3.2.1. High availability in MySQL Fabric

MySQL Fabric uses a concept of high-availability groups - a group contains two or more MySQL servers connected using replication (actually, you can have just a single host in a group but, obviously, it won't be highly-available). Each server may have several roles - it can be either a "primary" - that is, a master for a given high-availability group; it can be also a "secondary", when it's acting as a slave or "spare". A host can also be "offline" or "faulty", if something is not right with it or its replication setup.

MySQL Fabric can take care of the fault detection within a group, to make sure that your application will be able to query it. If the primary host fails, one of secondary hosts will take over its role and start serving writes.

3.2.2. Scaling out with MySQL Fabric

MySQL Fabric not only gives you the ability to maintain availability of your data - it also supports scaling out through sharding. The basic idea is - if we can configure a few servers into a single high-availability group, we can then scale by having more of them. Then we'd need to implement some kind of shard mapping - we need to decide which column to use for sharding and which tables should be sharded. Another decision has to be made on what algorithm should be used to shard your data - MySQL Fabric supports "hash", which, as long as there are no hash collisions, should result in even distribution of rows across shards, and "range", which works fairly similarly to what we discussed in Vitess - a user can define ranges of rows handled by a single shard - this allows for rather fine-grained control of where a particular set of data is located.

Shards are created out of groups of hosts - a group having one or more hosts in master-slave setup. If you want to deploy four shards, four groups have to be created. One interesting concept of MySQL Fabric is a "global" group. It also consists of one or more MySQL hosts. The idea here is that every master in each shard replicates from the master of "global" group. The global group is a place where all non-sharded tables are going to be updated. This group is also used when performing schema changes.

3.2.3. Query routing in MySQL Fabric

One of the challenges with a sharded setup is how to ensure that the application will be able to connect to the correct shard in order to issue queries. Some tool or module should have up-to-date state information about the database tier - which hosts are online, which hosts handle which shard etc. In MySQL Fabric, such data is stored in Fabric cache, which is then queried by the connector before it routes requests from the application. The best case scenario is when the application doesn't have to know anything about the complexity of the sharded infrastructure. Unfortunately, MySQL Fabric is not there yet - the application has to use MySQL Fabric-related code from the MySQL Connector and pass the sharding key - it has to pass an integer value or a hash, and based on that, MySQL Fabric will decide the shard to which the connection is to be routed to. This is not an ideal solution as your code has to be modified before your application can connect to the MySQL Fabric setup. The change is made in the connection properties. Note that you have to modify your application anyway to connect to the MySQL Fabric connector instead of opening a direct connection to MySQL.

One particular problem is with range queries - they may affect multiple shards and this is not possible to do in automated way. If your application needs to run range queries over the sharding key, your application will have to understand how data is sharded before such a query can be executed. It is not a particular flaw of MySQL Fabric as this particular issue is typical in sharded systems in general, but we wanted to make it clear that MySQL Fabric won't solve it for you.

To avoid some of the problems, MySQL Fabric can work with MySQL Router. This is another tool from Oracle which is intended to provide routing for highly available MySQL environments. It also has a "Fabric-integration" mode. When configured to work with MySQL Fabric, MySQL Router will connect to the Fabric cache, collect data on the state of the Fabric environment and use this data to route queries accordingly. Unfortunately, this works only on the "high availability group" level only - you cannot connect to MySQL Router and let it route your query to a correct shard - you'd have to expose connections to each group over the MySQL Router using different ports and then make your application pick one of them to connect to. This may sound cumbersome, but under some circumstances, it may work really nicely as we hope to showcase later in this ebook.

Migrating into sharded environment with MySQL Fabric

In this chapter, we'll walk you through the process of migrating from a master-slave replication setup to a sharded environment created and maintained by MySQL Fabric. We will be using MySQL Router to make the transition even easier for your application.

4.1. Environment overview

Our initial environment is a master-slave replication setup running MySQL 5.7. Our application will be sysbench. We will use four tables, each containing 1 million rows. Below you can find the exact command that was used to create and populate those tables.

```
sysbench --test=/root/sysbench/sysbench/tests/db/oltp.lua --num-threads=2 --max-requests=0 --max-time=0 --mysql-host=172.30.4.93 --mysql-user=sbtest --mysql-password=sbtest --mysql-port=3306 --oltp-tables-count=4 --report-interval=10 --oltp-skip-trx=off --oltp-table-size=1000000 prepare
```

For generating traffic, we will use the following flags in sysbench:

```
--oltp-skip-trx=on --oltp-simple-ranges=0 --oltp-sum-ranges=0 --oltp-order-ranges=0 --oltp-distinct-ranges=0
```

Our application will use only primary key-based queries, both selects and DML's. We are also not going to use transactions. This is a result of limitations of the sharding system. If you split your table across several shards, you can't really execute range queries. Let's imagine the following case - you have 1000 rows across two shards, i.e., 500 in each shard. Let's assume the following query:

```
SELECT COUNT(*) FROM tab1 WHERE id BETWEEN 400 AND 600;
```

Where should this query be executed? If you run it on the first shard, you'll see the result of 100 (as it contains id's 400 - 500). If you run it on the second shard, you'll again see the result of 100 (as it contains id's 500 - 600). None of those queries return a valid result (200) - your application has to understand how data is sharded before range queries would be feasible to execute. In this case, it should execute two queries on two shards and combine results into one. Similar situation is with transactions - as long as you execute a transaction within the scope of a single shard, it is perfectly fine. You cannot run cross-shard transactions, though - therefore, if you want to run a transaction, you have to be certain that the queries executed within it will be relevant to a single shard only.

Please also note, we are going to focus on the sharding setup - we won't cover all the details that are required in a real life environment to make your application sharding-compatible. For example, one of the issues you'll face when working with shards is that you cannot use an auto_increment primary key. This is due to the fact that if you split such a table across a couple of shards, you'll end up with the same id's generated in multiple shards due to auto_increment behavior. It is possible to manipulate it using auto_increment_increment and auto_increment_offset, but it's tricky and error-prone. The recommended solution is to use some kind of external id generator which will generate a new id for each insert - making sure there are no conflicts. An example of such generator may be 'Snowflake', created by Twitter. In our example, we will remove auto_increment from the primary key (PK) column and demo inserts using manually prepared statements.

The following SQL clears the auto_increment:

```
alter table sbtest.sbtest1 modify column id int unsigned NOT NULL;
alter table sbtest.sbtest2 modify column id int unsigned NOT NULL;
alter table sbtest.sbtest3 modify column id int unsigned NOT NULL;
alter table sbtest.sbtest4 modify column id int unsigned NOT NULL;
```

As proxy, we use ProxySQL configured to perform read/write split of the traffic between our master and slave. It will be also used to route our traffic across multiple shards and it will be useful in making sure our traffic can be moved from the old master to MySQL Router without any impact on our application. Configuration of the ProxySQL has been covered in one of our blog posts: http://severalnines.com/blog/how-proxysql-adds-failover-and-query-control-your-mysql-replication-setup

We'll remember to make sure that our slave has the ***read_only*** variable set to 1.

Once we are done with the ProxySQL setup, we can run our application using the following command:

```
while true ; do sysbench --test=/root/sysbench/sysbench/tests/db/oltp.lua --num-threads=2 --max-requests=0 --max-time=0 --mysql-host=172.30.4.185 --mysql-user=sbtest --mysql-password=sbtest --mysql-port=6033 --oltp-tables-count=4 --report-interval=10 --oltp-skip-trx=on --oltp-simple-ranges=0 --oltp-sum-ranges=0 --oltp-order-ranges=0 --oltp-distinct-ranges=0 --oltp-table-size=1000000 run ; done
```

4.2. Setting up MySQL Fabric

4.2.1. Installation

To install MySQL Fabric you need to install mysql-utilities package. You can download it for your OS version from this link:

https://dev.mysql.com/downloads/utilities/

In our case, system is Ubuntu 14.04 and we had to install one more package for dependencies - mysql-connector-python. It is available from the following site:

https://dev.mysql.com/downloads/connector/python/

Installation on our system required:

```
root@ip-172-30-4-185:~# wget http://cdn.mysql.com//Downloads/MySQLGUITools/mysql-utilities_1.5.6-1ubuntu14.04_all.deb
root@ip-172-30-4-185:~# wget http://cdn.mysql.com//Downloads/Connector-Python/mysql-connector-python_2.1.3-1ubuntu14.04_all.deb
root@ip-172-30-4-185:~# dpkg -i mysql-utilities_1.5.6-1ubuntu14.04_all.deb mysql-connector-python_2.1.3-1ubuntu14.04_all.deb
```

This is all you need to download and install to get started with MySQL Fabric.

4.2.2. Initial setup

MySQL Fabric requires to have an access to some backend MySQL server to store its configuration and cluster setup. It has to be MySQL 5.6.10 or newer but it's recommended to have a backend with the same version as the other hosts managed by MySQL Fabric. We need to create a MySQL user which would be used by MySQL Fabric.

```
mysql> CREATE USER ‘fabric_store’@’%’ IDENTIFIED BY ‘pass’;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> GRANT ALTER, CREATE, CREATE VIEW, DELETE, DROP, EVENT, INDEX, INSERT, REFERENCES, SELECT, UPDATE ON mysql_fabric.* TO ‘fabric_store’@’%’;
Query OK, 0 rows affected (0.00 sec)
```

Couple more will have to be created on the managed MySQL hosts.

```
mysql> CREATE USER ‘fabric_server’@’%’ IDENTIFIED BY ‘pass’;
Query OK, 0 rows affected (0.06 sec)
```

```
mysql> GRANT DELETE, PROCESS, RELOAD, REPLICATION CLIENT,
REPLICATION SLAVE, SELECT, SUPER, TRIGGER ON *.* TO 'fab-
ric_server'@'%';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
mysql> GRANT ALTER, CREATE, DELETE, DROP, INSERT, SELECT,
UPDATE ON mysql_fabric.* TO 'fabric_server'@'%';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CREATE USER 'fabric_backup'@'%' IDENTIFIED BY 'pass';
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> GRANT EVENT, EXECUTE, REFERENCES, SELECT, SHOW VIEW,
TRIGGER ON *.* TO 'fabric_backup'@'%';
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
mysql> CREATE USER 'fabric_restore'@'%' IDENTIFIED BY
'pass';
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> GRANT ALTER, ALTER ROUTINE, CREATE, CREATE ROUTINE,
CREATE TABLESPACE, CREATE VIEW, DROP, EVENT, INSERT, LOCK
TABLES, REFERENCES, SELECT, SUPER, TRIGGER ON *.* TO 'fab-
ric_restore'@'%';
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

Next step will require editing Fabric's configuration file which is located (for Ubuntu 14.04) in /etc/mysql/fabric.cfg. We want to edit [storage] and [servers] section, we also added password and IP address to [protocol.xmlrpc] and [protocol.mysql] sections. Below you can find a complete configuration file with all of our changes included.

```
root@ip-172-30-4-185:~# cat /etc/mysql/fabric.cfg
[DEFAULT]
prefix =
sysconfdir = /etc
logdir = /var/log

[storage]
address = 172.30.4.185:3306
user = fabric_store
password = pass
database = mysql_fabric
auth_plugin = mysql_native_password
connection_timeout = 6
connection_attempts = 6
connection_delay = 1

[servers]
user = fabric_server
```

```
password = pass
backup_user = fabric_backup
backup_password = pass
restore_user = fabric_restore
restore_password = pass
unreachable_timeout = 5

[protocol.xmlrpc]
address = 172.30.4.185:32274
threads = 5
user = admin
password = pass
disable_authentication = no
realm = MySQL Fabric
ssl_ca =
ssl_cert =
ssl_key =

[protocol.mysql]
address = 172.30.4.185:32275
user = admin
password = pass
disable_authentication = no
ssl_ca =
ssl_cert =
ssl_key =

[executor]
executors = 5

[logging]
level = INFO
url = file:///var/log/fabric.log

[sharding]
mysqldump_program = /usr/bin/mysqldump
mysqlclient_program = /usr/bin/mysql
prune_limit = 10000

[statistics]
prune_time = 3600

[failure_tracking]
notifications = 300
notification_clients = 50
notification_interval = 60
failover_interval = 0
detections = 3
detection_interval = 6
detection_timeout = 1
```

```
prune_time = 3600

[connector]
ttl = 1
```

Next step will be to create any required schemas and tables for the MySQL Fabric database:

```
root@ip-172-30-4-185:~# mysqlfabric manage setup
[INFO] 1470138218.386342 - MainThread - Initializing persister: user (fabric_store), server (localhost:3306), database (mysql_fabric).
[INFO] 1470138220.312645 - MainThread - Initial password for admin/mysql set
Password set for admin/mysql from configuration file.
[INFO] 1470138220.320492 - MainThread - Password set for admin/mysql from configuration file.
[INFO] 1470138220.321065 - MainThread - Initial password for admin/xmlrpc set
Password set for admin/xmlrpc from configuration file.
[INFO] 1470138220.327604 - MainThread - Password set for admin/xmlrpc from configuration file.
```

Once this is done, we can start the MySQL Fabric management service as a daemon:

```
root@ip-172-30-4-185:~# mysqlfabric manage start --daemon
```

4.2.3. Setting up global replication group

Before we proceed with setting up shards, we need to set up the global group. For that we prepared two hosts in master-slave setup as we'd like to maintain some level of availability. We will want the master of our global group to slave off our production master. We will use MySQL replication to keep our setup under MySQL Fabric up to date until cutover happens.

Replication requires hosts in the global group to be provisioned in some manner (unless you still have all binary logs) - you can use mysqldump if you have some free time (or if your dataset is small). You can also use xtrabackup to provision them. Any method of provisioning a slave will do. We are going to use xtrabackup to prepare nodes and then we will setup replication. But before we do that, let's add a replication user on our production master - we'll use it later to setup the replication:

```
mysql> CREATE USER rpl_user@'%' IDENTIFIED BY 'replpass';
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> GRANT REPLICATION SLAVE ON *.* TO rpl_user@'%';
Query OK, 0 rows affected (0.01 sec)
```

Once we accomplish this, we can start working on provisioning our global group. First, we will stream xtrabackup from the production slave to the master of the global group in MySQL Fabric:

```
root@ip-172-30-4-141:~# innobackupex --stream=xbstream /
backups/ | ssh root@172.30.4.17 “xbstream -x -C /var/lib/
mysql”
```

Next, we prepare the backup on the master of the global group:

```
root@ip-172-30-4-17:~# innobackupex --apply-log --use-memo-
ry=2G /var/lib/mysql
```

There’s still a slave to provision so we are going to scp the prepared backup to the slave:

```
root@ip-172-30-4-17:~# scp -r /var/lib/mysql/*
root@172.30.4.221:/var/lib/mysql/
```

As a next step - we need to ensure owners are set correctly:

```
root@ip-172-30-4-17:~# chown -R mysql.mysql /var/lib/mysql
root@ip-172-30-4-221:~# chown -R mysql.mysql /var/lib/mysql
```

The problem with GTID slave is that, by default, it may start to replicate from an old GTID. To make sure we won’t break the replication, we need to start from the exact transaction our backup ended at. Luckily, xtrabackup contains the GTID state in xtrabackup_binlog_info file:

```
root@ip-172-30-4-17:~# cat /var/lib/mysql/xtrabackup_bin-
log_info
binlog.000004    818523343   2f5b9100-5803-11e6-b442-12a1ea-
da5517:1-136,
cce85ca1-5802-11e6-a92f-12fa87e491f7:1-689255
```

Now, all we need to do is to clear current replication settings, set a value of gtid_purged correctly - marking all GTID’s covered by the backup as purged, and setup replication again:

```
root@ip-172-30-4-17:~# mysql -ppass
```

```
mysql> RESET SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> RESET MASTER;
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> SET GLOBAL gtid_purged=”2f5b9100-5803-11e6-b442-
12a1eada5517:1-136,
cce85ca1-5802-11e6-a92f-12fa87e491f7:1-689255”;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CHANGE MASTER TO MASTER_HOST='172.30.4.93', MASTER_USER='rpl_user', MASTER_PASSWORD='replpass', MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 2 warnings (0.00 sec)
```

```
mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

The same process has to happen on a slave in our global group:

```
root@ip-172-30-4-221:~# cat /var/lib/mysql/xtrabackup_binlog_info
binlog.000004   818523343  2f5b9100-5803-11e6-b442-12a1eada5517:1-136,
cce85ca1-5802-11e6-a92f-12fa87e491f7:1-689255
```

```
root@ip-172-30-4-221:~# mysql -ppass
```

```
mysql> RESET SLAVE;
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> RESET MASTER;
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> SET GLOBAL gtid_purged="2f5b9100-5803-11e6-b442-12a1eada5517:1-136,
    "> cce85ca1-5802-11e6-a92f-12fa87e491f7:1-689255";
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CHANGE MASTER TO MASTER_HOST='172.30.4.17', MASTER_USER='rpl_user', MASTER_PASSWORD='replpass', MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 2 warnings (0.03 sec)
```

```
mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

At this point we have our global group provisioned and set to replicate with our production system. Now it's time to setup the global group in MySQL Fabric. First, we need to create a group:

```
root@ip-172-30-4-185:~# mysqlfabric group create group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
9f002643-ea0a-4f8c-b7c7-170ad4c60a52        1       1      1
```

Then it's time to add our hosts to the group we've just created:

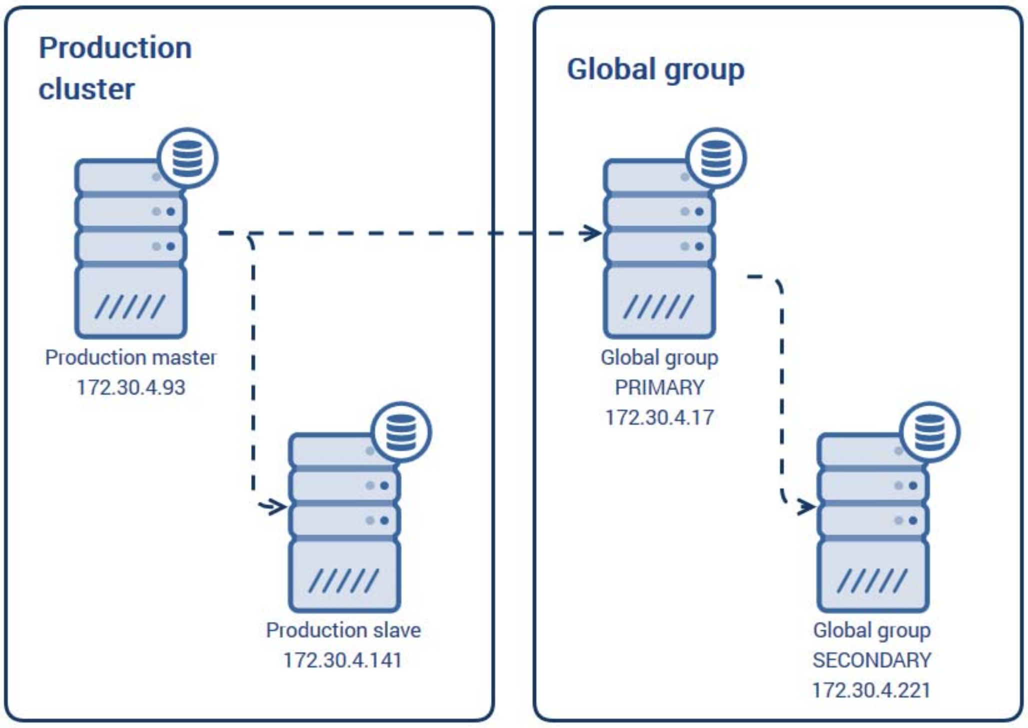
```
root@ip-172-30-4-185:~# mysqlfabric group add group-global
172.30.4.17:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
90a3552f-7f4a-4d77-8da1-fdf05c89fd10        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group add group-global
172.30.4.221:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
cbe2579a-beef-4eb8-9fa2-d7d5f5e8e4e2        1       1      1
```

Both commands finished successfully but let's check how MySQL Fabric sees our group:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers
group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address
status      mode weight
------------------------------------ ----------------- -----
---- --------- ------
2b0cf0dd-58b3-11e6-9360-12ca057f857d  172.30.4.17:3306 SEC-
ONDARY READ_ONLY    1.0
5aa4368f-58b3-11e6-beac-1262072f5c8d 172.30.4.221:3306 SEC-
ONDARY READ_ONLY    1.0
```

It seems like both hosts are in the group, but they are treated as read-only replicas (their status is set to SECONDARY). We need to promote one of them to act as a master. We could allow MySQL Fabric to pick one of them but, as we have a replication chain in place (see the diagram below), we want our 172.30.4.17 host to be the master of the global group. We need to find its UUID - it can be found in the output of the command we executed above: 2b0cf0dd-58b3-11e6-9360-12ca057f857d



Once we know the UUID, we can tell MySQL Fabric to promote this particular host to the master:

```
root@ip-172-30-4-185:~# mysqlfabric group promote
group-global --slave_id=2b0cf0dd-58b3-11e6-9360-12ca057f857d
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
4110e31f-c6f2-45bf-9266-64a0fe2b3b8d        1       1      1
```

Let's check the state of our global group:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers
group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid          address
status       mode weight
------------------------------------ ----------------- -----
---- ---------- ------
2b0cf0dd-58b3-11e6-9360-12ca057f857d  172.30.4.17:3306
PRIMARY READ_WRITE    1.0
5aa4368f-58b3-11e6-beac-1262072f5c8d 172.30.4.221:3306 SEC-
ONDARY  READ_ONLY    1.0
```

Everything looks as we expect - our master has a status of "PRIMARY" and it's in "READ_WRITE" mode.

At this point it may happen that our "PRIMARY" host stops replicating from the production hosts - you can check it by running SHOW SLAVE STATUS;. If that is the case, you will have to execute CHANGE MASTER command once more:

```
mysql> CHANGE MASTER TO MASTER_HOST='172.30.4.93', MASTER_USER='rpl_user', MASTER_PASSWORD='replpass', MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 2 warnings (0.00 sec)
```

```
mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

4.2.4. Define shard mappings

In the previous chapter, we set up replication between our production infrastructure and the global group. We also setup the global group within MySQL Fabric. Next step would be to decide how exactly we'd like to shard our application as we need to tell MySQL Fabric which tables will be sharded and how. As we mentioned earlier, our "application" is a sysbench with four tables created. Let's assume that we will shard three of them using "id" column. The fourth will not be sharded. Such setup may simulate, for example, an application which has several tables connected in some relation using the same column. Maybe it's an e-commerce site which has large number of users. In such case one table might contain user data like home address, shipping address, phone, email address and so on. Another table would contain information about current and previous transactions - who bought what and when? How much did he pay? Third table could contain information about some social elements on the site - maybe the user wrote some reviews of different products we sell? Maybe she took part in a discussion about new season sales? What's important is that those tables are all connected together using the "id" of the user and we may want to join them in a query. This is why we want to keep all of the data of a given user together, in a single shard.

Keeping all of above in mind, we are going to shard the first three tables (sbtest1, sbtest2 and sbtest3) using the "id" column. We will be using a RANGE sharding scheme.

```
root@ip-172-30-4-185:~# mysqlfabric sharding create_definition RANGE group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
2a8a62d4-af23-487b-bf38-c94993ebb88d        1       1      1
```

Once we create the sharding definition, we need to define which tables we are going to shard, and using which column.

```
root@ip-172-30-4-185:~# mysqlfabric sharding add_table 1 sbtest.sbtest1 id
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
16f0ad9d-a3db-4f56-a60e-9a7250cf78d1        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric sharding add_table 1 sbtest.sbtest2 id
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
20ac012a-2967-4a5b-be97-44196cfe8cbe        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric sharding add_table 1 sbtest.sbtest3 id
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
04c5499a-b938-4454-9cee-3a69f21bdd2e        1       1      1
```

4.2.5. Creating shards

Having configured shard mappings, we need to create shards. Obviously, we need to have MySQL hosts installed, but we also need to create them under MySQL Fabric. We'll start with two shards, splitting data in half. We need to repeat the process we went through while creating our global group - first we need to create groups:

```
root@ip-172-30-4-185:~# mysqlfabric group create shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
5fb70a5e-1d5e-4550-9b93-5e759caa93b8        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group create shard-2
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
90acd29e-3c90-4d0b-87f2-5ab6be58c04c        1       1      1
```

Next, we need to build our shard hosts. We assume MySQL is running on them and all grants have been executed, as we discussed earlier. If so, we can benefit from MySQL Fabric's feature to provision them with data. MySQL Fabric uses mysqldump for that so this may not be the most suitable option for large deployments - in that case, you can use your own method to provision the servers with data.

We'll start with the first host. Please note that we use group-global hosts as a source (in this case MySQL Fabric will pick one of them, it is also possible to tell it explicitly which host you'd like to provision from by using --source_id flag and passing the UUID of a source host).

```
root@ip-172-30-4-185:~# mysqlfabric server clone group-global 172.30.4.138
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
43490d9c-5371-44e7-8eb8-abc079976a18        1       1      1
```

Once provisioning completes, we need to add it to the shard-1:

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-1
172.30.4.138:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
7b119581-f279-42f4-9c31-c56dd01e1b74        1       1      1
```

And finally, we want this host to become a master in our shard - we need to promote it:

```
root@ip-172-30-4-185:~# mysqlfabric group promote shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
95b9c34f-3268-4f18-995f-728b6ea963a3        1       1      1
```

Then, our next host follows a similar path - we need to provision it and add it to the shard. Please note, this time we used shard-1 group as a source of our data - mysqldump will be executed on our master:

```
root@ip-172-30-4-185:~# mysqlfabric server clone shard-1
172.30.4.193
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                              uuid finished success result
------------------------------------ -------- ------- ------
e146a6f2-6dde-40a0-ae69-817fd5033b12        1       1      1
```

Once this process completes, we can add the host to the shard.

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-1
172.30.4.193:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                              uuid finished success result
------------------------------------ -------- ------- ------
53ccc8ca-3438-4a8d-ace1-5159eee68f03        1       1      1
```

Similar process has to be performed for the second shard. Steps are exactly the same so we will skip our explanations. The console output will look as per below:

```
root@ip-172-30-4-185:~# mysqlfabric server clone group-glob-
al 172.30.4.76
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                              uuid finished success result
------------------------------------ -------- ------- ------
c68a8f89-ff4b-466b-9106-093e5cd38339        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-2
172.30.4.76:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                              uuid finished success result
------------------------------------ -------- ------- ------
28658ea0-9bd5-48a1-90ef-f749e326fec1        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group promote shard-2
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                              uuid finished success result
------------------------------------ -------- ------- ------
374035cc-479c-425e-beb7-7bb468a9c220        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric server clone shard-2 
172.30.4.30
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
6c684e1f-7fb4-43b1-8513-5d4757fe7a8f        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-2 
172.30.4.30:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
90618548-e6d0-4f73-aea0-95850fd4496c        1       1      1
```

Now, it might be a good idea to activate failure detection in our groups. MySQL Fabric will start to monitor status of MySQL and replication and it will trigger slave promotions if needed. We will activate failure detection in both shards and our global-group.

```
root@ip-172-30-4-185:~# mysqlfabric group activate shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
3c40b110-6668-4f75-8134-a3788d5a166a        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group activate shard-2
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
08c81f60-ab28-4e52-89ac-9c4bbad435c3        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group activate 
group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
5b2d5a08-367a-44d6-a7b0-7459d6dbb718        1       1      1
```

Now it's time to define what data will be stored in which shard:

```
root@ip-172-30-4-185:~# mysqlfabric sharding add_shard 1
"shard-1/1, shard-2/500000" --state=ENABLED
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
0886b7ff-0879-4741-86ba-f7136b4444ca        1       1      1
```

At this point we have two shards defined but each of them contains the full data set. We cannot change it, at this moment, because we rely on replication to keep our MySQL Fabric setup in sync with production. If we'd just remove some data in the sharded tables, replication would fail. We need to switch our traffic to the MySQL Fabric setup first before we can do that. To accomplish this, we need to install and configure MySQL Router which will maintain our nodes' directory and expose them to ProxySQL. We will also have to modify ProxySQL routing to route traffic to relevant shards.

4.3. Setting up MySQL Router

4.3.1. Installation of MySQL Router

Installation of MySQL Router is rather simple. You need to grab the following deb package (or its RPM equivalent) to setup MySQL repository and install it:

```
root@ip-172-30-4-5:~# wget http://dev.mysql.com/get/mysql-
apt-config_0.7.3-1_all.deb
```

```
root@ip-172-30-4-5:~# dpkg -i mysql-apt-config_0.7.3-1_all.
deb
```

While doing so, please make sure you pick MySQL Tools & Connectors to be enabled. Then, all we need to do is to refresh apt cache and install the package.

```
root@ip-172-30-4-5:~# apt-get update
root@ip-172-30-4-5:~# apt-get install mysql-router
```

4.3.2. Configuring MySQL Router

As we mentioned above, our plan is to configure MySQL Router to expose our shards to ProxySQL. We also mentioned earlier that MySQL Router works on per "high availability group" basis. This means, in practice, that each of our shards will require an entry in MySQL Router configuration (we will actually use two, one for read-only and one for read-write traffic). We will then setup ProxySQL to route relevant queries to their destination shard. This also means that, when a change in sharding schema is made, MySQL Router's configuration will have to be added. One may ask - why bother using MySQL Router when we have to handle another proxy too? It actually makes sense in our case. Please keep in mind that, most of the time, data is not resharded often. On the other hand, servers go up and down much more frequently. You may be adding new hosts to the shard, you may be moving a shard to a new set of hosts, you may be promoting a slave to act as a master for a given shard. Those changes will be well hidden behind the MySQL Router and we won't have to change ProxySQL's config frequently to follow the topology changes. Of course, it all depends on your particular setup and you may find it more efficient to skip MySQL Router and focus on automating ProxySQL's configuration changes.

Back to the configuration - MySQL Router's configuration file is located by default (on Ubuntu 14.04) in /etc/mysqlrouter/mysqlrouter.ini

At this point we have three groups - one for our global-group and two for shards. For each of them we will use two entries - one for read-only access and one for read-write access. MySQL Router will route them to hosts in a "SECONDARY" state and "PRIMARY", respectively.

```
[DEFAULT]
logging_folder = /var/log/mysqlrouter/
plugin_folder = /usr/lib/x86_64-linux-gnu/mysqlrouter
runtime_folder = /var/run/mysqlrouter
config_folder = /etc/mysqlrouter

[logger]
level = info

# If no plugin is configured which starts a service, keep-alive
# will make sure MySQL Router will not immediately exit. It is
# safe to remove once Router is configured.
[keepalive]
interval = 60

[fabric_cache:sysbenchapp]
address = 172.30.4.185
user = admin

[routing:globalro]
bind_address = 172.30.4.5:9901
destinations = fabric+cache://sysbenchapp/group/group-global/
mode = read-only

[routing:globalrw]
bind_address = 172.30.4.5:9902
destinations = fabric+cache://sysbenchapp/group/group-global/
mode = read-write

[routing:shard1ro]
bind_address = 172.30.4.5:9903
destinations = fabric+cache://sysbenchapp/group/shard-1/
mode = read-only
```

```

[routing:shard1rw]
bind_address = 172.30.4.5:9904
destinations = fabric+cache://sysbenchapp/group/shard-1/
mode = read-write

[routing:shard2ro]
bind_address = 172.30.4.5:9905
destinations = fabric+cache://sysbenchapp/group/shard-2/
mode = read-only

[routing:shard2rw]
bind_address = 172.30.4.5:9906
destinations = fabric+cache://sysbenchapp/group/shard-2/
mode = read-write
```

At the time of writing, MySQL Router was in version 2.0.3 and it was not possible to set MySQL Fabric password in the configuration file. It makes sense from the security point of view, but, unfortunately, init scripts didn't allow for proper password setup and the connection to MySQL Fabric failed for us. We had to revert to starting mysqlrouter manually, in the screen session:

```
root@ip-172-30-4-5:~# mysqlrouter
Logging to /var/log/mysqlrouter/mysqlrouter.log
Password for [fabric_cache:sysbenchapp], user admin:
```

This finally allowed MySQL Router to connect to MySQL Fabric and reach the Fabric Cache.

4.4. Configuring ProxySQL for sharding

4.4.1. Configuring hostgroups

In this chapter we are going to go through the configuration process of the ProxySQL which will result in traffic being routed to the correct shards. We are going to use different query rules which will check values of the sharding key and based on those, will route our traffic to different hostgroups.

At first, let's create new hostgroups - we are going to add all six ports exposed by MySQL Router. ProxySQL will just forward the traffic there and MySQL Router's task will be to route queries to the backend MySQL hosts.

First, default-group:

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (91, ‘172.30.4.5’, 9901, 1000, 0);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (92, ‘172.30.4.5’, 9902, 1000, 0);
Query OK, 1 row affected (0.00 sec)
```

Then, shard-1:

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (11, ‘172.30.4.5’, 9903, 1000, 0);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (12, ‘172.30.4.5’, 9904, 1000, 0);
Query OK, 1 row affected (0.00 sec)
```

Followed by shard-2:

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (21, ‘172.30.4.5’, 9905, 1000, 0)
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into mysql_servers (hostgroup_id, hostname, port, max_connections, max_replication_lag) values (22, ‘172.30.4.5’, 9906, 1000, 0);
Query OK, 1 row affected (0.00 sec)
```

Finally, we load the config to runtime and save it to persistent storage.

```
mysql> LOAD MYSQL SERVERS TO RUNTIME;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SAVE MYSQL SERVERS TO DISK;
Query OK, 0 rows affected (0.18 sec)
```

4.4.2. Configuring query rules

Next, we need to define query rules which will match incoming queries and route them to the correct shard. To accomplish that, it’d be nice to have insight into what queries are executed on the system. There are numerous ways to do that, including enabling slow query log or capturing a traffic using tcpdump but, as we already use ProxySQL, we can use one of its features and check the list of queries executed through ProxySQL. One of its stats tables - *stats_mysql_query_digest*, contains data about different executed queries. We are interested in their digests:

```
mysql> select digest_text from stats_mysql_query_digest;
+-----------------------------------------------------------+
| digest_text                                               |
+-----------------------------------------------------------+
| INSERT INTO sbtest3 (id, k, c, pad) VALUES (?, ?, ?, ?) |
| DELETE FROM sbtest3 WHERE id=?                            |
| UPDATE sbtest3 SET c=? WHERE id=?                         |
| SELECT c FROM sbtest3 WHERE id=?                          |
| SELECT c FROM sbtest1 WHERE id=?                          |
| INSERT INTO sbtest4 (id, k, c, pad) VALUES (?, ?, ?, ?) |
| DELETE FROM sbtest4 WHERE id=?                            |
| UPDATE sbtest4 SET c=? WHERE id=?                         |
| SELECT c FROM sbtest4 WHERE id=?                          |
| INSERT INTO sbtest1 (id, k, c, pad) VALUES (?, ?, ?, ?) |
| SELECT c FROM sbtest2 WHERE id=?                          |
| INSERT INTO sbtest2 (id, k, c, pad) VALUES (?, ?, ?, ?) |
| DELETE FROM sbtest1 WHERE id=?                            |
| DELETE FROM sbtest2 WHERE id=?                            |
| UPDATE sbtest1 SET c=? WHERE id=?                         |
| UPDATE sbtest4 SET k=k+? WHERE id=?                       |
| UPDATE sbtest2 SET c=? WHERE id=?                         |
| UPDATE sbtest3 SET k=k+? WHERE id=?                       |
| UPDATE sbtest1 SET k=k+? WHERE id=?                       |
| UPDATE sbtest2 SET k=k+? WHERE id=?                       |
+-----------------------------------------------------------+
20 rows in set (0.00 sec)
```

As can be seen, every query uses our sharding key (id) to locate rows - this is the ideal case for us and we can easily route this traffic based on a value of that column. Let's remind our sharding configuration - tables sbtest1-3 are sharded in two shards, values 1-499999 are located in shard1 (defined as hostgroups 11 - ro access and 12 - rw access in ProxySQL), remaining values (500000 and more) are located in shard2 (hostgroup 21 for read only and 22 for read-write access in ProxySQL). We also have table sbtest4, which is not sharded and traffic should be routed to the global group (hostgroups 91 for read-only and 92 for read-write traffic).

Based on this data we are going to insert a couple of mysql_query_rules which will catch queries and route them to the relevant hostgroups. ProxySQL allows to use regex patterns to match queries (supported syntax can be found here: https://github.com/google/re2/wiki/Syntax). Let's start with first shard. A regex of:

```
[1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]{5}
```

should match values 1-499999. We'll use it to route SELECT queries:

```
mysql> INSERT INTO mysql_query_rules (active, match_pattern,
destination_hostgroup, apply) VALUES (0, "^SELECT .* FROM
sbtest[1-3] WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]
{5}$)", 11, 1);
Query OK, 1 row affected (0.00 sec)
```

Couple of comments - we set all our rules to be not active (active=0) because we definitely don’t want them to be executed before a cutover. Match pattern should be very strict - it has to match precisely queries you want to catch. Make no shortcuts here as you may end up routing your traffic incorrectly. Each rule will have apply=1 - this means that it won’t be tested by any further query rule. Of course, destination_hostgroup has to be set correctly, depending on how you configured MySQL Router ports in ProxySQL hostgroups.

After SELECT, we need to handle DML’s:

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^UPDATE
sbtest[1-3].* WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]
{5}$)”, 12, 1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^DELETE FROM
sbtest[1-3].* WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]
{5}$)”, 12, 1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^INSERT INTO
sbtest[1-3] \(id, k, c, pad\) VALUES \(([1-9],|[1-9][0-9]
{1,4},|[1-4][0-9]{5},).*\)$”, 12, 1);
Query OK, 1 row affected (0.00 sec)
```

Once we complete setting shard1, we need to setup rules for shard2 - rows with id greater or equal to 500000. Following regex will match those numbers:

```
mysql> INSERT INTO mysql_query_rules (active, match_pattern,
destination_hostgroup, apply) VALUES (0, “^SELECT .* FROM
sbtest[1-3] WHERE id=([5-9][0-9]{5}|[1-9]{6,}$)”, 21, 1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^UPDATE
sbtest[1-3].* WHERE id=([5-9][0-9]{5}|[1-9]{6,}$)”, 22, 1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^DELETE FROM
sbtest[1-3].* WHERE id=([5-9][0-9]{5}|[1-9]{6,}$)”, 22, 1);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pat-
tern, destination_hostgroup, apply) VALUES (0, “^INSERT INTO
sbtest[1-3] \(id, k, c, pad\) VALUES \(([5-9][0-9]{5}|[1-9]
{6,},).*\)$”, 22, 1);
Query OK, 1 row affected (0.00 sec)
```

We need also to add rules for queries which hit our global table, sbtest4 - currently we rely on the default hostgroup for our application user. Those rules require a bit of explanation. What we'll do here is to match all queries heading to sbtest4 table and send them to hostgroup 92 (read-write access to our global group). We don't finish our matching, though - next two rules pick SELECT and SELECT ... FOR UPDATE queries and route them to the correct hostgroups:

```
mysql> INSERT INTO mysql_query_rules (active, match_pattern, destination_hostgroup, apply) VALUES (0, “^.*sbtest4.*”, 92, 0);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pattern, destination_hostgroup, apply) VALUES (0, “^SELECT.*sbtest4.*”, 91, 0);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO mysql_query_rules (active, match_pattern, destination_hostgroup, apply) VALUES (0, “^SELECT.*sbtest4.*FOR UPDATE”, 92, 1);
Query OK, 1 row affected (0.00 sec)
```

Finally, we save this configuration to disk.

```
mysql> SAVE MYSQL QUERY RULES TO DISK;
Query OK, 0 rows affected (0.10 sec)
```

Please note, we set only exceptions which have to be routed to particular shards. Remaining of queries will be routed to the global group using query rules which, currently, are used to route traffic to our production setup. We will just have to change their hostgroups at the time of cutover.

4.4.3. Testing of query rules

It is important to make sure your query rules are correct and work properly. To ensure this, you have to test them. There are a couple of approaches to test query rules in ProxySQL. In our case, there was not that many queries we had to take care of, and one acceptable solution was to create a bogus, empty table (we called it sbtest5) with a schema of production tables - we then created rules related to this particular table and went through every query and shard range to make sure data has been looked up or modified in the correct shard.

4.5. Cutover process

4.5.1. Preparations

In order to perform a cutover we need to execute two steps.

1. We need to disable traffic routing to current production systems
2. We need to activate query rules which will route traffic to our sharding setup

With ProxySQL, as long as there are no long running transactions, you can stop sending traffic to a hostgroup and reroute it to some different location without any error showing up in the application. We are going to rely on this behavior.

For starters, we need to verify what changes we need to make in the query rules. Let's check how they look like:

```
mysql> select rule_id, active, match_pattern, destination_hostgroup, apply from mysql_query_rules\G
*************************** 1. row ***************************
              rule_id: 1
               active: 1
        match_pattern: ^SELECT.*
destination_hostgroup: 1
                apply: 0
*************************** 2. row ***************************
              rule_id: 2
               active: 1
        match_pattern: SELECT.*FOR UPDATE
destination_hostgroup: 0
                apply: 0
*************************** 3. row ***************************
              rule_id: 145
               active: 0
        match_pattern: ^SELECT .* FROM sbtest[1-3] WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]{5}$)
destination_hostgroup: 11
                apply: 1
*************************** 4. row ***************************
              rule_id: 146
               active: 0
        match_pattern: ^UPDATE sbtest[1-3].* WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]{5}$)
destination_hostgroup: 12
                apply: 1
*************************** 5. row ***************************
              rule_id: 147
               active: 0
        match_pattern: ^DELETE FROM sbtest[1-3].* WHERE id=([1-9]$|[1-9][0-9]{1,4}$|[1-4][0-9]{5}$)
destination_hostgroup: 12
                apply: 1
*************************** 6. row
```

```
****************************
              rule_id: 148
               active: 0
        match_pattern: ^INSERT INTO sbtest[1-3] \(id, k, c, 
pad\) VALUES \(([1-9],|[1-9][0-9]{1,4},|[1-4][0-9]{5},).*\)$
destination_hostgroup: 12
                apply: 1
*************************** 7. row 
****************************
              rule_id: 149
               active: 0
        match_pattern: ^SELECT .* FROM sbtest[1-3] WHERE 
id=([5-9][0-9]{5}|[1-9]{6,}$)
destination_hostgroup: 21
                apply: 1
*************************** 8. row 
****************************
              rule_id: 150
               active: 0
        match_pattern: ^UPDATE sbtest[1-3].* WHERE id=([5-9]
[0-9]{5}|[1-9]{6,}$)
destination_hostgroup: 22
                apply: 1
*************************** 9. row 
****************************
              rule_id: 151
               active: 0
        match_pattern: ^DELETE FROM sbtest[1-3].* WHERE 
id=([5-9][0-9]{5}|[1-9]{6,}$)
destination_hostgroup: 22
                apply: 1
*************************** 10. row 
****************************
              rule_id: 152
               active: 0
        match_pattern: ^INSERT INTO sbtest[1-3] \(id, k, c, 
pad\) VALUES \(([5-9][0-9]{5}|[1-9]{6,},).*\)$
destination_hostgroup: 22
                apply: 1
*************************** 11. row 
****************************
              rule_id: 153
               active: 0
        match_pattern: ^.*sbtest4.*
destination_hostgroup: 92
                apply: 0
*************************** 12. row 
****************************
              rule_id: 154
               active: 0
        match_pattern: ^SELECT.*sbtest4.*
destination_hostgroup: 91
                apply: 0
*************************** 13. row ***************************
              rule_id: 155
               active: 0
        match_pattern: ^SELECT.*sbtest4.*FOR UPDATE
destination_hostgroup: 92
                apply: 1
13 rows in set (0.00 sec)
```

It looks like we have to disable rules 1 and 2 and activate remaining rules (to route traffic to the correct shard or global group).

We also want to verify that our MySQL Fabric high availability groups work correctly:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers group-global
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address    status       mode weight
------------------------------------ ----------------- --------- ---------- ------
2b0cf0dd-58b3-11e6-9360-12ca057f857d  172.30.4.17:3306   PRIMARY READ_WRITE    1.0
5aa4368f-58b3-11e6-beac-1262072f5c8d 172.30.4.221:3306 SECONDARY  READ_ONLY    1.0
```

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address    status       mode weight
------------------------------------ ----------------- --------- ---------- ------
973fe4fd-5803-11e6-b421-12e1054e95cf 172.30.4.138:3306   PRIMARY READ_WRITE    1.0
b879e14a-5980-11e6-87f7-12a36f4a0473 172.30.4.193:3306 SECONDARY  READ_ONLY    1.0
```

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers shard-2
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid         address    status       mode weight
------------------------------------ --------------- --------- ---------- ------
1217822f-5805-11e6-be31-12fe58b52243 172.30.4.76:3306   PRIMARY READ_WRITE    1.0
60214a02-5805-11e6-820a-126c973d658b 172.30.4.30:3306 SECONDARY  READ_ONLY    1.0
```

Everything seems to be working perfectly, we are ready for the cutover.

4.5.2. Cutover

When logged into ProxySQL's admin interface, we need to execute the following query:

```
mysql> UPDATE mysql_query_rules SET active=0 WHERE rule_id IN (1, 2);
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> UPDATE mysql_query_rules SET active=1 WHERE rule_id NOT IN (1, 2);
Query OK, 11 rows affected (0.00 sec)
```

```
mysql> LOAD MYSQL QUERY RULES TO RUNTIME;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SAVE MYSQL QUERY RULES TO DISK;
Query OK, 0 rows affected (0.09 sec)
```

Cutover happened at the moment we loaded the updated query rules to runtime.

4.5.3. Cleanup

Until now both shards contain the full data set. We want to clear not needed data, to save disk space. This process is very simple in MySQL Fabric:

```
root@ip-172-30-4-185:~# mysqlfabric sharding prune_shard sbtest.sbtest1
^@Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
e876d4a2-eff3-4191-8547-403abf3975c3        1       1      1
```

This particular command clears unneeded data in both shards, completing the migration into sharded environment.

4.6. Typical operations in MySQL Fabric sharded environment

In the previous chapter we accomplished migration into a sharded environment using MySQL Fabric. We'd like to show a couple of typical operations you may need to execute in a sharded setup.

4.6.1. Add node to shard

Sometimes, a particular shard may become **hot** and the current set of servers is not large enough to handle the traffic. There are a couple of options to pick from - we could split the shard into two, but, as long as write load is acceptable, we can scale reads by adding another slave into the shard. This process is really simple in MySQL Fabric and it involves three steps.

First, we need to install MySQL on our new host, set correct server_id and make sure all grants required by MySQL Fabric are there. We covered this bit in chapter "4.2.2. Initial setup" where we discussed setting up MySQL Fabric therefore we'll skip it here.

Second, when all MySQL Fabric users have been added to the new host, we need to clone data from a shard:

```
root@ip-172-30-4-185:~# mysqlfabric server clone shard-1 172.30.4.165
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
840d1701-ec8c-4f3d-9cd4-744e8247b73b        1       1      1
```

Next step will be to add it to the shard, but unfortunately, there may be one more thing to do - when cloning, all data has been transferred to our new host, including "mysql" schema. This may lead to problems with replication because MySQL is expecting to use non-existing relay logs. In case you will run into problem like that, you may need to clear it before you can proceed with setting up replication.

```
mysql> RESET SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

Now we can add the host to a shard group:

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-1
172.30.4.165:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
e3de8c91-16a1-49a6-8f7c-f54bc3a45fdc        1       1      1
```

And verify that everything is ok:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers
shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address
status       mode weight
------------------------------------ ----------------- -----
---- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306 SEC-
ONDARY  READ_ONLY    1.0
973fe4fd-5803-11e6-b421-12e1054e95cf 172.30.4.138:3306
PRIMARY READ_WRITE    1.0
b879e14a-5980-11e6-87f7-12a36f4a0473 172.30.4.193:3306 SEC-
ONDARY  READ_ONLY    1.0
```

This is enough - traffic will start to hit our new host.

4.6.2. Remove node from a shard

After adding nodes to a shard we may need to remove some of them. Maybe you've added faster nodes and you want to, step by step, replace old hosts with new hardware? This is easy in MySQL Fabric. You need to find the UUID of a node you want to remove:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers
shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address
status       mode weight
------------------------------------ ----------------- -----
---- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306 SEC-
ONDARY  READ_ONLY    1.0
973fe4fd-5803-11e6-b421-12e1054e95cf 172.30.4.138:3306
PRIMARY READ_WRITE    1.0
b879e14a-5980-11e6-87f7-12a36f4a0473 172.30.4.193:3306 SEC-
ONDARY  READ_ONLY    1.0
```

Next, you just remove it:

```
root@ip-172-30-4-185:~# mysqlfabric group remove shard-1 b879e14a-5980-11e6-87f7-12a36f4a0473
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
fbe3109b-9f55-40af-9ccf-e32686efe5a6        1       1      1
```

Let's verify if we succeeded:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address    status       mode weight
------------------------------------ ----------------- --------- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306 SECONDARY  READ_ONLY    1.0
973fe4fd-5803-11e6-b421-12e1054e95cf 172.30.4.138:3306   PRIMARY READ_WRITE    1.0
```

Seems like we did succeed.

4.6.3. Promote a secondary node in a shard

Let's say we added a new node to the shard and then we want to promote it to a master. It's a matter of a single command:

```
root@ip-172-30-4-185:~# mysqlfabric group promote shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
25b87719-0c44-4a0c-a339-b68e75169750        1       1      1
```

We need to verify the result:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers shard-1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid          address    status       mode weight
------------------------------------ ---------------- --------- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306   PRIMARY READ_WRITE    1.0
973fe4fd-5803-11e6-b421-12e1054e95cf 172.30.4.138:3306 SECONDARY  READ_ONLY    1.0
```

If we had more than one "secondary" node in the shard, we can point MySQL Fabric to promote the exact node we want using "--slave_id=UUID" parameter. Please note that promotion requires a slight downtime so your application may report database errors for the duration of the switchover (usually just a few seconds).

4.6.4. Move shard to a new high availability group

We've discussed adding hosts to the high availability hostgroup but sometimes it may be more efficient to move a whole shard to a new high availability hostgroup. Let's take a look how we could accomplish that in our setup.

First of all, we need to create a new high availability group:

```
root@ip-172-30-4-185:~# mysqlfabric group create shard-1-new
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
2ac0b977-538b-40d8-9154-1d92e6f6a259        1       1      1
```

Then we need to add hosts to it:

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-1-new 172.30.4.165:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
90bdc7b1-f203-4951-bd48-238c2f4fa208        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-1-new 172.30.4.118:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
02338bc5-1731-44a6-960a-1078de4313d1        1       1      1
```

Next, we need to promote one of hosts in this group to master:

```
root@ip-172-30-4-185:~# mysqlfabric group promote shard-1-new
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
77bd998f-5b88-445f-8995-6946b512dc20        1       1      1
```

Let’s verify that everything is ok:

```
root@ip-172-30-4-185:~# mysqlfabric group lookup_servers shard-1-new
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid           address status       mode weight
------------------------------------ ----------------- --------- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306 SECONDARY  READ_ONLY    1.0
eb594f94-5807-11e6-a381-12200a8209c9 172.30.4.118:3306 PRIMARY READ_WRITE    1.0
```

Now, let’s stop for a moment and discuss next steps. What we need to do is to execute “move_shard” command which would move our shard to a new high availability group (in our case: “shard-1-new”. This leads to a serious problem - in MySQL Router configuration, where we define backends, we use the high availability group name:

```
[routing:shard1rw]
bind_address = 172.30.4.5:9904
destinations = fabric+cache://sysbenchapp/group/shard-1/
mode = read-write
```

If we want to move our shard to a new group, a new backend has to be created. This alone is not a problem as we can create it beforehand. You also have to change routing settings in ProxySQL - to route the traffic to a new MySQL Router port. Again, this is not a big problem. The main problem is that the process of moving data is not atomic and you can’t easily time when routing changes should happen. This, virtually, forces you to take a downtime for the whole duration of the “shard_move” command. Most of the time it will be not feasible and you'll rather add new hosts to the existing group, but we still wanted to cover this option to show it to you. There's also another workaround which would include building a set of custom scripts which will perform a process for you. We will show an example of such approach in the next chapter.

Move command looks like below:

```
root@ip-172-30-4-185:~# mysqlfabric sharding move_shard 13 shard-1-new
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
d7343aa1-657f-4019-9820-da5702a57a7c        1       1      1
```

Number "13" is the id of a shard you want to move while "shard-1-new" is the name of a high availability group we want it to move to. The best way to verify what id is assigned to which shard would be to execute the following SQL on the MySQL Fabric database:

```
mysql> select * from mysql_fabric.shards;
+----------+----------+---------+
| shard_id | group_id | state   |
+----------+----------+---------+
|       13 | shard-1  | ENABLED |
|       14 | shard-2  | ENABLED |
+----------+----------+---------+
2 rows in set (0.00 sec)
```

Once the move completes, you either need to create new hostgroups in ProxySQL or you can just edit the MySQL Router configuration and change the old high availability group name to the new one. As you are taking a downtime anyway, just do whatever will be easier for you.

Finally, we can verify that our shard has indeed been moved to the new set of hosts:

```
root@ip-172-30-4-185:~# mysqlfabric sharding lookup_servers sbtest.sbtest1 1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                         server_uuid          address    status       mode weight
------------------------------------ ----------------- --------- ---------- ------
65d19a41-5807-11e6-94ad-1236ea811dcd 172.30.4.165:3306 SECONDARY  READ_ONLY    1.0
eb594f94-5807-11e6-a381-12200a8209c9 172.30.4.118:3306   PRIMARY READ_WRITE    1.0
```

Here, value "1" is a sharding key which we want to lookup - the output means that the row where the sharding key has a value of "1" in sbtest.sbtest1 table is stored on those two hosts.

4.6.5. Splitting the shard

At some point it may happen that you will need to split the shard - maybe the load on the shard became too high or maybe it has grown too large on disk and outgrown the shard's hardware. One way or the other, you need to create a new shard and reroute some of the traffic there.

As mentioned in the previous chapter, we are going to use an approach of developing a custom script to execute this operation without impact to the application.

The split process executed from the CLI is done in couple of steps:

1. Create a new shard
2. Add a host to the new shard
3. Provision that host using data from the shard which we will split
4. Setup replication between the new host and the master of the old shard
5. Stop the replication when all data has been transferred and the new host has caught up
6. Setup replication between the new host and the master of the global group
7. Prune unneeded data from both shards

If we'd stick to this process, we'd face similar problem as with moving the shard - we need to accept downtime for the duration of the data transfer and replication catching up, or we will risk data being written to the old shard. This is because steps 3-6 are executed by a single CLI command and the user does not have any control or hooks to change the routing to the new shard at the exact time the new shard starts to replicate from a global group.

MySQL Fabric, luckily, gives the user the ability to execute CLI commands in a way they won't interfere with data - they will just make a change in the MySQL Fabric internal database. We are talking now about "--update_only" option which can be used with most of the CLI calls. Let's see how we can go around the limitations of our setup.

We are going to assume that the split will happen around the value of 800000 - it means that it'll happen in the second shard.

We have to start with creating a new high availability group for our third shard.

```
root@ip-172-30-4-185:~# mysqlfabric group create shard-3
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                             uuid finished success result
------------------------------------ -------- ------- ------
3e8422de-f6f0-4c29-8873-2e93bf2ab27c        1       1      1
```

Next, we need to prepare a host and then add it to the third shard. We will use data from shard-2 as our shard-3 will eventually use a subset of shard-2's data.

```
root@ip-172-30-4-185:~# mysqlfabric server clone shard-2
172.30.4.118
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
067fb096-3950-4891-9f0b-a8d3d604a7d5        1       1      1
```

```
root@ip-172-30-4-185:~# mysqlfabric group add shard-3
172.30.4.118:3306
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
47cd8261-f9d1-4adc-91bf-aa8f5d35059c        1       1      1
```

Finally, we need to promote our new host to become a PRIMARY in shard-3:

```
root@ip-172-30-4-185:~# mysqlfabric group promote shard-3
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
841c8f93-a696-405c-bc82-592c79ed986e        1       1      1
```

Once we set up a shard in MySQL Fabric, we need to make sure it is available in MySQL Router:

```
[routing:shard3ro]
bind_address = 172.30.4.5:9907
destinations = fabric+cache://sysbenchapp/group/shard-3/
mode = read-only
```

```
[routing:shard3rw]
bind_address = 172.30.4.5:9908
destinations = fabric+cache://sysbenchapp/group/shard-3/
mode = read-write
```

We've concluded our preparations and now it's time to execute our script. Please note this is not a production-ready script - it's more like a proof of concept which does what it should but doesn't implement any safety features which would be a requirement for a production script.

We will start with setting some of variables. The script accepts two parameters: id of a source shard and name of the target high availability group. We also hardcoded some usernames and passwords.

```
#!/bin/bash

org_shard_id=$1
dest_shard_name=$2
fabric_host='172.30.4.185'
fabric_user='fabric_store'
fabric_pass='pass'
repl_user='rpl_user'
repl_pass='replpass'
fabric_server_user='fabric_server'
fabric_server_pass='pass'
global_grp_name='group-global'
```

Next, we collect additional information required for the script. One step may require a bit of explanation - we want to collect the current uuid of the shard-3 host as we will reprovision that host. Should the uuid change, MySQL Fabric will complain about it.

```
org_shard_name=$(mysql -u${fabric_user} -p${fabric_pass} -h
${fabric_host} -e "select group_id from mysql_fabric.shards
WHERE shard_id=${org_shard_id}" | grep -v group_id)

org_ip=$(mysqlfabric group lookup_servers ${org_shard_name}
| grep PRIMARY | awk '{print $2}' | cut -d : -f 1)
echo $org_ip

dest_ip=$(mysqlfabric group lookup_servers ${dest_shard_
name} | grep PRIMARY | awk '{print $2}' | cut -d : -f 1)
echo $dest_ip

global_grp_ip=$(mysqlfabric group lookup_servers ${global_
grp_name} | grep PRIMARY | awk '{print $2}' | cut -d : -f 1)
echo ${global_grp_ip}

dest_shard_uuid=$(mysqlfabric group lookup_servers ${dest_
shard_name} | grep PRIMARY | awk '{print $1}')
echo ${dest_shard_uuid}
```

Now, reprovisioning using xtrabackup:

```
ssh ${dest_ip} "service mysql stop && rm -rf /var/lib/
mysql/*"
ssh ${org_ip} "innobackupex --stream=xbstream /backups/ |
ssh root@${dest_ip} \"xbstream -x -C /var/lib/mysql\""

gtid_after_backup=$(ssh ${dest_ip} "cat /var/lib/mysql/xtra-
backup_binlog_info" | tr -d '\n' | awk '{print $3}')
echo ${gtid_after_backup}
ssh ${dest_ip} "innobackupex --apply-log --use-memory=2G /
var/lib/mysql"
```

```

ssh ${dest_ip} “echo ‘[auto]
server-uuid=${dest_shard_uuid}’ > /var/lib/mysql/auto.cnf”

ssh ${dest_ip} “chown -R mysql.mysql /var/lib/mysql ; ser-
vice mysql start”
mysql -h${dest_ip} -u${fabric_server_user} -p${fabric_serv-
er_pass} -e “RESET SLAVE ALL; RESET MASTER”
mysql -h${dest_ip} -u${fabric_server_user} -p${fabric_serv-
er_pass} -e “SET GLOBAL gtid_purged=’${gtid_after_back-
up}’ ; CHANGE MASTER TO MASTER_HOST=’${org_ip}’, MASTER_US-
ER=’${repl_user}’, \
MASTER_PASSWORD=’${repl_pass}’, MASTER_AUTO_POSITION=1;
START SLAVE;”
```

This is a fairly typical process of provisioning GTID-based slave using xtrabackup. Only exception is the UUID step in which we recreated auto.cnf with original UUID.

Next, we wait a bit so the replication will kick in and we can collect the *seconds behind master* value. The idea here is that we want to make sure we catch up on the replication before we proceed further.

```
sleep 2s
SBM=$(mysql -h${dest_ip} -u${fabric_server_user} -p${fab-
ric_server_pass} -e “SHOW SLAVE STATUS\G” | grep Seconds_Be-
hind_Master | awk ‘{print $2}’)

while [ ${SBM} -gt 1 ]
do
        SBM=$(mysql -h${dest_ip} -u${fabric_server_user}
-p${fabric_server_pass} -e “SHOW SLAVE STATUS\G” | grep Sec-
onds_Behind_Master | awk ‘{print $2}’)
        echo ${SBM}
        sleep 1s
done
```

At this point, we redirect the traffic in ProxySQL - from now on, queries looking for rows with id >= 800000 will be routed to new hostgroup. The catch here is that such hostgroup still does not exist - we rely on ProxySQL to queue requests before our new shard will be ready to process them.

```
mysql -P6032 -uadmin -padmin -h 127.0.0.1 < ./rules.sql
```

Below is the content of rules.sql file:

```
UPDATE mysql_query_rules SET match_pattern=’^SELECT .* FROM
sbtest[1-3] WHERE id=([5-7][0-9]{5}$)’ WHERE match_pattern
LIKE ‘^SELECT%[5-9][0-9]{5}|[1-9]{6,}$%’;
UPDATE mysql_query_rules SET match_pattern=’^UPDATE
sbtest[1-3].* WHERE id=([5-7][0-9]{5}$)’ WHERE match_pattern
```

```
LIKE '^UPDATE%[5-9][0-9]{5}|[1-9]{6,}$%';
UPDATE mysql_query_rules SET match_pattern='^DELETE FROM
sbtest[1-3].* WHERE id=([5-7][0-9]{5}$)' WHERE match_pattern
LIKE '^DELETE%[5-9][0-9]{5}|[1-9]{6,}$%';
UPDATE mysql_query_rules SET match_pattern='^INSERT INTO
sbtest[1-3] \(id, k, c, pad\) VALUES \(([5-7][0-9]{5},).*\)'
WHERE match_pattern LIKE '^INSERT%[5-9][0-9]{5}|[1-9]{6,}%';
INSERT INTO mysql_query_rules (active, match_pattern, des-
tination_hostgroup, apply) VALUES (1, '^SELECT .* FROM
sbtest[1-3] WHERE id=([8-9][0-9]{5}|[1-9]{6,}$)', 41, 1);
INSERT INTO mysql_query_rules (active, match_pattern, des-
tination_hostgroup, apply) VALUES (1, '^UPDATE sbtest[1-
3].*WHERE id=([8-9][0-9]{5}|[1-9]{6,}$)', 42, 1);
INSERT INTO mysql_query_rules (active, match_pattern, desti-
nation_hostgroup, apply) VALUES (1, '^DELETE FROM sbtest[1-
3].*WHERE id=([8-9][0-9]{5}|[1-9]{6,}$)', 42, 1);
INSERT INTO mysql_query_rules (active, match_pattern,
destination_hostgroup, apply) VALUES (1, '^INSERT INTO
sbtest[1-3] \(id, k, c, pad\) VALUES \(([8-9][0-9]{5}|[1-9]
{6,},).*\)$', 42, 1);
LOAD MYSQL QUERY RULES TO RUNTIME;
```

Next, we collect the current value of GTID executed on the master of shard-2. We want to have a GTID _after_ traffic, which is intended to reach shard-3, stopped on shard-2.

```
gtid_target=$(mysql -h${org_ip} -u${fabric_server_user}
-p${fabric_server_pass} -e "show global variables like
'gtid_executed'\G" | tr -d '\n' | awk '{print $7}')
echo ${gtid_target}
```

Using this data, we start the replication on the shard-3 host up to that GTID - this will ensure all queries in shard-3 range, which hit shard-2 before the routing change, will be processed. Once this process completes, we add hosts to shard-3 hostgroups in ProxySQL (which makes the traffic hit those hosts and clean up the queue) and reslave shard-3 master to global group master.

```
mysql -h${dest_ip} -u${fabric_server_user} -p${fabric_serv-
er_pass} -e "STOP SLAVE; START SLAVE UNTIL SQL_AFTER_
GTIDS='${gtid_target}'"

sql_thread_running=$(mysql -h${dest_ip} -u${fabric_serv-
er_user} -p${fabric_server_pass} -e "SHOW SLAVE STATUS\G" |
grep Slave_SQL_Running | awk '{print $2}')

while [ "${sql_thread_running}" != "No" ]
do
        sql_thread_running=$(mysql -h${dest_ip} -u${fab-
ric_server_user} -p${fabric_server_pass} -e "SHOW SLAVE STA-
TUS\G" | grep Slave_SQL_Running | awk '{print $2}')
        SBM=$(mysql -h${dest_ip} -u${fabric_server_user}
-p${fabric_server_pass} -e “SHOW SLAVE STATUS\G” | grep Sec-
onds_Behind_Master | awk ‘{print $2}’)
        echo ${SBM}
        sleep 1s
done
mysql -P6032 -uadmin -padmin -h 127.0.0.1 < ./servers.sql

mysql -h${dest_ip} -u${fabric_server_user} -p${fabric_serv-
er_pass} -e “STOP SLAVE; RESET SLAVE ALL; CHANGE MASTER TO
MASTER_HOST=’${global_grp_ip}’, MASTER_USER=’${repl_user}’,
\
MASTER_PASSWORD=’${repl_pass}’, MASTER_AUTO_POSITION=1;”
```

Contents of the servers.sql file are:

```
INSERT INTO mysql_servers (hostgroup_id, hostname, port)
VALUES (41, ‘172.30.4.5’, 9907);
INSERT INTO mysql_servers (hostgroup_id, hostname, port)
VALUES (42, ‘172.30.4.5’, 9908);
LOAD MYSQL SERVERS TO RUNTIME;
```

This completes the process of preparing new shard and correctly configuring routing. We can then tell MySQL Fabric what we’ve done:

```
root@ip-172-30-4-185:~# mysqlfabric sharding split_shard 14
shard-3 --split_value=800000 --update_only
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
7d6860a1-f83f-4351-b1bd-7050795a15d6        1       1      1
```

Final step will be to prune out-of-range data from all shards.

```
root@ip-172-30-4-185:~# mysqlfabric sharding prune_shard
sbtest.sbtest1
Fabric UUID:  5ca1ab1e-a007-feed-f00d-cab3fe13249e
Time-To-Live: 1

                                uuid finished success result
------------------------------------ -------- ------- ------
b9d6ded7-aef6-43f7-bb86-6b3c56e56276        1       1      1
```

From now on, all shards will only contain their data.

4.7. High availability aspect

We have not covered much about high availability of our sharded environment. At the lowest level, it is covered by high availability groups - as long as you have more than one host per group, you have redundancy with auto-promotion of slaves in case of master failure.

MySQL Fabric itself is a single point of failure, although it is not critical to the operations on our setup. MySQL Fabric may be down but traffic will still be routed correctly, it’s just that we will not be able to perform new operations in MySQL Fabric during that time. The best method here is to setup an active - standby setup, replicating MySQL Fabric database and have it installed on both hosts. Additionally, detailed monitoring of MySQL Fabric process is needed to ensure it’ll be restarted when something goes wrong. A standby host could be promoted to active in case the active server fails.

MySQL Router and ProxySQL can be located on the application hosts - it is a common pattern to colocate the proxy with the application and treat them as one unit. As long as you have more than one application host, you could build a pretty nice redundant environment.

4.8. Summary

As you may have seen, we managed to build an environment and migrate into it without any changes to the application. Unfortunately, there are limitations about what options and operations on shards are available in the system. There are workarounds which include building your own tooling to work around these limitations.

Eventually, it is up to you to decide how to proceed with the migration into MySQL Fabric. For sure, you will be able to use it in a more flexible way by skipping ProxySQL and MySQL Router, and just using the Fabric connector to connect to MySQL. This involves changes in the application as you need to rewrite code to handle connections. This will require cooperation between the DBA/Sysadmin and the application developers.

About Severalnines

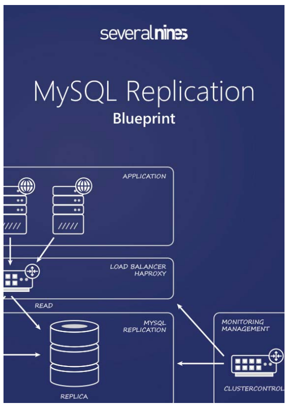
Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. The company has enabled over 8,000 deployments to date via its popular ClusterControl solution. Currently counting BT, Orange, Cisco, CNRS, Technicolour, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with offices in Singapore and Tokyo, Japan. To see who is using Severalnines today visit, http://severalnines.com/customers.

Deploy Manage Monitor Scale

Related Resources from Severalnines

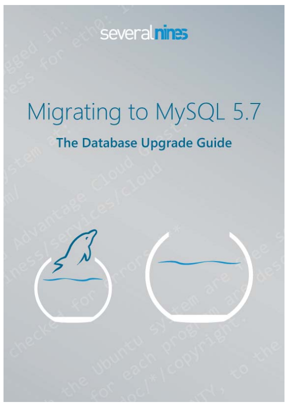
Whitepapers



MySQL Replication Blueprint

The MySQL Replication Blueprint whitepaper includes all aspects of a Replication topology with the ins and outs of deployment, setting up replication, monitoring, upgrades, performing backups and managing high availability using proxies.

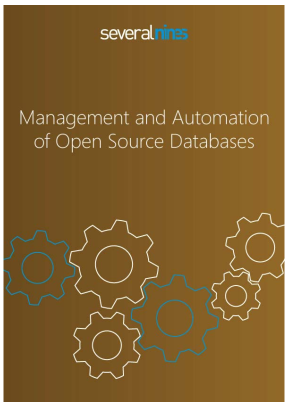
Download here



Migrating to MySQL 5.7 - The Database Upgrade Guide

Upgrading to a new major version involves risk, and it is important to plan the whole process carefully. In this whitepaper, we look at the important new changes in MySQL 5.7 and show you how to plan the test process. We then look at how to do a live system upgrade without downtime. For those who want to avoid connection failures during slave restarts and switchover, this document goes even further and shows you how to leverage ProxySQL to achieve a graceful upgrade process.

Download here



Management and Automation of Open Source Databases

Proprietary databases have been around for decades with a rich third party ecosystem of management tools. But what about open source databases? This whitepaper discusses the various aspects of open source database automation and management as well as the tools available to efficiently run them.

Download here

severalnines

Deploy

Manage

Monitor

Scale

© 2016 Severalnines AB. All rights reserved. Severalnines and the Severalnines logo(s) are trademarks of Severalnines AB. Other names may be trademarks of their respective owners.