MySQL™ Fabric

MySQL Fabric
A Guide to Managing MySQL High Availability and Scaling Out

A MySQL® White Paper

May 2014

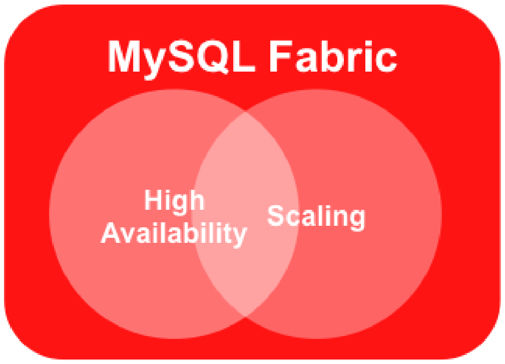
ORACLE®

Table of Contents

1 Executive Summary

MySQL is famous for being a very easy to use database and with the InnoDB storage engine it delivers great performance, functionality and reliability.

MySQL/InnoDB now scales-up extremely well as you add more cores to the server and this continues to improve with each release but at some point a limit is reached where scaling up is no longer enough. It could be you're already using the largest available machine or it's just more economical to use multiple, commodity servers. Scaling out reads is a simple matter using MySQL Replication - have one master MySQL Server to handle all writes and then load balance reads across as many slave MySQL Servers as you need. What happens when that single master fails though? High Availability (HA) also goes beyond coping with failures - with always connected, mobile apps and global services, the concept of a "maintenance window" where system downtime can be scheduled is a thing of the past for most applications.



It's traditionally been the job of the application or the DBA to detect the failure and promote one of the slaves to be the new master. Making this whole system Highly Available can become quite complex and diverts development and operational resources away from higher-value, revenue generating tasks.

While MySQL Replication provides the mechanism to scale out reads, a single server must handle all of the writes and as modern applications become more and more interactive the proportion of writes will continue to increase. The ubiquity of social media means that the age of the publish once and read a billions times web site is over. Add to this the promise offered by Cloud platforms - massive, elastic scaling out of the underlying infrastructure - and you get a huge demand for scaling out to dozens, hundreds or even thousands of servers.

The most common way to scale out is by sharding the data between multiple MySQL Servers; this can be done vertically (each server holding a discrete subset of the tables - say those for a specific set of features) or horizontally where each server holds a subset of the rows for a given table. While effective, sharding has required developers and DBAs to invest a lot of effort in building and maintaining complex logic at the application and management layers - once more, detracting from higher value activities.

The introduction of MySQL Fabric makes all of this far simpler. MySQL Fabric is designed to manage pools of MySQL Servers - whether just a pair for High Availability or many thousand to cope with scaling out huge web application.

For High Availability, MySQL Fabric will manage the replication relationships, detect the failure of the master and automatically promote one of the slaves to be the new master. This is all completely transparent to the application.

For scaling, MySQL Fabric automates sharding with the connectors routing requests to the server (or servers if also using MySQL Fabric for High Availability) based on a sharding key provided by the application. If one shard gets too big then MySQL Fabric can split the shard while ensuring that requests continue to be delivered to the correct location.

MySQL Fabric provides a simple and effective option for High Availability as well as the option of massive, incremental scale-out. It does this without sacrificing the robustness of MySQL and InnoDB; requiring major application changes or needing your Dev Ops teams to move to unfamiliar technologies or abandon their favorite tools.

This paper goes into MySQL Fabric's capabilities in more depth and then goes on to provide a worked example of using it - initially to provide High Availability and then adding sharding.

2 What MySQL Fabric Provides

MySQL Fabric is built around an extensible framework for managing farms of MySQL Servers. Currently two features have been implemented - High Availability and scaling out using data sharding. These features can be used in isolation or in combination.

Both features are implemented in two layers:

- The *mysqlfabric* process which processes any management requests - whether received through the *mysqlfabric* command-line-interface (documented in the reference manual[1]) or from another process via the supplied XML/RPC interface. When using the HA feature, this process can also be made responsible for monitoring the master server and initiating failover to promote a slave to be the new master should it fail. The state of the server farm is held in the state store (a MySQL database) and the *mysqlfabric* process is responsible for providing the stored routing information to the connectors.
- MySQL Connectors are used by the application code to access the database(s), converting instructions from a specific programming language to the MySQL wire protocol, which is used to communicate with the MySQL Server processes. A 'Fabric-aware' connector stores a cache of the routing information that it has fetched from MySQL Fabric and then uses that information to send transactions or queries to the correct MySQL Server. Currently the three supported Fabric-aware MySQL connectors are for PHP, Python and Java (and in turn the Doctrine and Hibernate Object-Relational Mapping frameworks). This approach means that the latency and potential bottleneck of sending all requests via a proxy can be avoided.

2.1 High Availability

High Availability (HA) refers to the ability for a system to provide continuous service - a system is available while that service can be utilized. The level of availability is often expressed in terms of the “number of nines" - for example, a HA level of 99.999% means that the service can be used for 99.999% of the time, in other words, on average, the service is only unavailable for 5.25 minutes per year (and that includes all scheduled as well as unscheduled down-time).

2.1.1 Different Points of High Availability

Figure 1 shows the different layers in the system that need to be available for service to be provided.

At the bottom is the data that the service relies on. Obviously, if that data is lost then the service cannot function correctly and so it's important to make sure that there is at least one extra copy of that data. This data can be duplicated at the storage layer itself but with MySQL it's most commonly replicated by the layer above - the MySQL Server using MySQL Replication. The MySQL Server provides access to the data - there is no point in the data being there if you can't get at it! It is a common misconception that having redundancy at these two levels is enough to have a HA system but it is also necessary to look at the system from the top-down.

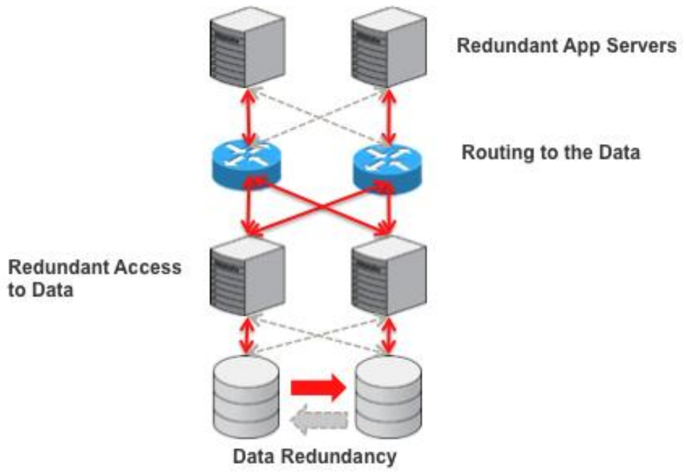


Figure 1 Layered High Availability

To have a HA service, there needs to be redundancy at the application layer; in itself this is very straight-forward, just load balance all of the service requests over a pool of application servers which are all running the same

[1] http://dev.mysql.com/doc/mysql-utilities/1.4/en/fabric.html

application logic. If the service were something as simple as a random number generator then this would be fine but most useful applications need to access data and as soon as you move beyond a single database server (for example because it needs to be HA) then a way is needed to connect the application server to the correct data source. In a HA system, the routing isn't a static function, if one database server should fail (or be taken down for maintenance) the application should be directed instead to an alternate database. Some HA systems implement this routing function by introducing a proxy process between the application and the database servers; others use a virtual IP address which can be migrated to the correct server. When using MySQL Fabric, this routing function is implemented within the Fabric-aware MySQL connector library that's used by the application server processes.

2.1.2 MySQL Replication

MySQL Replication is implemented by configuring one instance as a master, with one or more additional instances configured as slaves. The master will log the changes to the database, which are then sent and applied to the slave(s) immediately or after a set time interval. Figure 2 illustrates this replication flow.

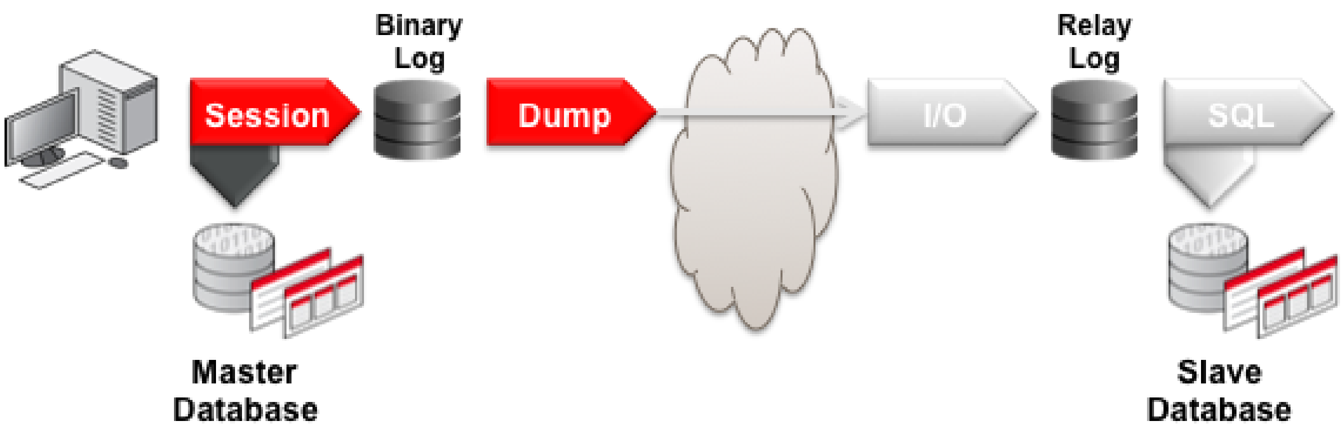


Figure 2 MySQL Replication

Beyond HA, MySQL replication is often employed to scale-out the database across a cluster of physical servers, as illustrated in the figure below. All write operations (and any reads which need to be guaranteed to retrieve the most recent changes) are directed to the master, while other SELECT statements are directed to the slave(s), with query routing implemented either via the appropriate MySQL connector (e.g. the Connector/J JDBC or PHP drivers) or within the application logic.

MySQL replication can be deployed in a range of topologies to support diverse scaling and HA requirements. To learn more about these options and how to configure MySQL replication, refer to this replication whitepaper[1].

MySQL replication is a mature and well-proven approach to scaling workloads while providing a foundation for HA.

2.1.2.1 MySQL Semi synchronous Replication

Semi-Synchronous Replication can be used as an alternative to MySQL's default asynchronous replication, serving to enhance data integrity.

Using semi-synchronous replication, a commit is returned to the client only when a slave has received the update. Therefore it is assured that the data exists on the master and at least one slave (note that the slave will have received the update but not necessarily applied it when a commit is returned to the master).

[1] http://www.mysql.com/why-mysql/white-papers/mysql-replication-introduction/

It is possible to combine the different modes of replication such that some MySQL slaves are configured with asynchronous replication while others use semi synchronous replication. This ultimately means that the Developer / DBA can determine the appropriate level of data consistency and performance on a per-slave basis.

The different replication modes described above can be contrasted with fully synchronous replication whereby data is committed to two or more instances at the same time, using a "two phase commit" protocol. Synchronous replication gives assured consistency across multiple systems and can facilitate faster failover times, but it can add a performance overhead as a result of additional messaging between nodes. Synchronous replication is only possible with MySQL Cluster[1] which is not currently part of MySQL Fabric.

2.1.3 What MySQL Fabric Adds in Terms of High Availability

MySQL Fabric has the concept of a HA group which is a pool of two or more MySQL Servers; at any point in time, one of those servers is the Primary (MySQL Replication master) and the others are Secondaries (MySQL Replication slaves). The role of a HA group is to ensure that access to the data held within that group is always available.

While MySQL Replication allows the data to be made safe by duplicating it, for a HA solution two extra components are needed and MySQL Fabric provides these:

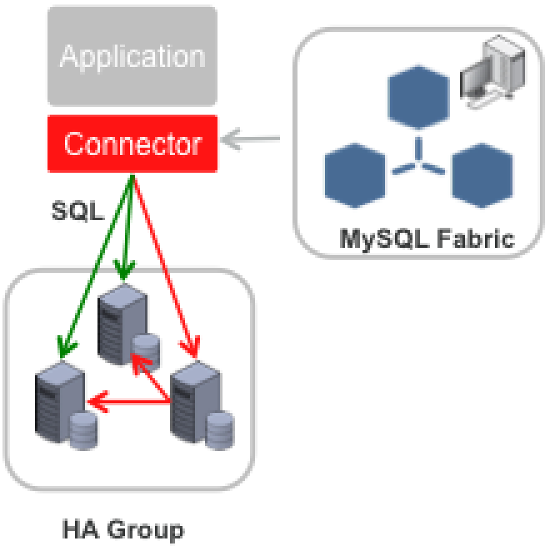


Figure 3 MySQL Fabric Implementing HA

- **Failure detection and promotion** - the MySQL Fabric process monitors the Primary within the HA group and should that server fail then it selects one of the Secondaries and promotes it to be the Primary (with all of the other slaves in the HA group then receiving updates from the new master). Note that the connectors can inform MySQL Fabric when they observe a problem with the Primary and the MySQL Fabric process uses that information as part of its decision making process surrounding the state of the servers in the farm.
- **Routing of database requests** - When MySQL Fabric promotes the new Primary, it updates the state store and that new routing information will be picked up by the connectors and stored in their caches. In this way, the application does not need to be aware that the topology has changed and that writes need to be sent to a different destination.

[1] http://www.mysql.com/products/cluster/

2.3 Scaling Out - Sharding

When nearing the capacity or write performance limit of a single MySQL Server (or HA group), MySQL Fabric can be used to scale-out the database servers by partitioning the data across multiple MySQL Server "groups". Note that a group could contain a single MySQL Server or it could be a HA group.

The administrator defines how data should be partitioned/sharded between these servers; this is done by creating shard mappings. A shard mapping applies to a set of tables and for each table the administrator specifies which column from those tables should be used as a shard key (the shard key will subsequently be used by MySQL Fabric to calculate which shard a specific row from one of those tables should be part of). Because all of these tables use the same shard key and mapping, the use of the same column value in those tables will result in those rows being in the same shard - allowing a single transaction to access all of them. For example, if using the subscriber-id column from multiple tables then all of the data for a specific subscriber will be in the same shard. The administrator then defines how that shard key should be used to calculate the shard number:

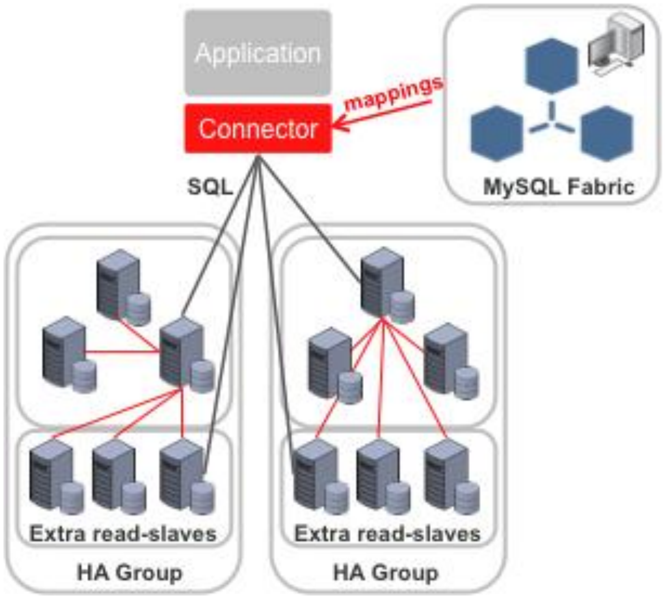


Figure 4 MySQL Fabric Implementing HA & Sharding

- **HASH** - A hash function is run on the shard key to generate the shard number. If values held in the column used as the sharding key don't tend to have too many repeated values then this should result in an even partitioning of rows across the shards.
- **RANGE** - The administrator defines an explicit mapping between ranges of values for the sharding key and shards. This gives maximum control to the user of how data is partitioned and which rows should be co-located.

When the application needs to access the sharded database, it sets a property for the connection that specifies the sharding key - the Fabric-aware connector will then apply the correct range or hash mapping and route the transaction to the correct shard.

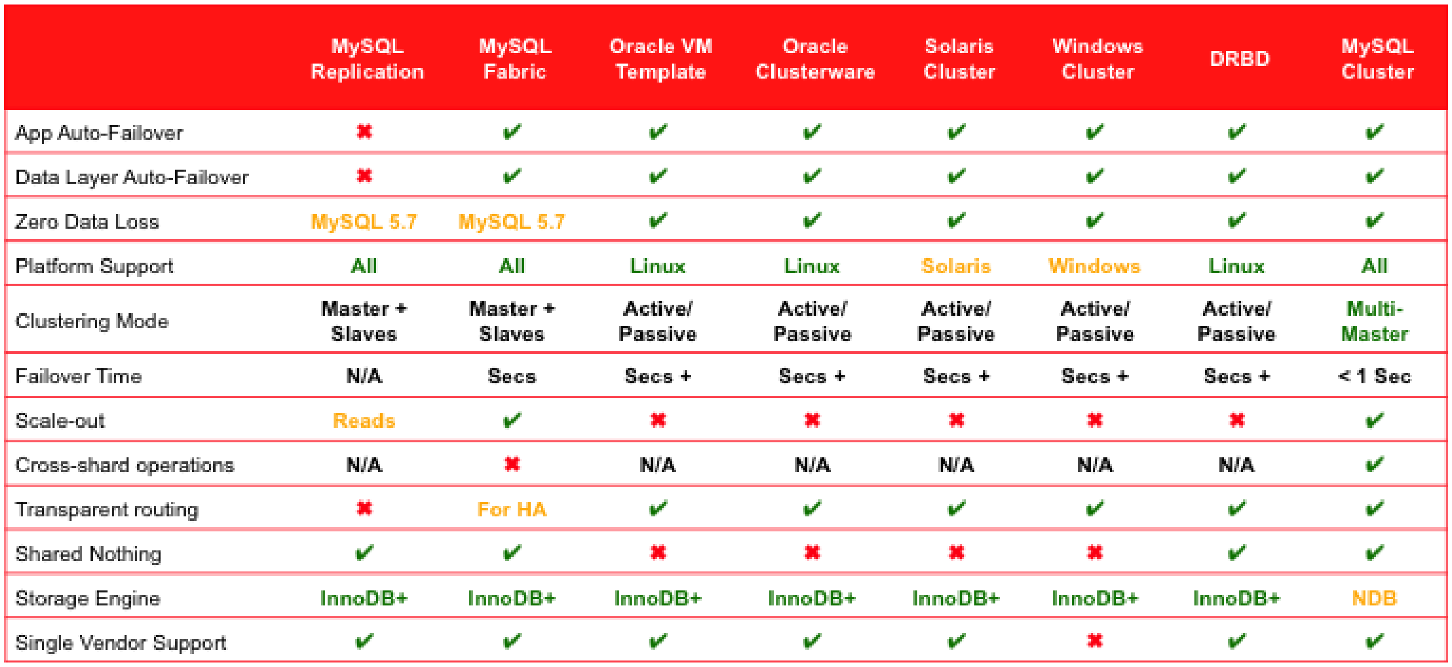
If further shards/groups are needed then MySQL Fabric can split an existing shard into two and then update the state store and the caches of routing data held by the connectors. Similarly, a shard can be moved from one HA group to another.

Note that a single transaction or query can only access a single shard and so it is important to select shard keys based on an understanding of the data and the application's access patterns. It doesn't always make sense to shard all tables as some may be relatively small and having their full contents available in each group can be beneficial given the rule about no cross-shard queries. These global tables are written to a 'global group' and any additions or changes to data in those tables are automatically replicated to all of the other groups. Schema changes are also made to the global group and replicated to all of the others to ensure consistency.

To get the best mapping, it may also be necessary to modify the schema if there isn't already a 'natural choice' for the sharding keys.

3 Where MySQL Fabric Sits with Other MySQL Offerings From Oracle

MySQL Fabric is one of a number of MySQL HA and scaling solutions that are offered by Oracle. These solutions are listed in Table 1 where each one is assessed against a number of criteria.



	MySQL Replication	MySQL Fabric	Oracle VM Template	Oracle Clusterware	Solaris Cluster	Windows Cluster	DRBD	MySQL Cluster
App Auto-Failover	✖	✔	✔	✔	✔	✔	✔	✔
Data Layer Auto-Failover	✖	✔	✔	✔	✔	✔	✔	✔
Zero Data Loss	MySQL 5.7	MySQL 5.7	✔	✔	✔	✔	✔	✔
Platform Support	All	All	Linux	Linux	Solaris	Windows	Linux	All
Clustering Mode	Master + Slaves	Master + Slaves	Active/ Passive	Active/ Passive	Active/ Passive	Active/ Passive	Active/ Passive	Multi-Master
Failover Time	N/A	Secs	Secs +	Secs +	Secs +	Secs +	Secs +	< 1 Sec
Scale-out	Reads	✔	✖	✖	✖	✖	✖	✔
Cross-shard operations	N/A	✖	N/A	N/A	N/A	N/A	N/A	✔
Transparent routing	✖	For HA	✔	✔	✔	✔	✔	✔
Shared Nothing	✔	✔	✖	✖	✖	✖	✔	✔
Storage Engine	InnoDB+	InnoDB+	InnoDB+	InnoDB+	InnoDB+	InnoDB+	InnoDB+	NDB
Single Vendor Support	✔	✔	✔	✔	✔	✖	✔	✔

Table 1 Comparison of MySQL HA & Scale-Out Technologies

More details of these other solutions can be found in the MySQL Guide to High Availability Solutions[1].

MySQL Fabric has a number of features that make it very suitable for adding either scale-out, HA or both to a MySQL based application:

- MySQL Fabric allows the application to use the excellent and well-proven InnoDB storage engine
- HA is built upon MySQL Replication which is again a technology that's extremely mature and used all over the world; what MySQL Fabric adds is a management layer together with automated failover which is transparent to the application
- Scale-out of writes in addition to reads
- Shared-nothing architecture - ideal for cost-effective, commodity hardware while also being able to exploit servers with many cores
- All elements of the solution are supported by Oracle

[1] http://www.mysql.com/why-mysql/white-papers/mysql-guide-to-high-availability-solutions

4 Current Limitations

The initial version of MySQL Fabric is designed to be simple, robust and able to scale to thousands of MySQL Servers. This approach means that this version has a number of limitations, which are described here:

- Sharding is not completely transparent to the application. While the application need not be aware of which server stores a set of rows and it doesn't need to be concerned when that data is moved, it does need to provide the sharding key when accessing the database.
- Auto-increment columns cannot be used as a sharding key
- All transactions and queries need to be limited in scope to the rows held in a single shard, together with the global (non-sharded) tables. For example, Joins involving multiple shards are not supported
- Because the connectors perform the routing function, the extra latency involved in proxy-based solutions is avoided but it does mean that Fabric-aware connectors are required - at the time of writing these exist for PHP, Python and Java
- The MySQL Fabric process itself is not fault-tolerant and must be restarted in the event of it failing. Note that this does not represent a single-point-of-failure for the server farm (HA and/or sharding) as the connectors are able to continue routing operations using their local caches while the MySQL Fabric process is unavailable

5 Architected for Extensibility

MySQL Fabric has been architected for extensibility at a number of levels. For example, in the first release the only option for implementing HA is based on MySQL Replication but in future releases we hope to add further options (for example, MySQL Cluster). We also hope to see completely new applications around the managing of farms of MySQL Servers - both from Oracle and the wider MySQL community.

Figure 5 illustrates how new applications and protocols can be added using the pluggable framework.

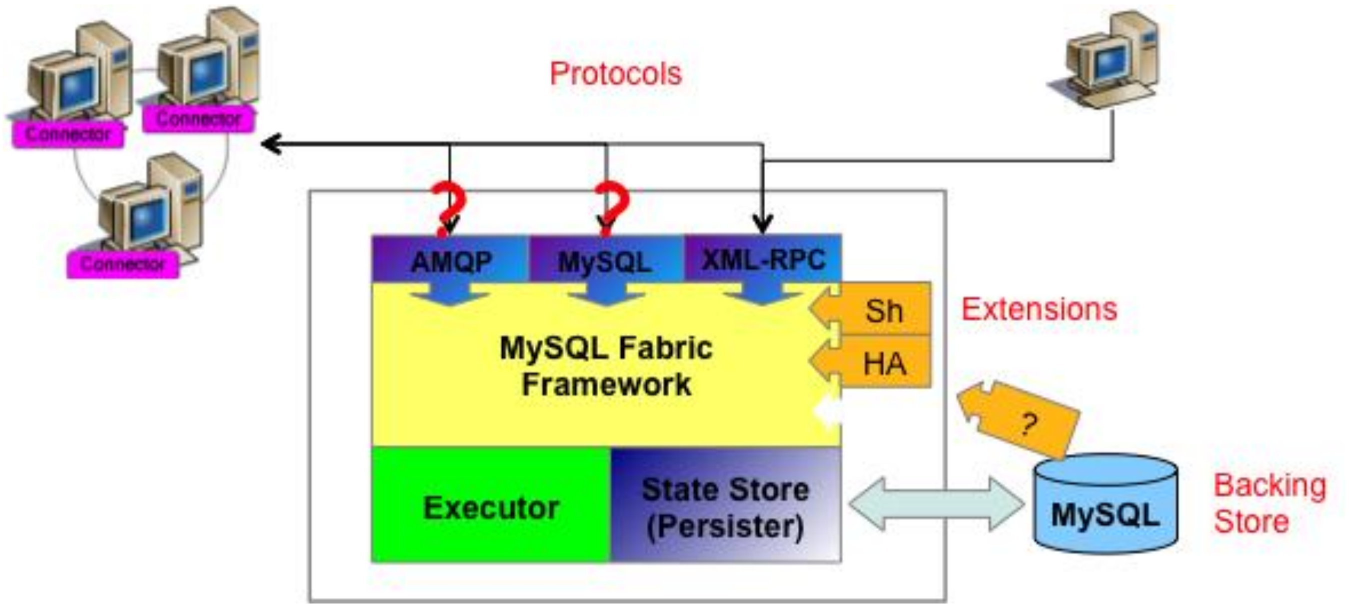


Figure 5 MySQL Fabric's Extensible Architecture

6 Getting Hands-On - Tutorials

This section focuses on how to actually use MySQL Fabric - initially to provide High Availability and then to augment that by scaling out using sharding. Most of the focus will be on the management tasks but a number of Python code examples will also be included to illustrate what the application needs to do. For simplicity, this paper skips configuring and starting all of the MySQL Servers (apart from the one used for MySQL Fabric's own state store); a complete end-to-end walkthrough (including configuring and running all of the MySQL Servers) can be found in MySQL Fabric - adding High Availability and Scaling to MySQL[1]. Additionally, if only HA is required then this is documented in MySQL Fabric – adding High Availability to MySQL[2] and if only sharding then MySQL Fabric - adding Scaling to MySQL[3].

Note that as any MySQL Server in these walkthroughs can be a Primary server (MySQL Replication master) their configuration files should enable Global Transaction IDs as well as binary logging and assign a unique `server-id`. There follows an example configuration file:

```
[mysql@fab2 myfab]$ cat my2a.cnf
[mysqld]
datadir=/home/mysql/myfab/data2a
basedir=/home/mysql/mysql
socket=/home/mysql/myfab/mysqlfab2a.socket
binlog-format=ROW
log-slave-updates=true
gtid-mode=on
enforce-gtid-consistency=true
master-info-repository=TABLE
relay-log-info-repository=TABLE
sync-master-info=1
port=3306
report-host=fab2
report-port=3306
server-id=21
log-bin=fab2a-bin.log
```

6.1 Adding High Availability

The intent of this section is to introduce MySQL Fabric to add High Availability to MySQL. Figure 6 illustrates the configuration that will be created.

There will be a single HA group that has the name `group_id-1` that will contain three MySQL Servers - each running on a different machine (`fab2`, `fab3` and `fab4`) - and at any point in time, one of those MySQL Servers will be the Primary (master) and the others will be Secondaries. Should the Primary fail, one of the Secondaries would be automatically promoted by the MySQL Fabric process to be the new Primary.

The MySQL Fabric process itself will run on a fourth machine (`fab1`) together with its state store (another MySQL Server) and the test application which uses the Fabric-aware Python connector.

[1] http://www.clusterdb.com/mysql-fabric/mysql-fabric-adding-high-availability-and-scaling-to-mysql
[2] http://www.clusterdb.com/mysql-fabric/mysql-fabric-adding-high-availability-to-mysql
[3] http://www.clusterdb.com/mysql-fabric/mysql-fabric-add-scaling-to-mysql

The first step is to install the Python connector as well as MySQL Fabric:

```
[root@fab1]# rpm -i mysql-connector-python-1.2.2-
             1.el6.noarch.rpm
[root@fab1]# rpm -i mysql-utilities-1.4.3-1.el6.noarch.rpm
```

The state store must then be configured and started:

```
[mysql@fab1 myfab]$ mkdir data
[mysql@fab1 myfab]$ cat my.cnf
[mysqld]
datadir=/home/mysql/myfab/data
basedir=/home/mysql/mysql
socket=/home/mysql/myfab/mysqlfab.socket
binlog-format=ROW
log-slave-updates=true
gtid-mode=on
enforce-gtid-consistency=true
master-info-repository=TABLE
relay-log-info-repository=TABLE
sync-master-info=1
port=3306
report-host=fab1
report-port=3306
server-id=1
log-bin=fab-bin.log

[mysql@fab1 mysql]$ scripts/mysql_install_db \
             --basedir=/home/mysql/mysql/ \
             --datadir=/home/mysql/myfab/data/ \
             --defaults-file=/home/mysql/myfab/my.cnf
mysql@fab1 ~]$ mysqld --defaults-file=/home/mysql/myfab/my.cnf &
```

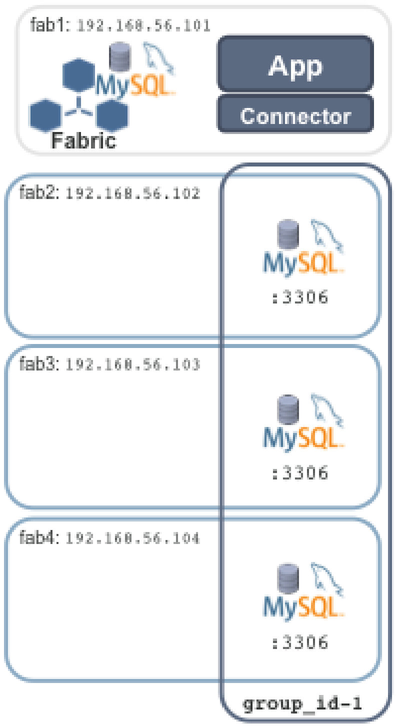


Figure 6 Single HA Group

The user account that will be used by MySQL Fabric can then be created:

```
mysql@fab1 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e \
            'CREATE USER fabric@localhost; GRANT ALL \
             ON fabric.* TO fabric@localhost;'
```

Note that the `fabric` schema has not yet been created - that will be done automatically by MySQL Fabric.

Before starting MySQL Fabric, its configuration file should be checked to make sure that it matches the actual environment that will be used for the MySQL Servers and MySQL Fabric itself:

```
[root@fab1 mysql]# cat /etc/mysql/fabric.cfg
[DEFAULT]
prefix =
sysconfdir = /etc
logdir = /var/log

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

[storage]
auth_plugin = mysql_native_password
database = fabric
user = fabric
address = localhost:3306
connection_delay = 1
connection_timeout = 6
password =
connection_attempts = 6

[failure_tracking]
```

```
notification_interval = 60
notification_clients = 50
detection_timeout = 1
detection_interval = 6
notifications = 300
detections = 3
failover_interval = 0
prune_time = 3600

[servers]
password =
user = fabric

[connector]
ttl = 1

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

[executor]
executors = 5

[sharding]
mysqldump_program = /home/mysql/mysql/bin/mysqldump
mysqlclient_program = /home/mysql/mysql/bin/mysql
```

Details on the meanings of these configuration parameters can be found in the MySQL Fabric documentation[1].

The fabric schema within the state store can now be created and (optionally) checked:

```
[root@fab1 mysql]# mysqlfabric manage setup --param=storage.user=fabric
[INFO] 1399476439.536728 - MainThread - Initializing persister: user \
             (fabric), server (localhost:3306), database (fabric).
[INFO] 1399476451.330008 - MainThread - Initial password for admin/xmlrpc set
Password set for admin/xmlrpc from configuration file.
[INFO] 1399476451.333563 - MainThread - Password set for admin/xmlrpc \
             from configuration file.

[mysql@fab1 ~]$ mysql --protocol=tcp -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.6.16-log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

[1] http://dev.mysql.com/doc/mysql-utilities/1.4/en/fabric-configuration-file.html

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| fabric             |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)

mysql> use fabric;show tables;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
+-------------------+
| Tables_in_fabric  |
+-------------------+
| checkpoints       |
| error_log         |
| group_replication |
| groups            |
| permissions       |
| role_permissions  |
| roles             |
| servers           |
| shard_maps        |
| shard_ranges      |
| shard_tables      |
| shards            |
| user_roles        |
| users             |
+-------------------+
14 rows in set (0.00 sec)
```

The MySQL Fabric process can now be started (note that the Fabric process can be run as a daemon by adding the `--daemonize` option:

```
[mysql@fab1 ~]$ mysqlfabric manage start
[INFO] 1399476805.324729 - MainThread - Fabric node starting.
[INFO] 1399476805.327456 - MainThread - Initializing persister: user (fabric), server
             (localhost:3306), database (fabric).
[INFO] 1399476805.335908 - MainThread - Initial password for admin/xmlrpc set
Password set for admin/xmlrpc from configuration file.
[INFO] 1399476805.339028 - MainThread - Password set for admin/xmlrpc from configuration
             file.
[INFO] 1399476805.339868 - MainThread - Loading Services.
[INFO] 1399476805.359542 - MainThread - Starting Executor.
[INFO] 1399476805.360668 - MainThread - Setting 5 executor(s).
[INFO] 1399476805.363478 - Executor-0 - Started.
[INFO] 1399476805.366553 - Executor-1 - Started.
[INFO] 1399476805.368680 - Executor-2 - Started.
[INFO] 1399476805.372392 - Executor-3 - Started.
[INFO] 1399476805.376179 - MainThread - Executor started.
[INFO] 1399476805.382025 - Executor-4 - Started.
[INFO] 1399476805.385570 - MainThread - Starting failure detector.
[INFO] 1399476805.389736 - XML-RPC-Server - XML-RPC protocol server ('127.0.0.1', 8080)
             started.
[INFO] 1399476805.390695 - XML-RPC-Server - Setting 5 XML-RPC session(s).
[INFO] 1399476805.393918 - XML-RPC-Session-0 - Started XML-RPC-Session.
[INFO] 1399476805.396812 - XML-RPC-Session-1 - Started XML-RPC-Session.
```

```
[INFO] 1399476805.399596 - XML-RPC-Session-2 - Started XML-RPC-Session.
[INFO] 1399476805.402650 - XML-RPC-Session-3 - Started XML-RPC-Session.
[INFO] 1399476805.405305 - XML-RPC-Session-4 - Started XML-RPC-Session.
```

Initially, there will be a single HA group (this is all that is required for HA - later, additional groups will be added to enable scaling out through partitioning/sharding of the data):

```
[mysql@fab1 myfab]$ mysqlfabric group create group_id-1
Procedure :
{ uuid        = 7e0c90ec-f81f-4ff6-80d3-ae4a8e533979,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
```

Before adding the MySQL Servers to the group, it is necessary to have a user on each MySQL Server that can access the server from the machine where the MySQL Fabric process is running (in this example - `fab1`/`192.168.56.101`). For simplicity this paper uses the `root` user, without a password and with permission to access all tables from any host - clearly this is only for experimentation and much tighter security should be used in any production database!

```
[mysql@fab2 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO fabric@"%"'
[mysql@fab3 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO fabric@"%"'
[mysql@fab4 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO fabric@"%"'
[mysql@fab2 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO root@"%"'
[mysql@fab3 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO root@"%"'
[mysql@fab4 ~]$ mysql -h 127.0.0.1 -P3306 -u root -e 'GRANT ALL ON *.* TO root@"%"'
```

MySQL Fabric is now able to access and manipulate those MySQL Servers and so they can now be added to the HA group.

```
[mysql@fab1 myfab]$ mysqlfabric group add group_id-1 192.168.56.102:3306
Procedure :
{ uuid        = 073f421a-9559-4413-98fd-b839131ea026,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1 myfab]$ mysqlfabric group add group_id-1 192.168.56.103:3306
Procedure :
{ uuid        = b0f5b04a-27e6-46ce-adff-bf1c046829f7,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1 myfab]$ mysqlfabric group add group_id-1 192.168.56.104:3306
Procedure :
{ uuid        = 520d1a7d-1824-4678-bbe4-002d0bae5aaa,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
```

At this point, all of the MySQL Servers are acting as Secondaries (in other words, none of them is acting as the MySQL Replication master). The next step is to promote one of the servers to be the Primary; in this case the `uuid` of the server we want to promote is provided but that isn't required - in which case MySQL Fabric will select one.

```
[mysql@fab1 myfab]$ mysqlfabric group promote group_id-1 \
             --slave_uuid 00f9831f-d602-11e3-b65e-0800271119cb
Procedure :
{ uuid         = c875371b-890c-49ff-b0a5-6bbc38be7097,
  finished     = True,
  success      = True,
  return       = True,
  activities   =
}
[mysql@fab1 myfab]$ mysqlfabric group lookup_servers group_id-1
Command :
{ success      = True
  return       = [
               {'status': 'PRIMARY', 'server_uuid': '00f9831f-d602-11e3-b65e-0800271119cb', \
                    'mode': 'READ_WRITE', 'weight': 1.0, 'address': '192.168.56.104:3306'}, \
               {'status': 'SECONDARY', 'server_uuid': 'f6fe224e-d601-11e3-b65d-0800275185c2', \
                    'mode': 'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.102:3306'}, \
               {'status': 'SECONDARY', 'server_uuid': 'fbb5c440-d601-11e3-b65d-0800278bafa8', \
                    'mode': 'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.103:3306'}]
  activities   =
}
```

Note that `fab4` is now showing as the Primary; any of the Secondary servers can also be queried to confirm that they are indeed MySQL replication slaves of the Primary.

```
[mysql@fab1 ~]$ mysql -h 192.168.56.103 -P3306 -u root -e "show slave status\G"
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.56.104
                  Master_User: fabric
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: fab3-bin.000003
          Read_Master_Log_Pos: 487
               Relay_Log_File: fab3-relay-bin.000002
                Relay_Log_Pos: 695
        Relay_Master_Log_File: fab3-bin.000003
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 487
              Relay_Log_Space: 898
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids:
             Master_Server_Id: 40
                  Master_UUID: 00f9831f-d602-11e3-b65e-0800271119cb
             Master_Info_File: mysql.slave_master_info
                    SQL_Delay: 0
          SQL_Remaining_Delay: NULL
      Slave_SQL_Running_State: Slave has read all relay log; waiting for the slave I/O \
                                                    thread to update it
           Master_Retry_Count: 86400
                  Master_Bind:
      Last_IO_Error_Timestamp:
     Last_SQL_Error_Timestamp:
               Master_SSL_Crl:
           Master_SSL_Crlpath:
           Retrieved_Gtid_Set: 00f9831f-d602-11e3-b65e-0800271119cb:1-2
            Executed_Gtid_Set: 00f9831f-d602-11e3-b65e-0800271119cb:1-2,\
                                     fbb5c440-d601-11e3-b65d-0800278bafa8:1-2
                Auto_Position: 1
```

At this stage, the MySQL replication relationship is configured and running but there isn't yet High Availability as MySQL Fabric is not monitoring the state of the servers - the final configuration step fixes that:

```
[mysql@fab1 ~]$ mysqlfabric group activate group_id-1
Procedure :
{ uuid        = 40a5e023-06ba-4e1e-93de-4d4195f87851,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
```

Everything is now set up to detect if the Primary (master) should fail and in the event that it does, promote one of the Secondaries to be the new Primary. If using one of the MySQL Fabric-aware connectors (initially PHP, Python and Java) then that failover can be transparent to the application.

The code that follows shows how an application can accesses this new HA group - in this case, using the Python connector. First an application table is created:

```
[mysql@fab1 myfab]$ cat setup_table_ha.py
import mysql.connector
from mysql.connector import fabric

conn = mysql.connector.connect(
    fabric={"host" : "localhost", "port" : 32274, "username": "admin",
           "password" : "admin"},
    user="root", database="test", password="",
    autocommit=True
)

conn.set_property(mode=fabric.MODE_READWRITE, group="group_id-1")
cur = conn.cursor()
cur.execute(
"CREATE TABLE IF NOT EXISTS subscribers ("
"   sub_no INT, "
"   first_name CHAR(40), "
"   last_name CHAR(40)"
")"
)
```

Note the following about that code sample:

- The connector is provided with the address for the MySQL Fabric process `localhost:32274` rather than any of the MySQL Servers

- The `mode` property for the connection is set to `fabric.MODE_READWRITE` which the connector will interpret as meaning that the transaction should be sent to the Primary (as that's where all writes must be executed so that they can be replicated to the Secondaries)
- The `group` property is set to `group_id-1` which is the name that was given to the single HA Group

This code can now be executed and then a check made on one of the Secondaries that the table creation has indeed been replicated from the Primary.

```
[mysql@fab1 myfab]$ python setup_table_ha.py
[mysql@fab1 myfab]$ mysql -h 192.168.56.103 -P3306 -u root -e "use test;show tables;"
+-----------------+
| Tables_in_test |
+-----------------+
| subscribers    |
+-----------------+
```

The next step is to add some rows to the table:

```
[mysql@fab1 myfab]$ cat add_subs_ha.py
import mysql.connector
from mysql.connector import fabric

def add_subscriber(conn, sub_no, first_name, last_name):
    conn.set_property(group="group_id-1", mode=fabric.MODE_READWRITE)
    cur = conn.cursor()
    cur.execute(
        "INSERT INTO subscribers VALUES (%s, %s, %s)",
        (sub_no, first_name, last_name)
        )

conn = mysql.connector.connect(
    fabric={"host" : "localhost", "port" : 32274, "username": "admin",
"password" : "admin"},
    user="root", database="test", password="",
    autocommit=True
    )

conn.set_property(group="group_id-1", mode=fabric.MODE_READWRITE)

add_subscriber(conn, 72, "Billy", "Fish")
add_subscriber(conn, 500, "Billy", "Joel")
add_subscriber(conn, 1500, "Arthur", "Askey")
add_subscriber(conn, 5000, "Billy", "Fish")
add_subscriber(conn, 15000, "Jimmy", "White")
add_subscriber(conn, 17542, "Bobby", "Ball")

[mysql@fab1 myfab]$ python add_subs_ha.py

[mysql@fab1 myfab]$ mysql -h 192.168.56.103 -P3306 -u root -e "select * from
             test.subscribers"
+--------+------------+-----------+
| sub_no | first_name | last_name |
+--------+------------+-----------+
|     72 | Billy      | Fish      |
|    500 | Billy      | Joel      |
|   1500 | Arthur     | Askey     |
|   5000 | Billy      | Fish      |
|  15000 | Jimmy      | White     |
|  17542 | Bobby      | Ball      |
+--------+------------+-----------+
```

And then the data can be retrieved (note that the `mode` parameter for the connection is set to `fabric.MODE_READONLY` and so the connector knows that it can load balance the requests across any MySQL Servers in the HA Group).

```
mysql@fab1 myfab]$ cat read_table_ha.py
import mysql.connector
from mysql.connector import fabric

def find_subscriber(conn, sub_no):
    conn.set_property(group="group_id-1", mode=fabric.MODE_READONLY)
    cur = conn.cursor()
    cur.execute(
        "SELECT first_name, last_name FROM subscribers "
        "WHERE sub_no = %s", (sub_no, )
        )
    for row in cur:
        print row

conn = mysql.connector.connect(
    fabric={"host" : "localhost", "port" : 32274, "username": "admin",
            "password" : "admin"},
    user="root", database="test", password="",
    autocommit=True
    )

find_subscriber(conn, 72)
find_subscriber(conn, 500)
find_subscriber(conn, 1500)
find_subscriber(conn, 5000)
find_subscriber(conn, 15000)
find_subscriber(conn, 17542)

[mysql@fab1 myfab]$ python read_table_ha.py
(u'Billy', u'Fish')
(u'Billy', u'Joel')
(u'Arthur', u'Askey')
(u'Billy', u'Fish')
(u'Jimmy', u'White')
(u'Bobby', u'Ball')
```

Note that if the Secondary servers don't all have the same performance then you can skew the ratio for how many reads are sent to each one using the `mysqlfabric server set_weight` command - specifying a value between `0` and `1` (default is `1` for all servers). Additionally, the `mysqlfabric server set_mode` command can be used to specify if the Primary should receive some of the reads (`READ_WRITE`) or only writes (`WRITE_ONLY`).

Because replication to the slaves is asynchronous, reads from them cannot be guaranteed to return the very latest committed data. If it's essential that the read is completely consistent with all committed transactions then the mode should be set to `fabric.MODE_READWRITE`.

For brevity, these code examples don't contain error handling – for a real application the code should deal with any database errors appropriately.

The next section describes how this configuration can be extended to add scalability by sharding the table data (and it can be skipped if that isn't needed).

6.2 Adding Scale-Out with Sharding

The example in this section builds upon the previous one by adding more servers in order to scale out the capacity and read/write performance of the database. The first step is to create a new group (which is named `global-group` in this example) - the Global Group is a special HA group that performs two critical functions:

- Any data schema changes are applied to the Global Group and from there they will be replicated to each of the other HA Groups
- If there are tables that contain data that should be replicated to all HA groups (rather than sharded) then any inserts, updates or deletes will be made on the Global Group and then replicated to the others. Those tables are referred to as global tables.

Figure 7 illustrates what the configuration will look like once the Global Group has been created.

The Global Group is defined and populated with MySQL Servers and then a Primary is promoted in the following steps:

```
[mysql@fab1]$ mysqlfabric group create global-group
Procedure :
{ uuid        = 5f07e324-ec0a-42b4-98d0-46112f607143,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}

[mysql@fab1 ~]$ mysqlfabric group add global-group \
    192.168.56.102:3316
Procedure :
{ uuid        = ccf699f5-ba2c-4400-a8a6-f951e10d4315,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1 ~]$ mysqlfabric group add global-group \
    192.168.56.102:3317
Procedure :
{ uuid        = 7c476dda-3985-442a-b94d-4b9e650e5dfe,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1 ~]$ mysqlfabric group add global-group \
    192.168.56.102:3318
Procedure :
{ uuid        = 476fadd4-ca4f-49b3-a633-25dbe0ffdd11,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}

[mysql@fab1 ~]$ mysqlfabric group promote global-group
Procedure :
{ uuid        = e818708e-6e5e-4b90-aff1-79b0b2492c75,
  finished    = True,
```

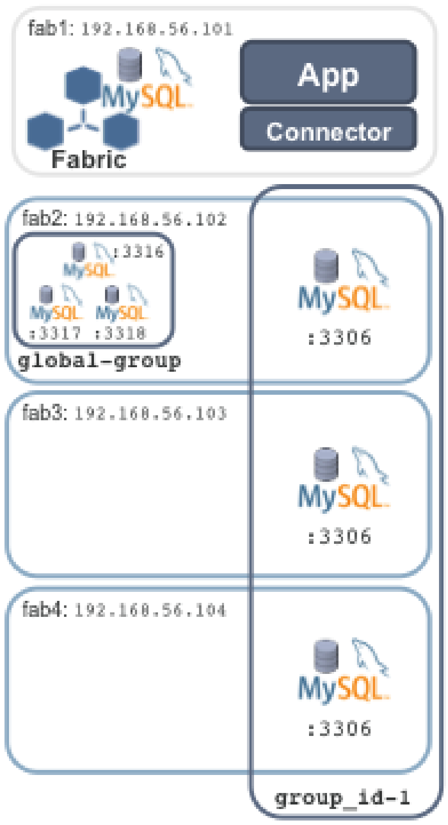


Figure 7 Addition of Global Group

```
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1 ~]$ mysqlfabric group lookup_servers global-group
Command :
{ success     = True
  return      = [
               {'status': 'PRIMARY', 'server_uuid': '56a08135-d60b-11e3-b69a-0800275185c2',\
                    'mode': 'READ_WRITE', 'weight': 1.0, 'address': '192.168.56.102:3316'}, \
               {'status': 'SECONDARY', 'server_uuid': '5d5f5cf6-d60b-11e3-b69b-0800275185c2', \
                    'mode': 'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.102:3317'}, \
               {'status': 'SECONDARY', 'server_uuid': '630616f4-d60b-11e3-b69b-0800275185c2', \
                    'mode': 'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.102:3318'}]
  activities  =
}
```

As an application table has already been created within the original HA group, that will need to copied to the new Global Group:

```
mysql@fab1 myfab]$ mysqldump -d -u root --single-transaction -h 192.168.56.102 -P3306 \
--all-databases > my-schema.sql
[mysql@fab1 myfab]$ mysql -h 192.168.56.102 -P3317 -u root -e 'reset master'
[mysql@fab1 myfab]$ mysql -h 192.168.56.102 -P3317 -u root < my-schema.sql
[mysql@fab1 myfab]$ mysql -h 192.168.56.102 -P3317 -u root -e 'show create table
             test.subscribers'
+-------------+------------------------------..-+
| Table       | Create Table                 .. |
+-------------+------------------------------..-+
| subscribers | CREATE TABLE `subscribers` (
  `sub_no` int(11) DEFAULT NULL,
  `first_name` char(40) DEFAULT NULL,
  `last_name` char(40) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-------------+------------------------------..-+
```

A shard mapping is an entity that is used to define how certain tables should be sharded between a set of HA groups. It is possible to have multiple shard mappings but in this example, only one will be used. When defining the shard mapping, there are two key parameters:

- The type of mapping - can be either `HASH` or `RANGE`
- The global group that will be used

The commands that follow define the mapping and identify the index number assigned to this mapping (in this example - 3) - in fact that same index is recovered in two different ways: using the `mysqlfabric` command and then reading the data directly from the state store:

```
[mysql@fab1 ~]$ mysqlfabric sharding create_definition HASH global-group
Procedure :
{ uuid        = 78ea7209-b073-4d03-9d8b-bda92cc76f32,
  finished    = True,
  success     = True,
  return      = 1,
  activities  =
}

[mysql@fab1 ~]$ mysqlfabric sharding list_definitions
Command :
{ success     = True
  return      = [[1, 'HASH', 'global-group']]
  activities  =
}
```

```
[mysql@fab1 ~]$ mysql -h 127.0.0.1 -P3306 -u root \
             -e 'SELECT * FROM fabric.shard_maps'
+------------------+-----------+--------------+
| shard_mapping_id | type_name | global_group |
+------------------+-----------+--------------+
|                1 | HASH      | global-group |
+------------------+-----------+--------------+
```

The next step is to define what columns from which tables should be used as the sharding key (the value on which the `HASH` function is executed or is compared with the defined `RANGE`s). In this example, only one table is being sharded (the `subscribers` table with the `sub_no` column being used as the sharding key) but the command can simply be re-executed for further tables. Note that the identifier for the shard mapping (`3`) is passed on the command-line:

```
[mysql@fab1 ~]$ mysqlfabric sharding add_table 1 test.subscribers sub_no
Procedure :
{ uuid        = 446aadd1-ffa6-4d19-8d52-4683f3d7c998,
  finished    = True,
  success     = True,
  return      = True,
  activities  =

}
```

At this point, the shard mapping has been defined but no shards have been created and so the next step is to create a single shard and that shard will be stored in the existing HA group `group_id-1`):

```
[mysql@fab1]$ mysqlfabric sharding add_shard 1 group_id-1 --state=enabled
Procedure :
{ uuid        = 8a351c36-9e80-41fd-a665-f3369aa3b31b,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
[mysql@fab1]$ mysql -h 127.0.0.1 -P3306 -u root -e 'select * from fabric.shards'
+----------+------------+---------+
| shard_id | group_id   | state   |
+----------+------------+---------+
|        1 | group_id-1 | ENABLED |
+----------+------------+---------+
```

At this point, the database has technically been sharded but of course it offers no scalability as there is only a single shard. The steps that follow evolve that configuration into one containing two shards as shown in Figure 8.

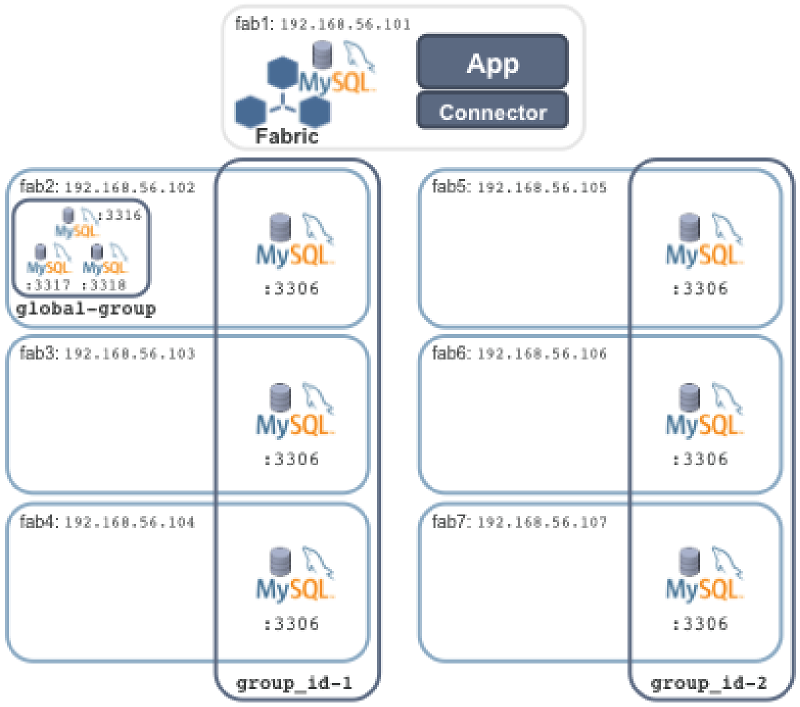


Figure 8 Sharded, HA MySQL Fabric Server Farm

Another HA group (`group_id-2`) is created, the three new servers added to it and then one of the servers is promoted to be the Primary:

```
[mysql@fab1 ~]$ mysqlfabric group create group_id-2
Procedure :
{ uuid          = 4ec6237a-ca38-497c-b54d-73d1fa7bbe03,
  finished      = True,
  success       = True,
  return        = True,
  activities    =
}
[mysql@fab1 ~]$ mysqlfabric group add group_id-2 192.168.56.105:3306
Procedure :
{ uuid          = fe679280-81ed-436c-9b7f-3d6f46987492,
  finished      = True,
  success       = True,
  return        = True,
  activities    =
}
[mysql@fab1 ~]$ mysqlfabric group add group_id-2 192.168.56.106:3306
Procedure :
{ uuid          = 6fcf7e0c-c092-4d81-9898-448abf2b113c,
  finished      = True,
  success       = True,
  return        = True,
  activities    =
}
[mysql@fab1 ~]$ mysqlfabric group add group_id-2 192.168.56.107:3306
Procedure :
{ uuid          = 8e9d4fbb-58ef-470d-81eb-8d92813427ae,
  finished      = True,
  success      = True,
  return       = True,
  activities   =
}
[mysql@fab1 ~]$ mysqlfabric group promote group_id-2
Procedure :
{ uuid         = 21569d7f-93a3-4bdc-b22b-2125e9b75fca,
  finished     = True,
  success      = True,
  return       = True,
  activities   =
}
```

At this point, the new HA group exists but is missing the application schema and data. Before allocating a shard to the group, a `reset master` needs to be executed on the Primary for the group (this is required because changes have already been made on that server - if nothing else, to grant permissions for one or more users to connect remotely). The `mysqlfabric group lookup_servers` command is used to first check which of the three servers is currently the Primary.

```
[mysql@fab1 ~]$ mysqlfabric group lookup_servers group_id-2
Command :
{ success      = True
  return       = [
                {'status': 'PRIMARY', 'server_uuid': '10b086b5-d617-11e3-b6e7-08002767aedd', \
                     'mode': 'READ_WRITE', 'weight': 1.0, 'address': '192.168.56.105:3306'}, \
                {'status': 'SECONDARY', 'server_uuid': '5dc81563-d617-11e3-b6e9-08002717142f', \
                     'mode': 'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.106:3306'}, \
                {'status': 'SECONDARY', 'server_uuid': '83cae7b2-d617-11e3-b6ea-08002763b127', 'mode':
                'READ_ONLY', 'weight': 1.0, 'address': '192.168.56.107:3306'}]
  activities   =
}

[mysql@fab1 myfab]$ mysql -h 192.168.56.105 -P3306 -uroot -e 'reset master'
```

The next step is to split the existing shard, specifying the shard id (in this case `1` - which we find from the `fabric.shards` table) and the name of the HA group where the new shard will be stored:

```
[mysql@fab1]$ mysql -h 127.0.0.1 -P3306 -u root \
    -e 'select * from fabric.shards'
+----------+------------+---------+
| shard_id | group_id   | state   |
+----------+------------+---------+
|        1 | group_id-1 | ENABLED |
+----------+------------+---------+

[mysql@fab1]$ mysqlfabric sharding split_shard 1 group_id-2
Procedure :
{ uuid         = 4c559f6c-0b08-4a57-b095-364755636b7b,
  finished     = True,
  success      = True,
  return       = True,
  activities   =
}
```

Before looking at the application code changes that are needed to cope with the sharded data, a simple test can be run to confirm that the table's existing data has indeed been split between the two shards:

```
[mysql@fab1]$ mysql -h 192.168.56.102 -P3306 -uroot -e 'select * from test.subscribers'
+--------+------------+-----------+
| sub_no | first_name | last_name |
+--------+------------+-----------+
|    500 | Billy      | Joel      |
|   1500 | Arthur     | Askey     |
|   5000 | Billy      | Fish      |
|  17542 | Bobby      | Ball      |
+--------+------------+-----------+

[mysql@fab1]$ mysql -h 192.168.56.107 -P3306 -uroot -e 'select * from test.subscribers'
+--------+------------+-----------+
| sub_no | first_name | last_name |
+--------+------------+-----------+
|     72 | Billy      | Fish      |
|  15000 | Jimmy      | White     |
+--------+------------+-----------+
```

The next example Python code adds some new rows to the `subscribers` table. Note that the `tables` property for the connection is set to `test.subscribers` and the `key` to the value of the `sub_no` column for that table - this is enough information for the Fabric-aware connector to choose the correct shard/HA group and then the fact that the `mode` property is set to `fabric.MODE_READWRITE` further tells the connector that the transaction should be sent to the Primary within that HA group.

```
[mysql@fab1 myfab]$ cat add_subs_shards2.py
import mysql.connector
from mysql.connector import fabric

def add_subscriber(conn, sub_no, first_name, last_name):
    conn.set_property(tables=["test.subscribers"], key=sub_no, mode=fabric.MODE_READWRITE)
    cur = conn.cursor()
    cur.execute(
        "INSERT INTO subscribers VALUES (%s, %s, %s)",
        (sub_no, first_name, last_name)
        )

conn = mysql.connector.connect(
    fabric={"host" : "localhost", "port" : 32274, "username": "admin", "password" :
             "admin"},
    user="root", database="test", password="",
    autocommit=True
)

conn.set_property(tables=["test.subscribers"], scope=fabric.SCOPE_LOCAL)

add_subscriber(conn, 22, "Billy", "Bob")
add_subscriber(conn, 8372, "Banana", "Man")
add_subscriber(conn, 93846, "Bill", "Ben")
add_subscriber(conn, 5006, "Andy", "Pandy")
add_subscriber(conn, 15050, "John", "Smith")
add_subscriber(conn, 83467, "Tommy", "Cannon")
[mysql@fab1 myfab]$ python add_subs_shards2.py
```

For brevity, these code examples don't contain error handling – for a real application the code should deal with any database errors appropriately.

The mysql client can then be used to confirm that the new data has also been partitioned between the two shards/HA groups.

```
[mysql@fab1 myfab]$ mysql -h 192.168.56.103 -P3306 -uroot -e 'select * from
             test.subscribers'
+--------+------------+-----------+
| sub_no | first_name | last_name |
+--------+------------+-----------+
|    500 | Billy      | Joel      |
|   1500 | Arthur     | Askey     |
|   5000 | Billy      | Fish      |
|  17542 | Bobby      | Ball      |
|     22 | Billy      | Bob       |
|   8372 | Banana     | Man       |
|  93846 | Bill       | Ben       |
|  15050 | John       | Smith     |
+--------+------------+-----------+

[mysql@fab1 myfab]$ mysql -h 192.168.56.107 -P3306 -uroot -e 'select * from
             test.subscribers'
+--------+------------+-----------+
| sub_no | first_name | last_name |
+--------+------------+-----------+
|     72 | Billy      | Fish      |
|  15000 | Jimmy      | White     |
|   5006 | Andy       | Pandy     |
|  83467 | Tommy      | Cannon    |
+--------+------------+-----------+
```

The final example application code reads the row for each of the records that have been added, the key thing to note here is that the mode property for the connection has been set to fabric.MODE_READONLY so that the Fabric-aware Python connector knows that it can load balance requests over the Secondaries within the HA groups rather than sending everything to the Primary.

```
[mysql@fab1 myfab]$ cat read_table_shards2.py
import mysql.connector
from mysql.connector import fabric

def find_subscriber(conn, sub_no):
    conn.set_property(tables=["test.subscribers"], key=sub_no, mode=fabric.MODE_READONLY)
    cur = conn.cursor()
    cur.execute(
        "SELECT first_name, last_name FROM subscribers "
        "WHERE sub_no = %s", (sub_no, )
        )
    for row in cur:
        print row

conn = mysql.connector.connect(
    fabric={"host" : "localhost", "port" : 32774, "username": "admin",
           "password" : "admin"},
    user="root", database="test", password="",
    autocommit=True
    )
find_subscriber(conn, 22)
find_subscriber(conn, 72)
find_subscriber(conn, 500)
find_subscriber(conn, 1500)
find_subscriber(conn, 8372)
find_subscriber(conn, 5000)
find_subscriber(conn, 5006)
find_subscriber(conn, 93846)
find_subscriber(conn, 15000)
find_subscriber(conn, 15050)
find_subscriber(conn, 17542)
find_subscriber(conn, 83467)

[mysql@fab1 myfab]$ python read_table_shards2.py
(u'Billy', u'Bob')
(u'Billy', u'Fish')
(u'Billy', u'Joel')
(u'Arthur', u'Askey')
(u'Banana', u'Man')
(u'Billy', u'Fish')
(u'Andy', u'Pandy')
(u'Bill', u'Ben')
(u'Jimmy', u'White')
(u'John', u'Smith')
(u'Bobby', u'Ball')
(u'Tommy', u'Cannon')
```

7 Operational Best Practices

High Availability is not only a function of the underlying technology, but also well established and tested operating procedures managed by a highly skilled operations team. Industry analysts estimate that 80% of downtime is the result of "people and process", so the importance of operational best practices cannot be overstated.

Oracle offers a range of tools and services to enable MySQL users to achieve operational excellence and deliver against their committed SLAs.

7.1 Oracle University

Training of operational and administrative teams reduces the risk of human error that can result in accidental system outages. Oracle University offers an extensive range of MySQL training from introductory courses (i.e. MySQL Essentials, MySQL DBA, etc.) through to advanced certifications such as MySQL High Availability and MySQL Cluster Administration. It is also possible to define custom training plans for delivery at customer site.

You can - Learn more about MySQL training from the Oracle University[1].

7.2 MySQL Consulting

To ensure adherence to best practices from the initial design phase of a project through to implementation and sustaining, users can engage Oracle's MySQL Professional Services consultants. Delivered remotely or onsite, these engagements help in optimizing the architecture and increasing operational efficiency.

Again, Oracle offers a full range of consulting services, from Architecture and Design through to High Availability, Replication and Clustering. Learn more about Oracle's MySQL Consulting services[2].

7.2.1 MySQL Enterprise Edition and MySQL Cluster Carrier Grade Edition (CGE)

The commercial editions of MySQL deliver the most comprehensive set of MySQL production, backup, monitoring, modeling, development, and administration tools so organizations can achieve the highest levels of availability, performance and security.

Key components of MySQL Enterprise Edition and MySQL Cluster CGE are discussed below.

7.2.1.1 24x7 Global Support

MySQL offers 24x7x365 access to Oracle's MySQL Support team, which is staffed by seasoned database experts ready to help with the most complex technical issues, with direct access to the MySQL development team. Oracle's Premier support provides you with:

- 24x7x365 phone and online support
- Rapid diagnosis and solution to complex issues
- Unlimited incidents
- Emergency hot fix builds
- Access to Oracle's MySQL Knowledge Base
- Consultative support services

The Support team partners with customers in the analysis and remediation of issues that are causing outages, leading to faster problem resolution, and if needed, generates hot fixes to restore service. This level of assistance offers significant benefits for HA over community or self-supported environments.

[1] http://www.mysql.com/training/
[2] http://www.mysql.com/consulting/

Access to best practices Knowledge Base is included within MySQL support agreements. The Knowledge Base offers great insight into how to configure, provision and manage highly available MySQL environments.

Learn more about Oracle's MySQL support services[1].

7.2.1.2 MySQL Enterprise Monitor

During normal operations, monitoring of the infrastructure is key to maintaining high availability and can help you detect problems BEFORE they occur. MySQL Enterprise Monitor provides at-a-glance views of the health of your databases, continuously monitoring your MySQL servers and alerting you to potential problems before they impact your system.

MySQL Enterprise Monitor automatically tracks hundreds of MySQL variables to analyze current status. A sophisticated rules-based engine alerts administrators whenever parameters exceed defined thresholds so that DevOps and DBA teams can proactively avoid downtime or performance degradation.

Administrators are alerted immediately should an outage occur, and are presented with diagnostics information to speed remediation of the issue and quickly restore service availability.

MySQL Enterprise Monitor also stores historical MySQL status data so that analysis of issues is greatly simplified.

Learn more about MySQL Enterprise Monitor[2].

7.2.1.3 MySQL Enterprise Backup

Database backups are well-established processes in production environments. Depending on the technique used, backup operations can affect on-going services in several ways:

- Increased server load, impacting performance of production queries
- Blocking of write operations, limiting the service to read-only queries during the backup process
- Complete (planned) downtime during backup.

Of course, for HA services none of these is acceptable. A full online backup that does not consume MySQL Server resources is therefore the right choice to achieve HA.

MySQL Enterprise Backup performs online "Hot", non-blocking backups of your MySQL databases. Full backups can be performed on all InnoDB data, while MySQL is online, without interrupting queries or updates. In addition, incremental backups are supported where only data that has changed is backed up. Also partial backups are supported when only certain tables or tablespaces need to be captured.

MySQL Enterprise Backup restores your data from a full backup with full backward compatibility. Consistent Point-in-Time Recovery (PITR) enables DBAs to perform a restore to a specific point in time.

Learn more about MySQL Enterprise Backup[3].

7.2.1.4 MySQL Enterprise Security

MySQL Enterprise Security provides ready to use external authentication modules to easily integrate MySQL with existing security infrastructures including PAM and Windows Active Directory. MySQL users can be authenticated using Pluggable Authentication Modules ("PAM") or native Windows OS services.

Learn more about MySQL Enterprise Security[4].

[1] http://www.mysql.com/support/
[2] http://www.mysql.com/products/enterprise/monitor.html
[3] http://www.mysql.com/products/enterprise/backup.html
[4] http://www.mysql.com/products/enterprise/security.html

7.2.1.5 MySQL Enterprise Audit

MySQL Enterprise Audit enables you to quickly and seamlessly add policy-based auditing compliance to new and existing applications. You can dynamically enable user level activity logging, implement activity-based policies, manage audit log files and integrate MySQL auditing with Oracle and third-party solutions.

Learn more about MySQL Enterprise Audit[1].

8 Conclusion

MySQL Fabric is an extensible framework for managing farms of MySQL Servers and enabling MySQL Connectors to get transactions and queries to the most appropriate server while hiding the topology of the server farm from the application.

The intent is that developers can focus on high value activities such as adding new features to their applications rather than spending time on the platform plumbing - that can now be handled by MySQL Fabric.

The first applications supported by MySQL Fabric are High Availability (built on top of MySQL Replication) and sharding-based scale-out. Over time we hope to add new options to these applications (for example, alternate HA technologies) as well as completely new applications. We look forward to hearing what users would like us to add as well as what they build for themselves.

9 Additional Resources

- MySQL Fabric
  http://www.mysql.com/products/enterprise/fabric.html
- Detailed MySQL Fabric Tutorials:
  - MySQL Fabric – adding High Availability to MySQL
    http://www.clusterdb.com/mysql-fabric/mysql-fabric-adding-high-availability-to-mysql
  - MySQL Fabric - adding Scaling to MySQL
    http://www.clusterdb.com/mysql-fabric/mysql-fabric-add-scaling-to-mysql
  - MySQL Fabric - adding High Availability and Scaling to MySQL
    http://www.clusterdb.com/mysql-fabric/mysql-fabric-adding-high-availability-and-scaling-to-mysql
- MySQL Guide to High Availability Solutions
  http://www.mysql.com/why-mysql/white-papers/mysql-guide-to-high-availability-solutions/
- Automated Sharding and High Availability with MySQL Fabric – Webinar replay
  http://www.mysql.com/news-and-events/web-seminars/automated-sharding-and-high-availability-with-mysql-fabric/

[1] http://www.mysql.com/products/enterprise/audit.html