MySQL 5.6 Replication

An Introduction

A MySQL® Technical White Paper by Oracle

June 2014

MySQL

Table of Contents

ORACLE

1 Introduction

MySQL Replication enables users to cost-effectively deliver application performance, scalability and high availability. Many of the world's most trafficked web properties like eBay, Facebook, Tumblr, Twitter and YouTube rely on MySQL Replication to elastically scale-out beyond the capacity constraints of a single instance, enabling them to serve hundreds of millions of users and handle exponential growth.

By mirroring data between instances, MySQL replication is also the most common approach to delivering High Availability (HA) for MySQL databases. In addition, the MySQL replication utilities can automatically detect and recover from failures, allowing users to maintain service in the event of outages or planned maintenance.

With the release of MySQL 5.6, a number of enhancements have been made to MySQL Replication, delivering higher levels of data integrity, performance, automation and application flexibility. We will discuss these enhancements in the whitepaper.

We will also explore the business and technical advantages of deploying MySQL Replication and describe the fundamental enabling technology behind replication.

This paper concludes by outlining the major differences when using replication with the MySQL Cluster database.

A companion white paper (MySQL 5.6 Replication – Tutorial[1]) provides a simple step-by-step guide on how to install and configure a master/slave topology, as well as handle failover events. It demonstrates how to do this using Global Transaction Identifiers (GTIDs) as well as the extra steps needed when taking the more traditional approach.

2 Replication Fundamentals

For the purposes of this paper we define “replication” as the duplication of data to one or more locations. These can be co-located within a data center, or geographically distributed. In the forthcoming sections we will cover the differences between popular types of replication.

Replication enables a database to copy or duplicate changes from one physical location or system to another (typically from the “master” to a “slave” system). This is typically used to increase the availability and scalability of a database, though users will often also perform back-up operations or run analytical queries against the slave systems, thereby offloading such functions from the master systems.

MySQL natively supports replication as a standard feature of the database. Depending on the configuration, you can replicate all databases, selected databases, or even selected tables within a database.

MySQL Replication works by simply having one server act as a master, while one or more servers act as slaves. The master server will log the changes to the database. Once these changes have been logged, they are then sent and applied to the slave(s).

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

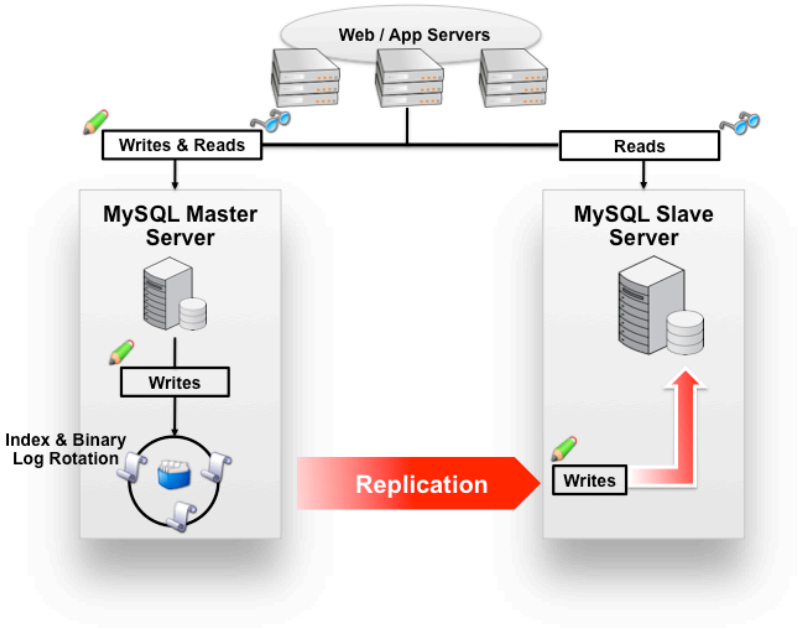

Figure 1 MySQL Replication Supports HA and Read Scalability "Out of the Box"

Using MySQL replication provides the ability to scale out large web farms where reads (SELECTs) represent the majority of operations performed on the database. Slaves present very little overhead to Master servers (typically a 1% overhead per slave), and it is not uncommon to find 30 slaves deployed per master in larger web properties[1].

Replication Modes and Data Consistency

There are multiple modes of replication, defined as asynchronous, semi- synchronous or synchronous.

Asynchronous Replication

By default, MySQL is asynchronous. Updates are committed to the database on the master and then relayed to the slave where they are also applied. The master does not wait for the slave to receive the update, and so is able to continue processing further write operations without being blocked as it waits for acknowledgement from the slave.

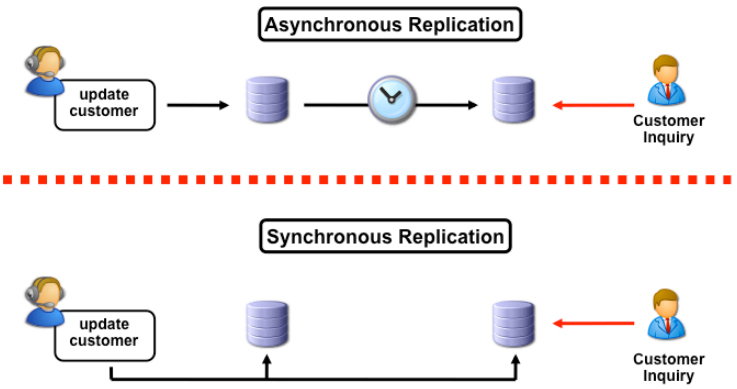

Figure 2 Contrasting different replication modes

When using asynchronous replication, there are no guarantees that all updates are replicated to the slave in the event of an outage of the master. Semi-Synchronous replication – discussed below – can be configured to enhance consistency between a MySQL master and its slaves, reducing the risk of data loss in the event of a failover.

[1] Refer to Ticketmaster materials in the Resources section for an example of complex MySQL replication topologies

Any delay (lag) of committed updates to the slaves is most noticeable with highly transactional applications where there is an abundance of write operations.

With the correct components and tuning, replication can appear to be almost instantaneous to the application.

Using asynchronous replication, slaves need not be connected permanently to receive updates from the master. This means that updates can occur over long-distance connections and even over temporary or intermittent connections. Depending on the configuration, you can replicate all databases, select databases, or even selected tables within a database.

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, or a timeout occurs. 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).

It is possible to combine the different modes of replication, so 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 facilitates faster failover times in the event of an outage, but it can add a performance overhead as a result of additional messaging between nodes.

Statement-Based Replication

By default, MySQL leverages statement-based replication where SQL statements (not the actual data changes) are replicated from the master to the slave(s).

An advantage of statement-based replication is that in some cases, less data ends up being written to log files, for example when updates or deletes affect many rows. For simple statements affecting just a few rows, then row-based replication can take up less space.

There are also some disadvantages to statement-based replication, most notably it cannot support statements which have nondeterministic behavior – like a current time function.

Row-Based Replication

Row-based replication logs the changes in individual table rows as opposed to statements. With row-based replication, the master writes messages, otherwise known as events to the binary log that indicate how individual table rows were changed. This is akin to more traditional forms of replication found in other RDBMSs. In general, row-based replication requires fewer locks on the master and slave, which means it is possible to achieve higher concurrency. A disadvantage of row-based replication is that it can generate more data that must be logged. For example, if a statement changes 100 rows in a table, that will mean 100 changes need to be logged with row-based replication, while with statement-based replication only the single SQL statement is required to be replicated.

It should be noted that development work to enhance MySQL replication is focussed on RBR over SBR and so it is recommended that you use RBR unless you have a compelling reason not to.

If MySQL Cluster is being used then row-based replication must be used.

Mixed-Format Replication

The binary logging format can be changed in real time according to the event being logged, using mixed-format logging. With mixed-format enabled, statement-based replication is used by default, but automatically switches to row-based replication under some conditions – for example:

- A DML statement that updates a MySQL Cluster table
- A statement contains `UUID()`
- Two or more tables with `AUTO_INCREMENT` columns are updated
- When any `INSERT DELAYED` is executed
- When the body of a view requires row-based replication, the statement creating the view also uses it — for example, this occurs when the statement creating a view uses the `UUID()` function
- A call to a User Defined Function (`UDF`) is made

For a complete list of conditions please visit: http://dev.mysql.com/doc/refman/5.6/en/binary-log-mixed.html

3 Summary of Replication Changes in MySQL 5.6

MySQL 5.6 introduces the most significant set of Replication enhancements ever released.

These enhancements deliver the following benefits:

High performance

- Improves consistency of reads from slaves (i.e. reduces the risk of reading "stale" data)
- Reduces the risk of data loss should the master fail before replicating all events in its binary log (binlog)

High availability

- Simpler failover through the use of GTIDs
- Proactive monitoring of the replication topology to identify issues before they can cause an outage

Data integrity

- Ensuring replicated data is correct, consistent and accessible

Developer / Operations (DevOps) Agility

- Automation to reduce administrative overhead
- Flexibility to support the rapid evolution of business requirements
- Maintaining low Total Cost of Ownership (TCO).

Each of these are discussed in more detail below.

HA Using MySQL Replication and GTIDs

The most significant HA enhancements come from the introduction of Global Transaction Identifiers (GTIDs). The primary motivation for introducing GTIDs is that it enables seamless failover or switchover with minimal manual intervention and service disruption.

GTIDs are unique identifiers comprising the server UUID (of the original master) and a transaction number. They are automatically generated as a header for every transaction and written with the transaction to the binary log. GTIDs make it simple to track and compare replicated transactions between the master and slaves, which in turn enables simple recovery from failures of the master. The default InnoDB storage engine must be used with GTIDs to get the full benefits of HA.

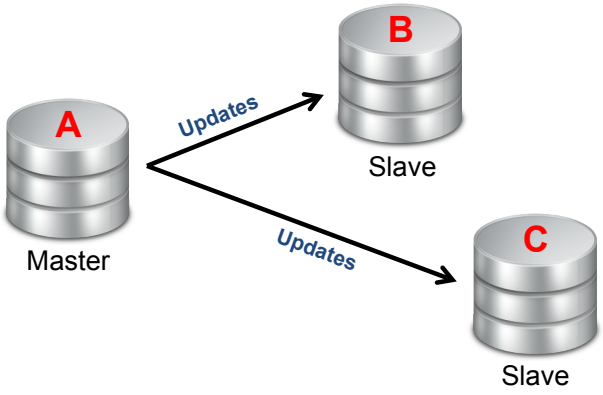

Figure 3 Simple master/slave topology

Global Transaction Identifiers (GTIDs)

To understand the implementation and capabilities of GTIDs, we will use one of the most common replication topologies in which a master (Server A) receives all updates, and replicates those to one or more slaves (Servers B and C). This is often referred to simply as “Master / Slave replication” or a “tree topology”, and is illustrated in Figure 3.

In the event of Server "A" crashing, we need to failover to one of the slaves, promoting it to master, while the remaining server becomes a slave of that new master, as illustrated in Figure 4.

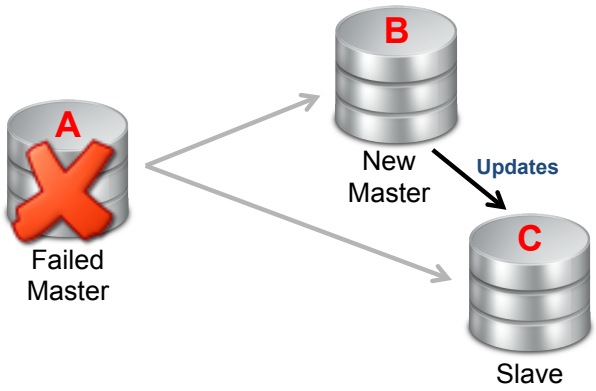

Figure 4 Failure & slave promotion

As MySQL replication is asynchronous by default, servers B and C may not have both replicated and executed the same set of transactions, i.e. one may be more complete than the other. Consider the following scenarios:

- Server B is more complete than C and B is chosen as the new master
- Server C needs to start to replicate from the first transaction in server B that it is yet to receive
- Server C may have executed transactions that have so far not been received by Server B
- Server B therefore needs to execute any missing transactions from Server C, before assuming the role of the master, otherwise lost transactions and conflicts can ensue

GTIDs apply a unique identifier to each transaction, so it is becomes easy to track when it is executed on each slave. When the master commits a transaction, it generates a GTID consisting of two components:

- The first component is the UUID of the server (a randomly generated 128-bit number)
- The second component is an automatically incremented integer; 1 for the first transaction committed on the server, 2 for the second transaction, and so on

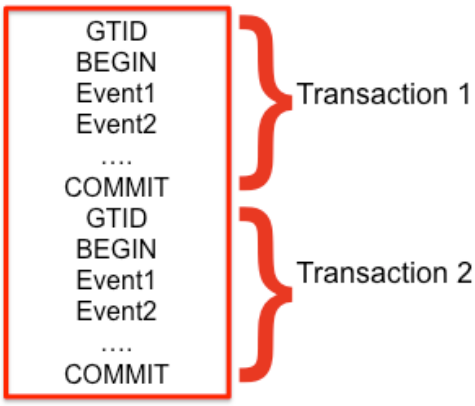

Figure 5 GTID positioning in the Binlog

The first component ensures that two transactions generated on *different* servers have different GTIDs. The second component ensures that two transactions generated on the *same* server have different GTIDs.

In ASCII, we write the GTID as "UUID:N", e.g. `22096C54-FE03-4B44-95E7-BD3C4400AF21:4711`

The GTID is written to the binary log prior to the transaction, as illustrated in Figure 5.

The GTID and the transaction are replicated to the slave. If the slave is configured to write changes to its own binary log, the slave ensures that the GTID and transaction are preserved and written after the transaction has been committed.

It is important to note that the slave does not generate a new GTID, even if the slave is configured as a master to other slaves in an n-tier (hierarchical or cascading) replication topology.

The set of GTIDs executed on each slave is exposed to the user in a new, read-only, global server variable, `gtid_executed`[1]. The variable can be used in conjunction with the `GTID_SUBTRACT()`[2] function to determine if a slave is up to date with a master, and if not, which transactions are missing.

A new replication protocol was created to make the process above automatic. When a slave connects to the master, the new protocol ensures it sends the range of GTIDs that the slave has executed and committed and requests any missing transactions. The master then sends all *other* transactions, i.e. those that the slave has not yet executed. This is illustrated in Figure 6 (note that the GTID numbering and binlog format is simplified for clarity).

In this example, Server B has executed all transactions before Server A crashed. Using the MySQL replication protocol, Server C will send "id1" to Server B, and then B will send "id2" and "id3" to Server C, before then replicating new transactions as they are committed.

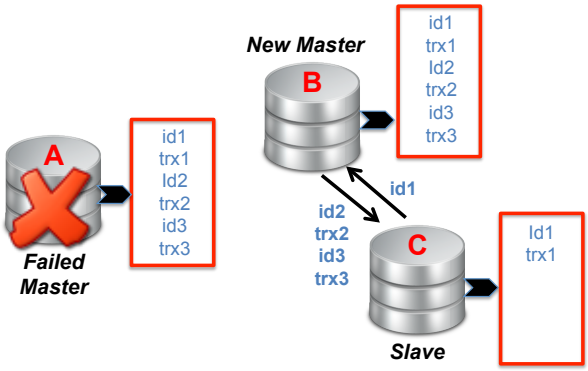

Figure 6 Automatically synchronizing a new master with its slaves

We therefore have a foundation for reliable slave promotion, ensuring that any transactions executed on a slave are not lost in the event of an outage to the master.

Utilities for Simplifying Replication

For convenience and ease-of-use, additional MySQL Replication Utilities are provided to accelerate the provisioning of new replication clusters.

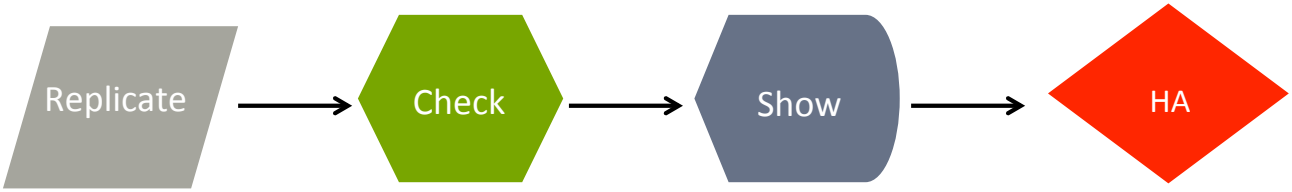

Figure 7 Fast provisioning with MySQL Replication Utilities

The utilities supporting failover and recovery are components of a broader suite of MySQL utilities[3] that simplify the maintenance and administration of MySQL servers, including the provisioning and verification of replication, comparing and cloning databases, diagnostics, etc. The utilities are available under the GPLv2 license, and are extendable using a supplied library. They are designed to work with Python 2.6 and 2.7. The source code is also available from the MySQL Utilities Launchpad page[4] and the package can be downloaded from the MySQL Website[5].

[1] https://dev.mysql.com/doc/refman/5.6/en/replication-options-gtids.html#sysvar_gtid_executed
[2] http://dev.mysql.com/doc/refman/5.6/en/miscellaneous-functions.html#function_gtid-subtract
[3] http://dev.mysql.com/doc/workbench/en/mysql-utils-intro-intro.html
[4] https://launchpad.net/mysql-utilities
[5] http://dev.mysql.com/downloads/tools/utilities/

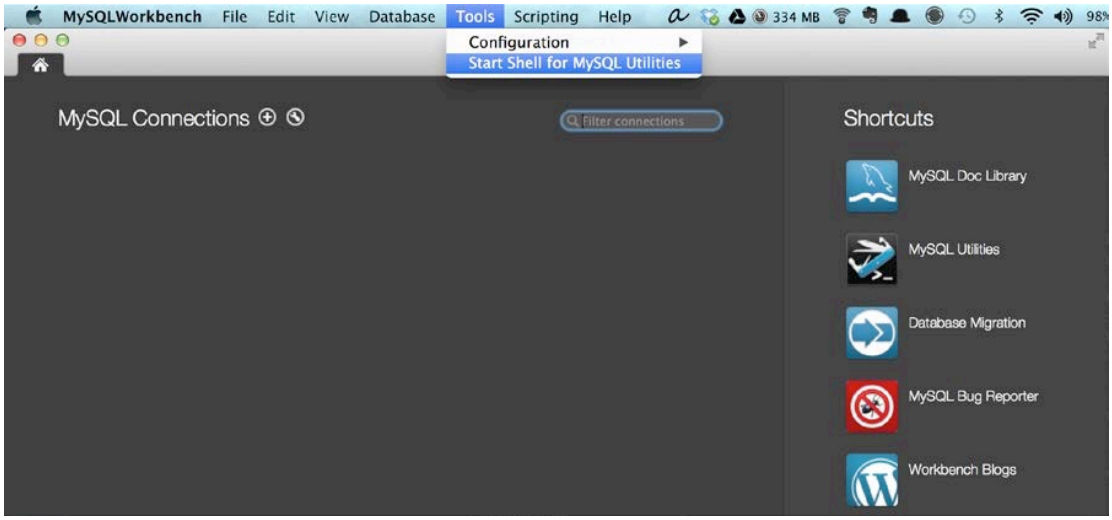

Figure 8 Launch MySQL Utilities using the pocket knife icon in MySQL Workbench

MySQL Utilities can be launched from the command line or from MySQL Workbench as shown in Figure 8. With MySQL WorkBench 6.0 and later, you must first install MySQL Utilities before they can be launched from the WorkBench GUI.

Replication Utility: mysqlreplicate

Enables fast and simple introduction of replication slaves, the `mysqlreplicate` utility is used to start the replication process. The user provides login and connection parameters for the master and optionally a point within the binary logs from where to start replication. Any GTIDs that have already been executed on the slave will be skipped. The utility also checks storage engine compatibility.

Replication Utility: mysqlrplcheck

Provides simple verification of deployment and fast fault resolution. The utility checks that the binlog is enabled and displays any configured exceptions. It then checks slave access and privileges to master and slave connection status. Each test is run sequentially, and status reports generated.

An example of the output is as follows:

```
$ mysqlrplcheck --master=root@host1:3310
  --slave=root@host2:3311
  --discover-slaves-login=root
 # master on host1: ... connected.
 # slave on host2: ... connected. Test Description      Status
 ------------------------------------------------------------
Checking for binary logging on master                 [pass]
Are there binlog exceptions?                          [pass]
Replication user exists?                              [pass]
Checking server_id values                             [pass]
Is slave connected to master?                         [pass]
Check master information file                         [pass]
Checking InnoDB compatibility                         [pass]
Checking storage engines compatibility                [pass]
Checking lower_case_table_names settings              [pass]
Checking slave delay (seconds behind master)          [pass]
# ...done.
```

Replication Utility: mysqlrplshow Discovers and displays the replication topology on-demand. It shows slaves attached to each master and labels each slave with hostname and port number.

```
$ mysqlrplshow --master=root@host1:3310 --discover-slaves-login=root
# master on host1: ... connected.
# Finding slaves for master: host1:3310

# Replication Topology Graph
host1:3310 (MASTER)
   |
   +--- host2:3310 - (SLAVE)
   |
   +--- host2:3311 - (SLAVE)
   |
   +--- host3:3310 - (SLAVE)
```

Replication Utility: mysqlfailover

While providing continuous monitoring of the replication topology, `mysqlfailover` enables automatic or manual failover to a slave in the event of an outage to the master. Its default behavior is to promote the first viable slave, as defined by the following slave election criteria.

- The slave is running and reachable
- GTIDs are enabled
- It is the most up-to-date slave
- The slaves replication filters do not conflict
- The replication user exists
- Binary logging is enabled

Once a viable slave is elected (called the candidate), the process to retrieve all transactions active in the replication cluster is initiated. This is done by connecting the candidate slave to all of the remaining slaves thereby gathering any missing transactions found in the cluster on the candidate slave. This ensures that no replicated transactions are lost.

The election process is configurable. The administrator can use a list to nominate a specific set of candidate slaves to become the new master (e.g. because they have better performing hardware).

To start the utility, users can specify a list of slaves or provide a default user and password to be used in discovering the slaves connected to the master. From this list of slaves, the user can define a specific subset of candidate slaves to be promoted to master in the event of an outage.

The utility connects to the master and its slaves from an independent host. When used in n-tier environments, users can run one instance for each master. The utility provides the ability to run a failover check and report the health of masters and slaves at specific intervals in one-second increments from five seconds and up.

The following screen shot shows the health reporting provided by the utility:

```
MySQL Replication Failover Utility
Failover Mode = auto     Next Interval = Wed Aug 15 13:19:30 2012
Master Information
------------------
Binary Log File    Position  Binlog_Do_DB  Binlog_Ignore_DB
black-bin.000001  2586

GTID Executed Set
A0F7E82D-3554-11E2-9949-080027685B56:1-5

Replication Health Status
+---------+-------+---------+--------+------------+---------+
| host    | port  | role    | state  | gtid_mode  | health  |
```

```
+---------+-------+---------+--------+------------+---------+
| black   | 3306  | MASTER  | UP     | ON         | OK      |
| blue    | 3306  | SLAVE   | UP     | ON         | OK      |
| green   | 3306  | SLAVE   | UP     | ON         | OK      |
| brown   | 3306  | SLAVE   | UP     | ON         | OK      |
| red     | 3306  | SLAVE   | UP     | ON         | OK      |
+---------+-------+---------+--------+------------+---------+
Q-quit R-refresh H-health G-GTID Lists U-UUIDs
```

At each interval, the utility will check to see if the server is alive via a ping operation, followed by a check of the connector to detect if the server is still reachable. If the master is found to be offline or unreachable, the utility will execute one of the following actions based on the value of the `failover mode` option, which enables the user to define failover policies:

- The `auto` mode tells the utility to failover to one of the list of specified candidates first and if none are viable, search the list of remaining slaves for a candidate
- The `elect` mode limits election to the candidate slave list and if none are viable, automatic failover is aborted
- The `fail` mode tells the utility to not perform failover and instead stop execution, awaiting further manual recovery actions from the DevOps team

In addition to these options, there are four "Extension Points" permitting users to interact with the utility during failover. It is therefore possible for users to extend HA policies by binding in their own scripts at each of the failover and recovery process:

- `exec-fail-check`: executes a script to determine if failover is needed. This replaces the default failure detection mechanism, allowing users to perform application-specific failure detection.
- `exec-before`: executes a script before failover is initiated. This can be used to tell the application to suspend write operations while a new master is provisioned.
- `exec-after`: executes a script immediately after failover to the new master. This permits users to inform the application that the new master is available and ready to accept write operations.
- `exec-post-fail`: execute a script after failover is complete and all slaves have been attached to the new master. This can be used to inform applications that it is safe to resume reads from the remaining slaves.

The combination of options to control failover and the ability to inform applications of the failover event are powerful features that enable self-healing for critical replication-based applications.

By default, `mysqlfailover` runs under the user's terminal process but it can instead be run as a daemon process using the `--daemon` option (to be used reliably the `--log`, `--pidfile` and `--report-values` options should be considered). This blog post[1] steps through how to run `mysqlfailover` as a daemon.

Review the mysqlfailover documentation[2] for more detail on configuration and options of this utility.

Replication Utility: mysqlrpladmin:
If a user needs to take a master offline for scheduled maintenance, mysqlrpladmin can perform a switchover to a specific slave (called the new master). When performing a switchover, the original master is locked and all slaves are allowed to catch up. Once the slaves have read all events from the original master, the original master is shutdown and control is switched to the new master.

[1] http://www.mysqlhighavailability.com/mysql-replication/standalone-mysql-utilities-now-ga-includes-running-mysqlfailover-as-a-daemon/
[2] http://dev.mysql.com/doc/workbench/en/mysqlfailover.html

There are many operations teams that prefer to take failover decisions themselves, and so `mysqlrpladmin` provides a mechanism for manual failover after an outage to the master has been identified, either by using the health reporting provided by the HA Utilities, or by alerting provided by a tool such as the MySQL Enterprise Monitor (discussed later in this whitepaper).

You can use the utility to perform one of the following actions:

- `elect`: Performs election of the candidate to promote to master in the event of a failover or switchover.
- `failover`: Conducts failover to the candidate. The command will test each candidate slave listed for the pre-requisites defined above. Once a candidate slave is elected, it is made a slave of each of the other slaves in turn, thereby collecting any transactions executed on other slaves but not on the candidate. In this way, the candidate has the superset of all transactions executed on any of the slaves and so, when promoted to master, no replicated transactions are lost.
- `gtid`: Displays the contents of the GTID variables used to report GTIDs in replication. The command also displays universally unique identifiers (UUIDs) for all servers.
- `health`: Displays the health of the master and its slaves.
- `reset`: Executes the `STOP SLAVE` and `RESET SLAVE` commands on all slaves.
- `start`: Executes the `START SLAVE` command on all slaves.
- `stop`: Executes the `STOP SLAVE` command on all slaves.
- `switchover`: Moves the master role to a specific slave.

Review the mysqlrpladmin documentation[1] for more detail on configuration and options of the utility.

Replication Utility: mysqlrplsync
This utility permits you to check replication servers for synchronization. It checks data consistency between a master and slaves or between two slaves. The utility reports missing schema objects as well as missing data.

The utility can be used while replication is still running and is dependent on Global Transaction IDs being enabled. When it detects that there is a data mismatch it will report the table name but not the specific rows.

Review the mysqlrplsync documentation for more detail on configuration and options of the utility.

Replication Utility: mysqlrplms
This utility enables replication from multiple master servers (also called sources) to a single slave. At any point in time, the slave will have a single master but the utility will rotate between the specified masters on a round-robin basis. The utility is dependent on Global Transaction IDs and can also be run as a daemon.

In addition to the documentation, a tutorial video[2] is also available demonstrating how to configure and use the Replication Utilities discussed above.

[1] http://dev.mysql.com/doc/workbench/en/mysqlrpladmin.html
[2] http://dev.mysql.com/tech-resources/articles/mysql-replication-utilities.html

Multi-Threaded Slaves

Replication performance is improved by using multiple execution threads to apply replication events to the slave(s).

As shown in Figure 9, the multi-threaded slave splits processing between worker threads based on schema, allowing updates to be applied in parallel, rather than sequentially. This delivers benefits to those workloads that isolate application data using databases – for example, multi-tenant systems. The splitting between worker threads happens as the replicated events are read from the relay log. The number of worker threads is configured using the `slave-parallel-workers`[1] parameter (where the default value of 0 turns off the functionality).

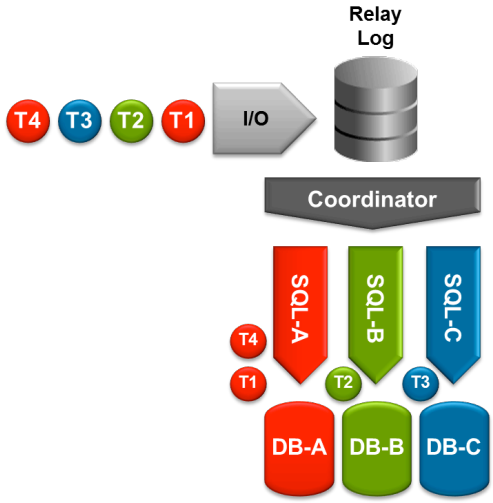

Figure 9 Multi-Threaded Slaves design

To demonstrate performance benefits of multi-threaded Slaves, the MySQL Engineering team ran a benchmark[2] that compared slave throughput when using single and multi-threaded replication. As the results in Figure 10 demonstrate, slave throughput was increased by a factor of 5x based on a configuration with 10 databases/schemas, directly translating to improved read consistency and reduced risk of data loss in the event of an outage of the master.

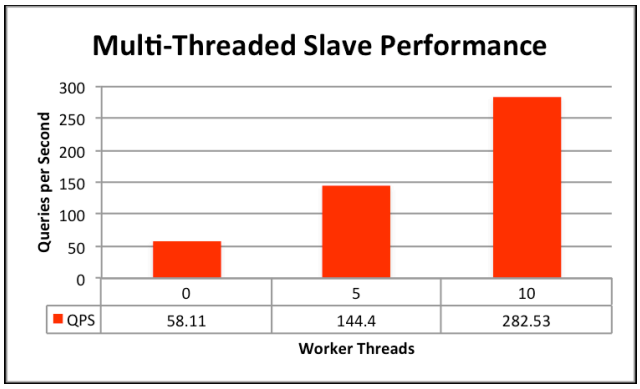

Figure 10 Multi-Treaded Slave benchmark

Slaves are better able to keep up with the master, and so users are much less likely to need to throttle the sustained throughput of writes, just so that the slaves don't indefinitely fall further and further behind (at the moment some users have to reduce the capacity of their systems in order to reduce slave lag).

Binary Log Group Commit

A significant improvement to replication performance has been enabled by the introduction of Binary Log Group Commit which reduces the frequency of writes to disk by flushing groups of binary log writes to the Binlog file, rather than flushing them one at a time.

This enhancement significantly improves the performance of the replication master, as demonstrated by the benchmark results[3] shown in Figure 11.

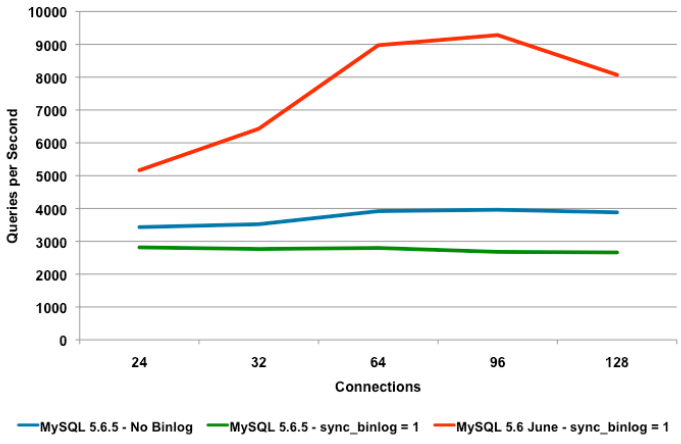

Figure 11 Binary Log Group Commit benchmark

[1] http://dev.mysql.com/doc/refman/5.6/en/replication-options-slave.html#option_mysqld_slave-parallel-workers `
[2] https://blogs.oracle.com/MySQL/entry/benchmarking_mysql_replication_with_multi
[3] http://mysqlmusings.blogspot.se/2012/06/binary-log-group-commit-in-mysql-56.html

The sync_binlog[1] parameter is used to control the frequency of flushes to disk where a (default) value of 0 leaves it up to the operating system to decide when to flush, a value of 1 results in a flush to disk after every transaction and a larger value groups the writes.

Optimized Row Based Replication

This feature is an optimization for Row Base Replication. By only replicating those elements of the row image that have changed following INSERT, UPDATE and DELETE operations, replication throughput for both the master and slave(s) can be increased while binary log disk space, network resource and server memory footprint are all reduced.

This functionality is controlled by the binlog-row-image[2] parameter which can be set to:

- full – the complete before and after images are included. This matches the behavior from previous releases and is the default
- minimal - Log only changed columns, and columns needed to identify rows
- noblob - Log all columns, except for unneeded BLOB and TEXT columns

Crash-Safe Replication

Also known as Transactional Replication, Crash Safe Slaves and Binlog extend the robustness, availability and ease-of-use of MySQL replication by making both the binary log and the slaves crash safe.

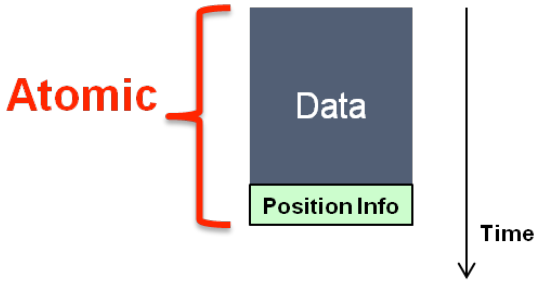

Figure 12 Table data & binlog position transactionally consistent

As shown in Figure 12, the Binary Log positioning information and the table data can now be written as part of the same transaction – ensuring that they are transactionally consistent when using InnoDB. This enables the slave to automatically roll back replication to the last committed event before a crash, and resume replication without administrator intervention. Not only does this reduce operational overhead, it also eliminates the risk of data loss or corruption caused by the failure of a slave.

If a crash to the master causes corruption of the binary log, the server will automatically recover it to a position where it can be read correctly.

This functionality is enabled by setting the master-info-repository[3] and relay-log-info-repository[4] parameters to TRUE.

[1] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#sysvar_sync_binlog
[2] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#sysvar_binlog_row_image
[3] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#option_mysqld_master-info-repository
[4] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#option_mysqld_relay-log-info-repository

Replication Event Checksums

Data integrity within a replication environment assures data is correct, consistent and accessible.

Replication Event Checksums ensure the integrity of data being replicated to a slave by detecting data corruption and returning an error, preventing the slave itself from becoming corrupt.

Checksums are written in the binary and relay logs can be checked at various points, allowing errors to be detected whether they are caused by memory, disk or network failures, or by the database itself. Checksums can be implemented on a per-slave basis, giving maximum flexibility in how and where it is deployed.

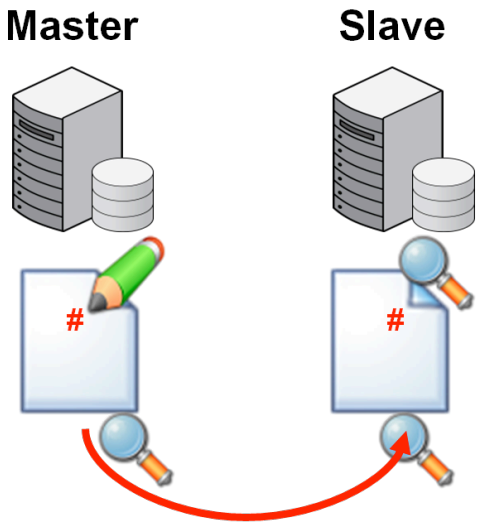

Figure 13 Replication checksums

Configuration parameters can be used to control if the checksums should be generated (and recorded in the binary log) and where they should be verified (as shown in Figure 13: `binlog-checksum`[1], `master-verify-checksum`[2] and `slave-sql-verify-checksum`[3]. If the checksum is included in the binary log of the master then it will always be verified when the replication event is received by the slave.

If a mismatch is detected then replication is halted so that the DBA can fix the issue before resuming replication.

Time-Delayed Replication

Time-Delayed Replication affords protection against operational errors made on the master, for example accidently dropping tables – allowing replication to be suspended before the mistake has propagated to all slaves – in turn, allowing the lost data to be recovered from a slave. It also becomes possible to inspect the state of a database without reloading a backup.

The user can define a time delay for events to be replicated from a master to each slave, defined in increments of one second, up to a maximum of 68 years – this is specified using the `MASTER_DELAY`[4] argument when invoking the `CHANGE MASTER` command.

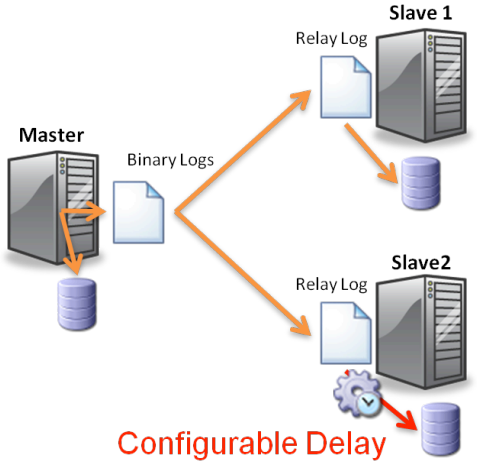

Figure 14 Replication delay on 1 slave

Time-Delayed Replication is implemented at a per-slave level (via holding execution of the SQL_THREAD), so a user could configure multiple slaves to apply replication events immediately, and another slave to apply them only after a delay of (for example) 10 minutes, therefore providing deployment flexibility - as illustrated in Figure 14. It's important to note that the slave still receives replicated events in as close to real-time as possible and that the delay is applied as those events are read from the slave's relay log.

[1] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#option_mysqld_binlog-checksum
[2] http://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#option_mysqld_master-verify-checksum
[3] http://dev.mysql.com/doc/refman/5.6/en/replication-options-slave.html#option_mysqld_slave-sql-verify-checksum
[4] http://dev.mysql.com/doc/refman/5.6/en/replication-delayed.html

4 Replication Use Cases

There are a variety of technical and business reasons why you may choose to replicate your data from one MySQL server to another. In this section we explore the various use cases and application scenarios in which MySQL Replication can be leveraged.

Scale-Out

This is easily the most popular reason that users choose to implement replication. In a scale-out topology the primary objective is spreading the workload across one or more slave servers in order to improve performance.

It is simple for users to rapidly create multiple replicas of their database on commodity hardware to elastically scale-out beyond the capacity constraints of a single instance, enabling them to serve rapidly growing database workloads.

This is the opposite of scale-up, in which the idea is to increase the resources (typically RAM and CPU) on the existing machine. Scaling-up can be thought of as a vertical, "fork-lift" approach to satisfying the need for increased capacity.

In a scale-out architecture, reads and writes are split amongst the master and slave server(s).

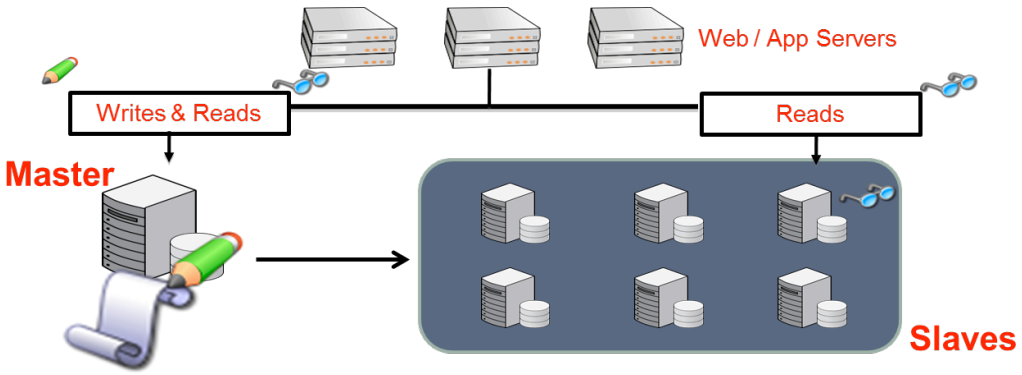

Figure 15 Scale-out with MySQL Replication

Specifically, all writes (`UPDATE`, `INSERT`, `DELETE`) are sent to the master for execution and reads (`SELECT`) are directed to the slave(s). This allows the read workload that was previously being executed on the master server to instead use the resources available on the slave server(s). This allows for a more efficient use of resources as now the workload has been effectively spread across more than one server.

The division of the writes and the reads can be handled at different layers in the system, such as within the application (it maintains connections to all of the servers and

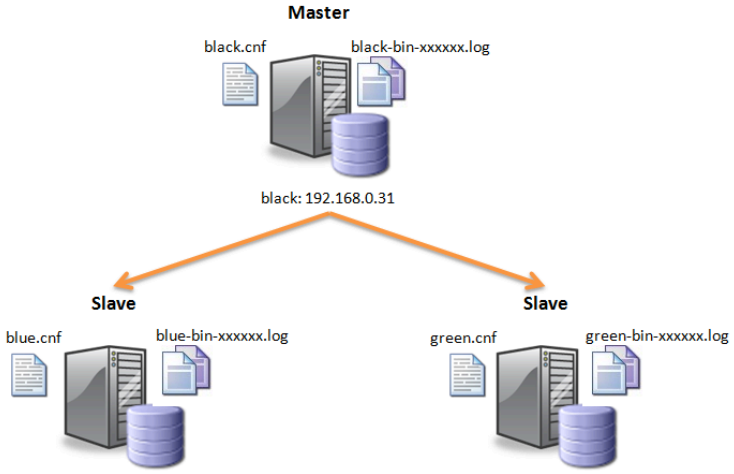

Figure 16 Master with dual slaves

decides when to use a connection to the master versus one of the slaves) or in the database connector.

As an example if using MySQL's JDBC connector (Connector/J) then when connecting to the database you can provide multiple servers in the connect-string, starting with the master and followed by the slaves. If we take the configuration shown in Figure 16 ("black" is the master and "blue" and "green" are slaves) then the application could connect to the "clusterdb" database by using the connect-string `jdbc:mysql:replication://black,blue,green/clusterdb`. Once connected in this way, Connector/J will route transactions to the appropriate server (master/slave) depending on the value of the `ReadOnly` attribute of the connection (queried using `connection.getReadOnly()` and written to using `connection.setReadOnly(bool)`).

As replication is asynchronous, if the application needs a read-only operation to use the latest values from the database with absolute certainty then it should send it to the master rather than the slaves. In the case of Connector/J that would mean running `connection.setReadOnly(false)`.

High Availability

In this scenario, the idea is to replicate changes from a master to a slave server with the goal being to fail-over to the slave server in the event that the master goes offline either due to an error, crash or for maintenance purposes.

As with scaling out, the selection of the correct server can be implemented in various ways. If for example you were using Connector/J with the configuration shown in Figure 17 ("black" is the master and "blue" is the slave) then the application can use `jdbc:mysql://black,blue/clusterdb` as the connect-string; Connector/J would then send all operations to "black" while it is available and then failover to "blue" when it wasn't.

Figure 17 Master-Slave replication configuration

Geographic Replication

Using geographic replication data can be replicated between two geographically dispersed locations, typically over large distances. Asynchronous replication will be the preferred solution in this scenario based on the potential impact of network latency. An example in this case might include replicating data from a central office in New York to a regional office on San Francisco allowing for queries to execute against a local data store. Clearly this approach is also ideal for providing disaster recovery in the event that there is a catastrophe at one of the sites (for example a loss of power or a natural disaster).

Figure 18 Replication for Geographic Redundancy

Backup Database

To avoid any performance degradation or locking that a backup may cause on the master you may choose to instead run the backup on a slave server instead. Note that when using MySQL Cluster or MySQL Enterprise Backup with InnoDB, reads and writes can continue while the database is being backed up.

Analytics

Many business intelligence or analytical queries can be resource intensive and take considerable time to execute. For this use case, slaves can be created for the purpose of servicing these analytical queries. In this configuration, the master suffers no performance impact by the execution of these queries.

This can be especially useful with MySQL Cluster which is ideal for applications that predominantly use Primary Key based reads and writes but can perform more slowly with very complex queries over large data sets[1]. Simply replicate the MySQL Cluster data to a second storage engine (typically InnoDB) and generate your reports there. This can be performed while simultaneously replicating to a remote MySQL Cluster site for geographic redundancy – as shown in Figure 19.

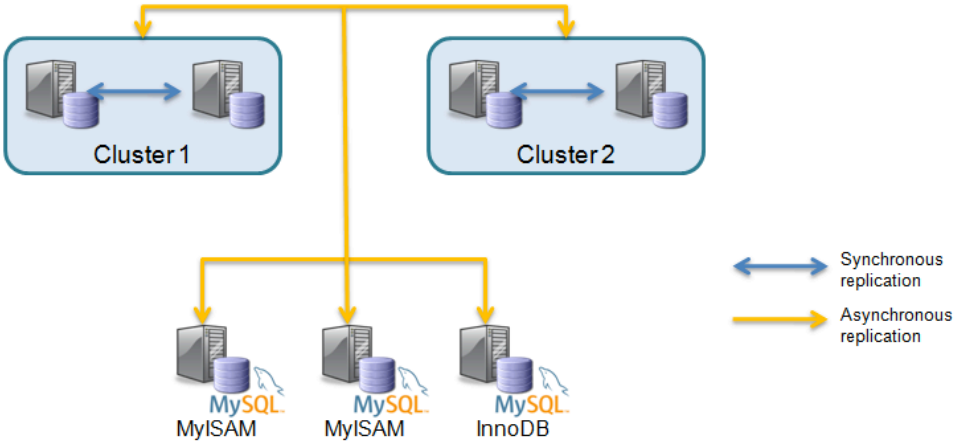

Figure 19 Combining Replication Use Cases

[1] The performance of joins was greatly improved by the addition of Adaptive Query Localisation in MySQL Cluster 7.2

5 Replication Topologies

MySQL supports a variety of replication topologies. Below we discuss some of these topologies, as well as some which are supported with reservations.

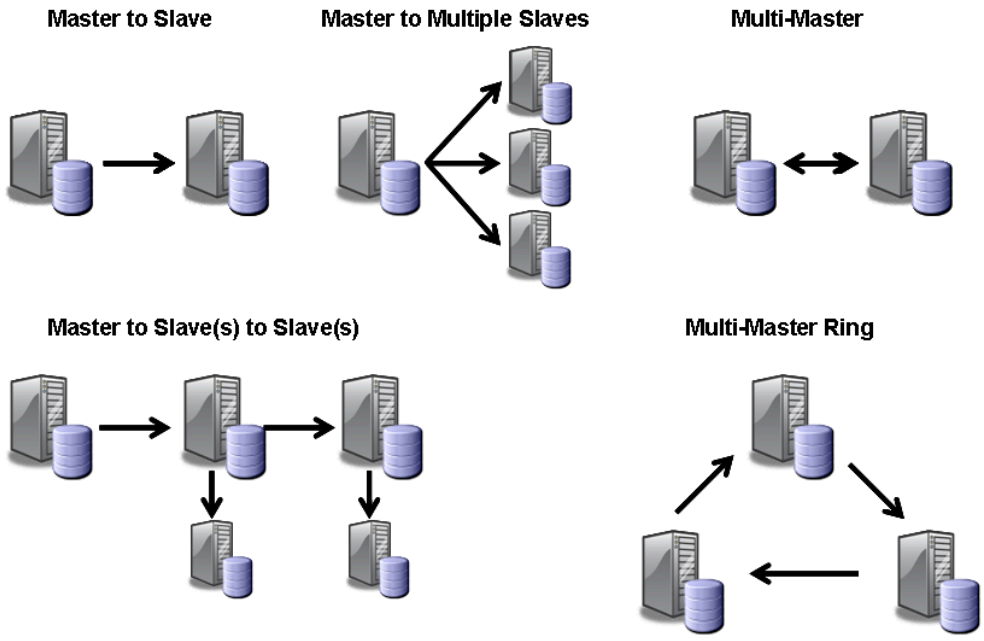

Figure 20 Common MySQL Replication Topologies

Master to Slave

This is the most popular and easiest to configure and administer. In this topology we have two servers, one master and one slave. All writes are performed on the master and reads can be split between the master and the slave.

Master to Multiple Slaves

In this scenario we have multiple slaves attached to a single master. This enables a greater degree of scale-out at the cost of increased administration.

Master to Slave(s) to Slave(s) (Hierarchical or Cascading Replication)

This configuration is an extension of either a master/slave or master/slaves configuration. In this case, an additional slave or slaves are attached to a slave already attached to the root master server. The slave(s) in the middle acts as both a master and slave. In this configuration, all writes are made to the primary master.

Master to Master (Multi-Master)

In a master/master configuration two servers are combined in a pair so that they are both masters and slaves to each other. Although this configuration yields the benefit on being able to write to either system knowing that the change will eventually be replicated, it does increase the degree of complexity in setup, configuration and administration. Additionally, unless you are using MySQL Cluster, there is no conflict detection/resolution and so the application must ensure that it does not update a row on one server while there is still a change to the same row on the other server that has not been replicated yet.

Note that a single slave may only have a single master at any point in time; to simulate multiple masters, the user would need to periodically switch between replication masters.

Multi-Master Ring

It is also possible to arrange a number of MySQL Servers in a ring, allowing even greater levels of scalability and performance (assuming the application can avoid sending conflicting updates to the same row to different servers). MySQL 5.5 introduced a new filtering feature that better handled cases where one server fails while its updates are still being replicated to the rest of the ring.

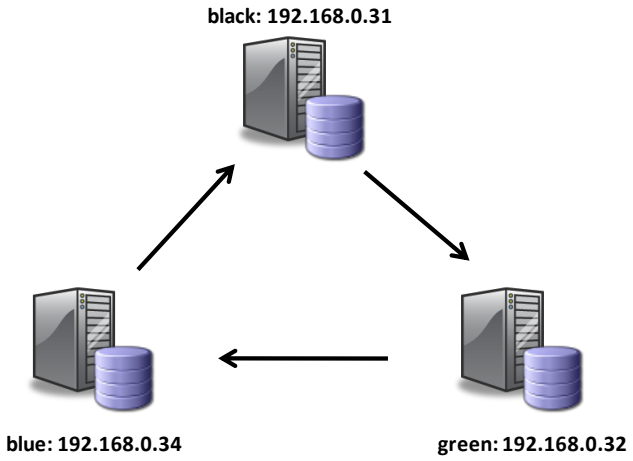

Figure 21 Multi-Master Replication Ring

When using multi-master replication together with auto-increment columns, you should use the `auto_increment_offset` and `auto_increment_increment` parameters on each server to make sure that there are no duplicate values assigned. An example for 3 servers (black, blue & green) is shown in Table 1.

Server	auto_increment_increment	auto_increment_offset	Values
black	3	1	1,4,7…
blue	3	2	2,5,8…
green	3	3	3,6,9,...

Table 1 Avoid Conflicting Auto-Increment Values

Multi-Master to Slave (Multi-Source)

This replication topology is currently *not* supported by MySQL. In a mutli-master configuration a slave essentially “serves two masters”, meaning that the slave receives the changes from more than one master at the same time. To simulate multiple masters, the user would need to periodically switch between replication masters.

MySQL Cluster allows multiple MySQL Servers to write to the same Cluster. Each server can act as a slave in its own right with its own master and so it would be possible to configure this functionality if required for an appropriate application.

6 Replication Internal Workflow

MySQL Replication is implemented by the master logging changes to data (DML: INSERT, UPDATE and DELETE) and changes to object structures (DDL, i.e. ALTER TABLE, etc.), which are then sent and applied to the slave(s) immediately or after a set time interval (when using Time Delayed Replication).

Figure 22 illustrates the implementation of MySQL replication.

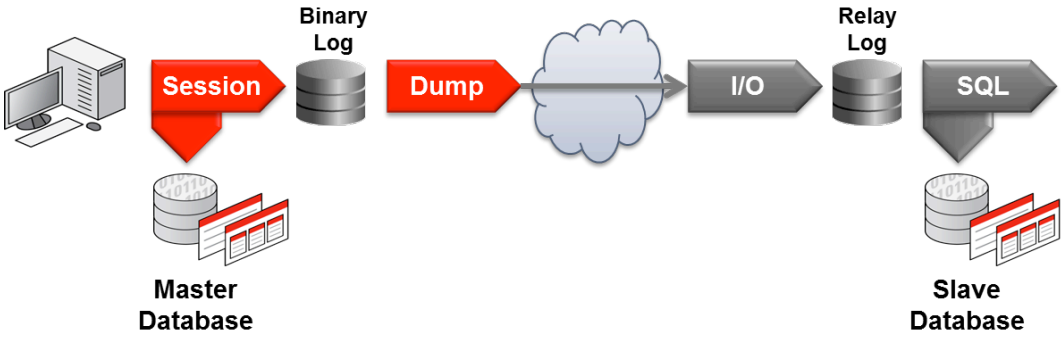

Figure 22 MySQL Replication workflow

With MySQL replication, the master writes updates to its binary log files and maintains an index of those files in order to keep track of the log rotation. The binary log files serve as a record of updates to be sent to slave servers. When a slave connects to its master, it determines the last position it has read in the logs on its last successful update. The slave then receives any updates which have taken place since that time. The slave subsequently blocks and waits for the master to notify it of new updates.

From MySQL 5.6, the user has the option of using Global Transaction Identifiers (GTIDs) where each slave records the GTIDs of all received transactions. When a slave connects it can then perform a handshake with the master to identify all changes that still need to be replicated (see section 3 for more details) and so knowledge of the current binary log position is not essential.

Replication Threads

A number of threads are used to implement the replication of updates from the master to the slave(s); each of those threads are described here. If using MySQL Cluster then an additional thread is involved - see Section 0 for details.

Binlog Dump Thread

The master creates this thread to send the binary log contents to the slave. A master that has multiple slaves “attached” to it creates one binlog dump thread for each currently connected slave, with each slave having its own I/O and SQL threads.

Slave I/O Thread

When a `START SLAVE` statement is issued on the slave, it creates an I/O thread, which connects to the master and asks it to send the updates recorded in its binary logs. The slave I/O thread then reads the updates that the master's binlog dump thread sends, and then copies them locally to the slave’s relay log files.

Slave SQL Thread

The slave creates this thread (on the call to `START SLAVE`) to read the relay logs that were written by the slave I/O thread and executes the updates contained in the relay logs. If using multi-threaded slaves, multiple SQL threads will be created on the slave.

Replication Log Files

During replication the MySQL server creates a number of files that are used to hold the relayed binary log from the master, and records information about the current status and location within the relayed log.

There are three file types used in the process by the slave:

relay log

The relay log on the slave contains events which have been read from the binary log of the master. The events in the binary log are ultimately executed on the slave by the slave's SQL thread.

master.info

The slave's status and current configuration information is located in the master.info file. This file contains the slave's replication connectivity information, including the master's host name, the login credentials being used and the slave's current position on the master's binary log. This file is not needed if `master-info-repository` is set to `table`.

relay-log.info

Status information concerning the execution point within the slave's relay log can be found in the relay-log.info file. This file is not needed if `relay-log-info-repository` is set to `table`.

On the master, there is the binary log and associated index file to track all updates to be replicated.

7 MySQL Fabric

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.

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:

- **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.

MySQL Fabric is beyond the scope of this paper but it is covered in detail in MySQL Fabric - A Guide to Managing MySQL High Availability & Scaling Out[1].

8 Differences when Replicating with MySQL Cluster

MySQL Cluster is a scalable, real-time, ACID-compliant transactional database, combining 99.999% availability with the low TCO of open source. Designed around a distributed, multi-master architecture with no single point of failure, MySQL Cluster scales horizontally on commodity hardware with transparent auto-sharding to serve read and write intensive workloads, accessed via SQL and NoSQL interfaces.

MySQL Cluster can be used as a pluggable storage engine for MySQL but the architecture is fundamentally different from other MySQL storage engines (for example InnoDB and MyISAM) and this impacts how replication works and is implemented. The architecture is shown in Figure 23. In terms of the impact on replication, the key architectural feature is that the data is not stored within a MySQL Server instance; instead the data is distributed over a number of Data Nodes. There is synchronous replication between pairs of Data Nodes within the Cluster to provide High Availability – note that this synchronous replication is not related to the replication described in this white paper. Data can either be accessed directly from the Data Nodes (for example using the native C++ API) or one or more MySQL Servers can be used to provide SQL access. Each MySQL server can read or write any table row and the change is immediately visible to every other MySQL Server.

MySQL replication is typically used with MySQL Cluster to provide geographic redundancy – the internal (synchronous) replication provides High Availability between data nodes co-located within the Cluster and then MySQL (asynchronous) replication to a remote site guards against a catastrophic site failure.

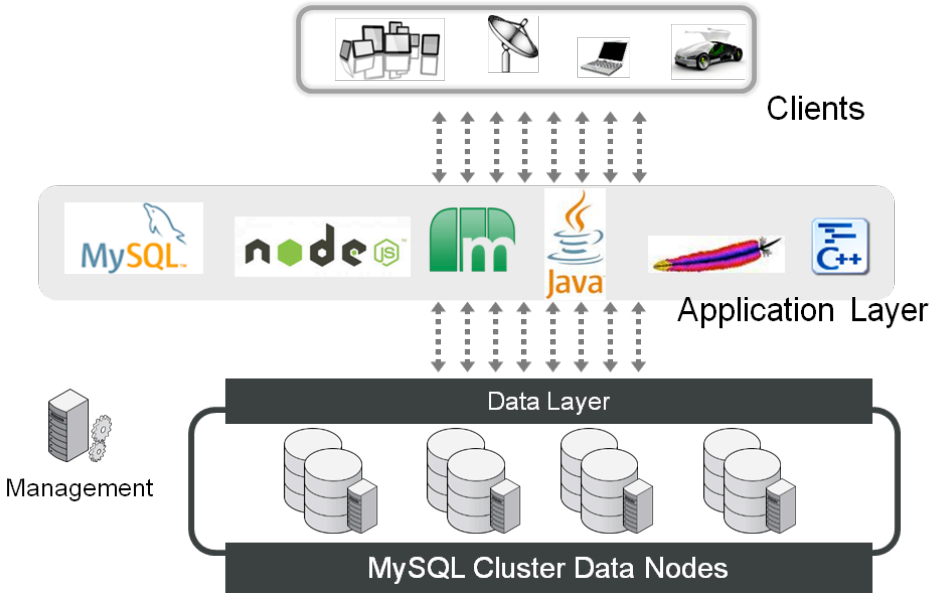

Figure 23 MySQL Cluster Architecture

How does this impact MySQL replication? Changes can be made from any MySQL Server or even directly to the Data Nodes, and so a mechanism has been implemented whereby all changes get

[1] http://www.mysql.com/why-mysql/white-papers/mysql-fabric-product-guide/

concentrated into the binary logs of one or more nominated MySQL Servers that then act as the MySQL replication master(s). This is performed by the NDB Binlog Injector thread. This thread ensures that all changes (regardless of where in the Cluster they get applied) are written to the binary log in the correct order.

As shown in Figure 24 this provides the opportunity for a more robust replication system architecture. Two MySQL Servers are shown where the binary log for each contains the exact same changes and either of those servers can be used as the master to replicate these changes to one or more slave MySQL Cluster deployments and/or to a MySQL Server which stores the data using the InnoDB or MyISAM storage engines.

While the binary logs on each master should contain the same changes, they are independent from each other. To facilitate slave failover from one master to another, the Cluster-wide 'Epoch' is used to represent the state of a Cluster at a single point in time. These advanced failover techniques are beyond the scope of this white paper but further information can be found in the MySQL Cluster Reference Guide[1].

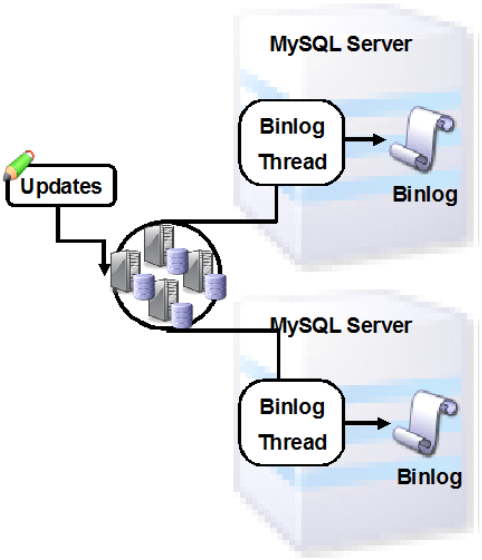

Figure 24 Multiple Masters within a Cluster

MySQL Cluster supports running MySQL replication in a multi-master active-active configuration or even with replication rings. Additionally, MySQL Cluster can provide conflict detection or resolution to cover cases where conflicting changes get written to the same rows on these different masters. This is beyond the scope of this white paper but details can be found in the MySQL Cluster Reference Guide[2].

MySQL Cluster releases run to a different schedule to the main MySQL releases and so the latest MySQL replication features will not always be available in the latest MySQL Cluster release. For example, at the time of writing the latest Generally Available release is MySQL Cluster 7.3 which uses a modified version of MySQL 5.6 for the MySQL Servers

When replicating from MySQL Cluster, row-based-replication (rather than statement-based-replication) is always used.

9 Replication Monitoring with MySQL Enterprise Monitor

The MySQL Enterprise Monitor with Query Analyzer is a distributed web application that you deploy on premise or in the cloud. The Monitor continually monitors all of your MySQL servers and proactively alerts you to potential problems and tuning opportunities before they become costly outages. It also provides you with MySQL expert advice on the issues it has found so you know where to spend your time in optimizing your MySQL systems.

[1] http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-replication-failover.htm
[2] http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-replication-multi-master.html

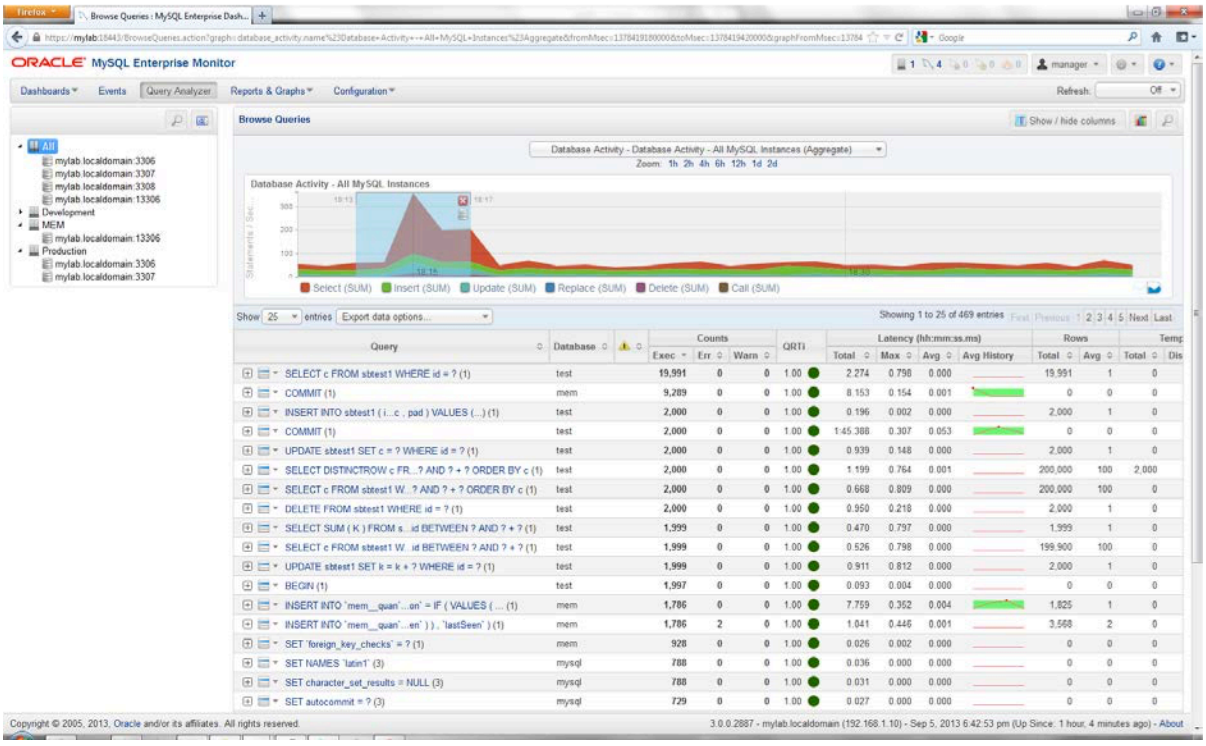

Figure 25 MySQL Enterprise Monitor

MySQL Enterprise Monitor makes it easier to scale-out and achieve high availability using the MySQL Replication Monitor, providing auto-detection, grouping, documenting and monitoring of all master/slave hierarchical relationships. Changes and additions to existing replication topologies are also auto-detected and displayed, providing DBAs with instant visibility into newly implemented updates.

As the Replication Advisor identifies a problem and sends out an alert, the DBA can use the alert content along with the Replication Monitor to drill into the status of the affected master and/or slave. Using the Replication Monitor and the expert advice from the Replication Advisor they can review the current master/slave status and view metrics (such as Slave I/O, Slave SQL thread, seconds behind master, last error, etc.) that are relevant to diagnosing and correcting any problems.

The Replication Monitor is designed and implemented to save DevOps time writing and maintaining scripts that collect, consolidate and monitor similar MySQL Replication status and diagnostic data.

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

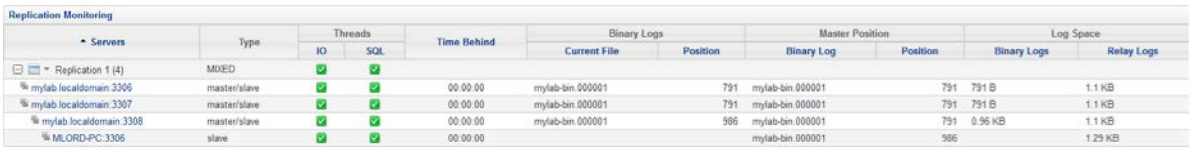

Figure 26 Replication Statistics in MySQL Enterprise Monitor

MySQL Enterprise Monitor is available as part of certain commercial MySQL editions as described at http://www.mysql.com/products/.

10 Conclusion

MySQL Replication has been proven as an effective solution for the extreme scaling of database-driven applications in some of the most demanding environments in web, mobile, social and cloud applications

This whitepaper has discussed the business and technical advantages of deploying MySQL Replication, as well as provide practical step-by-step guides to getting started.

As evidenced by the MySQL 5.6 release, replication is an area of active development, with continual improvements in areas such as data integrity, performance and deployment flexibility.

11 Resources

MySQL 5.6 Replication – A Tutorial
http://www.mysql.com/why-mysql/white-papers/mysql-replication-introduction

MySQL 5.6 Download:
http://dev.mysql.com/downloads/mysql/

MySQL Replication user guide:
http://dev.mysql.com/doc/refman/5.6/en/replication.html

MySQL Cluster Replication Documentation:
http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-replication.html

MySQL at Ticketmaster (heavy user of MySQL Replication):
http://www.mysql.com/customers/view/?id=684