MySQL Replication

Enhancing Scalability and Availability with MySQL 5.5

A MySQL® Technical White Paper by Oracle

October 2010

Table of Contents

1 Introduction

MySQL Replication has been widely deployed by some of the leading properties on the web and in the enterprise to deliver extreme levels of database scalability. It is simple for users to rapidly create multiple replicas of their database to elastically scale-out beyond the capacity constraints of a single instance, enabling them to serve rapidly growing database workloads.

Replication is also used as the foundation for delivering highly available database services, providing a means of mirroring data across multiple hosts to withstand failures of individual systems. In addition, many users replicate data to systems that are dedicated to back-up or data analysis functions, allowing for a more efficient use of resources by offloading such tasks from their production servers.

With the release of MySQL 5.5, a number of enhancements have been made to MySQL Replication, delivering higher levels of data integrity, performance and application flexibility. We will discuss these enhancements in the whitepaper. (Note that at the time of writing MySQL 5.5 is available as a Release Candidate only).

We will also explore the business and technical advantages of deploying MySQL Replication; describe the fundamental enabling technology behind replication and provide a simple step-by-step guide on how to install and configure a master/slave topology, as well as handle failover events.

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

2 Replication Fundamentals

For the purposes of this paper we define “replication” as the duplication of data to one or more locations. 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).

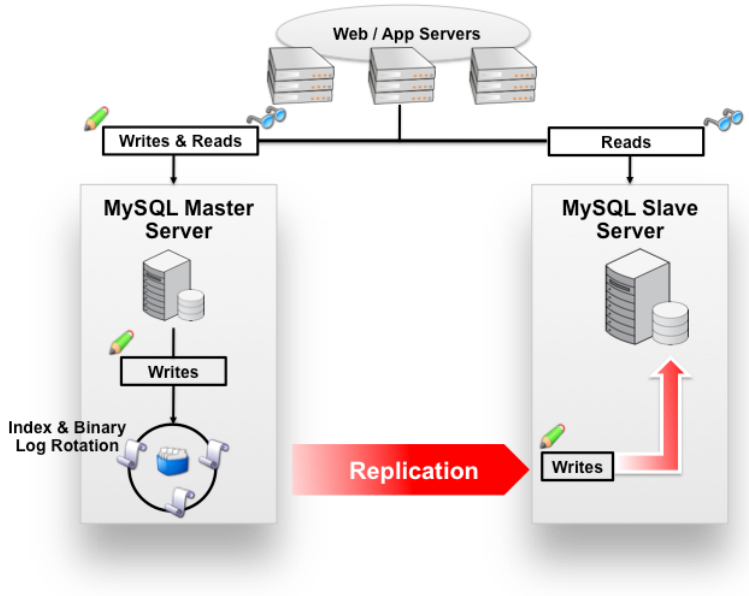

Figure 1 MySQL Replication Supports HA and Read Scalability "Out of the Box"

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].

Asynchronous Replication

MySQL natively supports one-way, asynchronous replication. Asynchronous replication means that data is copied from one machine to another, with a resultant delay in the actual copying of the data changes – most importantly this means that the data may not have been copied/applied to the slave at the point where the transaction commit is confirmed to the application. Often this delay is determined by networking bandwidth, resource availability and system load. However, with the correct components and tuning, replication itself can appear to be almost instantaneous to most applications.

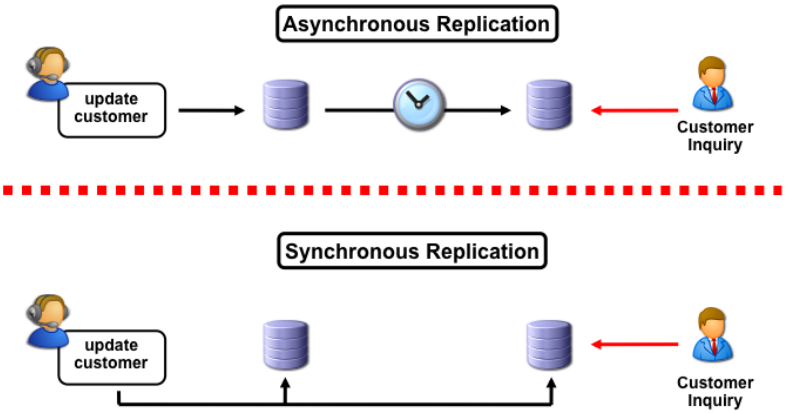

Figure 2 Asynchronous vs. Synchronous Replication

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

Synchronous Replication

Synchronous replication can be defined as data that is committed to one or more machines at the same time, usually via what is commonly known as a "two-phase commit". Although this does give you consistency across multiple systems, it comes with the performance penalty of additional messaging.

MySQL with the InnoDB or MyISAM storage engines does not natively support synchronous replication, however there are technologies like Distributed Replicated Block Device or DRBD, which provide synchronous replication of the underlying file system, allowing a second MySQL Server to take over (using that second copy) in the event of the loss of the primary copy/server. For more information, see: http://www.drbd.org/

If using MySQL Cluster, data is replicated synchronously between data nodes within the Cluster (site) and then asynchronously between geographically separated Clusters.

Semisynchronous Replication

New in MySQL 5.5 (at the time of writing MySQL 5.5 is at Release Candidate status) is a feature called semisynchronous replication. This means that if semisynchronous replication is enabled on the master and there is at least one semisynchronous slave configured, a thread that performs a transaction commit on the master blocks after the commit is applied locally and waits until at least one semisynchronous slave returns an acknowledgement to the master that it has received all the events for the transaction, or until a timeout occurs. In the event of a timeout, the master still commits the transaction but reverts to asynchronous mode.

Using asynchronous replication, if the master crashes, it is not immediately known whether the transactions that the master has committed might have been replicated to the slave. Consequently, failover from master to slave may result in failover to a server that is missing transactions relative to the master. Semisynchronous replication minimizes the potential of "orphan" transactions on the master server since all committed transactions were received by the slave – in other words, for any client thread, the changes being made to the master as part of its one "in-flight" transaction would be lost (and so the client should retry) but any changes from transactions where the commit has been acknowledged will survive.

Semisynchronous replication does have some performance impact because commits are slower due to the need to wait for slaves. This is the trade-off for increased data integrity. The amount of slowdown is at least the TCP/IP roundtrip time to send the commit to the slave, have the slave record it in its relay log and wait for the acknowledgment of receipt by the slave. This means that semisynchronous replication works most effectively for servers that are physically co-located, communicating over fast networks.

Semisynchronous replication is not currently available for tables using the MySQL Cluster storage engine.

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). Statement-based replication has been part of the MySQL server since version 3.23.

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

Introduced in MySQL 5.1, 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 usually generates 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.

Prior to MySQL 5.5, the column types have to be identical on the master and the slave when using row-based replication. As of 5.5, you may configure whether to allow type promotion and/or demotion or continue to enforce the strict type checking,

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

Mixed-Format Replication

As of MySQL 5.1.8, 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/5.1/en/binary-log-mixed.html

3 Summary of Replication Changes in MySQL 5.5

MySQL 5.5 introduces a number of enhancements to replication; a summary of these is provided in this section and some of these are described in more detail elsewhere in this document:

- **Semisynchronous replication**: Improved resilience by having master wait for slave to persist events.
- **Slave fsync tuning & Automatic relay log recovery**: Option to dictate when relay logs are written to disk rather than relying on default operating system behavior; set `sync_relay_log=1` to ensure that no more than 1 statement or transaction is missing from the relay log after a crash. The slave can now recover from corrupted relay logs by requesting corrupt entries to be resent from the master. Three new options are introduced (sync-master-info, sync-relay-log and sync-relay-log-info) and their use is described in the MySQL

Reference Manual at http://dev.mysql.com/doc/refman/5.5/en/replication-options-slave.html#sysvar_sync_master_info

- **Replication Heartbeat**: Automatically checks the status of the connection between the master and the slave(s), allowing a more precise failure detection mechanism. Can detect loss of connection within milliseconds (configurable). Avoid unnecessary relay log rotation when the master is idle.
- **Per server replication filtering**: When a server is removed from a replication ring, a surviving server can be selected to remove its outstanding replication messages once they've been applied by all servers.
- **Precise Slave Type Conversions**: Allows different types to be used on the master and slave, with automatic type promotion and demotion when using row-based replication (already possible with statement-based replication)
- **Individual Log Flushing**: Selectively flush server logs when using 'FLUSH LOGS' for greater control
- **Safe logging of mixed transactions**: Replicate transactions containing both InnoDB and MyISAM changes

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 an effort to improve performance.

It is simple for users to rapidly create multiple replicas of their database 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).

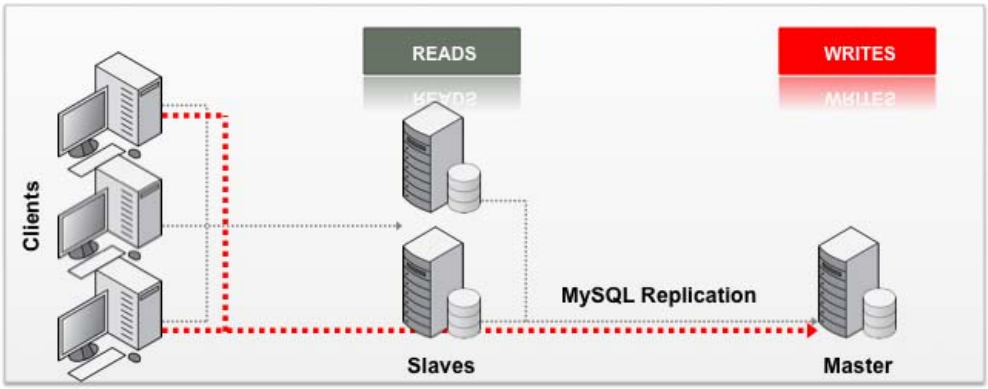

Figure 3 Scale-out with MySQL Replication

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 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 10 ("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 9 ("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.

Geographic Replication

With geographic replication the aim is to replicate data 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).

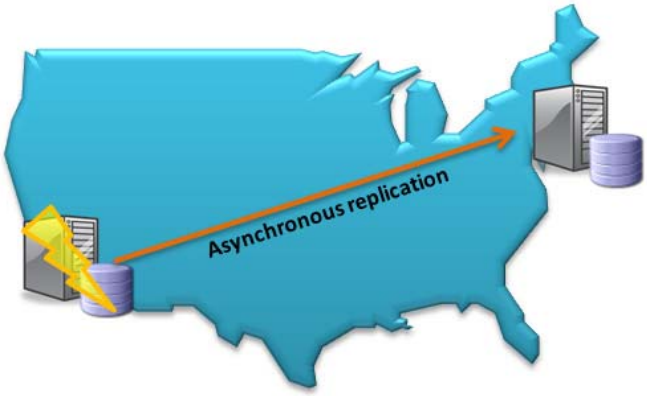

Figure 4 Replication for Geographic Redundancy

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, 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 slowly with very complex queries over large data sets. Simply replicate the MySQL Cluster data to a second storage engine (typically MyISAM or 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 5 below.

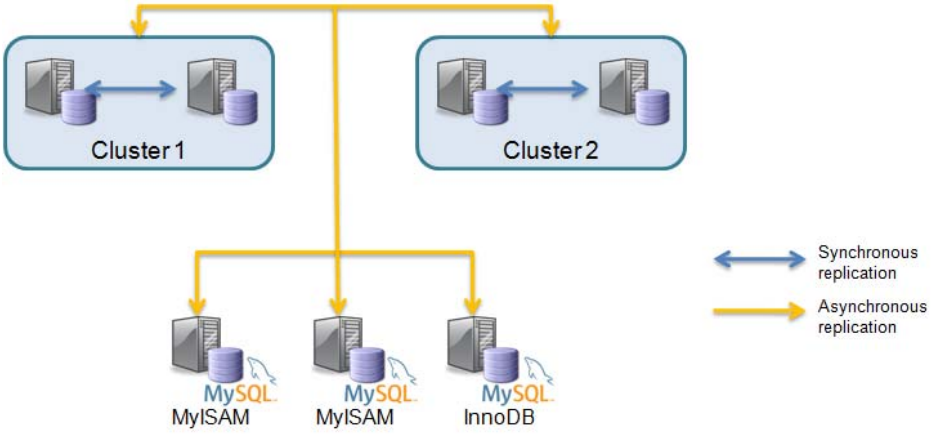

Figure 5 Combining Replication Use Cases

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.

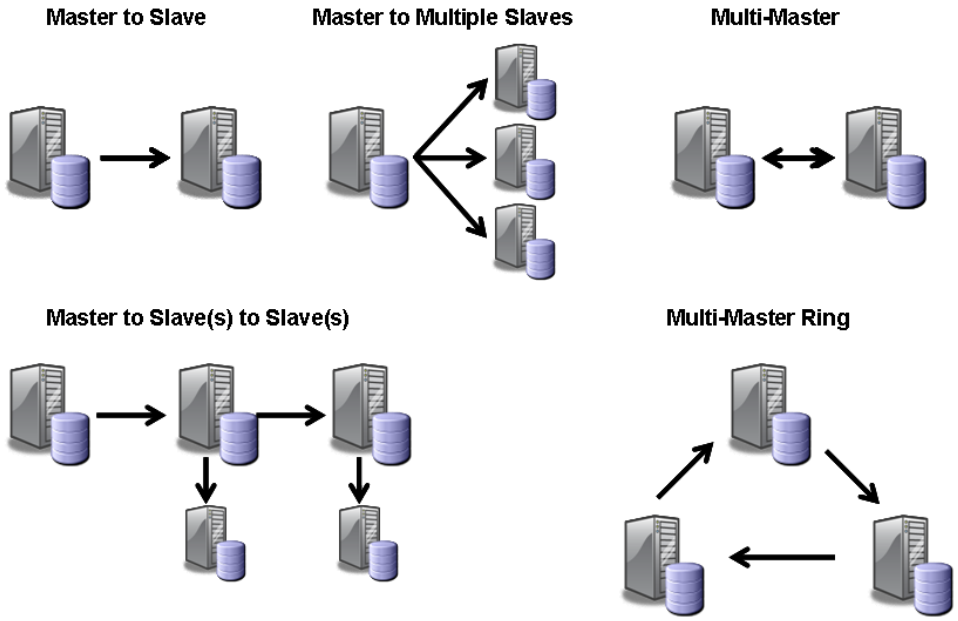

Figure 6 Common MySQL Replication Topologies

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)

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 original master server. In effect, 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 significantly 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.

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 introduces a new filtering feature that better handles cases where one server fails while its updates are still being replicated to the rest of the ring.

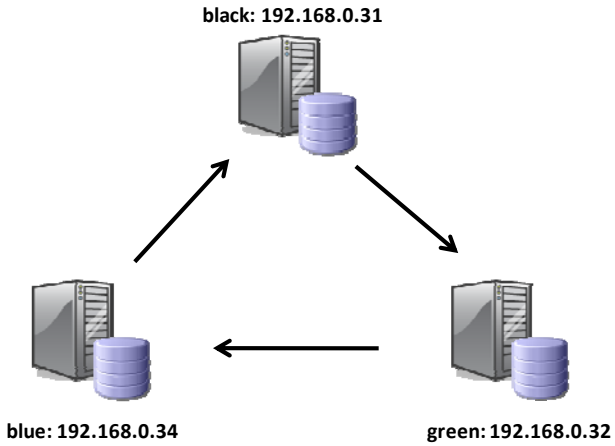

Figure 7 Mulit-Master Replication Ring

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 below.

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 replicates in the changes from more than one master.

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

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. A basic illustration of these concepts is depicted in Figure 8 below.

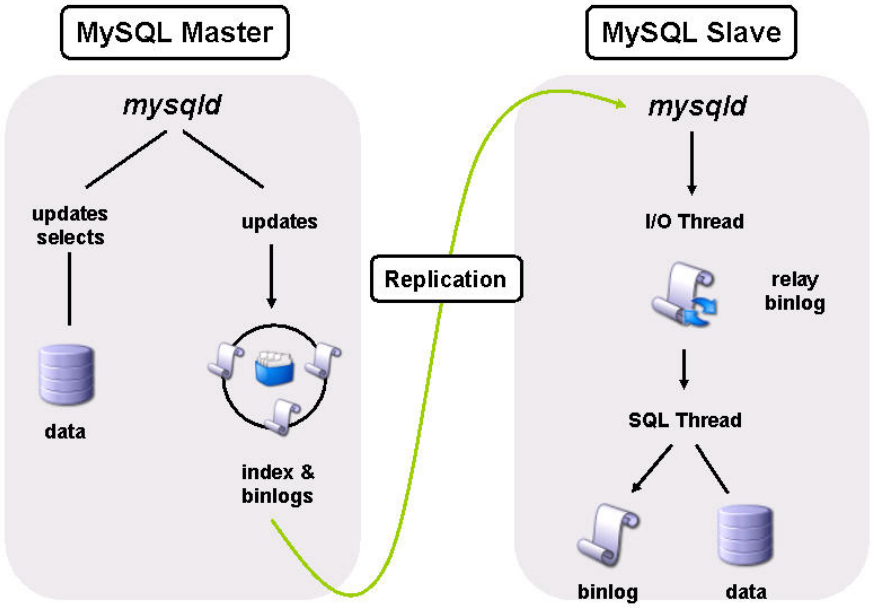

Figure 8 How MySQL Replication is Implemented

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 11 for details.

Binlog Dump Thread

The master creates this thread to send the binary log contents to the slave. The binlog dump thread acquires a lock on the master's binary log for reading each event that is to be sent to the slave. As soon as the event has been read, the lock is released, even before the event is sent 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 on the slave as relay logs in the slave's data directory.

Slave SQL Thread

The slave creates this thread to read the relay logs that were written by the slave I/O thread and executes the updates contained in the relay logs.

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.

relay-log.info

Status information concerning the execution point within the slave's relay log can be found in the relay-log.info file

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

7 Configuring MySQL Replication

In this section we describe one method for setting up MySQL replication. Although, this procedure is written in terms of setting up a single slave, it can be repeated to set up multiple slaves. For the purposes of this guide we are also assuming that you have successfully downloaded and installed at least two MySQL servers.

A Note Concerning the Example

For the purposes of this paper we have configured two MySQL 5.5 Servers which will serve the following roles:

Master
Host Name: black
IP: 192.168.0.31

Slave
Host Name: blue
IP: 192.168.0.34

Figure 9 Configuration used for example procedure

We will assume that you will be using the InnoDB storage engine. Refer to Chapter 11 to understand where things would be different for the MySQL Cluster (NDB) storage engine due to its extra capabilities.

To cover the more complex use-case, the example assumes that the MySQL Server which is to be used as a master is already in use and already contains data that needs to be replicated to the new slave. If starting with an empty database then steps 3 and 4 can be skipped.

Compatibility

If you are attempting to set up replication between two MySQL servers which have already been installed, ensure that the versions of MySQL installed on the master and slave are compatible. For a current list of compatible versions see: http://dev.mysql.com/doc/refman/5.5/en/replication-compatibility.html

Step 1: Configure the Master & Slave cnf Files

The first step in setting up replication involves editing the "my.cnf" file on the servers that will serve as the master and slave. A default is provided with the MySQL installation but in case there is already a production MySQL database running on these servers, we provide local configuration files "master.cnf" and "slave.cnf" that will be used when starting up the MySQL servers.

At a minimum we'll want to add two options to the [mysqld] section of the master.cnf file:

- log-bin: in this example we choose black-bin.log. The server cannot act as a replication master unless binary logging is enabled.
- server-id: in this example we choose 1. The server_id variable must be a positive integer value from 1 to 2^{32}

master.cnf:

```
[mysqld]
server-id=1
log-bin=black-bin.log
datadir=/home/billy/mysql/master/data
innodb_flush_log_at_trx_commit=1
sync_binlog=1
```

Note: For the greatest possible durability and consistency in a replication setup using InnoDB with transactions, you should also specify the innodb_flush_log_at_trx_commit=1, sync_binlog=1 options.

Next, you'll need to add the server-id option to the [mysqld] section of the slave's slave.cnf file. The server-id value, like the master_id value, must be a positive integer from 1 to 2^{32}, It is also necessary that the ID of the slave be different from the ID of the master. If you are setting up multiple slaves, each one must have a unique server-id value that differs from that of the master and from each of the other slaves. Think of server-id values as something similar to IP addresses: These IDs uniquely identify each server instance in the community of replication servers.

You can also define filenames to be used by the slave by setting relay-log-index and relay-log.

slave.cnf:

```
[mysqld]
server-id=2
relay-log-index=slave-relay-bin.index
relay-log=slave-relay-bin
datadir=/home/billy/mysql/slave/data
```

Now, restart the MySQL servers using the service manager or directly from the command line if not running as a service:

```
[billy@black ~]$ mysqladmin -u root shutdown # only needed if MySQL already running
[billy@black ~]$ mysqld --defaults-file=/home/billy/mysql/master/master.cnf &

[billy@blue ~]$ mysqladmin -u root shutdown # only needed if MySQL already running
[billy@blue ~]$ mysqld --defaults-file=/home/billy/mysql/slave/slave.cnf&
```

Note: If the slave has been replicating previously, start the slave server with the --skip-slave-start option so that it doesn't immediately try to connect to its master. You also may want to start the slave server with the --log-warnings option to get more messages in the error log about problems (for example, network or connection problems). The option is enabled by default, but aborted connections are not logged to the error log unless the option value is greater than 1.

Step 2: Create Replication User

The next step in setting up replication is creating an account on the master that will be used exclusively for replication. We strongly advise that a dedicated replication user be created for better security so you won't need to grant any additional privileges besides replication permissions. Create an account on the master server that the slave server can use to connect. As mentioned, this account must be given the REPLICATION SLAVE privilege. You can either execute the GRANT in the MySQL client or in your favorite MySQL administration tool:

```
[billy@black ~]$ mysql -u root --prompt='master> '
master> CREATE USER repl_user@192.168.0.34;
master> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.34 IDENTIFIED BY 'billy';
```

Step 3: Lock the Master, Note Binlog Position and Backup Master Database

On the master flush all the tables and block write statements by executing a FLUSH TABLES WITH READ LOCK statement:

```
master> FLUSH TABLES WITH READ LOCK;
```

While the read lock placed by FLUSH TABLES WITH READ LOCK is in effect, read the value of the current binary log name and offset on the master with the following statement:

```
master> SHOW MASTER STATUS;
```

```
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| black-bin.000001 |     1790 |              |                  |
+------------------+----------+--------------+------------------+
```

The “File” column shows the name of the log and the “Position” column shows the offset within the file. In this example, the binary log file is “black-bin.000001” and the Position is “1790”. You’ll want to write down these values as you will need them later on when you are setting up the slave. They represent the replication coordinates at which the slave should begin processing new updates from the master.

Note: If the master has been running previously without binary logging enabled, the log name and position values displayed by SHOW MASTER STATUS will be empty. If this is the case, the values that you need to use later when specifying the slave's log file and position are the empty string (‘’) and 4.

Next, leaving the MySQL client window open you used to execute the FLUSH TABLES WITH READ LOCK, you’ll need to dump out the contents of the databases on the master you will want to replicate on the slave. In our example we will dump the contents of the “clusterdb” database.

Note: You may not want to replicate the “mysql” system database if the slave server has a different set of user accounts from those that exist on the master. In this case, you should exclude it from dump process.

You can execute mysqldump either from the command line or in a graphically driven manner with MySQL Workbench.

```
[billy@black ~]$ mysqldump -u root clusterdb > /home/billy/mysql/master/clusterdb.sql
```

You can re-enable write activity on the master with the following statement:

```
master> UNLOCK TABLES;
```

Step 4: Load the Dump File on the Slave

Next, you’ll want to load the mysqldump file from the master onto the slave (of course, this requires copying the clusterdb.sql file from “black” to “blue”). This can be done at the command line or in a graphically driven manner with MySQL Workbench. As the “clusterdb” database does not yet exist on the slave, it must be created before loading in the tables and data:

```
[billy@blue ~]$ mysql -u root –e ‘create database clusterdb;’
[billy@blue ~]$ mysql -u root clusterdb < /home/billy/mysql/slave/clusterdb.sql
```

Step 5: Initialize Replication

We are now ready to initialize replication on the slave. Execute the following statement on the slave:

```
slave> SLAVE STOP;
```

Next, you’ll need to issue a CHANGE MASTER statement:

```
slave> CHANGE MASTER TO MASTER_HOST='192.168.0.31',
    -> MASTER_USER='repl_user',
    -> MASTER_PASSWORD='billy',
    -> MASTER_LOG_FILE='black-bin.000001',
    -> MASTER_LOG_POS=1790;
```

Where:

- **MASTER_HOST**: the IP or hostname of the master server, in this example black or 192.168.0.31
- **MASTER_USER:** this is the user we granted the REPLICATION SLAVE privilege to in Step 2, in this example, “repl_user”
- **MASTER_PASSWORD:** this is the password we assigned to ”repl_user” in Step 2
- **MASTER_LOG_FILE:** is the file name we determined in Step 3
- **MASTER_LOG_POS:** is the position we determined in Step 3

Finally, start replication on the slave:

```
slave> START SLAVE;
```

Step 6: Basic Checks

Now we are ready to perform a basic check to ensure that replication is indeed working. In this example we insert a row of data into the “simples” table on the master server and then verify that these new rows materialize on the slave server:

```
master> INSERT INTO clusterdb.simples VALUES (999);

slave> SELECT * FROM clusterdb.simples;
+-----+
| id  |
+-----+
|   1 |
|   2 |
|   3 |
| 999 |
+-----+
```

8 Migration to Semisynchronous Replication

This section assumes that you have asynchronous replication up and running but then want to migrate to semisynchronous replication – in other words, this walkthrough builds on section 7. You can of course start using semisynchronous replication directly if not already using asynchronous replication.

Note: Semisynchronous replication is only available in MySQL 5.5 and later.

Step 1: Install the Plugins on the Master and Slave

Install the appropriate plugins for each MySQL Server:

```
master> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';

slave> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
```

Step 2: Activate Semisynchronous Replication

For Semisynchronous replication to be active, it must be activated on the master and at least one of the slaves (in our case we have just one slave). This can be done by setting rpl_semi_sync_master_enabled to “ON” in master.cnf and rpl_semi_sync_slave_enabled to “ON” in slave.cnf; alternatively it can be done from the MySQL command line:

```
master> SET GLOBAL rpl_semi_sync_master_enabled = on;
```

```
slave> SET GLOBAL rpl_semi_sync_slave_enabled = on;
slave> STOP SLAVE IO_THREAD; START SLAVE IO_THREAD;
```

If replication was not already running then there would be no need to stop and then restart the slave IO thread, instead you would just issue START SLAVE.

Check that semisynchronous replication is running from the master's perspective and that at least one slave is connected in semisynchronous mode:

```
master> SHOW STATUS LIKE 'Rpl_semi_sync_master_status';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| Rpl_semi_sync_master_status | ON    |
+-----------------------------+-------+
master> SHOW STATUS LIKE 'Rpl_semi_sync_master_clients';
+-----------------------------+-------+
| Variable_name                | Value |
+-----------------------------+-------+
| Rpl_semi_sync_master_clients | 1     |
+-----------------------------+-------+
```

Step 3: Confirm that Replication is running in Semisynchronous Mode

We add a new row on the master and then check the status variables to ensure that it was replicated semisynchronously (in other words the slave acknowledged receipt of the change before the master timed-out and acknowledged the insert to the client asynchronously).

```
master> INSERT INTO clusterdb.simples VALUES (100);
master> SHOW STATUS LIKE 'Rpl_semi_sync_master_yes_tx';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| Rpl_semi_sync_master_yes_tx | 1     |
+-----------------------------+-------+

slave> SELECT * FROM clusterdb.simples;
+-----+
| id  |
+-----+
|   1 |
|   2 |
|   3 |
| 100 |
| 999 |
+-----+
```

9 Replication Administration and Troubleshooting

In this section we will cover how to perform a few basic administrative and troubleshooting tasks on MySQL replication.

Checking Replication Status

The most common task when managing a replication process is to ensure that replication is taking place and that there has not been any errors between the slave and the master.

The primary statement to use for this task is SHOW SLAVE STATUS which must be executed on the slave. For example:

```
slave> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
```

```
               Slave_IO_State:
                  Master_Host: 192.168.0.31
                  Master_User: repl_user
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: black-bin.000003
          Read_Master_Log_Pos: 790
               Relay_Log_File: slave-relay-bin.000013
                Relay_Log_Pos: 936
        Relay_Master_Log_File: black-bin.000003
             Slave_IO_Running: No
            Slave_SQL_Running: No
              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: 790
              Relay_Log_Space: 1238
              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: NULL
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: 1
```

Below is some guidance on how to interpret the results of the output:

- **Slave_IO_State -** indicates the current status of the slave
- **Slave_IO_Running -** shows whether the IO thread for reading the master's binary log is running
- **Slave_SQL_Running -** shows whether the SQL thread for executing events in the relay log is running
- **Last_Error -** shows the last error registered when processing the relay log. Ideally this should be blank, indicating no errors
- **Seconds_Behind_Master -** shows the number of seconds that the slave SQL thread is behind processing the master binary log. A high number (or an increasing one) can indicate that the slave is unable to cope with the large number of statements from the master. A value of 0 for Seconds_Behind_Master can *usually* be interpreted as meaning that the slave has caught up with the master, but there are some cases where this is not strictly true. For example, this can occur if the network connection between master and slave is broken but the slave I/O thread has not yet noticed this — that is, slave_net_timeout has not yet elapsed. If replication is stopped (as in the example above) then this will show the value 'NULL'.

On the master, you can check the status of slaves by examining the list of running processes on the server.

```
master> SHOW PROCESSLIST \G
```

Because it is the slave that drives the core of the replication process, very little information is available in this report.

Suspending Replication

You can stop and start the replication of statements on the slave using the STOP SLAVE and START SLAVE statements.

To stop execution of the binary log from the slave, use STOP SLAVE:

```
slave> STOP SLAVE;
```

When execution is stopped, the slave does not read the binary log from the master via the IO_THREAD and stops processing events from the relay log that have not yet been executed via the SQL_THREAD. You can pause either the IO or SQL threads individually by specifying the thread type. For example:

```
slave> STOP SLAVE IO_THREAD;
```

Stopping the SQL thread can be useful if you want to perform a backup or other task on a slave that only processes events from the master. The IO thread will continue to be read from the master, but the changes will not be applied yet, which will make it easier for the slave to catch up when you start slave operations again - this may be important if you need to failover and make this slave the new master.

Stopping the IO thread will allow the statements in the relay log to be executed up until the point where the relay log ceased to receive new events. Using this option can be useful when you want to allow the slave to catch up with events from the master, when you want to perform administration on the slave but also ensure you have the latest updates to a specific point. This method can also be used to pause execution on the slave while you conduct administration on the master while ensuring that there is not a massive backlog of events to be executed when replication is started again.

To start execution again, use the START SLAVE statement:

```
slave> START SLAVE;
```

If necessary, you can start either the IO_THREAD or SQL_THREAD threads individually.

Viewing Binary Logs

As mentioned previously, the server's binary log consists of files containing “events” that describe modifications to database contents. The server writes these files in binary format. To display their contents in text format, use the `mysqlbinlog` utility. You can also use `mysqlbinlog` to display the contents of relay log files written by a slave server in a replication setup because relay logs have the same format as binary logs.

For more information about the `mysqlbinlog` utility, please see:
http://dev.mysql.com/doc/refman/5.1/en/mysqlbinlog.html

10 Failover and Recovery

There will be times when the replication master fails or needs to be taken down for maintenance; this section goes through the required steps.

Step 1: Prerequisites

In Sections 7 and 8 we configured replication for two MySQL Servers in a master-slave configuration. In this section, we extend that in 2 ways:

1. Allow relationships to change – a server that was acting as a slave can become the master (e.g. when we need to shut down the original master for maintenance) and the original master can subsequently become a slave
2. We start with a more complex configuration as shown in Figure 10 – a single master (black) and 2 slaves (blue and green)

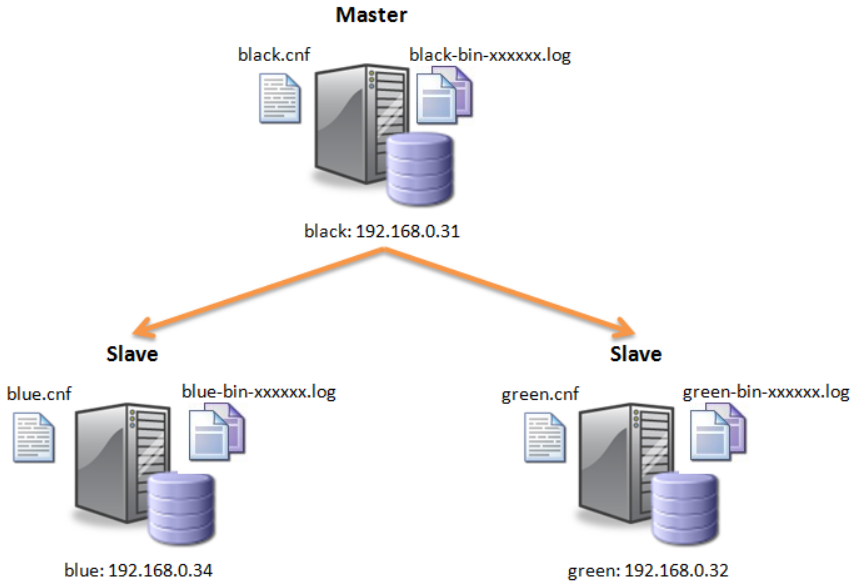

Figure 10 Master with dual slaves

Because any of these servers can become the master, even the slave servers should record changes to a binary log. Similarly, black could become a slave and so should have the appropriate configuration data. The configuration files could look like the following:

black.cnf:

```
[mysqld]
datadir=/home/billy/mysql/master/data
server-id=1

# Replication Master
log-bin=black-bin.log
innodb_flush_log_at_trx_commit=1
sync_binlog=1

# Replication Slave
relay-log-index=slave-relay-bin.index
relay-log=slave-relay-bin
```

blue.cnf

```
[mysqld]
datadir=/home/billy/mysql/slave/data
server-id=2

# Replication Master
log-bin=blue-bin.log
```

```
innodb_flush_log_at_trx_commit=1
sync_binlog=1

# Replication Slave
relay-log-index=slave-relay-bin.index
relay-log=slave-relay-bin
```

green.cnf

```
[mysqld]
datadir=/home/billy/mysql/slave/data
server-id=3

# Replication Master
log-bin=green-bin.log
innodb_flush_log_at_trx_commit=1
sync_binlog=1

# Replication Slave
relay-log-index=slave-relay-bin.index
relay-log=slave-relay-bin
```

As every server can act as a master with the other two replication users must be created on each server, each user with permissions for connecting from one of the other two servers.:

```
black> CREATE USER repl_user@192.168.0.34;
black> CREATE USER repl_user@192.168.0.32;
black> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.34 IDENTIFIED BY 'billy';
black> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.32 IDENTIFIED BY 'billy';

blue> CREATE USER repl_user@192.168.0.31;
blue> CREATE USER repl_user@192.168.0.32;
blue> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.31 IDENTIFIED BY 'billy';
blue> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.32 IDENTIFIED BY 'billy';

green> CREATE USER repl_user@192.168.0.34;
green> CREATE USER repl_user@192.168.0.31;
green> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.31 IDENTIFIED BY 'billy';
green> GRANT REPLICATION SLAVE ON *.* TO repl_user@192.168.0.34 IDENTIFIED BY 'billy';
```

Finally, if semisynchronous replication is being used then both the master and slave plugins should be installed on all 3 servers but the master/slave functionality should only be activated on the appropriate servers:

```
black> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
black> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
black> SET GLOBAL rpl_semi_sync_master_enabled = on;

blue> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
blue> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
blue> SET GLOBAL rpl_semi_sync_slave_enabled = on;
blue> STOP SLAVE IO_THREAD; START SLAVE;

green> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
green> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
green> SET GLOBAL rpl_semi_sync_slave_enabled = on;
green> STOP SLAVE IO_THREAD; START SLAVE;
```

Step 2: Detect if Master has Failed

In many cases, the master would be taken off-line intentionally, for example to perform hardware maintenance, in which case this step doesn't apply. In other cases, the failure will be very obvious – for example the hardware has crashed. However, there are many more subtle 'failures' that you may want to detect and use to trigger a failover.

One simple approach is to monitor the “Seconds_Behind_Master” value obtained from one or more of the slaves:

```
blue> show slave status \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.0.31
                  Master_User: repl_user
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: black-bin.000003
          Read_Master_Log_Pos: 599
               Relay_Log_File: slave-relay-bin.000013
                Relay_Log_Pos: 745
        Relay_Master_Log_File: black-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: 599
              Relay_Log_Space: 1047
              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: 1
```

If Seconds_Behind_Master increases and the level of updates being sent to the master has not changed significantly then that could be a signal that the master has problems (particularly if the same is observed by multiple slaves).

As shown in Figure 15, MySQL Enterprise Monitor can be used to monitor the status of replication and so that can be used to raise the alarm that a failover is required.

Step 3: Suspend Writes to Master

If the master MySQL Server process has died or the underlying hardware has failed then this step is likely to be obsolete but for more planned maintenance of the master or more subtle failures, you want to prevent the client applications from making changes during the transition.

There are many ways that updates could be stopped – by the application or by a load-balancer or the connector. Another approach would be to claim a lock on all tables:

```
black> FLUSH TABLES WITH READ LOCK;
```

Step 4: Promote Slave to Master

In our example, we have 2 slaves and so need to select which one should become the master; the selection should be based on which one is most up to date – in other words which one is further along processing updates replicated from the original master; this can be determined by running "`show slave status \G`" on each of the clients.

Once selected, the IO thread (see Figure 8) should be halted on the selected new master (in our example "blue" has been selected):

```
blue> STOP SLAVE IO_THREAD;
```

After this has been executed, the slaves SQL thread will continue to run and apply any remaining updates from the relay log. Once there has been enough time for all of the changes from the relay logs to be applied (wait until "Exec_Master_Log_Pos" = "Read_Master_Log_Pos" in the output from "`SHOW SLAVE STATUS \G`") replication can be stopped altogether on the server which will become the new master:

```
blue> STOP SLAVE;
```

If using semisynchronous replication then at this point, activate the master-side plugin on the new master:

```
blue> SET GLOBAL rpl_semi_sync_master_enabled = on;
```

Before starting replication from the new master (blue) to the remaining slave (green) you need to determine the current position in the new master's binary log:

```
blue> SHOW MASTER STATUS \G
*************************** 1. row ***************************
            File: blue-bin.000001
        Position: 807
    Binlog_Do_DB:
Binlog_Ignore_DB:
```

On the remaining slave ("green"), "re-master" it to the new master:

```
green> STOP SLAVE;
green> CHANGE MASTER TO MASTER_HOST='192.168.0.34',
    -> MASTER_USER='repl_user',
    -> MASTER_PASSWORD='billy',
    -> MASTER_LOG_FILE='blue-bin.000001',
    -> MASTER_LOG_POS=807;
green> START SLAVE;
```

If using semisynchronous replication then make sure that the remaining slave is registered for asynchronous replication by querying the new master (result should be 1):

```
blue> SHOW STATUS LIKE 'Rpl_semi_sync_master_clients';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| Rpl_semi_sync_master_clients | 1     |
+------------------------------+-------+
```

Replication is now up and running again as shown in Figure 11.

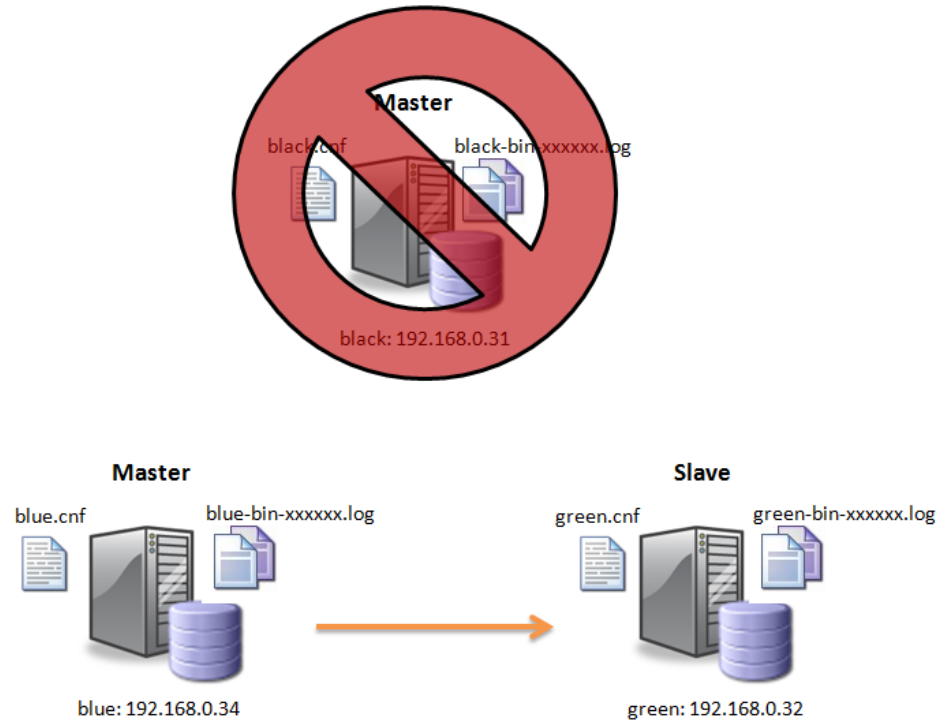

Figure 11 Replication restarted with new master

Step 5: Redirect Writes to New Master After Relay Log is Applied

The application can now start sending writes to the new master (and reads to both the new master and the remaining slave). This can be enabled using the connector, load balancer or within the application.

Step 6: Synchronize Failed Master with New Master

At some point, the original master (black) may be in a position to be added back into the configuration as a slave (initially at least) of the new master. There are 2 options to achieve this:

1. Treat is as a completely new slave and follow Steps 3 through 6 from Section 7. This might be the preferred option if a very large number of updates were applied to the new master before the original master is re-introduced or if there were changes made to the original master that were not replicated to the new master (this could be the case in an uncontrolled failure of the original master when not using semisynchronous replication).
2. Bring the original master back up to date by introducing it as a slave of the new master and allowing it to catch up with all of the changes made since it was promoted to master. The rest of this section shows this approach.

In either case, release the read-lock on the original master if not already done so:

```
black> UNLOCK TABLES;
```

If using semisynchronous replication then activate that plugin on the original master (new slave):

```
black> SET GLOBAL rpl_semi_sync_slave_enabled = on;
```

Start the new slave from the point recorded in Step 4:

```
black> CHANGE MASTER TO MASTER_HOST='192.168.0.34',
    -> MASTER_USER='repl_user',
```

```
    -> MASTER_PASSWORD='billy',
    -> MASTER_LOG_FILE='blue-bin.000001',
    -> MASTER_LOG_POS=807;
black> START SLAVE;
```

At this point, we are back to running with 1 master and 2 slaves as shown in Figure 12.

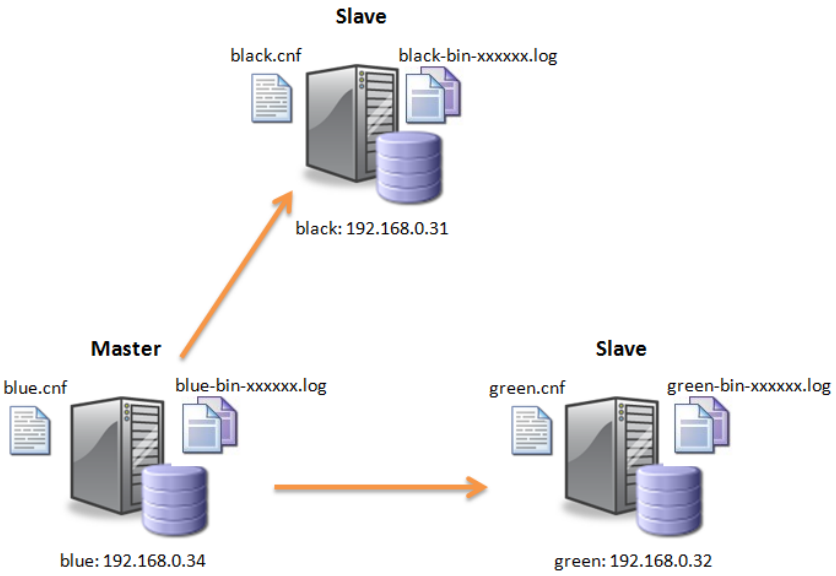

Figure 12 Back to 1 Master and 2 Slaves

An optional final step would be to run through this procedure again to make the original master (black) master again.

11 Differences when Replicating with MySQL Cluster

MySQL Cluster is a scalable, high-performance, clustered database, originally developed for some of the world's most demanding applications found in the telecommunications industry. Often these telecom applications required that the database's availability exceed 99.999%.

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 13** below. 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 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.

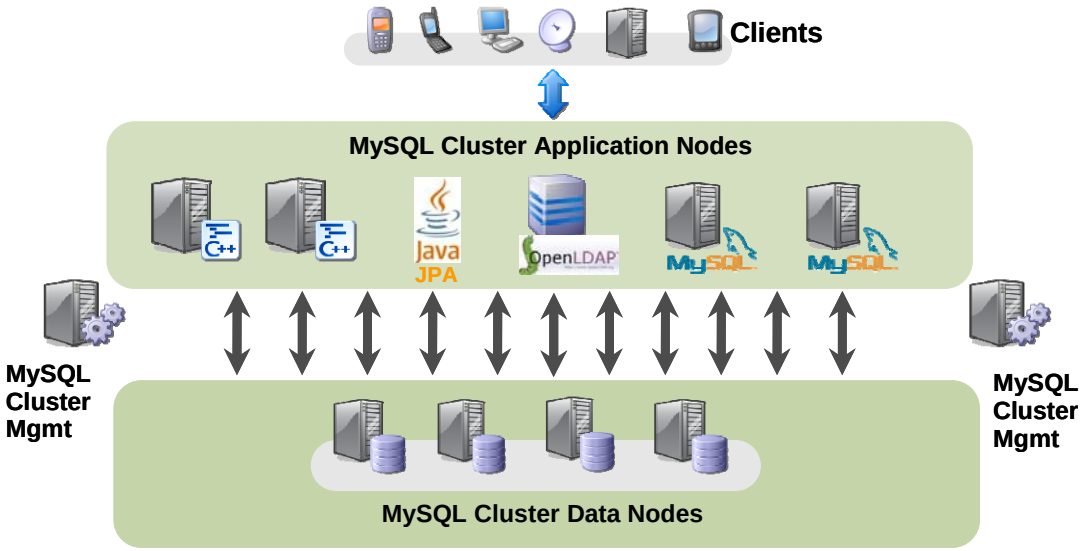

Figure 13 MySQL Cluster Architecture

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 data center and then MySQL (asynchronous) replication to a remote site guards against a catastrophic site failure.

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 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 14 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 at http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/mysql-cluster-replication-failover.html

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. There is some additional work required from the application for this conflict detection/resolution to work - this is beyond the scope of this white paper but details can be found in the MySQL Cluster Reference Guide: http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/mysql-cluster-replication-multi-master.html

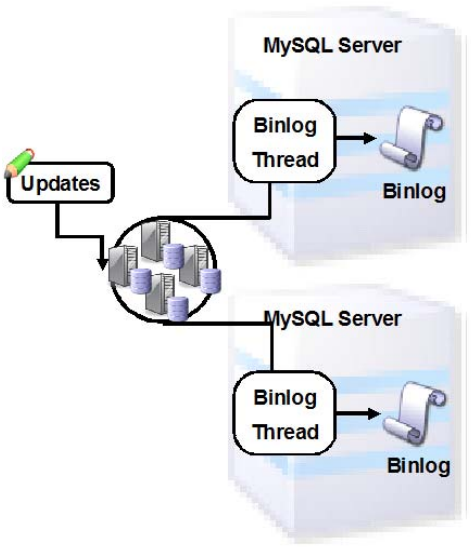

Figure 14 Multiple Masters within a Cluster

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 7.1 which uses a modified version of MySQL 5.1 for the MySQL Servers and so the enhancements introduced in MySQL 5.5 are not available. An exception to this is attribute promotion and demotion which was introduced in MySQL Cluster 7.1.3.

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

12 Replication Monitoring with MySQL Enterprise Monitor

The MySQL Enterprise Monitor with Query Analyzer is a distributed web application that you deploy within the safety of your corporate firewall. 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.

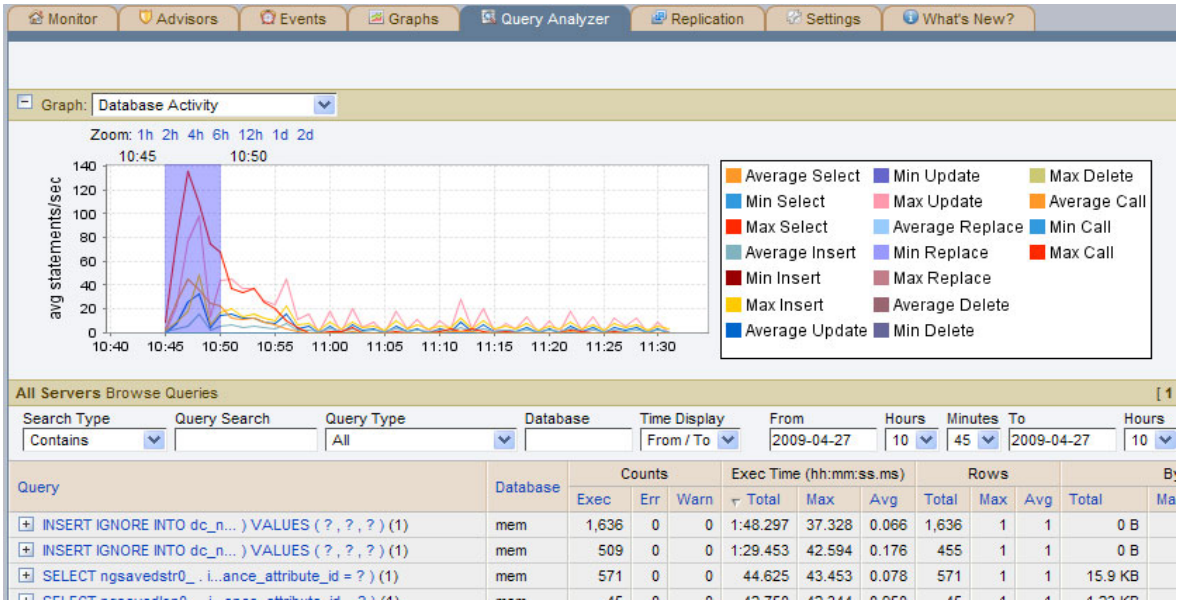

Figure 15 MySQL Enterprise Monitor

MySQL Replication-specific features that Enterprise Monitor provides, include**:**

- Replication relationships are auto-detected, auto-grouped and maintained without the need for DBA set up or configure
- A consolidated, real-time view into the performance and status of all Master/Slave relationships – as shown below in Figure 16.

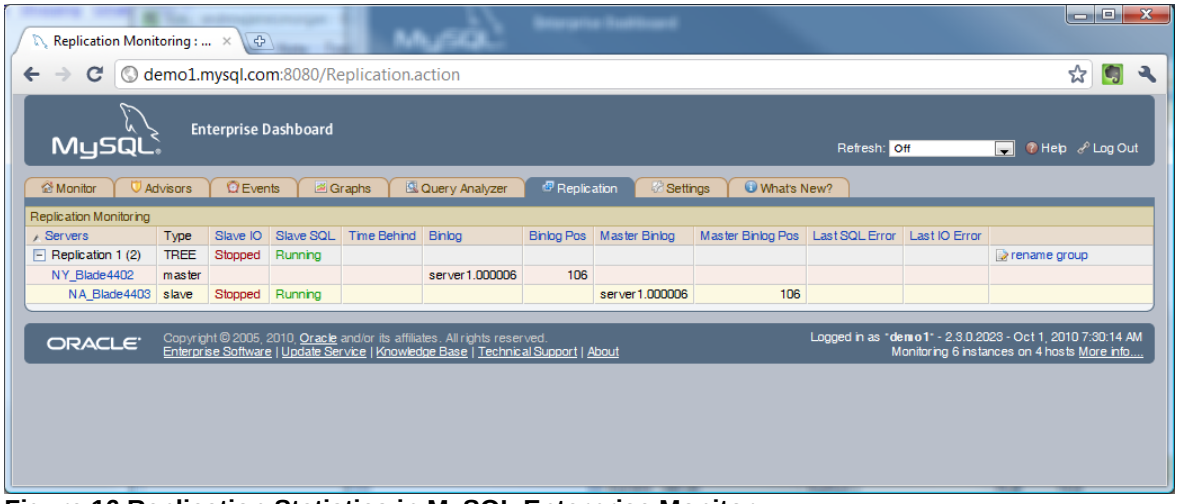

Figure 16 Replication Statistics in MySQL Enterprise Monitor

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

13 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 on the web and in the enterprise.

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 5.5 release, replication is an area of active development, with continual improvements in areas such as data integrity, performance and deployment flexibility.

The resources below complement the material presented in this whitepaper, and will accelerate your evaluation and deployment of MySQL Replication

14 Resources

MySQL 5.5 Download: http://dev.mysql.com/downloads/mysql/5.5.html

MySQL Replication user guide: http://dev.mysql.com/doc/refman/5.5/en/replication.html

Delivering Scalability and High Availability with MySQL 5.5 Replication Enhancements – webinar replay: http://www.mysql.com/news-and-events/on-demand-webinars/display-od-572.html

MySQL Cluster Geographic Replication webinar replay: http://www.mysql.com/news-and-events/on-demand-webinars/display-od-415.html

MySQL Cluster Replication Documentation: http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/mysql-cluster-replication.html

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