Advanced Tools & Techniques for MySQL Administrators

High Performance MySQL

Optimization, Backups, Replication & Load Balancing

O'REILLY®

Jeremy D. Zawodny & Derek J. Balling

CHAPTER 7
Replication

MySQL use often grows organically. In the corporate world, a single application developer may build the company's next killer app on top of MySQL. This initial success with MySQL development typically breeds more projects and more success. As the amount of data you manage using MySQL grows, you'll certainly appreciate its ability to handle large amounts of data efficiently. You may even find that MySQL has become the de facto standard backend storage for your applications.

At the same time, you may also begin to wish for an easy way to copy all the data from one MySQL server to another. Maybe you need to share data with a remote office in your organization, or you might just like to have a "hot spare" available in case your server dies. Fortunately, MySQL has a built-in replication system. You can easily configure a second server as a *slave* of your *master*, ensuring that it has an exact copy of all your data.

In this chapter, we'll examine all aspects of MySQL replication. We begin with an overview of how replication works, the problems it solves, and the problems it doesn't solve. We then move on to the ins and outs of configuring replication. After that we'll consider the various architectures you can construct using various numbers of masters and slaves. We'll continue with a discussion of administrative issues, including maintenance, security, useful tools, and common problems. Finally, we'll look ahead to some planned changes and improvements for MySQL's replication.

MySQL Versions 3.23.xx and 4.0.x have slightly different replication implementations. Much of the discussion in this chapter applies to both versions. There are sections that apply to only one, however, and they are explicitly noted.

Replication Overview

Database replication has an undeserved reputation for being complex to set up and prone to failure. The early versions of MySQL replication were difficult to configure

because the process was inadequately documented. In its most basic form, replication consists of two servers: a master and a slave. The master records all queries that change data in its *binary log*. The slave connects to the master, reads queries from the master's binary log, and executes them against its local copy of the data.

Before peering under the hood, let's look at the types of problems replication does and doesn't solve. If you're reading this in the hopes of deploying replication to cure a problem, this section may help you decide whether you're on the right track.

Problems Solved with Replication

Replication isn't perfect, but it can be quite useful in solving several classes of problems in the areas of scalability and backups.

Data distribution

Need to maintain a copy of your data 10,000 miles away? Replication makes it trivial to do so. As long as you have decent connectivity between two sites, you can replicate a MySQL database. Think of this as scaling geographically.

In fact, it's possible to use replication over a network connection that isn't "always on," such as traditional dial-up using PPP. You can simply let the slave fail and reconnect (it'll keep trying for a long time). Or you can use one of the `SLAVE STOP` commands (described later) to disable the slave's replication when no connection is available. The master doesn't mind if a slave disconnects for a few hours and then reconnects. But you can't let the slave go for too long without reconnecting to the master, because the older record of changes will eventually be purged to keep the master from running out of disk space.

Of course, you can also use replication between two servers that sit next to each other. Any time you need multiple up-to-date copies of your MySQL data, replication is often the easiest solution. You can even replicate data between two MySQL servers on the same machine, which is often a good way to test a new version of MySQL without using a second machine.

Load balancing

If you use MySQL on a large data warehousing application or a popular web site, odds are that your server is running many more read queries (`SELECT`) than write queries (`INSERT`, `UPDATE`, and `DELETE`). If that's the case, replication is an excellent way to support basic load balancing. By having one or more slave servers, you can spread most of the work among several servers.

The trick, of course, is coming up with an effective way to spread the queries among the available slaves so they get roughly equal workloads. One simple approach is to use round-robin DNS. Assign multiple IP addresses for a hostname such as *db-slave*.

example.com, and your application will connect to one at random each time it opens a new connection to MySQL.*

A more sophisticated approach involves the same solutions that are used in web server load balancing. Network load-balancing products from Foundry Networks, Cisco, Nortel, and others work just as well for MySQL as they do for web sites.† The same is true of software solutions such as the Linux Virtual Server (LVS) project (*http://www.linuxvirtualserver.org/*).

Load-balancing techniques are covered in greater detail in Chapter 8.

Backup and recovery

Backing up a busy MySQL server can be difficult when your clients demand access to the data 24 hours a day. Rather than deal with the complexity of implementing a backup process that minimizes the impact on your clients, you might find it easier simply to configure a slave server and back it up instead. Because the slave will have an exact copy of the data, it's just as good as backing up the master. Plus you'll never impact the clients who are using the master. You might even decide that you don't necessarily need or want to back up the data as long as you have the "hot spare" slave database available in the case of problems with the master.

Chapter 9 covers backup and recovery techniques in more detail.

High availability and failover

Using replication, you can avoid making MySQL (or the system hosting it) a single point of failure in your applications. Although there's no out-of-the-box, automated failover solution for MySQL, you can achieve a good degree of high availability using some relatively simple techniques.

Using a creative DNS setup, you can insulate your applications from having to know which server is the master and minimize the effort involved in switching to a slave when the master fails.

Let's suppose you have two MySQL servers, *db1.example.com* and *db2.example.com*. Rather than hardcoding the name of the master in your applications, you can set up *db.example.com* as a CNAME (or alias) in DNS for your master. By using a very low Time To Live (TTL) on the DNS record, you can ensure that clients will not cache the information longer than necessary.

In the event your master goes down, simply update your DNS to point `db.example.com` at the new master. As soon as the TTL expires, your applications will pick up the new information and connect to the proper server. There will be some time during

* Some operating systems don't randomize this very well.

† That's not entirely true, as you'll soon see.

which the applications can't contact MySQL, but that time will be relatively brief if you use a low enough TTL.*

If you'd like to eliminate entirely the need to use DNS, you can play similar games using IP addresses. Because it's trivial to add and remove additional IP addresses from a server, a scheme like this may serve you well:

- Use an IP address for each role, such as 192.168.1.1 for the master and an address in the 192.168.1.100–192.168.120 range for each slave.
- Make sure each machine has its own primary IP address in addition to the role-based IP address.
- When the master goes down, any of the slaves can be scripted to pick up the IP address and immediately take over.
- The master should be set so that if it ever loses its master address or goes down, it doesn't automatically pick up the address again (i.e., it assumes someone else will).

See the "High Availability" section of Chapter 8 for more on the topic.

Problems Not Solved with Replication

Replication doesn't solve every problem. Performance can become an issue with replication because every slave still needs to execute the same write queries as the master. In a very write-heavy application, slaves need to be at least as powerful as the master. If you attempt to use replication to set up a load-balancing system, you may be disappointed. It may be more productive to implement a partitioning system with multiple masters—one for each partition of the data.

Also, there's no guarantee that a slave will be completely in sync with the master at any given moment. If the load on a slave is relatively high, the slave may fall behind and need time to catch up.

Network bandwidth and latency can also become an issue. If the slave is far away from the master (in a network sense) and there isn't sufficient bandwidth, a slave may be able to keep up with the master's query load, but it won't be able to get data fast enough to do so.

Let's look at two specific examples that illustrate problems not easily solved with replication.

* Be careful not to set it too low, however. The DNS resolvers shipped with some operating systems have been known to simply ignore TTLs that are deemed to be too low. When in doubt, test the implementation before depending on it to work.

Real-time data transmission

MySQL's replication isn't the ideal vehicle for transmitting real-time or nearly real-time data such as a stock quote feed or an online auction site. In those applications, it's important that the user sees up-to-date data no matter which database server they use.

The only way to combat MySQL's lack of any latency guarantee is to implement your own monitoring system. It needs to use some sort of heartbeat to verify that each server has a reasonably up-to-date copy of the data. In the event that a server falls too far behind, the monitoring system needs to proactively remove it from the list of active servers until it can catch up.

Of course, you can also build your application in such a way that it updates all the slaves with the newest data. However, that can add a lot of complexity and may not be worth the effort. You'd end up writing a lot of code to handle the exceptional conditions, such as when a single server falls behind or is intermittently inaccessible. Testing and debugging all those situations can be very time-consuming and difficult.

As Derek went over this, he thought, "Wouldn't it be cool if MySQL could provide a query response that signified, 'Go ask another server, I'm really busy right now'?" This would allow clients to automatically find willing servers in a multihost DNS rotation.

For example, the client wants to connect to *db.example.com* (which is *db1*, *db2*, and *db3*). It connects (randomly) to *db2*, and the server answers the query with "I'm busy; go ask someone else," whereupon the client knows enough to try *db1* or *db3*. Because the client library would be connecting to the same virtual server, it could transparently disconnect from the busy server and connect to some other (hopefully less busy) server.

As a result, all you would need is some automated way for a slave server to know how far behind they are and to shut themselves off from queries when they get too far behind, and you'd have some protection. Of course, this could also be subject to a cascading failure. If all the slaves are very busy, the last thing you'd want is for them to start removing themselves from the pool of available servers. Continue on to Chapter 8 for a deeper discussion of these issues.

Online ordering

An ordering system is different from a real-time stock quote feed or an auction site in a couple of important ways. First, the ratio of reads to writes is likely to be much lower. There isn't a constant stream of users running read-only queries against the data. Also, when users are running read queries, they're often part of a larger transaction, so you can't send those read queries to a slave. If you did, the slave might not have the correct data yet. Transactions aren't written to the binary log and therefore

sent to slaves until they first commit on the master. A slave will contain only committed transactions.

Replication can still be very useful for an order processing system. It's reasonable to use a slave for running nightly reports and queries that don't need the most recent data.

Replication Performance

Having considered the problems that replication does and doesn't solve, you may still be a bit unsure about using it. Maybe replication is fast enough to get the job done, despite the lack of any performance guarantees built into the system. Wouldn't it be nice to have a general idea of how fast replication really is?

That's exactly what we wanted to know when we first began using replication—partly for our own sanity and partly because we knew a lot of people would soon be interested in MySQL replication. The first question they'd ask is, "How fast is it?" To answer that question, we devised the following simple test to measure the practical minimum replication latency in a particular environment.

A Perl script opened two database connections, one to the master and one to the slave. The master and slave were on the same 100-Mbit switched Ethernet network. The script then inserted a record into the master and immediately attempt to retrieve it from the slave. If the record wasn't available, the script immediately retried. We kept the records intentionally small, containing just an `auto-increment` column and a `VARCHAR` field into which we inserted the current time.

The results were encouraging. Of the 1,000 records inserted, 950 of them were available on the first attempt. That left 50 records that required at least a second try. Of those 50, 43 were available on the second attempt. The remaining 7 were there on the third try. The test was quick and very unscientific, but it can help to set realistic expectations.

Configuring Replication

With the theory out of the way, let's get our hands on some servers and configure a master and slave. We'll cover two scenarios. In the first, we'll assume that you have a fresh installation of MySQL on the master and slave, with no data on either server aside from the default `test` and `mysql` databases. Later, we'll examine the ways you can configure replication on a running master with minimal hassle and interruptions for your users.

On a New Server

Configuring replication on a new server is a straightforward process. The tasks you need to perform are:

1. Create a replication account on each server.
2. Add configuration entries in *my.cnf* on each server.
3. Restart the master and verify the creation of a binary log.
4. Restart the slave and verify that replication is working.

That's it. Four steps.

For the sake of clarity, we'll use the hostnames *master* and *slave* for our master and slave, respectively.

Account creation

When the slave connects to the master, it must authenticate itself just like any other MySQL client, so it needs a username and password. We'll create an account named *repl* with a password of `c0pyIT!` on both the master and slave.

Why create the account on the slave? Should the master ever fail, you'll want the slave to become the new master. When the old master is repaired, it can be put back online as a slave of the new master (which is the old slave). If the account didn't exist on the new master, the new slave wouldn't be able to replicate.

So, on each server let's create the account and give it only the minimum privileges necessary for replication: `REPLICATION SLAVE` and `REPLICATION CLIENT`. (In MySQL 3.23 you'd use `USAGE` and `FILE`.)

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO
repl@"192.168.1.0/255.255.255.0" IDENTIFIED BY 'c0pyIT!';
Query OK, 0 rows affected (0.00 sec)
```

After creating the account, verify that the settings are correct:

```
mysql> SHOW GRANTS FOR repl;
+-----------------------------------------------------------------------------+
|Grants for repl@"192.168.1.0/255.255.255.0"                                   |
+-----------------------------------------------------------------------------+
| GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'repl'@'...' IDENTIFIED BY ...|
+----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

If that command returns no rows or doesn't list the privileges, double-check that you entered the `GRANT` command properly.

Configuration file entries

The next step is to update the *my.cnf* file on each server. You need to tell the master to enable binary logging and to tell the slave about its master, login credentials, and so on.

Finally, each server needs to be assigned an ID number, known as a *server ID*. As you'll see later, the server ID is recorded in each server's binary log entries so that any other server can know which server first executed and logged a query. The server ID can be any number in the range 1–4294967295.*

So, on the master, make sure the following lines are present in the [*mysqld*] section of the *my.cnf* file:

```
log-bin
server-id = 1
```

The `log-bin` option tells MySQL to enable binary logging. By default, MySQL places the log file in its data directory. To place the logs elsewhere, you can supply a path and filename:

```
log-bin = /var/db/repl/log-bin
```

The slave requires a bit more information than the master. Add the following lines to its *my.cnf* file:

```
server-id = 2
master-host = master.example.com
master-user = repl
master-password = c0pyIT!
master-port = 3306
```

The slave's settings are self-explanatory. It just needs to know how to contact the master and authenticate itself. Later we'll look at some optional replication settings that can also appear in *my.cnf*.

Restart master

With the settings on the master, it's time to stop and start MySQL and verify that the binary log appears. No replication-specific method is necessary; simply stop and restart MySQL using your normal scripts that handle MySQL when the machine boots and shuts down.

If you didn't specify a path and filename after `log-bin` in your *my.cnf* file, MySQL writes the log files in your data directory. Otherwise, the logs are written in the location you specified.

You should find a binary log file with a *.001* extension. By default, the filename will be *hostname-bin*. On the host *master.example.com*, the first log file will be *master-bin.001*.

* A server ID of 1 is assumed if not explicitly assigned.

If you haven't run any write queries yet, the file will be less than 100 bytes in size. Each log file contains a short header and some meta information.

If you then execute a few write queries, you should notice the size of the binary log file increasing. If not, check the error log for hints about what might have gone wrong.

You can use the *mysqlbinlog* utility to examine the data stored in a binary log file. It reads the entries and prints out the SQL for each one. It also prints some comments that contain other helpful information. For example, running it on a fresh log produces output like this:

```
$ mysqlbinlog master-bin.001
# at 4
#020922 14:59:11 server id 1  log_pos 4        \
 Start: binlog v 3, server v 4.0.4-beta-log created 020922 14:59:11
```

The first comment indicates that this entry is at offset 4 in the log. The second comment indicates when the log was created, the server ID, the log version, and the server version.

Restart slave

With the master logging properly, a simple restart of the slave should be sufficient to get replication running. When a MySQL server is started, it checks to see whether it should connect to a master and begin (or continue) replicating queries. Upon connecting to the master, MySQL logs a message in its error log to indicate whether the connection succeeded or failed:

```
021103 13:58:10  Slave I/O thread: connected to master 'repl@master:3306',
replication started in log 'log-bin.001' at position 4
```

This entry indicates that the slave has connected to the master and begun reading the binary log file *master-bin.001* at position (or offset) 4, that of the first query.

Run some write queries on the master and verify that the data on the slave reflects those changes. Once the slave is happily replicating from the master, it can continue to do so indefinitely.

On an Existing Server

Setting up replication on a new server is easy. A few config file entries and a couple of restarts are all you need. On an existing server, however, there's a bit more work to do because you can't simply point a new slave at a master and ask it to clone all the existing data.*

* There are plans to fix that in a future version of MySQL.

There are a couple of ways to do get the job done. We'll look at two specific solutions in a moment. First, let's outline the work that needs to be done; we'll deal then with common solutions.

What needs to happen

Normally, to enable replication, you have to add binary logging to your server, which means subsequently restarting the server. If you happen to have binary logging already enabled, you don't have to restart the server. As described earlier, you'll need to add at least two lines to the server's *my.cnf* file:

```
log-bin
server-id = 1
```

Optionally, specify a full path and base filename for the binary logs:

```
log-bin = /var/db/repl/binary-log
```

The other task involves getting a copy of all the data from the master and putting it on the new slave. But there's a twist. The data given to the slave must correspond to the exact moment in time the binary log begins. Said another way, the binary log should contain *all* the queries that are executed on the master after the snapshot was taken and *none* of the queries from before the snapshot.

If the binary log contains queries that are already reflected in the data given to the slave, the slave has no way to know that. Consequently, it reexecutes the queries, possibly producing strange errors or otherwise making the data inconsistent with what is on the master.

If the binary log misses a few queries that weren't reflected in the slave's copy of the data, it won't see those queries. This can cause strange and hard-to-diagnose problems. Maybe records that were supposed to have expired are still there, or perhaps there's data on the master that doesn't appear on the slave.

Getting the initial data from the master to the slave may be complicated. If you're using only MyISAM tables and can afford to shut down the master for enough time to copy all the data, it will be easy. Otherwise, you'll need to perform an online copy or dump of the data.

Snapshot or backup, then copy

The easiest way to get the necessary data is to perform a snapshot (online backup) or a more traditional offline backup and then copy the data to the slave. Using archive tools such as *tar* or *zip*, or your traditional backup software, shut down MySQL and copy the contents of the data directory to your slave; then extract the data on the slave.

This method works well if you intend to replicate all the data and can shut down MySQL for the time required to make a copy of the data. If, however, you can't

afford to have MySQL offline for more than a few seconds, there's an alternative approach: restart the server once after making the config file changes and then perform an online snapshot of the data.

A snapshot works well only for MyISAM tables. InnoDB and BDB tables are best backed up when MySQL isn't running at all. A snapshot also requires a read lock on the data for the duration of the snapshot. So you'll be able to service read requests during the snapshot process, but all writers will be blocked.

To perform the actual snapshot, you can write your own script to do the job, or you can use *mysqlhotcopy* or *mysqlsnapshot*. If you roll your own script, you need to ensure that the binary log is reset before the locks are released. The easiest way to do that is by executing `FLUSH TABLES WITH READ LOCK` and then `RESET MASTER` (or `FLUSH MASTER` in versions older than 3.23.26).

Chapter 9 covers backups as well as the *mysqlhotcopy* and *mysqlsnapshot* utilities.

Online table copies

Another approach is to use MySQL's command:

```
LOAD TABLE mytable FROM MASTER
```

Doing so instructs a slave to load an entire table from the master. By writing a relatively simple script, you can instruct the slave to clone all the tables it needs using a series of those commands.

The usefulness of this technique is relatively limited, however. Like the previous option, it requires a master that isn't being updated. In an environment in which there are frequent updates to the master, this technique is simply not viable. Furthermore, the slave copies only the data from the master. It then reconstructs the indexes locally, for which large amounts of data can take hours or even days.

Online copy and synchronize (MySQL 4.x only)

MySQL 4.0 introduced the `LOAD DATA FROM MASTER` command. It combines the previous two approaches by first obtaining a read lock on all the master's tables, then loading each table one by one using the `LOAD TABLE` mechanism.* It respects any slave-side database or table filtering. Once it completes the loading process, it releases the locks on the master and begins replicating.

While this option is very appealing, it suffers from the same limitations as scripting the `LOAD TABLE` command yourself. It is much slower than using a master snapshot. It also requires that you grant the `repl` user `SUPER` and `RELOAD` privileges on the master. Finally, it works only with MyISAM tables.

* This doesn't include the tables in the `mysql` database. Put another way, `LOAD DATA FROM MASTER` doesn't clone users and permissions from the master.

Under the Hood

What really happens during replication? What does the binary log contain? What's different in Version 4.0? To help answer those questions, let's get deeper into the details and then walk through the steps that MySQL performs during replication. We'll start with an insert on the master and follow it to completion on the slave. We'll also look at how MySQL 3.23 and 4.x differ.

Replication in 3.23

MySQL's original replication code provides basic replication services. The master logs all write queries to the binary log. The slave reads and executes the queries from the master's binary log. If the two are ever disconnected, the slave attempts to reconnect to the master.

If you follow a query from start to finish, here's what's happening behind the scenes:

1. The client issues a query on the master.
2. The master parses and executes the query.
3. The master records the query in the binary log.
4. The slave reads the query from the master.
5. The slave parses and executes the query.
6. The slave performs a sanity check, comparing its result with the master's. If the query failed on the slave but succeeded on the master, replication stops. The reverse is also true. If the query partially completed on the master but succeeds on the slave, the slave stops and complains.
7. The slave updates the *master.info* file to reflect the new offset at which it is reading the master's binary log.
8. The slave waits for the next query to appear in the master's binary log. When one appears, it starts over at Step 4.

That's a relatively simple arrangement. The master simply logs any queries that change data. The slave reads those queries from the master, one by one, and executes each of them. If there are any discrepancies between the results on the master and the slave, the slave stops replicating, logs an error, and waits for human intervention.

The simplicity of this system has problems, however. If the master and slave are separated by a slow network, the speed at which replication can occur becomes bounded by the network latency. Why? Because the process is highly serialized. The slave runs in a simple "fetch query, execute query, fetch query, ..." loop. If the "fetch query" half of the loop takes more than a trivial amount of time, the slave may not be able to keep up with the master during very heavy workloads. The master may be able to execute and log 800 queries per second, but if the slave requires 25 msec to fetch each query over the network, it can replicate no more than 40 queries per second.

This can be problematic even with a fast network connection. Suppose the master executes a query that takes five minutes to complete. Maybe it's an `UPDATE` that affects 50 million records. During the five minutes the slave spends running the same query, it isn't pulling new queries from the master. By the time it completes the query, it's effectively five minutes behind the master, in terms of replication. It has a fair bit of catching up to do. If the master fails during that five-minute window, there's simply no way for the slave to catch up until the master reappears. Some of these problems are solved in 4.0.

Replication in 4.0

To solve the problem of slaves falling behind because of slow queries or slow networks, the replication code was reworked for Version 4.0. Instead of a single thread on the slave that runs in a "fetch, execute, fetch, ..." loop, there are two replication threads: the *IO thread* and the *SQL thread*.

These two threads divide the work in an effort to make sure the slave can always be as up to date as possible. The IO thread is concerned only with replicating queries from the master's binary log. Rather than execute them, it records them into the slave's *relay log*.* The SQL thread reads queries from the local relay log and executes them.

To put this in context, let's look at the step-by-step breakdown for replication in MySQL 4.0:

1. The client issues a query on the master.
2. The master parses and executes the query.
3. The master records the query in the binary log.
4. The slave's IO thread reads the query from the master and appends it to the relay log.
5. The slave's IO thread updates the *master.info* file to reflect the new offset at which it is reading the master's binary log. It then returns to Step 4, waiting for the next query.
6. The slave's SQL thread reads the query from its relay log, parses it, and then executes it.
7. The slave's SQL thread performs a sanity check, comparing its result with the master's. If the query failed on the slave but succeeded on the master, replication stops.
8. The slave's SQL thread updates the *relay-log.info* file to reflect the new offset at which it is reading the local relay log.
9. The slave's SQL thread waits for the next query to appear in the relay log. When one appears, it starts over at Step 6.

* To keep things simple, the relay log file uses the same storage format as the master's binary log.

While the steps are presented as a serial list, it's important to realize that Steps 4–5 and 6–9 are running as separate threads and are mostly independent of each other. The IO thread never waits for the SQL thread; it copies queries from the master's binary log as fast as possible, which helps ensure that the slave can bring itself up to date even if the master goes down. The SQL thread waits for the IO thread only after it has reached the end of the relay log. Otherwise it is working as fast as it can to execute the queries waiting for it.

This solution isn't foolproof. It's possible for the IO thread to miss one or more queries if the master crashes before the thread has had a chance to read them. The amount of data that could be missed is greatly reduced compared to the 3.23 implementation, however.

Files and Settings Related to Replication

There are several files and configuration options related to replication in this chapter. Without going into a lot of detail on any one of them (that's done elsewhere), the files fall into three categories: log files, log index files, and status files.

Log files

The log files are the binary log and the relay log. The binary log contains all write queries that are written when the log is enabled. The `log-bin` option in *my.cnf* enables the binary log. Binary log files must be removed when they're no longer needed because MySQL doesn't do so automatically.

The relay log stores replicated queries from a MySQL 4.0 slave (from the master's binary log) before it executes them. It's best thought of as a spool for queries. The relay log is enabled automatically in 4.0 slaves. The `relay-log` option in *my.cnf* can customize the name and location of the relay log's base filename:

```
relay-log = /home/mysql/relay.log
```

Like the binary log, MySQL always appends a sequence number to the base name, starting with 001. Unlike the binary log, MySQL takes care of removing old relay logs when they are no longer needed. MySQL 3.23 servers don't use relay logs.

Log index files

Each log file has a corresponding index file. The index files simply list the names of the log files on disk. When logs are added or removed, MySQL updates the appropriate index file.

You can add settings to *my.cnf* to specify the log index filenames and locations:

```
log-bin-index = /var/db/logs/log-bin.index
relay-log-index = /var/db/logs/relay-log.index
```

Never change these settings once a slave is configured and replicating. If you do, MySQL uses the new values when it is restarted and ignores the older files.

Status files

MySQL 3.23 and 4.0 slaves use a file named *master.info* to store information about their master. The file contains the master's hostname, port number, username, password, log file name, position, and so on. MySQL updates the log position and log file name (as necessary) in this file as it reads queries from the master's binary log. While you should never need to edit the file, it's worth knowing what it is used for.

The `master-info-file` option in *my.cnf* can be used to change the name and location of the *master.info* file:

```
master-info-file = /home/mysql/master-stuff.info
```

However, there's rarely a need to do so.

MySQL 4.0 slaves use an additional status file for the SQL thread to track its processing of the relay log, in much the same way the *master.info* file is used. The `relay-log-info-file` setting can be used to change the filename and path of this file:

```
relay-log-info-file = /home/mysql/logs/relay-log.info
```

Again, you won't need to change the default.

Filtering

There may be times when you don't need to replicate *everything* from the master to the slave. In such situations you can use the various replication filtering options to control what is replicated. This is well covered in the MySQL documentation, so we'll just recap the important parts.

There are two sets of configuration options for filtering. The first set applies to the binary log on the master and provide per-database filtering:

```
binlog-do-db=dbname
binlog-ignore-db=dbname
```

Any queries filtered on the master aren't written to its binary log, so the slave never sees them either.

The second set of options applies to the relay log on the slave. That means the slave still has to read each query from the master's binary log and make a decision about whether or not to keep the query. The CPU overhead involved in this work is minimal, but the network overhead may not be if the master records a high volume of queries.

Here is the second set of options:

```
replicate-do-table=dbname.tablename
replicate-ignore-table=dbname.tablename
replicate-wild-do-table=dbname.tablename
replicate-wild-ignore-table=dbname.tablename
replicate-do-db=dbname
replicate-ignore-db=dbname
replicate-rewrite-db=from_dbname->to_dbname
```

As you can see, the slave options are far more complete. They not only offer per-table filtering but also allow you to change the database or table names on the fly.

Replication Architectures

Though MySQL's replication system is relatively simple compared to some commercial databases, you can use it to build arbitrarily complex architectures that solve a variety of problems. In this section we'll look at some of the more popular and exotic configurations. We'll also review how MySQL's replication design makes this possible.

The Replication Rules

Before looking at the architectures, it helps to understand the basic rules that must be followed in any MySQL replication setup:

- Every slave must have a unique server ID.
- A slave may have only one master.
- A master may have many slaves.
- Slaves can also be masters for other slaves.

The first rule isn't entirely true, but let's assume that it is for right now, and soon enough you'll see why it isn't always necessary. In any case, the rules aren't terribly complex. Those four rules provide quite a bit of flexibility, as the next sections illustrate.

Sample Configurations

Building on the four rules, let's begin by constructing simple replication configurations and discussing the types of problems they solve. We'll also look at the types of configurations that don't work because they violate the second rule. We'll use the simple configuration as a building block for arbitrarily complex architectures.

Each configuration is illustrated in a figure that includes the server ID of each server as well as its role: master, slave, or master/slave.

Master with slaves

The most basic replication model, a single master with one or more slaves, is illustrated in Figure 7-1. The master is given server ID 1 and each slave has a different ID.

This configuration is useful in situations in which you have few write queries and many reads. Using several slaves, you can effectively spread the workload among many servers. In fact, each of the slaves can be running other services, such as Apache. By following this model, you can scale horizontally with many servers. The only limit you are likely to hit is bandwidth from the master to the slaves. If you have

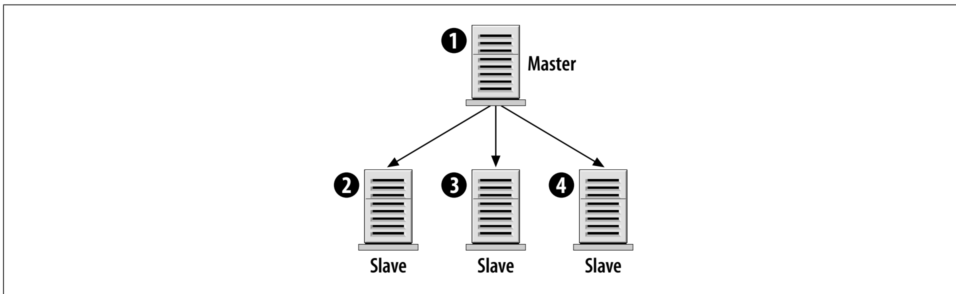

Figure 7-1. Simple master/slave replication

20 slaves, which each need to pull an average of 500 KB per second, that's a total of 10,000 KB/sec (or nearly 10 Mbits/sec) of bandwidth.

A 100-Mbit network should have little trouble with that volume, but if either the rate of updates to the master increases or you significantly increase the number of slaves, you run the risk of saturating even a 100-Mbit network. In this case, you need to consider gigabit network hardware or an alternative replication architecture, such as the pyramid described later.

Slave with two masters

It would be nice to use a single slave to handle two unrelated masters, as seen in Figure 7-2. That allows you to minimize hardware costs and still have a backup server for each master. However, it's a violation of the second rule: a slave can't have two masters.

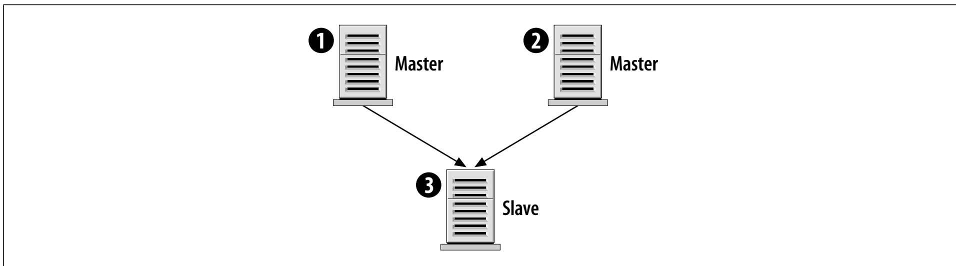

Figure 7-2. A slave can't have two masters

To get around that limitation, you can run two copies of MySQL on the slave machine. Each MySQL instance is responsible for replicating a different master. In fact, there's no reason you couldn't do this for 5 or 10 distinct MySQL masters. As long as the slave has sufficient disk space, I/O, and CPU power to keep up with all the masters, you shouldn't have any problems.

Dual master

Another possibility is to have a pair of masters, as pictured in Figure 7-3. This is particularly useful when two geographically separate parts of an organization need write access to the same shared database. Using a dual-master design means that neither site has to endure the latency associated with a WAN connection.

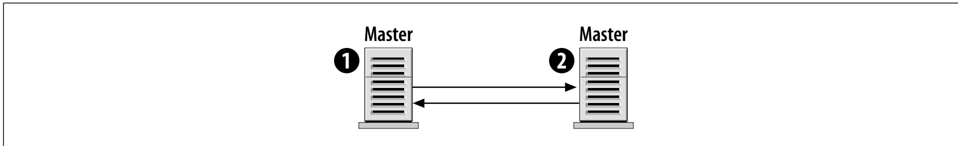

Figure 7-3. Dual master replication

Furthermore, WAN connections are more likely to have brief interruptions or outages. When they occur, neither site will be without access to their data, and when the connection returns to normal, both masters will catch up from each other.

Of course, there are drawbacks to this setup. The section, "Safe Multi-Master Replication," later in this chapter, discusses some of the problems associated with a multi-master setup. However, if responsibility for your data is relatively well partitioned (site A writes only to customer records, and site B writes only to contract records) you may not have much to worry about.

A logical extension to the dual-master configuration is to add one or more slaves to each master, as pictured in Figure 7-4. This has the same benefits and drawbacks of a dual-master arrangement, but it also inherits the master/slave benefits at each site. With a slave available, there is no single point of failure. The slaves can be used to offload read-intensive queries that don't require the absolutely latest data.

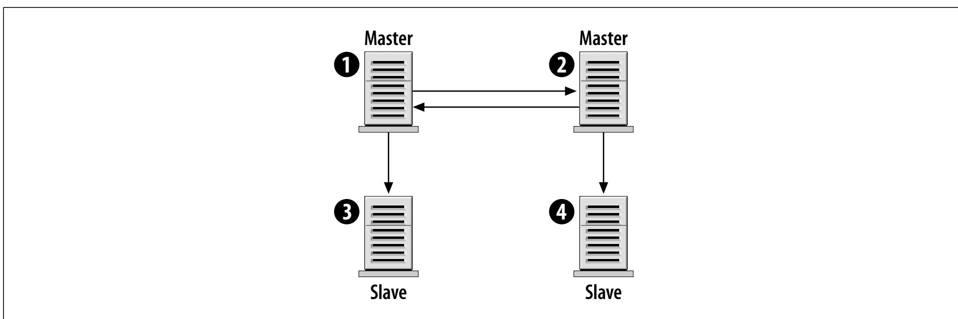

Figure 7-4. Dual master replication with slaves

Replication ring (multi-master)

The dual-master configuration is really just a special case of the master ring configuration, shown in Figure 7-5. In a master ring, there are three or more masters that form a ring. Each server is a slave of one of its neighbors and a master to the other.

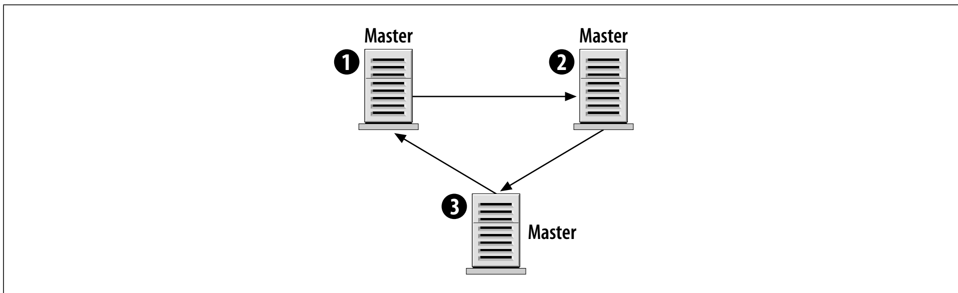

Figure 7-5. A replication ring or multi-master replication topology

The benefits of a replication ring are, like a dual-master setup, geographical. Each site has a master so it can update the database without incurring high network latencies. However, this convenience comes at a high price. Master rings are fragile; if a single master is unavailable for any reason, the ring is broken. Queries will flow around the ring only until they reach the break. Full service can't be restored until all nodes are online.

To mitigate the risk of a single node crashing and interrupting service to the ring, you can add one or more slaves at each site, as shown in Figure 7-6. But this does little to guard against a loss of connectivity.

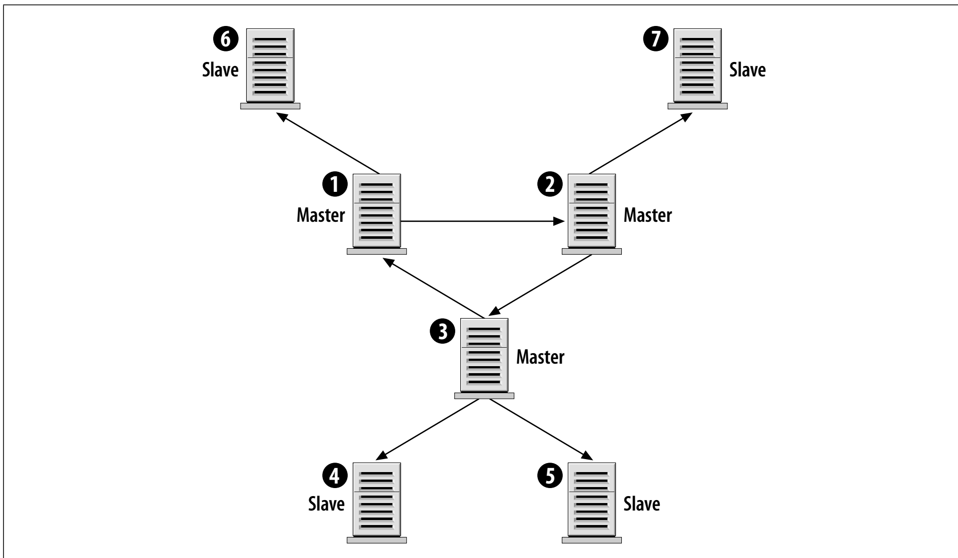

Figure 7-6. A replication ring with slaves at each site

Pyramid

In large, geographically diverse organizations, there may be a single master that must be replicated to many smaller offices. Rather than configure each slave to contact the

master directly, it may be more manageable to use a pyramid design as illustrated in Figure 7-7.

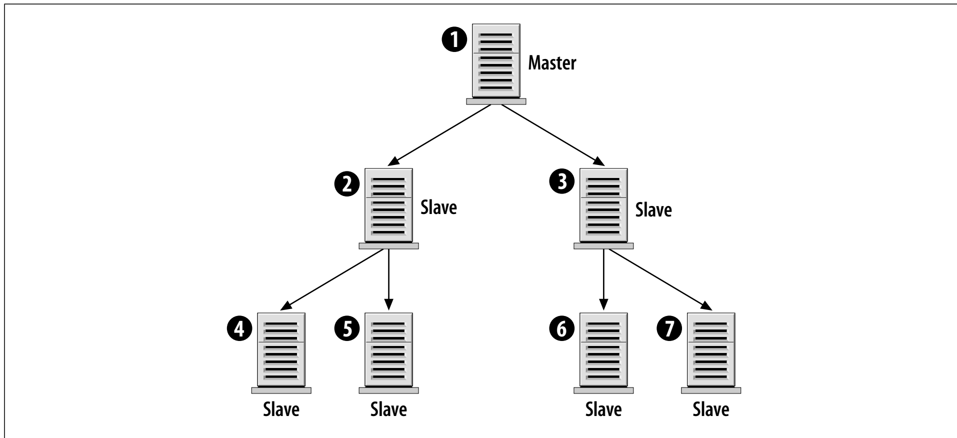

Figure 7-7. Using a pyramid of MySQL servers to distribute data

The main office in Chicago can host the master (1). A slave in London (2) might replicate from Chicago and also serve as a local master to slaves in Paris, France (4), and Frankfurt, Germany (5).

Design your own

There's really no limit to the size or complexity of the architectures you can design with MySQL replication. You're far more likely to run into practical limitations such as network bandwidth, management and configuration hassles, etc. Using the simple patterns presented here, you should be able to design a system that meets your needs. And that's what all this really comes down to: if you need to replicate your data to various locations, there's a good chance you can design a good solution using MySQL.

You can often combine aspects of the architectures we've looked at. In reality, however, the vast majority of needs are handled with less complicated architectures. As load and traffic grows, the number of servers may increase, but the ways in which they are organized generally doesn't. We'll return to this topic in Chapter 8.

Administration and Maintenance

Configuring replication isn't something you do every day (unless you're writing a book about MySQL, of course.) Once replication is set up and running, you're likely just to let it run. But there are a few administrative tasks you'll likely need to deal with at some point. You should, of course, consider automating them as much as possible.

Monitoring

Replication increases the complexity of monitoring MySQL's health. Are all the slaves replicating? Have the slaves encountered errors? How far behind is the slowest slave?

As you may have come to expect, MySQL provides all the data necessary to answer those questions (and many questions you probably haven't even considered), but extracting and understanding the data is something it won't do for you. In the section "Tools," later in this chapter, we'll try to provide some details to help you make sense of all the data MySQL provides, which should help you understand the tools that are helpful in processing that data.

Master status

Using the `SHOW MASTER STATUS` command, the master will tell you about its replication status:

```
mysql> SHOW MASTER STATUS \G
*************************** 1. row ***************************
            File: binary-log.004
        Position: 635904327
    Binlog_do_db:
Binlog_ignore_db:
1 row in set (0.00 sec)
```

The output includes the current binary log filename and the position (or offset) into the binary log where the next query will be written. The other two fields correspond to the `binlog-do-db` and `binlog-ignore-db` filtering options in the server's *my.cnf* file. If you are filtering binary log records on the master, one or both of these will list the database names affected by your filters.

You can also ask the master which binary logs still exist on disk:

```
mysql> SHOW MASTER LOGS;
+----------------+
| Log_name       |
+----------------+
| binary-log.001 |
| binary-log.002 |
| binary-log.003 |
| binary-log.004 |
+----------------+
4 rows in set (0.02 sec)
```

But the output is quite limited. It would be helpful to know the sizes and ages of the files as well. MySQL is doing little more than reading and displaying the contents of the *log-bin.index* file. To get more information, you need to log on to the server and examine the files by hand.

Slave status

There is significantly more information available on the slave side of replication, mostly because the slaves have more information to keep track of. To start, the SHOW SLAVE STATUS command provides a good summary of the information from both the *master.info* and *relay-log.info* files:

```
mysql> SHOW SLAVE STATUS \G
*************************** 1. row ***************************
          Master_Host: master.example.com
          Master_User: repl
          Master_Port: 3306
        Connect_retry: 15
      Master_Log_File: binary-log.004
  Read_Master_Log_Pos: 635904807
       Relay_Log_File: relay-log.004
        Relay_Log_Pos: 846096118
Relay_Master_Log_File: binary-log.004
     Slave_IO_Running: Yes
    Slave_SQL_Running: Yes
      Replicate_do_db:
  Replicate_ignore_db:
           Last_errno: 0
           Last_error:
         Skip_counter: 0
  Exec_master_log_pos: 635904807
      Relay_log_space: 846096122
1 row in set (0.00 sec)
```

In addition, there is some other metadata in the output. The Last_errno and Last_error fields provide information about the most recent replication-related error, if any. The Relay_log_space tells you how much space the relay log is consuming.

The two most important fields are Slave_IO_Running and Slave_SQL_Running. They tell you if the IO and slave threads are running.

Replication heartbeat

Watching the values produced by SHOW MASTER STATUS and SHOW SLAVE STATUS can give you a rough idea of how up to date a slave is. The trouble with relying on that information is that you're only looking at bytes. You can determine how many more bytes of log data the slave needs to execute before it is current. However, that doesn't tell you how many queries need to be executed. There's no good way to figure that out, short of running the binary log through *mysqlbinlog* and counting queries.

It is possible to determine how out of date the slave is with some degree of accuracy by implementing a simple heartbeat system. The heartbeat principle is easy. At a fixed interval, say 20 seconds, a process on the master inserts a record with the latest timestamp into a table. On the slave, a corresponding process reads the most recent record every 20 seconds. Assuming that the system clocks on both machines are in sync, you can tell how far behind the slave is to within 20 seconds of accuracy.

See the *write_heartbeat* and *read_heartbeat* scripts in the "Tools" section for a sample implementation.

Log Rotation

Binary log files accumulate on the server until they are explicitly removed. The `SHOW MASTER LOGS` command tells you how many logs there are at any given time. To remove one or more logs, use the `PURGE MASTER LOGS TO ...` command. It removes all the logs up to but not including the given log name.

Here's an example:

```
mysql> SHOW MASTER LOGS;
+----------------+
| Log_name       |
+----------------+
| binary-log.001 |
| binary-log.002 |
| binary-log.003 |
| binary-log.004 |
+----------------+
4 rows in set (0.02 sec)

mysql> PURGE MASTER LOGS TO 'binary-log.004';
```

The command tells MySQL to remove *binary-log.001*, *binary-log.002*, and *binary-log.003*. Be careful not to remove logs too quickly. If a slave is offline for a significant period of time, there's a chance that it still needs one or more of the logs you removed. If you're in doubt, run `SHOW SLAVE STATUS` on each slave to verify which log it is using.

To automate this process, see the `purge_binary_logs` script in the "Tools" section.

Changing Masters

Sooner or later you'll need to point your slaves at a new master. Maybe the old one is being replaced with a newer, faster computer; perhaps there was a failure, and you are promoting a slave to master. In MySQL 3.23 and 4.0, you need to inform the slaves about their new master. A future version of MySQL is supposed to include a fail-safe replication feature that automates the process.

A planned changing of masters is a straightforward process. (In the event of a master failure, it may not be so easy.) You simply need to issue the `CHANGE MASTER TO ...` command on each slave. In doing so, you inform the slave of the new master's parameters—the same ones specified in the *my.cnf* file. The slave will begin replicating from its new master, and MySQL will also update the *master.info* with the new information.

Using the right values

As usual, the devil is in the details. How do you decide which values to use? What if you get them wrong?

First, let's consider the easy case. If you are in control of the situation, the process is easy. Follow these steps:

1. Disconnect all clients (not slaves) from the master.
2. Make sure the new master is completely caught up.
3. Execute `RESET MASTER` on the new master.
4. Make sure each slave is caught up.
5. Shut down the old master.
6. Let all clients connect to the new master.
7. Issue a `CHANGE MASTER TO ...` command on each slave, pointing it to the new master.

The `RESET MASTER` command tells the master to flush all its binary logs and start fresh. By starting with a clean slate on the new master, there's no guesswork involved in determining the right log position. Since it's a brand new log, we know the position is 4, because each binary log has a 4-byte header that consumes positions 0–3.

The complete `CHANGE MASTER TO ...` command looks like this:

```
mysql> CHANGE MASTER TO
    -> MASTER_HOST='newmaster.example.com',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='MySecret!',
    -> MASTER_PORT=3306,
    -> MASTER_LOG_FILE='log-bin.001',
    -> MASTER_LOG_POS=4;
```

If, on the other hand, the master crashes and you can't bring it back online in a reasonable amount of time, things aren't so clear-cut. If you have only one slave, of course, there's no decision to make. You use the slave. But if you have multiple slaves, you need to determine which one is the most up to date.

By examining the output of `SHOW SLAVE STATUS` on each slave, you can easily determine which one is closest to matching the master at the time it crashed. Once you know the log name and position, you can construct a `CHANGE MASTER TO ...` command to run on the remaining slaves.

In doing so, however, you'll likely cause some the slaves to be slightly out of sync with their new master. To illustrate why, assume that each query is assigned an increasing unique ID number. The original master had just executed query 500 when it crashed. The "most up-to-date" slave, the new master, had executed query 496. That means that your best slave is missing four queries, and there's no way to recover them unless your application logs every query it writes, which is unlikely.

Now, let's assume that there are two more slaves, *slave2* and *slave3*; *slave2* executed query 490, and *slave3* executed query 493. You have a choice. You can either point both slaves at the new master's current position (query 496) or you can try to figure the corresponding offsets for each slave in the new master's binary log. That will take more time, but it means you lose less data.

To find the matching log position for each slave, you need to have the binary log enabled on each slave. Use the *mysqlbinlog* command (described in the "Tools" section) to locate the last query executed. Then locate exactly the same query in the new master's binary log. Once you find the query, you'll have the offset you need. The output of *mysqlbinlog* always includes the offset in a comment right before the query. For example:

```
$ mysqlbinlog log-bin.001
...
# at 683
#021103 18:36:33 server id 1  log_pos 683   Query  thread_id=288 exec_time=0
error_code=0
SET TIMESTAMP=1036377393;
insert into test1 values (8);
```

The `# at 683` line lists the position of the `insert into test1 values (8)` query in the log.

Tools

In this section, we'll look at some tools that can make dealing with replication a bit easier. A couple of the tools come straight out of the MySQL distribution, while others are home-grown and often ripe for improvement. The home-grown tools can serve as a starting point for solving your specific needs; such tools are available (and kept up to date) at *http://highperformancemysql.com*.

mysqlbinlog: Viewing data in logs

The *mysqlbinlog* utility has been mentioned several times in this chapter. It is used to decode the binary formats used by the binary log and relay log. Given a log file, it outputs the SQL queries contained in the log. Furthermore, it precedes each query with several pieces of metadata as comments.

```
$ mysql log-bin.001
...
# at 683
#021103 18:36:33 server id 1  log_pos 683   Query  thread_id=288 exec_time=0
error_code=0
SET TIMESTAMP=1036377393;
insert into test1 values (8);
```

The first line contains the offset (or position) of the query in the log. The second line begins with a date and timestamp followed by the server ID of the server that first

executed the query. The log position is repeated on this line and followed by the event type.

Finally, there's the ID of the thread that executed the query, followed by the time the query took to execute (in seconds) and the error code generated by the query.

You can use *mysqlbinlog* to pull the logs from a remote server by specifying a hostname, username, and password. Using the `-o` command-line option, you can specify the offset from which you'd like to start reading. For example:

```
$ mysqlbinlog -h slave3.example.com -u root -p -o 35532 log-bin.032
```

check_repl: Ensuring that replication takes place

As discussed earlier, it's important to check that your slaves are replicating properly when you expect them to. The following script connects to the local MySQL server and makes sure that replication is running by examining the output of `SHOW SLAVE STATUS` and checking for the both the 3.23.xx and 4.x values:

```
#!/usr/bin/perl -w

## On a slave server, check to see that the slave hasn't stopped.

use strict;
use DBIx::DWIW;

my $conn = DBIx::DWIW->Connect(
    DB      => "mysql",
    User    => "root",
    Pass    => "password",
    Host    => "localhost",
) or exit;

my $info = $conn->Hash("SHOW SLAVE STATUS");

if (exists $info->{Slave_SQL_Running} and $info->{Slave_SQL_Running} eq 'No')
{
    warn "slave SQL thread has stopped\n";
}
elsif (exists $info->{Slave_IO_Running} and $info->{Slave_IO_Running} eq 'No')
{
    warn "slave IO thread has stopped\n";
}
elsif (exists $info->{Slave_Running} and $info->{Slave_Running} eq 'No')
{
    warn "slave has stopped\n";
}
```

This script makes no effort to repair a problem; it simply reports when something is wrong. Without knowing why the failure occurred, it's probably not wise to blindly restart replication. To skip the problem query and restart replication, see the next section.

fix_repl: Skipping a bad query to continue replication

In the event that replication has stopped on a slave, you should tell the slave to skip the problem query and continue, unless the problem warrants further investigation. No restart of MySQL is necessary.

In MySQL 3.23.xx, execute:

```
SET SQL_SLAVE_SKIP_COUNTER=1
SLAVE START
```

In Versions 4.0.0–4.0.2, execute:

```
SET SQL_SLAVE_SKIP_COUNTER=1
SLAVE START SQL_THREAD
```

In Version 4.0.3 and beyond, execute:

```
SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1
SLAVE START SQL_THREAD
```

Yuck. If you're using a mixture of 3.23.xx and 4.0.x servers, it may be difficult to remember the exact syntax for each version. It's much easier to have a copy of the following *fix_repl* script on hand to do the hard work for you:

```
#!/usr/local/bin/perl -w
#
# fix mysql replication if it encounters a problem

$|=1;      # unbuffer stdout

use strict;
use DBIx::DWIW;

my $host = shift || 'localhost';
my $conn = DBIx::DWIW->Connect(
    DB   => "mysql",
    User => "root",
    Pass => "pa55word",
    Host => $host,
) or die "Couldn't connect to database!";

print "checking $host ... ";

my $info = $conn->Hash("SHOW SLAVE STATUS");
my $version = $conn->Scalar("SHOW VARIABLES LIKE 'Version'");
my $fix_cmd;
my $start_cmd;

# 3.23
if ($version =~ /^3\.23/ and $info->{Slave_Running} eq 'No')
{
    $fix_cmd = "SET SQL_SLAVE_SKIP_COUNTER = 1";
    $start_cmd = "SLAVE START";
}
```

```
# 4.0.0 - 4.0.2
elsif ($version =~ /^4\.0\.[012]/ and $info->{Slave_SQL_Running} eq 'No')
{
    $fix_cmd = "SET SQL_SLAVE_SKIP_COUNTER = 1";
    $start_cmd = "SLAVE START SQL_THREAD";
}

# 4.0.3 - 4.0.xx, 4.1.xx.  Don't know what 5.0 will be like.
elsif ($version =~ /^4\.[01]\./ and $info->{Slave_SQL_Running} eq 'No')
{
    $fix_cmd = "SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1";
    $start_cmd = "SLAVE START SQL_THREAD";
}

# things are okay or unknown version?
else
{
    print "GOOD\n";
    exit;
}

print "FIXING ... ";
$conn->Execute($fix_cmd);
$conn->Execute($start_cmd);
print "DONE\n";

exit;
```

Be careful with this technique. Blindly skipping queries on a slave may cause it to become out of sync with the master. If the query is failing due to a duplicate key error, it's probably safe, but you should investigate how that happened in the first place.

purge_binary_logs: Reclaiming space used by binary logs

To make log rotation easier, you can use something like the following *purge_binary_logs* Perl script. It connects to the specified server and checks to see how many binary logs are sitting around. If there are more than the specified threshold, it removes any extras.

```
#!/usr/bin/perl -w

## On a slave server, purge the replication logs if there are too many
## sitting around sucking up disk space.

use strict;
use DBIx::DWIW;

my $MIN_LOGS = 4; ## keep main log plus three old binary logs around

my $conn = DBIx::DWIW->Connect(
    DB   => "mysql",
    User => "root",
```

```
        Pass => "password",
        Host => 'localhost',
    );

    die "Couldn't connect to database!" if not $conn;

    ## see if there are enough to bother, exit if not
    my @logs = $conn->FlatArray("SHOW MASTER LOGS");
    exit if (@logs < $MIN_LOGS);

    ## if so, figure out what the last one we want to keep is, then purge
    ## the rest
    my $last_log = $logs[-$MIN_LOGS];
    print "last log is $last_log\n" unless $ENV{CRON};
    $conn->Execute("PURGE MASTER LOGS TO '$last_log'");

    exit;
```

Depending on your needs, there's a lot of room for improvement in this script. It would be nice if the script took command-line arguments so you wouldn't need to hardcode the hostname, password, and so on. It would also be nice if the script could check the sizes of the log files. If a master is restarted very frequently, using the number of log files as a metric probably isn't as useful as checking the volume of log data. However, the script can't be run remotely if it checked log file sizes, because it needs to examine the files directly.

A valuable but difficult addition would be for the script to remove logs only if it can tell that all slaves had already read them. That requires knowing all the slaves and contacting each one to verify its progress in the replication process.

mysqldiff: Replication sanity checks

As with anything new, you may not trust replication right away. To help convince yourself that it is really doing what it should do, it's good to perform spot checks on the data, making sure that the slaves have exactly the data they should have.

This checking can be done to varying degrees of paranoia:

- Simple metadata checks: make sure each table on the slaves contains the same number of rows that the same master table does.
- Verify all or some of the data by comparing rows on the master and slaves.
- Perform application-specific checks by running custom queries and comparing the results across servers.

The first check is quite easy to implement with a bit of Perl code:

```
#!/usr/bin/perl -w

use strict;
use DBIx::DWIW;
```

```
$|=1;       # unbuffer stdout

my $db_user = 'root';
my $db_pass = 'password';
my $db_name = 'test';
my $master  = 'master.example.com';

my @slaves = qw(
    slave1.example.com
    slave2.example.com
    slave3.example.com
);

my %master_count;

for my $server ($master)
{
    print "Checking master... ";
    my $conn = DBIx::DWIW->Connect(User => $db_user, Host => $server,
        Pass => $db_pass, DB => $db_name) or die "$!";

    for my $table ($conn->FlatArray("SHOW TABLES"))
    {
        my $count = $conn->Scalar("SELECT COUNT(*) FROM '$table'");
        $master_count{$table} = $count;
    }
    print "OK\n";
}

for my $server (@slaves)
{
    print "Checking $server... ";
    my $conn = DBIx::DWIW->Connect(User => $db_user, Host => $server,
        Pass => $db_pass, DB => $db_name) or die "$!";

    for my $table ($conn->FlatArray("SHOW TABLES"))
    {
        my $count = $conn->Scalar("SELECT COUNT(*) FROM '$table'");

        if ($count != $master_count{$table})
        {
           print "MISMATCH (got $count on $table, expecting $master_count{$table}\n";
        }
    }
    print "OK\n";
}

exit;
```

The script connects to the master and gets the number of rows in each table of the given database. Then it connects to each slave and checks to see that the counts match. If they don't, it issues a `MISMATCH` warning.

This framework can easily be extended to handle multiple databases, perform more specific checks, and even attempt to take corrective action. It is even ready to handle multiple masters.

write_heartbeat: Generating a periodic health check heartbeat

The following script can implement a heartbeat monitoring system as described earlier. To use it, create a database named `MySQL_Admin` and a table named `Heartbeat` with the following structure:

```
CREATE TABLE Heartbeat
(
    unix_time    INTEGER   NOT NULL,
    db_time      TIMESTAMP NOT NULL,
    INDEX        time_idx(unix_time)
)
```

The `unix_time` field holds the timestamp that is explicitly inserted into the table. The `db_time` field is set automatically by MySQL. By keeping track of both times and inserting new records instead of simply running an `UPDATE` on a single record, you maintain historical data in the event someone wants to graph or analyze it.

Let's look the script to add records:

```
#!/usr/bin/perl -w

use strict;
use DBIx::DWIW;

my $conn = DBIx::DWIW->Connect(
    DB   => "MySQL_Admin",
    User => "root",
    Pass => "password",
    Host => 'localhost',
) or die;

my $unix_time = time();
my $sql = "INSERT INTO Heartbeat (unix_time, db_time) VALUES ($unix_time, NULL)";
$conn->Execute($sql);

exit;
```

Running the script at a fixed interval generates a heartbeat that can be used by the `read_heartbeat` script to monitor replication latency.

read_heartbeat: Measuring replication log using heartbeat

The companion to *write_heartbeat* reads the most recent timestamp from the database and computes how far behind the slave might be. Remember that we can't know this time exactly unless the heartbeat records are generated every second, which is probably overkill for most installations.

```
#!/usr/bin/perl -w

use strict;
use DBIx::DWIW;

my $conn = DBIx::DWIW->Connect(
    DB   => "MySQL_Admin",
    User => "root",
    Pass => "password",
    Host => 'localhost',
) or die;

my $sql = "SELECT unix_time, db_time FROM Heartbeat
           ORDER BY unix_time DESC LIMIT 1";

my $info = $conn->Hash($sql);
my $time = $info->{unix_time};
my $delay = time( ) - $time;

print "slave is $delay seconds behind\n";

exit;
```

This script can also be extended to do far more than report on latency. If the latency is too great, it can send email or page a DBA.

Common Problems

Breaking MySQL's replication isn't difficult. The same simple implementation that makes it easy to set up also means there are many ways to stop, confuse, and otherwise disrupt it. In this section, we'll look at common problems, how they manifest themselves, and what can be done to solve or even prevent them.

Slave Data Changes

It should go without saying that manually changing data on a slave is usually a very bad idea. The same holds true for programmatically changing slave data. By accidentally making changes to data on a slave, you can easily introduce data inconsistencies that may cause replication to fail. It may take hours, days, weeks, or even months for the problem to surface, and when it does, you'll be hard pressed to explain what's going on.

Before MySQL 4.0.14 there was no way to tell MySQL not to allow any changes that don't originate from replication. Instead, the best solution in versions prior to 4.0.14 has an ironic aspect to it: you need to make a change on all the slaves, removing the permissions (or even the accounts) of users who can change data.

But that solution is problematic for other reasons. You'd probably forget about the change after a while. Then, late one night, the master would fail and you would need

to promote a slave to master. You'd have to spend a bit of time trying figure out why applications are mysteriously failing.

As of Version 4.0.14, adding `read-only` to the slave's configuration file allows the slave to process write queries only via replication.

It's worth remembering that MySQL is very trusting when it comes to replication. The slave threads don't switch identities to run each query as the same user that originally executed it on the master. Instead, the slave thread runs with the equivalent of root access on the slave. It can, by design, change any data it needs to change. The trust comes from the fact that the slaves never verify that a particular user has the necessary privileges to run a query that appears in the binary log. It blindly trusts the master and that the master's logs haven't been tampered with.

Nonunique Server IDs

This has to be one of the most elusive problems you can encounter with MySQL replication. If you accidentally configure two slaves with the same server ID they'll appear to work just fine if you're not watching closely. But if you watch their error logs carefully or watch the master with *mytop* (covered in Appendix B), you'll notice something very odd.

On the master, you'll see only one of the two slaves connected at a given moment. Usually all slaves are connecting and replicating all the time. On the slave you'll see frequent disconnect/reconnect messages appearing in the error log, but none of those messages will lead you to believe that the server ID of one slave might be misconfigured.

The only real harm in this situation is that the slaves can't replicate very quickly. Because the slaves (not the master) keep track of their replication progress, there's no need to worry about giving one query to the first slave, one to the other slave, and so on. Both slaves get all the data; they just get it much more slowly.

The only solution to this problem is to be careful when setting up your slaves. If you see symptoms like this, double check the configuration of each slave to ensure that it has the server ID you expect it to. You may find it helpful to keep a master list of slave-to-server-ID mappings so that you don't lose track of which ID belongs to each slave. Consider using numeric values that have some sort of meaning in your setup, such as the last octet of each machine's IP address.

Log Corruption or Partial Log Record

The second most elusive problem occurs when a binary log somehow becomes corrupted. When that happens, the slave will typically fail with an error message like:

```
Error in Log_event::read_log_event(): '...', data_len=92,event_type=2
```

If that ever happens, there's little you can do. The slave is often confused enough that you can't simply try to skip the query and go to the next one. The only solution is to resync with the master and start over.

How does this happen? It's difficult to say. As long as the software is working properly, it could be a hardware or driver problem. Jeremy once saw a system have this problem repeatedly before he found that it had faulty RAM installed. We have heard of it happening on systems with disk controllers that don't have reliable drivers.

Bulk-Loading Data

While you can write code to load a lot of data into MySQL quickly, nothing beats the performance of using MySQL's `LOAD DATA INFILE` and `LOAD DATA LOCAL INFILE` commands to read data in from a flat file. In fact, the *mysqlimport* command-line tool uses `LOAD DATA INFILE` behind the scenes.

In all 3.23.xx versions of MySQL, replicating the `LOAD DATA INFILE` command is problematic. The contents of the file aren't stored in the binary log; only the query is, so the file must exist on the master until all slaves have copied it (they will do so automatically when they need it). If the file is removed prematurely, slaves can't copy the file, and replication will fail.

The `LOAD DATA LOCAL INFILE` command isn't affected. When the `LOCAL` option is specified, the *mysql* client reads the file from the client and generates the appropriate SQL to insert the data.

To avoid this problem, it's best either to load the data remotely using the latter syntax or to import the data programmatically. Either option ensures that the inserting is done via normal SQL statements that will all be properly logged.

Starting with Version 4.0, MySQL doesn't have this limitation. When a `LOAD DATA INFILE` command is issued, MySQL actually copies the entire file into the binary log. Slaves don't need to pull a copy of the original file from the master's disk.

Nonreplicated Dependencies

If you perform binary log filtering on either the master or the slave, it's quite easy to inadvertently break replication. For example, you may want to have a production database called `production` and a staging database called `staging`. The idea is to do all the necessary testing, development, and retesting in the staging database. When all the interim work is complete, you copy the data into the production database.

If the slave ignores queries from the staging database because of a filtering rule like the following, you'll probably end up frustrated:

```
replicate-do-db = production
```

You might try to run a query like this one to populate one of the production tables:

```
INSERT INTO production.sales SELECT * FROM staging.sales
```

This query works fine on the master, but the slaves will all fail because they don't have copies of the staging database. In fact, there's no easy way to make it work. Any attempt to reference the staging database is doomed to fail.

The only real solution in a case like this is to export all the data from the staging database and import it into the production database. You can do this programmatically if you want fine control over the process, or you can simply use *mysqldump* to dump the data to a text file and reimport it using *mysql*.

Missing Temporary Tables

This is really a special case of the previous example, but it warrants special attention because the real cause is a bit different. Instead of a filtering problem, this is a problem of restarting the slave at the wrong time.

Temporary tables replicate just fine, but if a series of queries that create and use a temporary table are interrupted on a slave by a restart or by stopping and starting replication, replication will fail.

Temporary tables are, by definition, temporary. When the server is restarted, they vanish. When the thread vanishes (such as with a `SLAVE STOP` or `SLAVE STOP SQL_THREAD` command), any temporary tables created by that thread vanish.

There is no good solution for this problem. On the application side, it's best if temporary tables are created as late as possible, which helps minimize the time between the creation of the table and when it is actually needed. But even this solution only decreases the likelihood of the problem occurring.

You can avoid temporary tables completely, but that may involve time-consuming application changes. You'd have to ensure that the nontemporary tables created by your application always have unique names and that they are dropped when appropriate.

Because they are transient, this problem also affects Heap tables. They are always dropped explicitly, however so they vanish only when a slave is restarted. Stopping and restarting replication on the slave doesn't affect Heap tables.

Binary Log Out of Sync with Transaction Log

We know that MySQL records queries in the binary log after it executes them. We also know that MySQL writes transactions to the binary log after they have been committed. What happens if MySQL crashes, or someone pulls the plug in the microseconds after a transaction has been committed but before it writes the transaction to the binary log?

The result is that the master will contain the results of having completed the transaction, but the slaves will never see it. Ever. The transaction may have been a simple insert, or it could have been something as dramatic as a `DROP TABLE` command.

There is currently no workaround for this problem. Luckily MySQL crashes are rare. Make sure the power cables are plugged in tightly!

Slave Wants to Connect to the Wrong Master

If you change the hostname of your master, it's important to tell slaves using the `CHANGE MASTER` command:

```
mysql> CHANGE MASTER TO MASTER_HOST='newmaster.example.com';
```

You can't simply shut down the slave, edit the *my.cnf* file, and start it back up. MySQL always uses the *master.info* file if it exists, despite the settings contained in the *my.cnf* file.*

Alternatively, you can manually edit the *master.info* file, replacing the old hostname with the new one. The danger in relying on this method is that the *master.info* file can be deprecated, replaced, or radically changed in a future version of MySQL. It's best to stick to the documented way of doing things.

The Future of Replication

To solve some of MySQL's current shortcomings and to provide the infrastructure for handling problems that MySQL hasn't yet seen, a number of future enhancements have been proposed for MySQL. Let's look at them briefly.

Eliminating the Snapshot

With MySQL's current implementation, it's difficult to add a slave to a master after the master has been running for a long period of time. Many of the original binary logs have probably been removed to save space. Without all the logs, you can't simply configure the slave and point it at the master.

Even if you have all the binary logs on the master, it may take days, weeks, or even months for a slave to execute all the queries and finally catch up to the master. If you're looking to add slaves in a hurry, this clearly isn't the way to do it.

In either case, the ideal solution is simply to configure the new slave and tell it to begin replicating. Behind the scenes, the slave contacts the master and requests copies of the all the tables it needs, probably using a mechanism similar to `LOAD TABLE`

* This is, in my opinion, an easy-to-fix bug, but the MySQL maintainers don't agree. The workaround is to always use the `CHANGE MASTER TO` command for configuring slaves.

FROM MASTER. The master will need a way to track all changes to tables between the time that the slave begins and finishes copying the tables. Upon completion of the copy, the slave receives all the necessary changes and begins replicating from the binary log.

An alternative is for all of MySQL's storage engines to implement a versioning scheme similar to InnoDB's. When a new slave connects and begins to copy the tables, it can get a snapshot from that moment in time. When the copy is complete, the slave can begin replicating from the binary log position corresponding to the moment when the snapshot was marked.

Fail-Safe Replication

When a master fails, you must select a new master and instruct all the slaves to connect to the new master and begin replicating. Not only is that process prone to errors, it can be time-consuming too. Ideally, MySQL should handle failover automatically.

The proposed solution involves each slave registering itself with the master so that the master can keep track of it. Not only will the master know which servers are slaves, it can also keep track of how up to date each slave is. The slaves, in turn, will also keep track of who all the other slaves are.

In the event that the master fails, the slaves can elect a master based on the available information. Ideally, they will find the slave that was the most up to date when the master went down.

Safe Multi-Master Replication

Today it's possible to use replication in a multi-master architecture, as depicted earlier (see Figure 7-3). The major drawback to doing so, however, is that you can't rely on AUTO_INCREMENT columns to function properly.

Each MyISAM table has a single counter that controls the next AUTO_INCREMENT value. Once that value has increased, it can't easily be decreased. If inserts are timed properly, they cause data to become inconsistent between the two masters.

Imagine the following events occurring on two servers, *master1* and *master2*:

1. Both servers start with an empty orders table.
2. *master1* inserts a record for customer 58, which is assigned ID 1.
3. *master2* inserts a record for customer 1232, which is assigned ID 1.
4. *master2* replicates *master1*'s insert, adding the record for customer 58 and trying to assign it an ID of 1. That fails and results in a duplicate key error.
5. *master1* replicates *master2*'s insert, adding the record for customer 1232 and trying to assign it an ID of 1. That fails and results in a duplicate key error.

Each master was given an insert by some client before it had replicated the other master's insert. The result is that both masters are out of sync.

The current solution is to avoid using AUTO_INCREMENT fields completely and assign primary keys through some other means. You might use an MD5 hash of some values in the record, or perhaps use another library to generate a globally unique identifier (GUID).

Let's look at the two proposed solutions for the future.

Multipart auto-increment unique keys

The first is to use MyISAM's multipart auto-increment unique keys. Rather than using a single column as a primary key, you'd set up a table like this:

```
CREATE TABLE orders (
    server_id       INTEGER UNSIGNED NOT NULL,
    record_id       INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    stuff           VARCHAR(255)     NOT NULL,
    UNIQUE mykey (server_id, record_id)
);
```

Notice that the record_id is an AUTO_INCREMENT field and is the second part of a two-part unique key. When you insert NULL into the record_id column, MySQL will consider the value of server_id when automatically generating a value.

To illustrate this, notice the following:

```
mysql> insert into orders values (1, NULL, 'testing');
Query OK, 1 row affected (0.01 sec)

mysql> insert into orders values (1, NULL, 'testing');
Query OK, 1 row affected (0.00 sec)

mysql> insert into orders values (2, NULL, 'testing');
Query OK, 1 row affected (0.00 sec)

mysql> select * from orders;
+-----------+-----------+---------+
| server_id | record_id | stuff   |
+-----------+-----------+---------+
|         1 |         1 | testing |
|         1 |         2 | testing |
|         2 |         1 | testing |
+-----------+-----------+---------+
3 rows in set (0.03 sec)
```

MySQL, in effect, allows you to select from multiple AUTO_INCREMENT sequences based on the prefix you use. By adding a function such as SERVER_ID( ) to MySQL and rewriting the previous queries, you can use AUTO_INCREMENT with multi-master replication safely.

```
mysql> insert into orders values (SERVER_ID( ), NULL, 'testing');
Query OK, 1 row affected (0.01 sec)
```

```
mysql> insert into orders values (SERVER_ID(), NULL, 'testing');
Query OK, 1 row affected (0.00 sec)

mysql> insert into orders values (SERVER_ID(), NULL, 'testing');
Query OK, 1 row affected (0.00 sec)
```

There are three problems with this approach. First, it works only for MyISAM tables. An ideal solution works across all table types. Another issue is that all slaves require some special logic. Today, when a slave reads the binary log of a master, it knows the master's server ID as well as its own, but it doesn't really do anything with the master's server ID. In this solution, the slave has to actually use the master's server ID any time that it replicated a query that involved the mythical SERVER_ID( ) function. That makes the replication logic a bit trickier on the slaves.

You could work around the lack of a SERVER_ID( ) function by simply using the actual server ID in your SQL statements. If you know you're talking to server 12, write the query accordingly:

```
mysql> insert into orders values (12, NULL, 'testing');
Query OK, 1 row affected (0.01 sec)
```

But there's the rub. You need to know, in advance of each query, what the server's ID is. Granted, the server's ID doesn't change, but if you're accessing one of many servers via a load balancer or don't have a persistent connection, the server you're talking to may change often. So you'd have to deal with the overhead of obtaining the server's ID whenever you need it.

```
mysql> show variables like 'server_id';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| server_id     | 102   |
+---------------+-------+
1 row in set (0.00 sec)
```

Finally, and most importantly, using two columns as the primary key just doesn't feel natural. It feels like a hack or a workaround. If this solution became widespread, others problems might arise. For example, setting up foreign-key relationships would be troublesome. Putting aside the fact that InnoDB doesn't even support multipart auto-increment unique keys, how would you define a foreign-key relationship with multipart keys?

Partitioned auto-increment fields

The second solution is to make auto-increment fields a bit more complex. Rather than simply using a 32-bit integer that starts at 1 and keeps counting, it might make sense to use more bits and partition the key-space based on the server ID. Currently, server IDs are 32-bit values, so by using a 64-bit auto-increment value, the two can be combined. The high 32 bits of the value would be the server ID of the server that

originally generated the record, and the low 32 bits would be the real auto-increment value.

Internally, MySQL needs to treat the 64-bit auto-increment value a lot like the multipart auto-increment unique keys previously discussed. The value generated for the low 32 bits is dependent on the value of the high 32 bits (the server ID). The benefit is that from the user's point of view, it's a single column and can be used just like any other column. Insert statements are no more complex; all the magic is conveniently under the hood, where it belongs.

There are some downsides to this approach, however. The most apparent issue is that there would be large gaps in the values. For the sake of simplicity, MySQL can always subtract 1 from the server ID when generating the high bits of the auto-increment value. This allows values to continue starting at 1 when the server ID is 1. However, as soon as a second server is introduced, with server ID 2, it inserts values starting from 4,294,967,297 ($2^{32} + 1$) and counting up from there.

Another problem is that columns will require more space on disk (both in the data and index files). `BIGINT` columns are already 8 bytes (64 bits) wide. Adding another 4 bytes (32 bits) for the server ID portion of the auto-increment value means a 50% increase in the space required. That may not sound like a lot, but an application that requires 64-bit values in the first place is likely to be storing billions of rows. Adding an additional 4 bytes to a table containing 10 billion rows means storing an additional 40 GB of data!

It makes sense to break compatibility with existing MySQL versions (which use 32-bit server IDs) and reduce the size of the server ID to 8 or 16 bits. After all, with even 8 bits available, you can have up to 255 unique servers in a single replication setup; with 16 bits, that jumps to 65,535. It's unlikely anyone will have that many servers in a single replication setup.*

* Perhaps Google will decide to run MySQL on their growing farm of 100,000+ Linux servers. They'd need more than 8 bits.