MySQL SERVER PERFORMANCE TUNING

Making sure your **MySQL** server flies

In the open source world, truly great software starts in the hands of enthusiasts and hobbyists. Given time, it matures and develops a more robust community. Then, before most of us realize what is happening, it gains critical mass and moves into the broader industry. Companies that were using expensive commercial software just a year ago are suddenly using a free product — one of the rising stars from the world of Open Source.

We're all familiar with software products that have followed that pattern recently: Linux, Sendmail, Perl, Apache, and so on. Few people who have worked with MySQL will tell you that it is any different. MySQL is becoming an increasingly popular choice for building business-class database applications on Linux.

As a result of MySQL's growing role in larger organizations, its use is becoming more high-profile. This means, of course, that MySQL needs to provide responsiveness, high performance, and reliability. Already known in the industry for being a lightning-fast database server, MySQL is often up to the task straight out of the box. However, there are quite a few things that can easily slow it down. Sometimes it's the result of poor application design. Sometimes MySQL's default configuration simply isn't good enough for the task at hand. And sometimes all you need to do is throw a little more hardware at the problem.

In the June issue (http://www.linux-mag.com/2001-06/mysql_01.html) we looked at the first case: performance tuning from an application point of view. When trying to speed up a database applica-

BY JEREMY ZAWODNY

tion, it's always best to start with the application itself and make sure that the tables are properly normalized, columns are indexed, and queries are fast. But if you've done all that and things are still slow, it's time to look at the MySQL server itself.

This time around, let's begin by looking at how to tune a MySQL server to increase performance under more demanding workloads.

IT'S NOT ROCKET SCIENCE

While it may sound impressive, performance tuning is simply about squeezing as much performance as possible out of your system. The way we'll do this is by understanding the variables that are involved and learning how they are likely to affect performance.

Before diving into the details, it's worth reiterating an important fact: the techniques we'll look at will not "fix" poorly written or un-optimized queries, poor database design, or other application design problems. They *may* help alleviate the stress on a busy server, but you're just postponing the inevitable. The *only* solution for a poorly written application or poor database design is to go to the source and fix it.

Really — fixing slow queries and/or a poorly designed application will generally yield much better results than spending time on server tuning.

If you're not sure where to start, enable MySQL's *slow query log* as explained in the manual (http://www.mysql.com/doc/S/l/Slow_query_log.html). Then just watch for any query that you don't normally expect to be "slow" and figure out why that is. And, of course, make sure that any queries that are "slow" are run infrequently. It will be time well spent.

You may find that some fast queries appear in the slow query log, too. This is because MySQL considers any query "slow" if it does not use an index. This means it would call the query:

```
SELECT * FROM us_states
```

"slow" even though there are probably only 50 rows in the table. Indexing simply can't help such a query, but MySQL really doesn't know that.

We'll look at a few specific methods for speeding up MySQL. While this isn't a comprehensive look (that would require an entire book!), we'll examine some of the factors that typically give you the most "bang for your buck" when performance is suffering.

MEMORY USAGE

On the server side, the single most important factor in determining how well MySQL will perform is memory. It's not enough to simply have a lot of memory available. You *need* to tell MySQL precisely how you would like it to use that memory. MySQL's default settings are rather conservative for today's hardware. In fact, if you have a dedicated MySQL server with several hundred megabytes (or a few gigabytes) of RAM, you'll be able to give MySQL quite a large portion of it to work with. By default, it will only use a small fraction of what is available; this is because it has no way of knowing if it is running on a dedicated server where it will be used continuously or if it's running on an already stressed laptop where it'll only be used to hold an address book.

Much of our discussion will focus on memory usage and assume that you are using MySQL's default table type — MyISAM. If you are using one of the more advanced transactional table types (InnoDB or Gemini), please see the

MEMORY MANAGEMENT FOR NON-MyISAM TABLES

The more advanced table handlers (such as InnoDB or Gemini) allocate their own global memory areas that are completely separate from MySQL's key buffer, table cache, and similar variables. As a result, when you use either of these table handlers, you must decide just how much memory you're willing to commit to them.

These memory areas are called *buffer pools* rather than simply "buffers" because they serve as a combined cache for both index and record data. Whenever you are using InnoDB tables, for example, you should have a line like this in your `my.cnf` file:

```
set-variable = innodb_buffer_pool_size=256M
```

This line simply tells the InnoDB table handler that it may use up to 256 MB of memory for its index and record cache.

Because the buffer pool will cache both record *and* index data, it's likely that you'll need to choose a buffer pool setting which is larger than the key buffer that you might have used for MyISAM tables. It is common to use as much as 70 percent (or more) of the available memory for InnoDB's buffer pool.

If you find yourself making heavy use of two or more table types, you should use extra caution when allocating memory. Begin with conservative numbers and gradually increase them after you are familiar with MySQL's performance under your typical workloads. Few of MySQL's buffers are shared across table handlers.

USING A MY.CNF FILE

Like all good software, MySQL comes with sample configuration files to get you started. But MySQL doesn't require a `my.cnf` file to operate; it will simply use the built-in defaults if it cannot find one. As a result, many people who use MySQL never realize that there is actually a configuration file.

When you need to perform testing and tuning, it is easiest to make changes to a configuration file and restart the server. You can make all of the changes in this article simply by passing the right command-line arguments to *mysqld*, but that becomes more unwieldy as the number of arguments grows.

Rather that construct a configuration file from scratch, it's best to begin with one of the sample files that come with MySQL. If you use the binary version of MySQL or build your own from source, you will find the sample files in the `support-files` sub-directory.

The sample files are: `my-huge.cnf`, `my-large.cnf`, `my-medium.cnf`, and `my-small.cnf`. If MySQL was already installed on your system, you may already have a `my.cnf` file in `etc`.

Each of the four files is targeted at a different "sized" MySQL installation. By reading the comments in each of those files, you can get a good idea of which one most closely matches your need.

Once you've decided which file to use, simply copy it to either `/etc/my.cnf` or `my.cnf` in the data directory of your MySQL installation. Putting the file in `/etc` means that every MySQL client and server (on that machine) will read the settings.

Storing it in MySQL's data directory allows you to have several instances of MySQL installed on the same server. All you need to do is give each installation its own `my.cnf` file. In either case, MySQL's startup scripts will find the file automatically.

Memory Management for Non-MyISAM Tables sidebar, pg. 39, for additional information.

FILE AND DISK LAYOUT

Because MyISAM tables store records and indexes in normal files, the layout of the files can impact performance. If you have multiple disks in your server, it's a very good idea to put the most active databases on different disks and even different disk controllers.

If you happen to have software or hardware RAID on your server, putting MySQL's data directory on a RAID array can improve performance as well. RAID 0, 1, or 5 can boost read performance, while RAID 0 will also increase write performance. Also, if you can use SCSI rather than IDE disks, you will free up more of your server's CPU to deal with actually running MySQL.

THE FILESYSTEM

Similarly, the filesystem itself can affect how MySQL performs. With the number of alternative filesystems available for Linux today, it's hard to say which one(s) may work best for you. Some users have reported noticeable increases in MySQL performance after moving from the standard *ext2* filesystem to either *Reiserfs* or SGI's *XFS*, both of which are journaling filesystems. See the "Journaling Filesystems" article (http://www.linux-mag.com/2000-08/journaling_01.html) in our August, 2000 issue for more about the benefits of moving to a newer filesystem.

If you have the luxury of time, consider testing with a journaling filesystem (or two) using the latest 2.4.x Linux kernels. Not only are you likely to find that performance improves, you'll never have to wait hours for *fsck* to finish on your gigabytes of data after a crash.

If you do test on a 2.4.x Linux kernel, try to use a version later than 2.4.9. Early 2.4 kernels contained a Virtual Memory (VM) subsystem that could cause dramatic loss of performance during heavy MySQL use in some circumstances. The VM system has since been rewritten to alleviate most of those problems.

HOW MYSQL USES MEMORY

MySQL uses memory for a variety of internal buffers and caches that influence how often it must access files that reside on disk. The more often it has to wait for a disk to respond, the slower it will be. As fast as modern disk drives are, they're still an order of magnitude (or more) slower than RAM. And given the recent drops in memory prices, odds are pretty good that you can easily afford to add memory to a server if it will speed things up. Upgrading to faster disks should be a last resort.

MySQL's buffers and caches come in two flavors, global and per-thread:

GLOBAL: As its name suggests, these memory areas are allocated once and are shared among all of MySQL's threads. Two of the ones we'll look at are the *key buffer* and the *table cache*. Because these are shared buffers, the goal is to make them as large as possible (without unnecessarily taxing our resources).

PER-THREAD: These buffers allocate memory indivi-

dually to queries as they need to perform particular operations, such as sorting or grouping. Incidentally, most of MySQL's buffers are allocated on this per-thread basis. The per-thread buffers we'll be looking at are the *record buffer* and the *sort buffer*.

Let's first examine what function each of the buffers serves and how to set and inspect their values. Then we'll look at how to examine MySQL's performance counters and judge whether or not changes you make are having any significant impact.

KEY BUFFER

The key buffer is where MySQL caches index blocks for MyISAM tables. Anytime a query uses an index, MySQL will first check to see if the relevant index is in memory or not. The `key_buffer` parameter in your `my.cnf` file controls how large the buffer is allowed to get. Once the buffer is full, MySQL will make room for new data by replacing older data that hasn't been used recently. (See the *Using a my.cnf File* sidebar if you're not familiar with MySQL's configuration file.)

The size of the key buffer appears as `key_buffer_size` in the output of `SHOW VARIABLES`. With a 384 MB key buffer, you'd see:

```
| key_buffer_size              | 402649088
```

As a general recommendation, on a dedicated MySQL server, you should allocate somewhere between 20 percent and 50 percent of your RAM for MySQL's key buffer. If you have a gigabyte of memory, start with something like:

```
set-variable = key_buffer=128M
```

or even:

```
set-variable = key_buffer=256M
```

in your `my.cnf` file and see if you notice a difference. Odds are that you will.

If you were only allowed to adjust one parameter on your MySQL server, the key buffer would be the one to try. Indexes are so important to the overall performance of any database server that it's hard to go wrong with making more room in memory for them.

If you do not specify a size for the key buffer, MySQL will use its default size, which is in the neighborhood of 8 MB. Of course, it makes little sense to set the value for your key buffer too high. Doing so could potentially starve the operating system of memory that it needs for disk buffering and other tasks.

It might also be helpful to look at how much index data you have on disk. Simply find the size of all the `.MYI` files under MySQL's data directory:

```
$ du -sh */*.MYI
```

Knowing how much index data you have, you can better judge how much benefit you are likely to see from increasing the size of the key buffer beyond a certain point. If some of your index files belong to tables that are infrequently used, there is little point in making room for them. But it's clear that any large or medium-sized database will normally benefit from a larger key buffer.

TABLE CACHE

MyISAM tables are composed of three separate files on disk: the data file `tablename.MYD`, the index file `tablename.MYI`, and, lastly, the table definition (or schema) file named `tablename.frm`. In order to use a single table, MySQL actually needs to open all three files. The `.frm` file will be closed after it reads the schema, but the others will remain open. MySQL will not close them until it needs to. This avoids the overhead associated with opening and closing the files if the table is used frequently.

The files usually are not closed until one of the following events occurs:

1. The table has been explicitly closed via `FLUSH TABLES`.
2. The table has been dropped.
3. The server is being shut down.
4. The total number of open tables has reached the value of the `table_cache` parameter.

The last event can be particularly important if you have many tables that are often used across all your databases. The default value of MySQL's table cache is 64. So if you have a few hundred (or thousand!) tables that are actively used, MySQL is going to waste a lot of time and effort needlessly opening and closing those files.

Increasing the size of the table cache will certainly help in this situation. But you must be careful not to make the value too large. All operating systems have a limit on the number of open file descriptors a single process may have. Some also have limits on the total number of open file descriptors that a single user may have. If MySQL tries to open too many files, the operating system will refuse to allow it and MySQL will generate an error message in the error log. When in doubt, check OS limitations.

In extreme cases, it is possible to increase the number of available file descriptors via kernel configuration options. Also, keep in mind that even though MySQL on Linux appears as though it is many processes in the output of *ps* and *top*, it is actually one multi-threaded daemon. The open file descriptors are allocated by a single process and shared among all of its threads.

Unlike many of the other parameters, the table cache applies to all of MySQL's disk-based table types.

RECORD BUFFER

Whenever MySQL must sequentially scan a table (known as a "full table scan"), the thread performing the scan will allocate a record buffer for each table it must scan. This typically happens when MySQL decides it is more efficient to scan the table than to use an index for a query. It also happens when there simply is no index it could use.

A query such as:

```
SELECT *
    FROM table_a
    WHERE field4 = 'Linux'
```

will require a full table scan if `field4` is not indexed.

By increasing the value of `record_buffer` in your `my.cnf` file, you allow MySQL to read the table in larger chunks. This will likely reduce the number of disk seeks involved and make the scan significantly faster on a busy server.

However, you must be very careful with the size of the record buffer if you have a lot of clients which run queries that perform full table scans. Because the record buffer is allocated on a per-thread basis, you may end up in a situation where individual clients cause record buffers to be allocated at the same time. If the remaining memory is limited, you'll likely encounter swapping and dramatically reduced performance.

In version 3.23.41, a related setting was introduced — `record_rnd_buffer`. Like `record_buffer`, it is used in scanning a larger number of rows. The `record_rnd_buffer` is used for queries that result in an intermediate file-sort being performed as well as in some non-sequential record reads. Fortunately, if you don't set the size of `record_rnd_buffer`, it will default to the size of `record_buffer`.

SORT BUFFER

As its name implies, the sort buffer is used to answer queries that invoke sorting data — those with an `ORDER BY` clause in them. In addition, the sort buffer is used for queries that involve grouping data (those with a `GROUP BY` clause). Like the other buffers we've looked at, the sort buffer is relatively small by default. By adjusting the `sort_buffer` entry in your `my.cnf` file:

```
set-variable = sort_buffer=8M
```

you can often dramatically reduce the amount of time that is used for sorting large result sets.

The sort buffer appears as `sort_buffer` in the output of `SHOW VARIABLES`, such as:

```
| sort_buffer                | 8388600
```

The same caveat applies to the sort buffer as the record buffer. It's a buffer that MySQL allocates frequently and is allocated on a per-thread basis. So, increase it with caution on a server that runs a lot of concurrent queries.

GENERAL TUNING GUIDELINES

Before discussing how to measure or judge the effects of any changes you make, we need to briefly consider a common sense approach to tuning. There are a few things to keep in mind when you begin making and testing changes:

1. **Only change one parameter at a time.** Changes will not always result in the behavior you might expect. If you change too many things at once, you risk attributing a change in behavior to the wrong parameter.

2. **Don't make changes in production.** If at all possible, have a test server available that is similar in nature to your production database server. Making changes in MySQL's settings re-

RESOURCES

How to Write Efficient MySQL Applications
http://www.linux-mag.com/2001-06/mysql_01.html

Journaling Filesystems
http://www.linux-mag.com/2000-08/journaling_01.html

MySQL Manual: Tuning Server Parameters
http://www.mysql.com/doc/S/e/Server_parameters.html

MySQL Manual: SHOW VARIABLES
http://www.mysql.com/doc/S/H/SHOW_VARIABLES.html

MySQL Manual: SHOW STATUS
http://www.mysql.com/doc/S/H/SHOW_STATUS.html

mytop — A *top* Clone for MySQL
http://public.yahoo.com/~jzawodn/mytop

Installing MySQL
http://www.linux-mag.com/2001-03/mysql_01.html

MySQL Mailing Lists
http://lists.mysql.com

quire that you stop and start MySQL, which will cause your users to experience service interruptions. (In MySQL 4.x, you'll be able to change settings on the fly.)

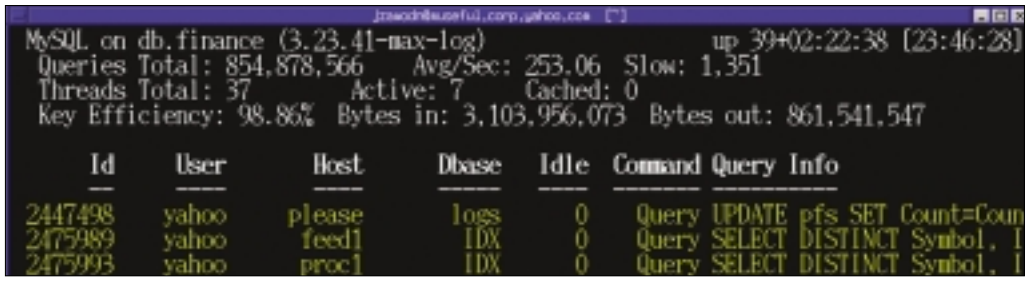

FIGURE ONE: A partial snapshot of *mytop* in action, showing the performance statistics it reports.

3. Use real data. The type of data that you are using affects how MySQL responds to queries. Ideally, you should use a copy of your production databases. If it's not possible to do this, then you should try to construct a representative subset of them.

4. Perform realistic tests. It is easy to assume that you know what to test simply because you know where the problem areas are. However, some configuration changes may speed up slow parts of your application while simultaneously slowing down things that previously were quite fast.

5. Be systematic and record your findings. It is important to track the changes you make and how they affected performance. After several hours (or even days) of testing, you won't likely be able to remember exactly what was changed and whether the effects were positive or negative. By putting your `my.cnf` file under a version control system (RCS, CVS, SCCS), you can keep an accurate record of all of your changes and how well they worked.

If you are comfortable with MySQL's replication features, you can use the binary log to generate some of your test data. Simply replaying a binary log using the *mysqlbinlog* command and piping it to the MySQL client:

```
mysqlbinlog binary-log.001 | mysql
```

will generate a lot of INSERT/UPDATE/DELETE queries on your server. Since SELECT queries do not appear in the binary log, you'll need to generate those by creating a test program/script or by simply running your application against the server. MySQL will run the queries from the binary log one at a time, as fast as possible, so the timing will not match the original conditions under which the queries will run. But it is a useful technique nonetheless.

WATCHING DATABASE PERFORMANCE NUMBERS

With a few starting points in mind and a concept of how to test, we now need to consider how to monitor progress. Fortunately, MySQL exposes more than 50 internal counters (or status variables), which track how many times various events occur: a table is opened, a key is used in a record lookup, etc.

While there isn't room in this article to discuss more than a few of MySQL's status variables, the MySQL manual contains a section (http://www.mysql.com/doc/S/H/SHOW_VARIABLES.html) that describes each one of them.

To see these numbers, you can use the `SHOW STATUS` command. In this case, we're going to focus on the variables related to the key buffer:

```
mysql SHOW STATUS LIKE 'Key%';

+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| Key_read_requests       | 3844786889 |
| Key_reads               | 16525182   |
| Key_write_requests      | 303516563  |
| Key_writes              | 152315649  |
+-------------------------+------------+
```

Those four variables tell you a lot about the performance of MySQL's key buffer. Anytime MySQL is able to read a key (or index) from the key buffer (rather than going to disk), it will increase the value of `Key_read_requests`). If MySQL must actually read the key from disk because it was not already cached, it will increase `Key_reads`. The same logic holds true for key writes.

Knowing this, we can calculate the efficiency (or hit rate) for the key buffer. Using a formula like:

```
100 - ( ( Key_reads / Key_read_requests ) * 100 )
```

we can obtain a percentage that represents how often MySQL is able to read keys directly from the cache rather than going to disk. The closer this value is to 100, the better. Using the numbers above, we find a hit ratio of roughly 99.57 percent. It's generally a good idea to try and keep this value over 90 percent.

However, running this calculation yourself can be quite a tedious task. A much easier approach would be to utilize a tool that can do all of the calculations for you. *mytop*

See MySQL, page 62

MySQL, from page 43

(http://public.yahoo.com/~jzawodn/mytop/) is a Perl script modeled after the Unix *top* command. It displays a list of the top threads that MySQL is running;, It also shows a summary of various performance statistics at the top of the display. As seen in *Figure One*, pg. 43, *mytop* displays the "Key Efficiency" as well as the average number of queries per second, slow queries, and more.

Fortunately, not all of MySQL's tunable parameters require a formula to measure. If you find that the value of `opened_tables` in the `SHOW STATUS` output is either very large or regularly increasing, it's a sure sign that you need to increase the table cache.

By either watching the output of SHOW STATUS or using a tool like *mytop*, you should quickly be able to determine whether or not the changes you made are having a measurable impact on MySQL.

WATCHING SYSTEM PERFORMANCE NUMBERS

Monitoring performance changes in MySQL is only part of the picture. You also need to watch what is happening from the operating system point of view. Has your server begun to swap? Is the CPU being much harder than it used to? Has disk I/O increased or decreased?

All of those are valid questions that directly affect how fast MySQL can operate. Like every application, it is at the mercy of what Linux (or any operating system) will allow it to do. So, it is important for you to keep a watch on overall system activity.

In order to do this, acquaint yourself with the tools that seasoned system administrators use to understand what their servers are doing. The most common tools for the job are: *top, vmstat, iostat,* and *sar.*

Keep in mind that you should get a feel for your system's current activity and performance characteristics before you begin testing. Without a baseline for comparison, you really won't know how MySQL's impact on the system may have changed.

FINAL HINTS

Believe or not, we've merely scratched the surface of server-side performance tuning for MySQL. The MySQL manual contains many other ideas about how to increase MySQL's performance and monitor your progress.

If you're struggling with performance tuning, or nearly any other MySQL related issue, consider joining the MySQL mailing list: http://lists.mysql.com/.

Jeremy Zawodny uses open source tools to process news and data feeds for Yahoo! Finance and is writing a MySQL book for O'Reilly. He can be reached at jeremy@zawodny.com.

Advertisers' Index

The Advertisers' Index lists each company's Web address and advertisement page. To advertise in *Linux Magazine*, please contact adsales@linux-mag.com for a media kit containing an editorial schedule, rate card, and ad close dates.

Linux Magazine (ISSN 1536-4674) is published monthly by InfoStrada LLC at 234 Escuela Avenue #64 Mountain View, CA 94040. The U.S. subscription rate is $29.95 for 12 issues. In Canada and Mexico, a one-year subscription is $39.95 US. In all other countries, the annual rate is $49.95 US. Non-US subscriptions must be pre-paid in US funds drawn on a US bank. Application to Mail at Periodicals Postage Rates is Pending at Mountain View, CA and additional mailing offices.

Article submissions and letters should be e-mailed to editors@linux-mag.com. *Linux Magazine* reserves the right to edit all submissions and assumes no responsibility for unsolicited material. Subscription requests should be e-mailed to linuxmag@neodata.com or visit our Web site at www.linux-mag. com.