

Beginning MySQL Database Design and Optimization:

From Novice to Professional

JON STEPHENS AND CHAD RUSSELL

Beginning MySQL Database Design and Optimization: From Novice to Professional
Copyright © 2004 by Jon Stephens and Chad Russell

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-332-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Dominic Shakeshaft and Jason Gilmore

Technical Reviewer: Mike Hillyer

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,
Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Tracy Brown Collins

Copy Edit Manager: Nicole LeClerc

Copy Editors: Ami Knox and Marilyn Smith

Production Manager: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Dina Quan

Proofreader: Christy Wagner

Indexer: Kevin Broccoli

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Finding the Bottlenecks

SO FAR, WE'VE CONCENTRATED MOSTLY on database design and writing queries through this book, and we'll continue to discuss aspects of those in this chapter. But there are other areas where you can work to improve the performance of MySQL and MySQL-backed applications. This chapter addresses those areas.

For example, many aspects of the MySQL server's operation can be modified by setting configuration variables. Although their default values are often "good enough," sometimes changing these can make a big difference in performance. In addition, you can obtain a lot of information regarding how well MySQL is actually performing by checking the values of system variables.

In the first part of this chapter, we'll look at the commands you need to read configuration and system variables, which ones are likely to be most useful to you (and why), and how to change them when necessary. We'll also take a very brief look at some freely available tools that can help you monitor your server's performance and make changes in its configuration, including *mytop* (a top clone written in Perl), *WinMySQLAdmin*, *phpMyAdmin*, and the new MySQL Administrator, available from MySQL AB. MySQL Administrator promises to become a standard and valuable part of every MySQL database administrator's toolkit.

We'll also look at caching of tables, keys, and queries. MySQL's caches, when used properly, can save a lot of memory and processing overhead. They can speed up your applications considerably by cutting down on the number of times that the server must read and/or write to disk instead of RAM. The query cache, new in MySQL 4.0, is a major resource for improving efficiency. The query cache can have dramatic effects on the speed of frequently repeated queries on tables that are not updated often, particularly if those queries yield large resultsets.

It's also true that the efficiency of your MySQL application is going to be no better than that of your queries. The cardinal rule here is: *Don't do what isn't necessary*. So don't perform unneeded queries. Don't return columns and rows that aren't required by your application. Don't join tables that aren't relevant to the problem you're trying to solve. We'll try to point out the most common errors of these types and what you can do to correct them, or better yet, to avoid making them in the first place. We'll also try to point out some common issues with application logic that affect an application's efficiency, such as repeated queries and connections, unneeded calculations, and the matter of database interoperability layers.

Configuration Issues

In addition to optimizing MySQL databases and applications, you can do a lot toward optimizing the MySQL server itself by way of various configuration settings. The first step is to read the configuration and system variables. Once you've done this, you can take appropriate action if these variables indicate performance could be improved. This action might be one or more of the following:

- Changing a value in the my.cnf (or my.ini) configuration file
- Making changes in the design of one or more tables, or adding or modifying table indexes
- Rewriting the queries that are being used by the application
- Upgrading the server hardware or changing the network configuration

In this section, we'll concentrate on reading configuration and system variables, and changing configuration settings. Later in this chapter, we'll look at some of the other possible solutions.



NOTE For more about the MySQL commands for viewing configuration and system variables—how to run them from the system shell, additional information you can get from them, and so on—see Chapter 10 of Martin Kofler's *The Definitive Guide to MySQL*, Second Edition, published by Apress.

System and Status Variables

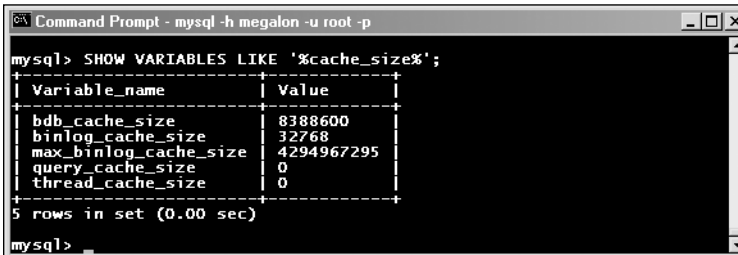
In order to understand what's happening with a running MySQL server and to see how well it's performing, you need to be able to read four types of status settings or variables:

- Configuration variables
- System variables
- Running processes
- Table variables

In the following sections, we'll look at the SQL commands you can use to accomplish these tasks and discuss how to interpret the results.

SHOW VARIABLES

The `SHOW VARIABLES` command is used to read the configuration settings currently in effect for the MySQL server daemon. As there can be in excess of 150 of these (181 on our test server running MySQL 5.0.1-alpha), it's usually a good idea to run this command using a `LIKE` clause. Here's an example:



```

mysql> SHOW VARIABLES LIKE '%cache_size%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| bdb_cache_size | 8388600 |
| binlog_cache_size | 32768 |
| max_binlog_cache_size | 4294967295 |
| query_cache_size | 0 |
| thread_cache_size | 0 |
+-----+-----+
5 rows in set (0.00 sec)

mysql>

```



NOTE All of the `SHOW` commands discussed in this section support `LIKE` clauses, which can be very useful in narrowing the result to those few variables and values in which you're most interested at any given time. This `LIKE` clause follows the same syntax rules as the `LIKE` clause used with a `SELECT` command (discussed in Chapter 1).

You can run the equivalent to a `SHOW VARIABLES` command from a system shell using this command:

```
shell> mysqladmin variables
```

Don't worry—we won't cover *all* of the configuration variables in this chapter. We'll focus on the ones that are most useful to fine-tuning MySQL and MySQL applications. An alphabetical listing of the 40 or so variables that you're most likely to need to know about when doing so is shown in Table 6-1.

Table 6-1. Some Common MySQL Configuration Variables

VARIABLE	DESCRIPTION/COMMENTS
<code>back_log</code>	Maximum number of outstanding connection requests. If your application requires a great many simultaneous requests (and there's no easy way to avoid that), you may want to increase this value. Note that there are limits on this value imposed by the operating system.
<code>binlog_cache_size</code>	Size of the cache used to store SQL statements during a transaction before they're committed. If your application uses a great many statements per transaction, you can increase this value for better performance.
<code>bulk_insert_buffer_size</code>	Size of the cache used to perform bulk inserts. This affects MyISAM tables only. The default value is 8MB.
<code>concurrent_inserts</code>	When set to ON, this allows inserts to be performed on MyISAM tables while running SELECT queries on them.
<code>connect_timeout</code>	Number of seconds that MySQL will wait for a connection packet before rejecting the connection.
<code>delay_key_write</code>	When this is enabled by being set to ON or ALL, writing to MyISAM tables with keys is faster because the key buffer is flushed to disk only when the table is closed, but tables should be checked frequently with <code>myisamchk -fast -force</code> . ON means that MySQL will honor the DELAY KEY WRITE option when used in a CREATE TABLE statement. OFF means the option will be ignored. ALL means that all tables will be treated as though they were created with this option.
<code>delayed_insert_limit</code>	When using INSERT DELAYED, MySQL will insert this many rows before checking to see if the thread has any SELECT statements to be performed. If your application performs a great many INSERTs and relatively few SELECTs, you may be able to increase performance by raising this number.

Table 6-1. Some Common MySQL Configuration Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
<code>delayed_insert_timeout</code>	Number of seconds a DELAYED INSERT thread should wait for INSERT statements.
<code>delayed_queue_size</code>	Number of rows to be queued before performing inserts from a DELAYED INSERT thread.
<code>flush</code>	If this is set to ON, MySQL will free resources after executing each SQL command; this will slow down MySQL and should be used only for troubleshooting crashes.
<code>flush_time</code>	If this is not zero (0), MySQL will stop each <code>flush_time</code> seconds to close all tables in order to free all resources. This will slow down MySQL considerably, and should not be used except on systems with very low memory or disk space.
<code>ft_max_word_length</code>	Maximum length for a word to be included in a full-text index (added in MySQL 4.0).
<code>ft_min_word_length</code>	Minimum length for a word to be included in a full-text index (added in MySQL 4.0).
<code>init_connect</code>	Beginning with MySQL 4.1.2, this can be set to a string containing SQL commands to be executed for each client connecting to MySQL.
<code>interactive_timeout</code>	MySQL waits this many seconds for activity on an interactive connection before closing it.
<code>join_buffer_size</code>	Size of the buffer used for full joins. For large joins where it's not possible to add indexes, you may be able to increase efficiency by increasing this value.
<code>key_buffer_size</code>	Size of the buffer used for index blocks. On a dedicated server, this should usually be about 25% of total RAM. Depending on the operating system, you may be able to increase it beyond this value, but anything above 50% of RAM is liable to be counterproductive due to paging effects caused by the fact that MySQL does not cache data reads from the files, leaving this to be handled by the operating system.

(Continued)

Table 6-1. Some Common MySQL Configuration Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
log	Will be ON if logging of all queries is enabled; this will tend to slow down MySQL by a very small amount and the log file will grow extremely rapidly. You may gain some improvement in performance by disabling it, and doing so is recommended if binary logging is enabled. Discontinued as of MySQL 5.0.
log_bin	Will be ON if binary logging is enabled. This is much more efficient than the query log and is recommended instead of it.
log_update	Will be ON if update logging is enabled. As with log, a very small performance increase may be gained by turning this OFF.
long_query_time	If a query takes longer than long_query_time seconds, it will be recorded in the slow query log.
max_allowed_packet	Largest packet allowed. This should be as small as you can make it without impacting your application. You should increase its size only if you need to store and retrieve large BLOB values.
max_connections	Maximum number of simultaneous connections. Increase this only as needed, since doing so incurs filesystem overhead.
max_delayed_threads	Maximum number of threads allowed for DELAY_INSERT operations. Once this number of INSERT DELAYED threads is in use, any additional insertions will be performed as if the DELAYED attribute wasn't specified. This value can be set to 0.
max_join_size	Joins that are likely to read more than max_join_size records will return an error. This can be used to help you catch joins that lack a WHERE clause, that are likely to take a very long time, and that return many excess rows.

Table 6-1. Some Common MySQL Configuration Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
<code>max_seeks_for_key</code>	Maximum number of seeks when looking up rows based on a key. By setting this to a low value (try 100 as a starting point), you can force MySQL to prefer keys instead of table scans, which may improve performance if you're using keys to good effect.
<code>max_sort_length</code>	Number of bytes used from TEXT or BLOB values when sorting them. Decreasing this value can increase the speed of ORDER BY queries. However, you should be careful not to make it too small, or you will lose accuracy in performing sorts.
<code>max_user_connections</code>	Maximum number of active connections per user (0 = no limit). This can be used to keep individual users or applications from tying up too many resources.
<code>max_write_lock_count</code>	After this many write locks are in effect, allow some read locks to take place. Normally, update operations take precedence over SELECT queries. Decreasing this value forces MySQL to let some selects to take place after fewer updates have occurred than normal, so that the SELECTs don't get put on hold for so long when large numbers of INSERT and UPDATE queries are taking place.
<code>net_buffer_length</code>	Size to which MySQL's communication buffer is reset between queries. This normally should not be changed; however, to gain a small performance improvement on systems with little memory, it can be set to the expected length of SQL statements sent by clients.
<code>query_alloc_block_size</code>	Size of memory blocks created for use during processing of queries. It can be increased slightly to help prevent memory fragmentation problems.
<code>query_cache_limit</code>	Query results larger than this are not cached. Default is 1MB.
<code>query_cache_size</code>	Memory used to store results of previous queries. The default is 0 (disabled).

(Continued)

Table 6-1. Some Common MySQL Configuration Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
query_cache_type	Used with SELECT NO_CACHE and SELECT_CACHE. Its settings are 0 = OFF; 1 = cache all results except those where SELECT NO_CACHE is used; 2 = cache only result of SELECT_CACHE queries.
read_buffer_size	Each thread that does a sequential scan allocates a buffer of this size for each table it scans. If you do many sequential scans, you may want to increase this value.
slow_launch_time	If creation of a thread takes longer than slow_launch_time seconds, it will increment the slow_launch_threads counter.
sort_buffer_size	Size of the sort memory buffer allocated to each thread. This can be increased to speed up ORDER BY and GROUP BY queries. The default is 2MB.
table_cache	Number of open tables for all threads. You can see if this needs to be increased by checking the value of the Open_tables variable (see Table 6-2).
thread_cache_size	Number of threads kept in cache for immediate reuse. New threads are taken from this cache first if any are available. You can sometimes improve performance in cases where there are many new connections by increasing this variable.
tmp_table_size	Temporary tables larger than this are stored on disk. If the server has plenty of memory, this can be increased to improve performance with large resultsets.
transaction_alloc_block_size	Amount of memory allocated for storing queries that are part of a transaction that is to be stored in the binary log when doing a commit.
transaction_prealloc_block_size	Buffer for transaction allocation blocks that are not freed between queries. You can often increase performance by making this large enough to fit all queries in a common transaction.

Memory and cache sizes are in bytes unless otherwise noted.

You will probably need to do some experimenting to get the right “mix” of configuration values for your system, and requirements may (and very likely will) change over time in response to changes in the size and numbers of your databases and tables, number and types of queries being run, number of users, hardware changes, and so forth.

When testing, you can set system variables using the SET command, for example:

```
SET GLOBAL key_buffer_size = 10000000;
```

Once you’ve determined the best value for your setup, you can force MySQL to use this value from startup by adding the appropriate line to the my.ini file, as shown here:

```
set-variable = key_buffer_size=10000000
```

The following are the most important of these variables in terms of overall performance:

key_buffer_size: This should be about 25% of available system memory. This can be increased somewhat if you have a lot of memory (more than 256MB), but should probably never be more than 45% to 50% of the system’s total RAM.

table_cache: If your application requires a lot of tables to be open at the same time, try increasing the size of the table_cache variable. (For more information about caching issues, see the “Caching” section later in this chapter.)

read_buffer_size: If you’re doing a lot a sequential scans (see the entry for Handler_read_rnd_next in Table 6-2), you should first try adding table indexes or optimizing existing ones. If that doesn’t work or isn’t feasible, you may want to increase the size of read_buffer_size.

sort_buffer_size: If you’re doing a lot of ORDER BY and/or GROUP BY queries that return large resultsets, you may find that increasing the value of sort_buffer_size helps. You may need to experiment with this. Try increasing it in increments of 5% to 10% of the starting value to see if and by how much this speeds up large queries of this type.

net_buffer_length: In situations where memory is at a premium or you have a very high number of connections, you may be able to improve matters by adjusting the size of net_buffer_length. However, if you set this value to be too small, you’ll waste any performance gain you might have otherwise obtained, because MySQL will need to keep resetting this value in order to accommodate queries that are longer than the stated number of bytes.

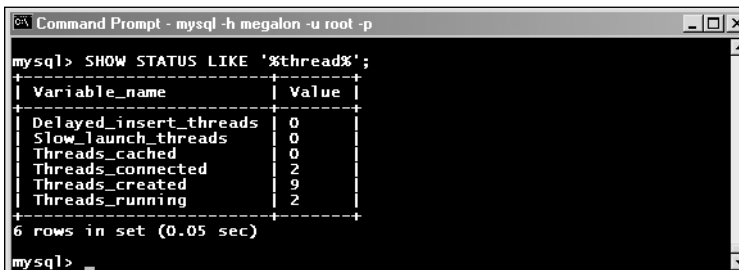
You can optimize the existing `my.cnf` configuration file or select one of those supplied with MySQL. These are named `my-small.cnf`, `my-medium.cnf`, `my-large.cnf`, and `my-huge.cnf`. For serious applications, you'll probably want to use one of the latter two as your starting point.

The best way to optimize these settings is to check the values of a number of MySQL status variables while your application is running, adjust system variables accordingly, and then check the status variables again. To examine status variables, you'll need to use the `SHOW STATUS` command, which is described in the next section.

SHOW STATUS

The `SHOW STATUS` command displays information about the status of the running MySQL server. Using this command will show you status information such as how many queries of a given type have been run since MySQL was last restarted, current uptime, caching data, and so on.

As with `SHOW VARIABLES`, there are about 150 values returned by an unmodified `SHOW STATUS` command, so it's usually best to use this command with a `LIKE` clause. Here's an example showing how you might obtain current data about how MySQL is handling threads:



```

mysql> SHOW STATUS LIKE '%thread%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Delayed_insert_threads | 0 |
| Slow_launch_threads | 0 |
| Threads_cached | 0 |
| Threads_connected | 2 |
| Threads_created | 9 |
| Threads_running | 2 |
+-----+-----+
6 rows in set (0.05 sec)
mysql>

```

You can run the equivalent command from a system shell or DOS prompt as follows:

```
shell> mysqladmin extended-status
```

You can pipe this to a file for later review and analysis using something like this:

```
shell> mysqladmin extended-status > ext-status.txt
```

The file will be created relative to the current directory; you can also specify a system absolute path (such as `/home/users/mystuff/ext-status.txt` or `C:\Documents and Settings\Jon\My Documents\ext-status.txt`) if desired. Of course, you can also save the results of a `mysqladmin variables` command to a file using the same technique.

Table 6-2 shows those status variables that are likely to be of the most use to you when analyzing the performance of your MySQL server. Most of these values are counters; all are reset each time MySQL is restarted.



NOTE *MySQL configuration variables are displayed in lowercase; status variables are displayed with a leading capital letter.*

Table 6-2. Common MySQL Status Variables

VARIABLE	DESCRIPTION/COMMENTS
Aborted_clients	Number of connections that were aborted without closing the connection properly. If this is a high proportion of the Connections count, there may be problems with your application code (such as waiting too long without activity or failing to close a connection when finished) or networking problems.
Aborted_connects	Number of times that connections to MySQL failed. This could be high compared to the value of Connections for a number of reasons, such as networking problems, failure to employ a correct user/password, incorrect database privileges, or malformed packets. Always investigate the situation when you note a high Aborted_clients / Connections ratio, because this may indicate security problems, such as someone trying to break into your MySQL server! This may also be a sign that the value of max_allowed_packet (see Table 6-1) is set too low. Note that the default value for max_allowed_packet should be large enough for most purposes and probably shouldn't be increased unless you're consistently running queries that return result rows larger than this.
Bytes_received	Number of bytes received from all clients.
Bytes_sent	Number of bytes sent to all clients.
Com_xxx	Number of times each xxx command has been executed (For example, Com_insert gives the number of INSERT commands performed; Com_select, Com_show_status, Com_update, and so on work the same way for their associated commands.)

(Continued)

Table 6-2. Common MySQL Status Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
Connections	Total number of connection attempts to the MySQL server.
Created_tmp_disk_tables	Number of implicit temporary tables on disk created while executing statements.
Created_tmp_files	How many temporary files have been created by MySQL.
Created_tmp_tables	Number of implicit temporary tables in memory created while executing statements.
Delayed_insert_threads	Number of delayed insert handler threads in use.
Delayed_errors	Number of rows written using INSERT DELAYED for which some error occurred (probably duplicate key).
Delayed_writes	Number of rows written using INSERT DELAYED.
Handler_delete	Number of times a row was deleted from a table. (Com_delete counts the number of actual DELETE commands.)
Handler_read_first	Number of times the first entry was read from an index. If this is high compared to Handler_read_rnd_next, MySQL is probably doing a lot of full-index scans (this is usually a good thing).
Handler_read_key	Number of requests to read a row based on a key. A high Handler_read_key value compared to Handler_read_rnd_next is a good indicator that your queries are optimized and tables are properly indexed.
Handler_read_next	Number of requests to read next row in key order, and is incremented whenever you perform a query on an index column with a range constraint. This count is also incremented when you do an index scan.
Handler_read_prev	Number of requests to read the previous row in key order. This is mainly used to optimize ORDER BY ... DESC.
Handler_read_rnd	Number of requests to read a row based on a fixed position. This will be high if you are doing a lot of queries that require sorting of the result.

Table 6-2. Common MySQL Status Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
Handler_read_rnd_next	Number of requests to read the next row in the datafile. This will be high if you are doing a lot of table scans, which usually indicates that tables aren't properly indexed. It can also mean that queries aren't being written to take advantage of existing indexes.
Handler_update	Number of requests to update a row in a table. (Com_update represents the number of actual UPDATE queries.)
Handler_write	Number of requests to insert a row in a table. (Com_insert is the number of actual INSERT commands.)
Key_blocks_used	Number of used blocks in the key cache.
Key_read_requests	Number of requests to read a key block from the cache.
Key_reads	Number of physical reads of a key block from disk. (See the "Key Cache" section later in this chapter.)
Key_write_requests	Number of requests to write a key block to the cache. (See the "Key Cache" section later in this chapter.)
Key_writes	Number of physical writes of a key block to disk.
Max_used_connections	Maximum number of connections that have been in use simultaneously. If this is close to the value of the max_connections configuration variable, it may be time to increase this value, or to look for ways to decrease the number of simultaneous connections required for your purposes.
Not_flushed_delayed_rows	Number of rows waiting to be written in INSERT DELAY queues. If this is very high compared to delayed_insert_limit or delayed_queue_size (see Table 6-1), you may need to increase the value of one or both of these.
Not_flushed_key_blocks	Key blocks in the key cache that have changed but haven't yet been flushed to disk. If this appears persistently high, you may need to increase the value of key_buffer_size (see Table 6-1).
Open_files	Number of files that are currently open.
Open_streams	Number of streams that are currently open (used mainly for logging).

(Continued)

Table 6-2. Common MySQL Status Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
Open_tables	Number of tables that are currently open.
Opened_tables	Total number of tables that have been opened.
Questions	Total number of queries that have been sent to the server.
Select_full_join	Number of joins that have been made without using any keys. Ideally, this value should always be 0; if it isn't, you should check all of your table indexes.
Select_full_range_join	Number of joins where a range search was used on a reference table.
Select_range	Number of joins where a range search was used on the first table. (Normally not critical even if quite large.)
Select_range_check	Number of joins without keys where key usage was checked for after each row. Ideally, this value should be 0. If it's not, you should review your tables and joins to see if there are sufficient indexes and if they're being used properly.
Select_scan	Number of joins where a full scan was done on the first table. You should review your joins to see if there are any that could benefit from additional indexing.
Slow_launch_threads	Total number of threads that have taken more than <code>slow_launch_time</code> to create.
Slow_queries	Total number of queries that have taken more than <code>long_query_time</code> seconds to execute. If this number is a very large proportion of the total number of queries, you should check the query log to determine which ones are running slowly and try to remedy this.
Sort_merge_passes	Number of merge passes that MySQL's internal sorting algorithms have needed to perform. If this value is large, you should consider increasing the value of the <code>sort_buffer</code> configuration variable.
Sort_range	Number of sorts that were done with ranges.
Sort_rows	Number of sorted rows.
Sort_scan	Number of sorts that were done by scanning the table. If this isn't 0, you might want to look at indexing the columns used in <code>ORDER BY</code> or <code>GROUP BY</code> queries.

Table 6-2. Common MySQL Status Variables (Continued)

VARIABLE	DESCRIPTION/COMMENTS
Table_locks_immediate	Number of times a table lock was acquired immediately.
Table_locks_waited	Number of times a table lock could not be acquired immediately and a wait was needed. If this is high, and you have performance problems, you should first optimize your queries, and then either split your table or use replication.
Threads_cached	Number of threads in the thread cache.
Threads_connected	Number of currently open connections. This should be fairly close to the value of Threads_running and Threads_created.
Threads_created	Number of threads created to handle connections.
Threads_running	Number of threads that are not sleeping.
Uptime	How many seconds the server has been up.

Of all the variables shown in Table 6-2, the following are probably the most important with regard to index and query optimization:

Handler_read_key, Handler_read_next, and Handler_read_rnd_next: The higher that `Handler_read_rnd_next` is, the more queries there are being run without the use of indexes (the *rnd* is short for *random*). When taken in relation to each of the first two values, this provides a rough measure of how efficiently you're using indexes. If either of these ratios is greater than a very small fraction, you need to examine your tables and queries for proper use of indexes.

Key_reads and Key_read_requests: The ratio of these two values should be a very small fraction. If it isn't, you may need to increase the size of the `key_buffer_size` configuration variable. If you can't increase this without going past the upper limit value of 50% of system RAM, consider adding more physical memory to the server. See the "Key Cache" section later in this chapter for more information.

Select_full_joins and Select_range_check: If either of these numbers is anything other than 0, it means that there are queries being run that don't use any indexes at all. This is the worst possible thing that can happen with regard to efficiency. You should definitely take the time to determine which queries these are, and either add indexes on the appropriate table columns or rewrite the queries to take advantage of existing indexes.

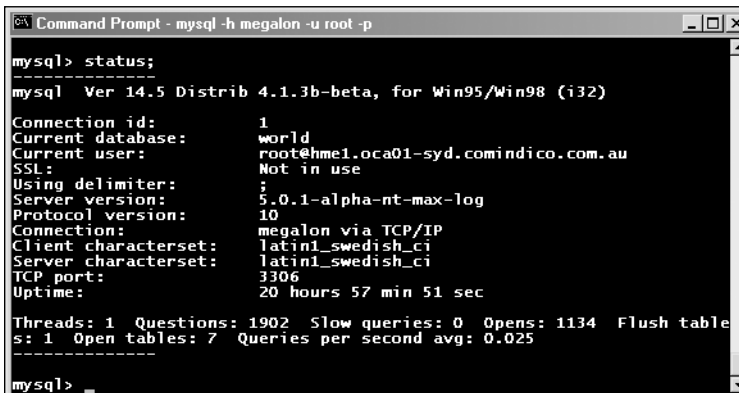
Select_scan: If this number is not 0, you have joins where no indexes are being used for the first table in a join. You should check your joins to see where adding indexes or making use of existing ones can take care of these.

Slow_launch_threads and **Slow_queries:** These indicate, respectively, the number of threads taking longer than `slow_launch_time` to begin and the number of queries taking longer than `long_query_time` (see Table 6-1 for descriptions of these) to run. The default values for these configuration variables are 2 and 10 seconds, respectively. Reasonable values for them under actual usage conditions will vary; we recommend 1 and 3 seconds as a good starting point.

Sort_merge_passes: If you see a large value for this compared with `Sort_rows`, you likely need to increase the value of the `sort_buffer_size` configuration variable, as MySQL is needing to make multiple passes to perform sorts required by `ORDER BY` and `GROUP BY` queries.

Sort_scan: This many sorts were performed without using any indexes. This can cause a major slowdown of `ORDER BY` and `GROUP BY` queries on large tables and large resultsets. You should determine which of these queries is incrementing the `Sort_scan` count, and add indexes or make use of existing ones.

You can also obtain a short summary of the server status by using the `STATUS` command (or the abbreviated form `\s`). As shown in the following example, this command displays basic client and server information, along with an abbreviated version of what you would obtain using the `SHOW PROCESSLIST` command (described in the next section).



```

Command Prompt - mysql -h megalon -u root -p

mysql> status;
-----
mysql Ver 14.5 Distrib 4.1.3b-beta, for Win95/Win98 (i32)

Connection id:          1
Current database:       world
Current user:           root@hme1.oqa01-syd.comindico.com.au
SSL:                   Not in use
Using delimiter:        ;
Server version:         5.0.1-alpha-nt-max-log
Protocol version:       10
Connection:             megalon via TCP/IP
Client characteraset:   latin1_swedish_ci
Server characteraset:   latin1_swedish_ci
TCP port:               3306
Uptime:                 20 hours 57 min 51 sec

Threads: 1 Questions: 1902 Slow queries: 0 Opens: 1134 Flush table
s: 1 Open tables: 7 Queries per second avg: 0.025
-----
mysql>

```

SHOW PROCESSLIST

The SHOW PROCESSLIST command shows the processes currently running on the server, and comes in two versions:

```
SHOW PROCESSLIST
SHOW FULL PROCESSLIST
```

Including the FULL keyword forces the complete display of all SQL commands currently being run; without it, only the first 100 characters of each one is shown.

Here is some sample output from a SHOW PROCESSLIST command (using the \G switch to make it fit nicely within the DOS window):

```
mysql> SHOW PROCESSLIST\G
***** 1. row *****
  Id: 1
  User: root
  Host: hme1.oqa01-syd.comindico.com.au:4283
  db: NULL
  Command: Query
  Time: 0
  State: NULL
  Info: SHOW PROCESSLIST
***** 2. row *****
  Id: 13
  User: root
  Host: hme1.oqa01-syd.comindico.com.au:4332
  db: NULL
  Command: Sleep
  Time: 3
  State: NULL
  Info: NULL
***** 3. row *****
  Id: 16
  User: zontar
  Host: localhost:1250
  db: xx
  Command: Query
  Time: 0
  State: freeing items
  Info: SELECT query FROM `phpmyadmin`.`pma_bookmark` WHERE dbase = '
xx' AND (user = 'zontar' OR user = '
3 rows in set (0.00 sec)
mysql>
```

Table 6-3 describes the information displayed by SHOW ProcessList.

Table 6-3. SHOW PROCESSLIST Information

COLUMN	DESCRIPTION
Id	Process ID; use with the KILL command to kill a process
User	Database user account name
Host	Host in <i>hostname:port</i> format or IP address
db	Name of current database
Command	Type of command; usually either Sleep or Query

(Continued)

Table 6-3. *SHOW PROCESSLIST Information (Continued)*

COLUMN	DESCRIPTION
Time	Seconds that this command has been running
State	Shows the current state of the process; see Table 6-4
Info	Text of the current SQL command (or NULL for a sleeping thread)

With this command, you can see at a glance what every MySQL user is doing. This is particularly useful if you get a “Too Many Connections” error and need to see what’s going on. Unfortunately, there’s no simple way to page the results from the MySQL Monitor on Windows systems, but you can pipe the output of the equivalent system shell command `mysqladmin processlist` to a file. On Linux and other Unix platforms, you can use `PAGER less;` to page the result. Note that you cannot use a `LIKE` clause with `SHOW PROCESSLIST`.

In order to get the most out of `SHOW PROCESSLIST`, you need to run it as the MySQL root user or as a user with the `SUPER` privilege. MySQL always reserves one thread for a user with this privilege; for this reason, you should never assign this privilege to an ordinary user. Users with the `SUPER` privilege can view all threads and kill any thread. Ordinary users can view or kill only their own threads.



NOTE *The `SUPER` privilege is not supported prior to MySQL 4.0.2. On Win32 platforms, the old `PROCESS` privilege remains in use through MySQL 4.0.10.*

If you observe a process that has been running for an overly long time, you can force it to be terminated using the `KILL` command:

```
KILL processId;
```

where *processId* is the process ID of the thread.

Generally, any command (other than `Sleep`, of course) that is taking a very long time to execute has probably run into trouble, so you should investigate to determine the cause of the problem. This may be the result of an incompetent or abusive user or of a “hung” application, and you may need to kill such threads manually. Of course, what constitutes “a very long time” will vary according to your specific situation. If your server is being used in a data warehousing application involving many thousands (or even millions) of records, it may be normal for a single `SELECT` or `SELECT INSERT` query to run for 10 or 15 minutes. On the other hand, if the server is supporting a relatively small web site or two, and a single query takes that long to execute, it’s a safe bet that something has gone wrong.

Something else to consider is that as systems grow, what may once have been acceptable may no longer be so. For example, programmers may have used `SELECT *` because tables were small and didn't contain very many rows. As the number of records increases, it may be necessary to fine-tune those queries and retrieve only the columns and rows actually needed by the application. However, this isn't the only possibility for corrective action, as you can see from Table 6-4.

Table 6-4. Common State Values Shown by SHOW PROCESSLIST

STATE VALUE	DESCRIPTION/EXPLANATION
Checking table	The process is examining a table, which is entirely normal.
Closing tables	The thread is saving changed table data to disk and closing the tables used. This should happen very quickly, unless the disk is full, very badly fragmented, or in very heavy use.
Connect out	A replication slave is connecting to the master server. (Used in replication scenarios only.)
Copying to tmp table on disk	A temporary resultset was larger than the value set for the <code>tmp_table_size</code> configuration variable in <code>my.cnf</code> (or possibly <code>my.ini</code> on Windows) that determines the maximum amount of memory in bytes that a resultset may take up; the thread is now copying the temporary table from RAM to disk in order to save memory. If you observe this happening a great deal and your system has sufficient memory, you can safely increase this value and thus the speed at which such large queries are executed.
Creating tmp table	The thread is creating a temporary table to hold the result of a query (or part of the result).
Deleting from main table	The thread is executing the first part of a multiple-table delete (deleting from the first table only).
Deleting from reference tables	The thread is executing the second part of a multiple-table delete (deleting matched rows from other tables).
Flushing tables	The thread is reloading tables and is waiting for all other threads to close their tables before proceeding.
Killed	A <code>KILL</code> command has been issued for this thread, but has not yet taken effect. (Once it has been killed, the thread will no longer be listed.)

(Continued)

Table 6-4. Common State Values Shown by SHOW PROCESSLIST (Continued)

STATE	VALUE	DESCRIPTION/EXPLANATION
Sending data		The thread is processing a SELECT statement and sending the resulting rows of data to the user.
Sorting for group		The thread is performing a sort as the result of a GROUP BY query.
Sorting for order		The thread is doing a sort due to an ORDER BY query.
Opening table		The thread is attempting to open a table, which should normally occur very quickly. If this persists, it is likely that a previous ALTER or LOCK command hasn't yet finished.
Removing duplicates		This sometimes occurs when a SELECT DISTINCT can't easily be optimized by MySQL and an extra step must be performed to remove duplicate rows before returning the final result.
Reopen table		This occurs when a thread attempts to obtain a lock for a table, but the table structure changed before the lock was complete; the thread has released the lock, closed the table, and is now trying to reopen it.
Searching rows for update		This happens when an UPDATE query has changed the index that is being used to find rows by the UPDATE query itself. In other words, a query of the form UPDATE <i>table</i> SET <i>column=newvalue</i> WHERE <i>column=oldvalue</i> ; is being executed, which may take a long time when the table is extremely large, <i>newvalue</i> and/or <i>oldvalue</i> are the result of a calculation, or the WHERE clause is particularly complex and is comparing a great many values.
Sleep		A connection for this thread is open, but isn't currently executing any commands from the client that opened it.
System lock		The thread is waiting for an external system lock for the table to be released. If you are not using multiple MySQL servers, you can (and probably should) disable system locks by starting the MySQL daemon with <code>--skip-external-locking</code> . You can also set <code>skip_lock=0n</code> in your <code>my.cnf</code> or <code>my.ini</code> file to accomplish this.

Table 6-4. Common State Values Shown by *SHOW PROCESSLIST* (Continued)

STATE VALUE	DESCRIPTION/EXPLANATION
Upgrading lock	An INSERT DELAYED is waiting to obtain a lock on the table before inserting rows. (INSERT DELAYED causes INSERT statements not to be executed until the table is no longer in use by any threads executing SELECT or DELETE statements on the same table. The server then locks the table and performs all pending INSERT statements for that table before unlocking it again.)
Updating	The thread is performing an UPDATE query on a table.
User Lock	The thread is waiting on a locked table to be released. If this persists, you may have a problem and need to restart the server. In such cases, you should examine the table after the restart to make sure that it hasn't been corrupted. If it has been corrupted, restore it from a backup.
Waiting for tables	The thread was notified that a table that it is trying to open has been changed by another thread. The thread must wait until any other threads using the table have closed it before reopening it, so that it can obtain the updated version of the table.

SHOW TABLE STATUS

It can sometimes be helpful to see how much data has been stored in one or more tables, when they were last accessed, their types, and how much memory has been allocated to them. *SHOW TABLE STATUS* provides this sort of information. You can use it on a database that is not currently selected by adding a *FROM dbname* clause, and its output can be filtered with a *LIKE* clause (and wildcards if desired).

The following example shows how to get the status of all tables in the **mdbd** database whose names begin with the string “orders.” It also serves to illustrate the columns returned by this command and the type of information displayed in each one.

```

C:\ Command Prompt - mysql -h megalon -u root -p

mysql> SHOW TABLE STATUS FROM mdbd LIKE 'orders%\G'
***** 1. row *****
      Name: orders
      Engine: MyISAM
      Version: 9
      Row_format: Dynamic
      Rows: 26
      Avg_row_length: 31
      Data_length: 820
      Max_data_length: 4294967295
      Index_length: 2048
      Data_free: 0
      Auto_increment: 27
      Create_time: 2003-11-25 12:59:37
      Update_time: 2004-03-29 13:40:55
      Check_time: 2004-05-30 12:51:03
      Collation: latin1_swedish_ci
      Checksum: NULL
      Create_options:
      Comment: orders table
***** 2. row *****
      Name: orders2
      Engine: MyISAM
      Version: 9
      Row_format: Fixed
      Rows: 8
      Avg_row_length: 35
      Data_length: 280
      Max_data_length: 150323855359
      Index_length: 2048
      Data_free: 0
      Auto_increment: 9
      Create_time: 2004-03-13 00:58:02
      Update_time: 2004-03-14 19:33:30
      Check_time: 2004-05-30 12:51:03
      Collation: latin1_swedish_ci
      Checksum: NULL
      Create_options:
      Comment:
2 rows in set (0.00 sec)

mysql>

```

For InnoDB tables, the **Create_time**, **Update_time**, **Check_time**, and **Max_data_length** columns will be NULL. Available free space will be shown in the **Comment** column, along with any foreign key constraints defined for the table.



NOTE MySQL 4.1.1 adds two new columns: **Collation** and **Checksum**. The **Collation** column shows the table's character set and collation. The **Checksum** column shows the checksum for the table (if there is one). In MySQL 5.0.1, views are also represented in the output of **SHOW TABLES**. If the table is a view, all columns except **Name** and **Comment** will be shown as NULL, with the value for **Comment** being view.

Tools for Monitoring Performance

There are some administration tools available that can make the job of monitoring the MySQL server much simpler and easier. Space does not permit us to go into a great amount of detail concerning these, but we thought it would be a good idea to mention four of the more commonly used ones: mytop, WinMySQLAdmin, MySQL Administrator, and phpMyAdmin.



NOTE For more information about MySQL administration tools, check the product or project web sites, or consult another reference, such as Enterprise MySQL (which will soon be available from Apress).

mytop

The mytop utility is an Open Source, text-mode tool written in Perl that allows you to monitor server status in real time. This is particularly useful on Unix systems where you want something a little more sophisticated than the output of a SHOW command, but don't want the added overhead of running a GUI on your database server.

However, we've also run this on Windows NT and Windows 2000 systems under ActivePerl from ActiveState.com without any problems. mytop was originally created by Yahoo programmer Jeremy Zawodny and is modeled after the top utility, which is commonly used for monitoring Unix system processes. He continues to develop it and has accepted contributions from several others. The latest release at the time of this writing was version 1.4. You can visit the mytop home page at <http://jeremy.zawodny.com/mysql/mytop>.

WinMySqlAdmin

WinMySqlAdmin is a Windows-only GUI configuration tool that allows you to read configuration and status data and to update the my.ini file with new configuration variable values using a simple built-in text editor. (One slight drawback is that you can't update a my.cnf file on a Windows machine using this utility.) This program is included with the Win32 distribution of MySQL and should run on all Windows flavors.

This tool is being superseded by MySQL Administrator (described in the next section), but may remain useful with legacy installations of MySQL, versions 3.23 and earlier.

MySQL Administrator

MySQL Administrator is a full-featured GUI tool for configuring and administering a MySQL server and is available for Windows and Linux systems. Still under development at this writing (the latest version was 1.0.9), it is already very powerful and usable and can perform nearly every task that you would otherwise do using the

command line and/or a text editor. The interface is extremely intuitive and has a great deal of helpful information built directly into it, such as descriptions of all the configuration variables as part of the appropriate displays. Because MySQL Administrator employs the newer version of the MySQL client programming libraries, it can be used only with servers running MySQL versions 4.0 or newer.

You can probably expect this utility (or one quite similar to it) to become part of the standard MySQL toolkit by the time that MySQL 4.1 is in a production release. In Chapter 8, we'll take a look at another new tool that MySQL AB is developing, the MySQL Query Browser, which provides a graphical interface for working with queries and tables.

phpMyAdmin

phpMyAdmin is an Open Source application written in PHP. It will run on nearly any web server supporting both PHP and MySQL, including both Apache and Internet Information Server (IIS). It can be used with MySQL 3.21 through 4.1 (we have tested releases 2.5.x through 2.6.0 with MySQL 4.1.3-beta and 5.0.1-alpha on servers running PHP 4 and PHP 5; it seems to work fine with these as well), and with PHP 3.0.8–5.0. As of this writing, the latest production release was 2.5.7 and version 2.6.0 was in beta.



CAUTION *phpMyAdmin versions previous to 2.6.0 do not employ the new MySQLi library (see Chapter 7). If you wish to use an older version of phpMyAdmin on a web server running PHP 5, you'll need to make sure that the older PHP 4-style mysql library is present.*

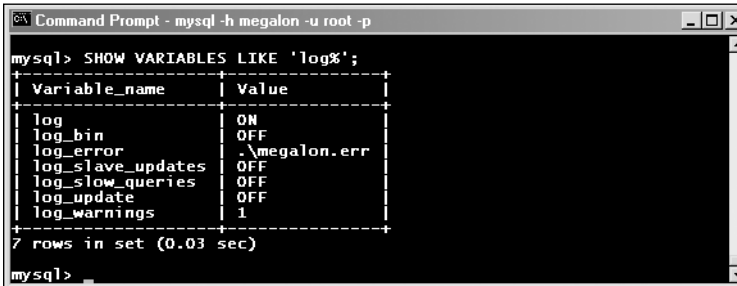
This tool is very simple to install and configure, and allows users who have the correct privileges to accomplish most MySQL database administration and query-related functions through any relatively recent web browser. For example, you can view processes, check server and table status, and check configuration variables. Although you can't use it to update a `my.cnf` or `my.ini` file, you can update configuration variables at least temporarily using the appropriate `SET` commands.

Administration of multiple MySQL servers is also possible with phpMyAdmin. Another big plus is that phpMyAdmin is internationalized quite well, currently supporting more than 45 languages. For more information about phpMyAdmin and to obtain a copy of the latest version, visit the phpMyAdmin home page at <http://www.phpmyadmin.net/> or <http://phpmyadmin.sourceforge.net/>.

Log Files

MySQL can keep a number of different types of useful records of its activity. Those relating directly to performance issues include the query log, the update log, the binary log, and the slow query log. We'll look briefly at each of these and how to use them in this section.

Before proceeding to descriptions of the individual logs, here's a quick and simple way to see which logs and logging options are enabled on your server:



```

C:\> Command Prompt - mysql -h megalon -u root -p

mysql> SHOW VARIABLES LIKE 'log%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| log           | ON   |
| log_bin       | OFF  |
| log_error     | .\megalon.err |
| log_slave_updates | OFF  |
| log_slow_queries | OFF  |
| log_update    | OFF  |
| log_warnings  | 1    |
+-----+-----+
7 rows in set (0.03 sec)

mysql>

```

The first three entries show whether the query, update, or binary logs are enabled. The `log_slow_queries` setting indicates whether the slow query log is being kept. The `log_error` setting shows the name of the error log if it's not the default.

Normally, all logs are kept in MySQL's data directory. You can override this behavior by specifying a path in the `=filename` portion of the appropriate lines in your server's `my.cnf` or `my.ini` file. For example, to force the binary logs to be saved to the directory `/usr/log/mysql`, you would need a line that reads like this:

```
-log-bin=/usr/bin/log/mysql
```

General Query Log

The *query log* (sometimes referred to as the *general query log* in order to distinguish it from the slow query log) keeps a record of all connections made to the server and of all queries, the dates and times they were made, and the users (with process IDs) who made them. This log is a plain text file whose format is quite simple, as you can see from this sample:

```

MySQL, Version: 5.0.0-alpha-max-nt-log, started with:
TCP Port: 3306, Named Pipe: MySQL
Time           Id Command  Argument
040524 17:59:39    12 Connect  root@localhost on
040524 17:59:43    12 Query    show databases

```

```

040524 18:00:26      12 Quit
040524 18:02:19      34 Connect    pytest@localhost on test
                        34 Query      INSERT INTO employees
                                (empid, firstname, lastname)
                                VALUES
                                ('', 'Joan', 'Newhouse')
                        34 Query      SHOW WARNINGS
                        34 Quit
040525 18:05:11      13 Connect    root@localhost on
040525 18:05:24      13 Query      show variables like 'query_cache'
040525 18:05:28      13 Query      show variables like 'query_cache%'
040525 21:28:58      13 Query      show variables like '%cache%'
040525 21:36:41      13 Query      show variables like '%open%'
040525 21:52:49      13 Query      show status like '%open%'
040525 22:08:44      13 Query      show variables like '%key%'
040526  4:07:45      13 Quit

```

For instance, you can tell that the user **pytest@localhost** logged in to the **test** database at 18:02:19, was given process ID 34, ran an insert query, ran a **SHOW WARNINGS** command, and then immediately logged out. It's important to note that all SQL commands are logged as they're received, and not necessarily in the order that they're actually executed.



NOTE *Access error messages (caused by trying to use unauthorized privileges) are recorded in the general query log, but query errors and warnings are not logged there. To view those, you need to use a **SHOW ERRORS** or **SHOW WARNINGS** command, or the equivalent API function, such as PHP 4's `mysql_error()`, in your application code.*

Enabling the general query log does slow down MySQL a bit, since it takes time to write a record of each connection and query. In addition, the query log file will very likely grow at a tremendous rate on a busy server! It's usually best to use it only when testing or debugging, and to rely on the update or binary log (preferably the latter) once the server goes into normal production use.

Update Log

In MySQL 3.x and 4.x, the update log keeps a record of all issued statements that update data. This log can be useful when you're trying to determine whether statements that are supposed to change data are actually doing so.

To enable update logging, use the `-log-update=filename` option in your MySQL configuration file or when running `mysqld`. The `=filename` portion is

optional; the filename defaults to *hostname.###*, where *###* is a three-digit numeral, unless you specify a file extension as part of *filename*. If present, this number is incremented for each new update log. A new update log is started whenever the logs are flushed or MySQL is restarted.



NOTE *The update log has been removed in MySQL 5.0, and starting with that version, you must use the binary log instead. In earlier versions, it's still preferable to use the binary log, as it's faster and uses fewer resources. See the "Binary Log" section in this chapter for more information.*

The update log records only statements that actually update data. So an SQL command such as this:

```
UPDATE products SET prodname='Blender' WHERE prodid='147042';
```

does *not* get recorded in the update log if there's no product in the **products** table whose **prodid** is **147042**. An UPDATE statement that sets a column to the same value that column already has also won't be written to the update log.

The update log can also be useful if you need to restore a database following a crash or another severe problem and you have a good known starting point. Note that update queries are logged in the order in which they're actually executed, unlike the case with the general query log.

Binary Log

The binary log, like the update log it's intended to replace, records all statements that update data. Its primary purpose is to make it easy to restore your databases following a critical failure and to assist in replication. However, it can also be useful for debugging purposes, when you need to know whether a particular query, which should have updated a table, has in fact done so. It is faster and less wasteful of space than the update log, and beginning with MySQL 5.0, binary logs replace the update logs entirely.

To enable binary logging, you need to include the following line in the `[mysqld]` section of a MySQL `my.cnf` or `my.ini` configuration file:

```
log-bin[=filename]
```

Alternatively, you can use `-log-bin[=filename]` as a startup option to `mysqld`. The default name of the binary log file is *hostname-bin*. MySQL automatically supplies a three-digit file extension when it creates a binary log file, so if you try to supply an extension as part of the filename, the extension will be ignored and will *not* be used by MySQL in naming the file.

You can't usefully read a binary log with a text editor, as you can MySQL's other log files. Instead, you must use the `mysqlbinlog` utility, which is supplied as part of all MySQL distributions, as in this example:

```
shell> mysqlbinlog localhost-bin.001
```

You can save the output of `mysqlbinlog` to a text file for later analysis, similar to how you can redirect output from other MySQL utilities. For instance, you might use something like this from a system shell or DOS prompt:

```
shell> mysqlbinlog localhost-bin.001 > binlog1.txt
```



NOTE For information about the use of `mysqlbinlog` with binary logs for replication purposes, run `mysqlbinlog --help`, consult the MySQL documentation for `mysqlbinlog`, or consult a reference such as the upcoming *Enterprise MySQL from Apress*.

Slow Query Log

When slow query logging is enabled, MySQL logs all statements taking longer than `long_query_time` (see Table 6-1) seconds to execute. This can be used to find queries that are taking too long to execute, so that they can be optimized.

You can enable the slow query log by adding this line to your MySQL configuration file:

```
log-slow-queries[=filename]
```

Alternatively, you can use `--log-slow-queries[=filename]` as one of the startup options for `mysqld`. By default, the filename is `hostname-slow.log`. In addition, by using `log-long-format` (or `--log-long-format`) in MySQL 4.0 or earlier, you can specify that all queries that don't use any indexes are written to the slow query log, no matter how long those queries take to run. Beginning with MySQL 4.1, you should use `[-]log-queries-not-using-indexes` for this purpose.



NOTE The time needed by MySQL to acquire table locks is not counted as part of the query execution time.

Caching

MySQL has some caching capabilities that can enhance performance considerably. Here, we will discuss MySQL table, key, and query caching.

Table Cache

It's very important to remember that MySQL tables are actual, discrete files on disk, so that when you run queries, you're causing `mysqld` to open, read and/or write, and close files for each database table involved. In order to speed up these tasks, MySQL keeps a *table cache*, which is another way of saying that it keeps files open in between queries so that they may be accessed again quickly without the overhead of closing them and then reopening them each time they're needed. The maximum number of files the server keeps open is affected by the `table_cache`, `max_connections`, and `max_tmp_tables` server variables (see Table 6-1).

The optimum value for `table_cache` is directly related to that of `max_connections`, as well as to the number of tables that need to be open simultaneously in order to perform multiple-table joins. The `table_cache` value should be equal to no less than the number of concurrent connections you're expecting to your MySQL server times the largest number tables involved in any one join.

For example, if you know that your server needs to support up 100 simultaneous running connections, and the largest join used by your application involves 5 tables, you should have a table cache size of at least 500. (If you think this implies that each table is opened as many times as there are threads accessing the table, then you're absolutely correct. Three threads running the same three-table join at the same time use nine open tables.) You also need to reserve some extra file descriptors for temporary tables and files as well. This will vary according to how heavily you use temporary tables, but a good rule of thumb is to allow an extra 20%, due to the fact that MySQL also creates temporary tables behind the scenes (whether or not you're creating explicit temporary tables as part of your application). So, in this example, you would want to make sure that `table_cache` was set to at least 600.

However, there are limits imposed by the operating system on the number of open file descriptors. If you increase the size of the table cache, you need to check your system's documentation and make sure that you're not exceeding this limit; otherwise, MySQL may refuse connections, fail to perform queries, and be very unreliable. It's also necessary to keep in mind that the MyISAM engine uses two file descriptors per open table, so make sure that the value of the `open_files_limit` configuration variable is high enough to accommodate this. Note that the default value of zero means that MySQL will use as many file descriptors as necessary, up to the maximum allowed by the operating system.

Once opened, a table remains in the table cache until the table cache is full, the table is no longer in use, and a new table needs to be opened. Using a `FLUSH TABLES` command or the equivalent causes MySQL to attempt to clear the table cache by closing all unused tables. MySQL will, if necessary, temporarily increase the size of the table cache if possible to accommodate all queries being run at the same time.

You should check the `Open_tables` and `Open_files` status variables (see Table 6-2) while your application is running, and if these are large compared to `table_cache` and `open_files_limit`, you should consider increasing their values. However, don't forget about the operating system limits just mentioned when you do this.

Key Cache

In order to save reading from and writing to MyISAM table index files (.MYI files), MySQL also caches table indexes in a *key cache*. The size of this cache is determined by the value of the `key_buffer_size` configuration variable. In determining your server's performance with regard to key caching (and thus what the best key buffer size is likely to be), you need to look at two different ratios, which can be derived from status variable values.

The first of these is the cache miss rate, which can be calculated like this:

$$\text{Key_cache_misses} = \text{Key_reads} / \text{Key_read_requests}$$

This figure, which represents the proportion of keys that are being read from disk instead of the key cache, should normally be less than 0.01 for optimum efficiency. If it's much larger than this, you may want to try to increase the value set for `key_buffer_size`.

The other ratio you need to consider concerns updated keys, which need to be written to disk as quickly as possible. Therefore, you should check this ratio:

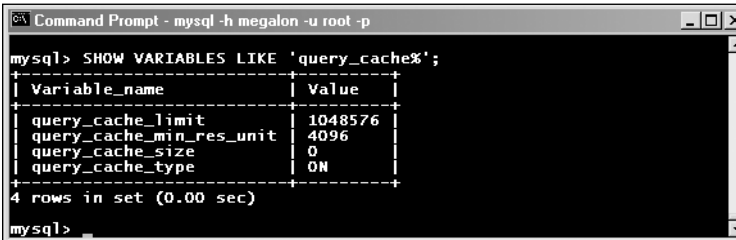
$$\text{Key_write_flushes} = \text{Key_writes} / \text{Key_write_requests}$$

You want this to be as close to 1 as possible. Again, if this figure doesn't approach the optimum, you'll want to increase `key_buffer_size`, if it's possible to do so without interfering with other memory allocations in the MySQL configuration.

Query Cache

Beginning with MySQL version 4.0.1, MySQL also has a *query cache*, which can help increase an application's speed dramatically when performing repetitive queries against your databases.

In order to make effective use of the query cache, you will need to make sure it is active and configured correctly. You can check for this by using `show variables`. The default values for the variables are shown in this example:



```

mysql> SHOW VARIABLES LIKE 'query_cache%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| query_cache_limit | 1048576 |
| query_cache_min_res_unit | 4096 |
| query_cache_size | 0 |
| query_cache_type | ON |
+-----+-----+
4 rows in set (0.00 sec)
mysql>

```

These variables control the query cache as follows:

query_cache_size: To enable query caching, set this to a nonzero value. This variable holds the total amount of memory (in bytes) set aside for storing cached queries. You might want to try 20MB or 40MB.

query_cache_limit: This is the maximum size for a cached result set. Resultsets larger than this won't be cached.

query_cache_min_res_unit: (MySQL 4.1 and above only) The default value is adequate in most cases. However, if you have a lot of small queries with small results, you may find that decreasing the value to 2048 or even 1024 bytes may improve performance. As you might expect, if you have a lot of very large queries and/or very large resultsets, increasing it to 8192, 16384, or even 32768 may speed up performance a bit.

query_cache_type: This can take one of three values: 0 = OFF (no results are cached), 1 = ON (all queries except those run with `SQL_NO_CACHE` are cached), and 2 = DEMAND (only queries run with `SQL_CACHE` are stored and retrieved).

When in use, the query cache stores the text and value of each `SELECT` statement. When another query is passed later, MySQL will check the cache first to see if a copy of it already exists; if it does, MySQL will return the result of the cache, rather than needing to process the entire query again. This can prove to be very useful and will provide a great speed advantage in an application such as an online catalog, where repetitive queries of products are being issued.



NOTE *The query cache does not return “stale” data. When data is modified, any relevant entries in the query cache are flushed, so that those queries are processed again to produce new resultsets.*

There is some overhead caused by having the query cache enabled. If you use many simple `SELECT` queries that aren't often repeated, having the query cache enabled may actually impede performance by 5% to 10%. However, using the query cache when your `SELECT` queries have large resultsets and are often reused, you may see performance increases on the order of 200% or even more.

By careful use and configuration of the query cache and the `SQL_CACHE` and `SQL_NO_CACHE` options for `SELECT` queries, you can cache only those queries that are largest and/or most often repeated, and not bother with those that are small, seldom repeated, or are most likely to return different results each time they're run. In this way, you'll be able to maximize the query cache's efficiency and thus that of your application.

Why Aren't My Queries Being Cached?

If you find that your queries are not being cached, there are two possible sources of problems that you can check. First, checking for cached queries is case-sensitive. Suppose you run this query:

```
SELECT * FROM mytable WHERE id=23;
```

Now let's say that later in the same application you run the same query as:

```
select * from mytable where id=23;
```

The second query will be considered a different query from the first one and rerun, rather than the results being pulled from the query cache. This is because MySQL's matching algorithm uses hashes in its query-matching routines.

Another reason that a query might not be cached is that in order to be cached, a query must begin with the `SELECT` keyword. It's perfectly legal in MySQL to begin a query with a comment, such as this:

```
/* get data from mytable for record 23 */ SELECT * FROM mytable WHERE id=23;
```

However, this query won't be cached because it doesn't begin with `SELECT`. Instead, placing your comment at the end of the query:

```
SELECT * FROM mytable WHERE id=23; /* get data from mytable for record 23 */
```

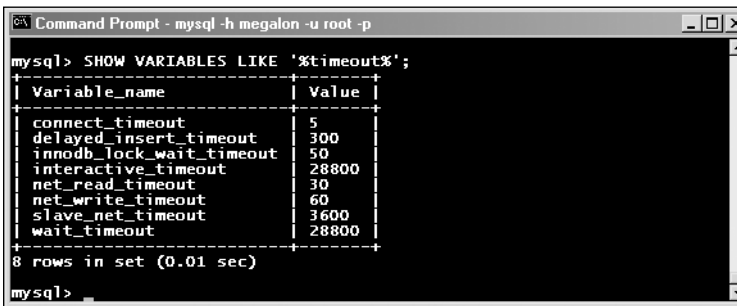
By observing these two rules—using uppercase or lowercase consistently and always beginning select queries with `SELECT`—you'll save yourself a lot of frustration as you're trying to fine-tune the performance of your MySQL applications.

Application Logic

Many people find that once they build an optimized database scheme for their application, they encounter bottlenecks and performance lags in their applications when trying to perform certain tasks. In this section, we'll discuss some of the causes of these. They include excessive connections, unnecessary or repetitive queries that could be combined into fewer queries, manipulating data in application code that could be handled just as well in a query, and database interoperability or database abstraction layers.

Repetitive Connections

Making repetitive connections to the database from within your application can cause a great amount of server overhead and can drastically reduce the performance of your application. Some people even have the mistaken idea that you must establish a new connection to MySQL each time you send a new query. They don't really understand the concept of a MySQL user session, or they don't realize how much time they have in between queries before MySQL closes the connection. You can easily find out how long a session will last using the appropriate `SHOW VARIABLES` command:



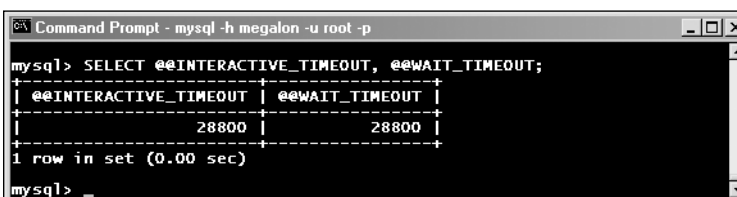
```

mysql> SHOW VARIABLES LIKE '%timeout%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| connect_timeout | 5 |
| delayed_insert_timeout | 300 |
| innodb_lock_wait_timeout | 50 |
| interactive_timeout | 28800 |
| net_read_timeout | 30 |
| net_write_timeout | 60 |
| slave_net_timeout | 3600 |
| wait_timeout | 28800 |
+-----+-----+
8 rows in set (0.01 sec)

mysql>

```

The important values to consider here are those for `interactive_timeout` and `wait_timeout`. As you can see, the default value for each of these is quite high: 28,800 seconds, which works out to eight hours. You can also obtain these values using a `SELECT` query, as shown here:



```

mysql> SELECT @@INTERACTIVE_TIMEOUT, @@WAIT_TIMEOUT;
+-----+-----+
| @@INTERACTIVE_TIMEOUT | @@WAIT_TIMEOUT |
+-----+-----+
| 28800 | 28800 |
+-----+-----+
1 row in set (0.00 sec)

mysql>

```

For web applications, the story is a bit different: a new connection to MySQL must be made on each new page. Even so, it's almost never necessary to establish a new connection more than once per page, unless you need to interact with more than one database.

We'll discuss connection-related issues and programming strategies in the next two sections.

One Connection, Multiple Queries

If you need to retrieve data in several different places in your application, it is quite unnecessary to make multiple connections to MySQL to perform each query.

Consider the following pseudocode:

```
connect to db
if order form submitted then
    insert order data into db
    if insert is successful
        print success message
    else
        print failure message

close db connection

connect to db

query db for customer info
while recordset is not empty
    get name, address, city, state, zip
    print name, address, city, state, zip

close db connection

connect to db

query db for order info
while recordset is not empty
    get orderID, total, date
    print orderID, total, date

close db connection

connect to db
```

```

query db for order details
while recordset is not empty
    get items from db where orderID is the same as customer
    print items

close db connection

```

By opening (and closing) multiple connections to the database, we are causing our application to perform much more slowly than if we used only one connection to the database, performed all of our needed queries, and then closed the connection.

Here is a better approach than in the previous example, once again using pseudocode, which you should be able to implement easily in your programming or scripting language of choice:

```

if form submitted then
    connect to database
    insert into database
    if insert is successful
        print success message
    else
        print failure message

query database for customer info
while recordset is not empty
    get name, address, city, state, zip
    print name, address, city, state, zip

query database for order info
while recordset is not empty
    get orderID, total, date
    print orderID, total, print date

query database for order details
while recordset is not empty
    get items from database where orderID is the same as customer
    print items

close database connection

```

Here, we made two changes to how the database connection was used that will help improve the performance of the application:

- In the first code section relative to the form submission, we moved the “connect to db” function to inside of the first `if` block, so that we connect to the database *only* if the form was submitted.
- We removed the repeated openings and closings of connections before and after each query. By doing this, we use only a single connection for all queries, and thus improve the application’s overall performance.

In addition, you should note that this simplifies the application code and makes it easier to debug and maintain.

Persistent Connections

The PHP 4 MySQL API provides both persistent and nonpersistent connection options for connecting to MySQL from within your application. There are no set rules that say when you should use either one; however, it is best to sometimes measure the performance of your application with each and determine which works better.

With nonpersistent connections, your application must establish a connection with the MySQL database server, authenticate itself, execute any queries, and, finally, close this connection when all database interaction by the script has been completed. However, with persistent connections, PHP will first check to see if there is already an open database connection using the same username and password, and, if one is found, it will execute the query using the existing connection. The connection will remain available for the next script executed by this user that may try to connect to the database using persistent connections.

PHP 4 uses the `mysql_pconnect()` function to establish persistent connections. Here’s the function prototype:

```
resource mysql_pconnect([string server[,
                        string username[,
                        string password[,
                        int client_flags]]]])
```

This function is employed as follows:

```
<?php
// mysql_test_pconnection.php

if (!mysql_pconnect("localhost", "mysql_user", "mysql_password"))
{
    printf("Could not connect: %s\n", mysql_error());
}
```

```

else
{
    print("Connection was successful");
}
?>

```

The downside to using persistent connections is that connections created by one user or application can persist unused for some time, and thus not be available to other users or applications. The PHP 5 MySQLi API does not support persistent connections for this reason.

Repetitive Queries

Repetitive use of queries in applications can also drastically reduce the performance of your application. Often, multiple SQL queries are written to perform a task that could otherwise be condensed into a single join, or could be better evaluated with your application code.

Consider our pseudocode from earlier; instead of making multiple queries to the database for the customer and order information, it can be condensed into one query that performs all of the given tasks.

```

if form submitted then
    connect to database
    insert into database
    if insert is successful
        print success message
    else
        print failure message

query database for customer info, order info and order details
while recordset is not empty
    get name, address, city, state, zip
    print name, address, city, state, zip

    get orderID, total, date
    print orderID, total, date

    get items from database where orderID is the same as customer
    print items

close db connection

```

For this example, our SQL query would change from three separate queries that looked like this:

```
# First Query
SELECT name, address, city, state, zip
FROM customers;
```

```
# Second Query
SELECT order_id, total, date
FROM orders
WHERE customer_id = '$customer_id';
```

```
# Third Query
SELECT items
FROM order_details
WHERE orderID = 'orderID';
```

to one query that looks like this:

```
SELECT c.name, c.address, c.city, c.state, cust.zip,
       o.orderID, o.total, o.date,
       d.items
FROM customers c
JOIN orders o USING (cust_id)
JOIN order_details d USING (order_id)
WHERE o.customer_id = '$customer_id';
```

Although these changes may seem small and insignificant, when used throughout your application, and for large datasets, they can help increase the overall performance of your application.



NOTE *If you need to repeat queries often, or submit queries that are very similar (differing only in the limiting values used), and you're running MySQL 4.1 or newer, you should look into using prepared statements for these. See Chapter 7 for more information about the MySQL Prepared Statements API, the programming platforms that currently support it, and the requirements for its use.*

Unnecessary Calculations

Frequently, mathematical operations that are done at the application level can be moved into the database level and can help increase the performance of your

application. We already discussed this and provided a fairly complex example in Chapter 4, but we wanted to touch on this again in a more general way.

For example, consider the following pseudocode example of a simple calculation:

```
connect to database
query database for var1 and var2
return data array
var3 = var1 * var2
print "The answer is: ", var3
disconnect from database
```

With this example, we must retrieve two variables from the database, load the values into an array for our application, perform the multiplication and assign the value to another variable, and then print it to our users. However, this query and process can be simplified by moving it to the database level. Consider the next example.

```
connect to database
query database for value of the expression (var1 * var2)
return value
print "The answer is: ", value
disconnect from database
```

Now the database performs the calculation and returns only the result. All that we need to accomplish with our application code is printing the answer. This is much simpler, quicker, easier to maintain, and easier to port between programming platforms and even to other databases.

Interoperability and Abstraction Layers

Interoperability and abstraction layers exist for most databases. Although they provide a simple and somewhat standard (to each interoperability layer) approach to connecting your application to multiple brands of databases, they can add a significant performance drop to any database-powered application.

The main reason that interoperability layers can be a performance bottleneck for your application is that they add multiple layers between your application and the database that you are trying to query. For example, most interoperability layers add at least two to three layers between your application and the database. This is illustrated in Figure 6-1.

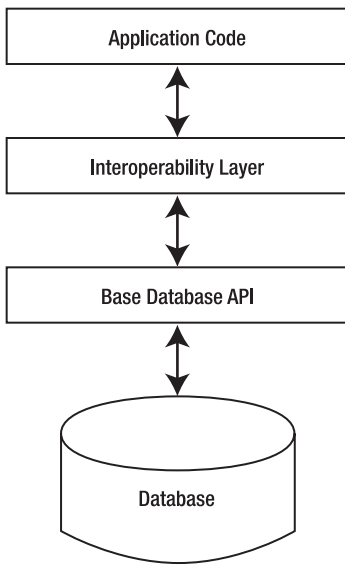


Figure 6-1. Relationships between a database, database interoperability layer, and an application

Generally, when you connect to the interoperability layer, it must translate your application's connection and query code to the correct database API before it can perform the desired operation. Then it must take the database server's response and translate it back into the format used in the application. However, if you don't use the interoperability layer, and you employ a native API for the database instead, the application will connect directly to the database, and then process the response from the database directly. This will eliminate the translation layers between, and thus eliminate the overhead of processing additional code for each of your queries to the database.

A database abstraction layer provides a “wrapper” for native API functions that simplifies working with a database. What we've said here about database interoperability layers also holds true for database abstraction layers: although database abstraction can make things easier for the programmer, there will be a performance penalty imposed by the transformation of abstracted function or method calls to the database's native API.

Summary

In this chapter, we discussed MySQL configuration issues as well as some others that may impact MySQL or MySQL-backed application performance. You can obtain a great deal of information about how well MySQL is operating by reading

the values of configuration and status variables using `SHOW VARIABLES` and `SHOW STATUS`. We discussed how these and some other useful `SHOW` commands are employed and what their output represents, concentrating on what they mean in terms of efficiency. Together with some of the log files that can be generated by MySQL, these can provide you with a valuable guide to fine-tuning the server, as well as pinpointing queries that are executing too slowly and other problems that might not be apparent until you've actually starting running your MySQL-backed applications.

We also took a very brief look at four common tools used for monitoring MySQL server performance: `mytop`, `phpMyAdmin`, `WinMySQLAdmin` for Win32 platforms, and the new multiplatform MySQL Administrator currently under development by MySQL AB. Each of these applications simplifies the task of keeping tabs on what and how the server is doing; the last two also provide GUI access for changing the server's configuration.

Another way in which MySQL allows you to improve performance is by taking advantage of its caching capabilities. By doing so, you can cut down dramatically on the number of times the server must read or write to disk instead of RAM, and this can speed up things considerably. MySQL has had good table and key caching for quite some time, and beginning with version 4.0, it also has query caching capabilities that, when understood and used properly, can dramatically reduce the time needed to perform repetitive queries—sometimes 200% or more.

We also looked at some application-oriented issues. Many of these we've touched on throughout this book, but we wanted to restate them as simply and clearly as possible. For instance, it probably can't be said enough times that it's silly and wasteful to send several queries separately from application code when these can be combined into a single query with a single resultset to be returned to the client. Another common source of inefficiency occurs when you perform calculations in application code that could be done as part of your queries. Doing the latter is almost always faster and means that there are fewer elements to return in query results. This also helps to make application code more compact and easier to maintain.

Finally, we talked a bit about database interoperability and abstraction layers, which are very popular among some application developers. While these can make it easy to write and port database-enabled applications, they can also incur a serious performance penalty because they interpose additional layers between the client and the database. It is always more efficient to write directly to the database's native API, such as the MySQL C API, or as close to it as the programming environment will allow. If portability is a concern, it's much better to design standards-compliant tables and queries than it is to rely on database-specific features and depend on an interoperability or abstraction layer to smooth out the differences for you.

What's Next

With a few exceptions, what we've discussed in this book so far can be accomplished from the command line. However, it's not very practical to type in queries and read them from a shell or DOS window every time you wish to use MySQL. You need to be able to connect your applications with MySQL, and to send data back and forth between the database and your applications' users. In Chapter 7, we'll look at some of the more common APIs available for use with MySQL, such as PHP 4's `mysql` extension, the new `ext/mysql` for PHP 5, and Python's `MySQLdb` module. While we'll concentrate on Open Source programming languages in our discussion, it's also true that, no matter which language or platform your applications run on, chances are very good that an interface to MySQL is available.

Some of these APIs have extra functions or methods for making it easier to work with MySQL features such as transactions, and we'll discuss these and show you examples. In addition, MySQL 4.1 and higher can provide enhanced functionality for programmers when using newer programming libraries or modules that take advantage of it. We'll also talk about the new Prepared Statements API, which allows for greater efficiency through the reuse of precompiled queries, and the Multiple Statements API, which permits you to transmit more than one SQL statement in a single query string and receive the results for all the queries sent in a single response.