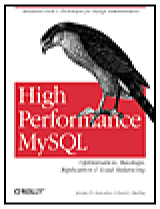


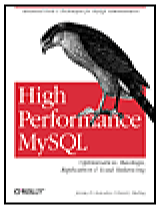
- Table of Contents
- Index
- Reviews
- Reader Reviews
- Errata
- Academic

High Performance MySQL

By Derek J. Balling, Jeremy Zawodny

Publisher :	O'Reilly
Pub Date :	April 2004
ISBN :	0-596-00306-4
Pages :	294
Slots :	1.0

In *High Performance MySQL* you will learn about MySQL indexing and optimization in depth so you can make better use of these key features. You will learn practical replication, backup, and load-balancing strategies with information that goes beyond available tools to discuss their effects in real-life environments. And you'll learn the supporting techniques you need to carry out these tasks, including advanced configuration, benchmarking, and investigating logs.



- Table of Contents
- Index
- Reviews
- Reader Reviews
- Errata
- Academic

High Performance MySQL

By

Derek J. Balling, Jeremy Zawodny

Publisher : O'Reilly
Pub Date : April 2004
ISBN : 0-596-00306-4
Pages : 294

Slots : 1.0

Copyright © 2004 O'Reilly Media, Inc.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *High Performance MySQL: Optimization, Backups, Replication, and Load Balancing* the image of a sparrow hawk; and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Preface

We had several goals in mind for this book. Many of them are derived from thinking about that mythical perfect MySQL book neither of us had read but kept looking for on bookstore shelves. Others come from a lot of experience helping other users put MySQL to work in their environments.

We wanted a book that wasn't just a SQL primer. We wanted a book with a title that didn't start or end

in some arbitrary time frame ("...in Thirty

Days," "Seven Days To a

Better...") and didn't imply that

the reader was a moron of some sort because he was reading our book.

Most of all we wanted a book that would help the reader take her MySQL skills to the next level. Every book we read focused almost exclusively on SQL command syntax or covered MySQL only at a very basic level. None really helped us to understand the deeper issues.

We wanted a book that went deeper and focused on real-world problems.

How can you set up a cluster of MySQL servers capable of handling millions upon millions of queries and ensure that things keep running even if a couple of the servers die?

We decided to write a book that focused not just on the needs of the MySQL application developer but also on the rigorous demands of the MySQL administrator, who needs to keep the system up and running no matter what his programmers or users may throw at the server.

Having said that, we assume that you are already relatively experienced with MySQL and, ideally, have read an introductory book on MySQL. In several chapters, we'll refer to common Unix tools for monitoring system performance, such as *top*, *vmstat*, and *sar*. If you're not already familiar with them (or their equivalent on your operating system), please take a bit of time to learn the basics. It will serve you well when we look at system performance and bottlenecks.

The Basic Layout of This Book

We fit a lot of complicated topics in this book. Here we'll explain how we put them together in an order that hopefully makes them easy for you to learn.

Back to Basics

The first two chapters are dedicated to the basicsthings you'll need to be familiar with before you get to

additional configuration details.

Chapter 1, reviews some rudimentary configuration basics. This book assumes a pretty good command of foundational MySQL

administration, but we'll go over the fundamentals briefly before digging deeper into the world of MySQL.

After that, Chapter 2, covers the various storage engines, or table types, that are part of MySQL. This is important because storage engine selection is one of the few things that can be nontrivial to change after you create a table. We review the various benefits (and potential pitfalls) of the various storage engines, and try to provide enough information to help you decide which engine is best for your particular application and environment.

Things to Reference as You Read the Rest of the Book

The next two chapters cover things you'll find yourself referencing time and again throughout the course of the book.

Chapter 3, discusses the basics of benchmarkingdetermining what sort of workloads your server can handle, how fast it can perform certain tasks, and so on.

You'll want to benchmark your application both

before and after a major change, so you can judge how effective your changes are. What seems to be a positive change may turn out to be a negative one under real-world stress.[1]

[1] Management folks also tend to like metrics they can point at and say, "See, this is how much our system improved after we spent $39.95 on that O'Reilly book!

Wasn't that a great

investment?"

In Chapter 4, we cover the various nuances of indexes. Many of the things we discuss in later chapters hinge on how well your application puts MySQL's indexes to work.

A firm understanding of indexes and how to optimize their use is something you'll find yourself returning to

repeatedly throughout the process.

Places to Tune Your Application

The next two chapters discuss areas in which the MySQL administrator, application designer, or MySQL programmer can make changes to improve performance of a MySQL application.

In Chapter 5, we discuss how the MySQL programmer might improve the performance of the MySQL queries themselves. This includes basics, such as how the query parser will parse the queries provided, as well as how to optimize queries for ideal performance.

Once the queries are optimized, the next step is to make sure the server's configuration is optimized to return those queries in the fastest possible manner. In Chapter 6, we discuss some ways to get the most out of your hardware, and to suggest hardware configurations that may provide better performance for larger-scale applications.

Scaling Upward After Making Changes

Once you've got a server up and running as best it can, you may find that one server simply isn't

enough. In Chapter 7, we discuss replicationthat is, getting your data copied automatically to multiple servers. When combined with the load-balancing lessons in Chapter 8, this will provide you with the groundwork for scaling your applications in a significant way.

Make Sure All That Work Isn't for Naught

Once you have configured your application, gotten it up and running, and replicated your database across multiple servers, your next task as a MySQL administrator is to keep it all going.

In Chapter 9, we discuss various backup and recovery strategies for your MySQL databases. These strategies help minimize your downtime in the event of inevitable hardware failure and ensures that your data survives such catastrophes.

Finally, Chapter 10, provides you with a firm grasp of some of the security issues involved in running a MySQL server.

More importantly, we offer many suggestions to allow you to prevent outside parties from harming the servers you have spent all this time trying to configure and optimize.

The Miscellany

There's a couple things we delve into that either don't "fit" in a particular chapter or are referenced often enough by multiple chapters that they deserve a bit of special attention all to themselves.

In Appendix A, we cover the output of the `SHOW STATUS` and `SHOW INNODB STATUS` commands. We attempt to decipher for the average administrator what all those variables mean and offer some ways to find potential problems based on their values relative to each other.

Appendix B, covers a program called *mytop*, which Jeremy wrote as an easy-to-use interface to what your MySQL server is presently doing. It functions much like the Unix *top* command and can be invaluable at all phases of the tuning process to find which MySQL

threads are using the most resources.

Finally, in Appendix C, we discuss *phpMyAdmin*, a web-based tool for administration of a MySQL server. *phpMyAdmin* can simplify many of the administrator's routine jobs and allow users to issue queries against the database without having to build a client or have shell access to the server.

Software Versions and Availability

Writing a

MySQL

book has proven to be quite a challenge. One reason is that MySQL is a moving target. In the two-plus years since Jeremy first wrote the outline for this book, numerous releases of MySQL have appeared.

MySQL 4.0 went from testing to stable, and as we go to press, MySQL

4.1 and 5.0 are both available as alpha versions. We had to revise the older text occasionally to remove references to limitations that were fixed after the fact.[2]

[2] Note to budding authors: write as fast as you can. The longer you drag it out, the more work you have to do.

We didn't use a single version of MySQL for this book. Instead, we used a handful of MySQL 4.0 and 4.1 releases, while occasionally looking back at how things used to be in the 3.23 days.

MySQL 5.0 is still in so much flux that we simply could not attempt to cover it in the first edition. The same is true for the (currently) new MySQL Administrator GUI tool.

Throughout this book, we assume a baseline version of MySQL 4.0.14

and have made an effort to note features or functionality that may not exist in older releases or that may exist only in the 4.1 series.

However, the definitive reference for mapping features to specific versions is the MySQL documentation itself. We expect that you'll find yourself visiting the

annotated online documentation (http://www.mysql.com/doc/) from time to time as you read this book.

Another great aspect of MySQL

is that it runs on all of today's popular platforms: Mac OS X, Windows, Linux, Solaris, FreeBSD: you name it! However, our experience is heavily skewed toward Linux and FreeBSD. When possible, we've tried to note differences Windows users are

likely to encounter, which tend to come in two flavors. First, file paths are completely different.

Chapter 1

contains numerous references to *C:\mysql* and the location of configuration files on Windows.

Perl is the other rough spot when dealing with MySQL on Windows.

MySQL comes with several useful utilities that are written in Perl and certain chapters in this book present example Perl scripts that form the basis of more complex tools you'll build.

However, Windows doesn't come with Perl. In order to use these scripts, you'll need to download a Windows version of Perl from ActiveState and install the necessary add-on modules (`DBI` and `DBD::mysql`) for MySQL access.

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text

Indicates menu titles, menu options, menu buttons, and keyboard accelerators (such as Alt and Ctrl).

Italic

Indicates new terms, example URLs, example email addresses, usernames, hostnames, filenames, file extensions, pathnames, directories, and utilities.

`Constant width`

Indicates elements of code, configuration options, variables, functions, modules, the contents of files, or the output from commands.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You don't need to

contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book doesn't require permission. Selling or distributing a CD-ROM of examples from O'Reilly books

does require permission. Answering a question by citing this book and quoting example code doesn't require permission. Incorporating a significant amount of example code from this book into your product's

documentation *does* require permission.

We appreciate, but don't require, attribution. An attribution usually includes the title, author, publisher, and ISBN.

For example: "*High Performance MySQL: Optimization, Backups, Replication, and Load*

Balancing, by Jeremy D. Zawodny and Derek J. Balling. Copyright 2004

O'Reilly Media, Inc.,

0-596-00306-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

Please address comments and questions concerning this book to the

publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and

any additional information. You can access this page at:

http://www.oreilly.com/catalog/hpmysql/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers,

and the O'Reilly Network, see our web site at:

http://www.oreilly.com

The

authors maintain a site called:

http://highperformancemysql.com

There you will find new information on MySQL releases, updates to the

tools shown in the book, and possibly other goodies such as

question-and-answer forums. Visit regularly!

Acknowledgments

A book like this doesn't come into being without help from literally dozens of people. Without their assistance, the book you hold in your hands would probably still be a bunch of sticky notes on the side of our monitors. This is the part of the book where we get to say whatever we like about the folks who helped us out, and we don't have to worry about music playing in the background telling us to shut up and go away, as you might see on TV

during an awards show.

We couldn't have completed this project without the constant prodding, begging, pleading, and support from our editor, Andy Oram.[3] If there is one person most responsible for the book in your hands, it's Andy. We really do appreciate the weekly nag sessions.

[3] Then again, if there's a second edition on the horizon, one might argue that this project is *not* complete.

Andy isn't alone, though. At O'Reilly there are a bunch of other folks who had some part in getting those sticky notes converted to a cohesive book that you'd be willing to read, so we also have to thank the production, illustration, and marketing folks for helping to pull this book together. And, of course, thanks to Tim O'Reilly for his continued commitment to producing some of the industry's finest documentation for popular open source software.

Finally, we'd both like to give a big thanks to the folks who agreed to look over the various drafts of the book and tell us all the things we were doing wrong: our reviewers. They spent part of their 2003 holiday break looking over roughly formatted versions of this text,

full of typos, misleading statements, and outright mathematical errors. In no particular order, thanks to Brian "Krow" Aker, Mark

"JDBC" Matthews, Jeremy

"the other Jeremy" Cole, Mike

"VBMySQL.com" Hillyer, Raymond

"Rainman" De Roo, Jeffrey

"Regex Master" Friedl, Jason

DeHaan, Dan Nelson, Steve "Unix

Wiz" Friedl, and last but not least, Kasia

"Unix Girl" Trapszo.

From Jeremy

I would again like to thank Andy for agreeing to take on this project and for continually beating on us for more chapter material.

Derek's help was essential for getting the last 20-30% of the book completed so that we wouldn't miss yet another target date. Thanks for agreeing to come on board late in the process and deal with my sporadic bursts of productivity, and for handling XML grunt work, Chapter 10

Appendix C, and all the other stuff I threw your way.

I also need to thank my parents for getting me that first Commodore 64 computer so many years ago. They not only tolerated the first 10

years of what seems to be a life-long obsession with electronics and computer technology, but quickly became supporters of my never-ending quest to learn and do more.

Next I'd like to thank a group of people I've had the distinct pleasure of working with while spreading MySQL religion at Yahoo during the last few years. Jeffrey Friedl and Ray Goldberger provided encouragement and feedback from the earliest stages of this undertaking. Along with them, Steve Morris, James Harvey, and Sergey Kolychev put up with my seemingly constant experimentation on the Yahoo! Finance MySQL servers, even when it interrupted their important work. Thanks also to the countless other Yahoos who have helped me find interesting MySQL

problems and solutions. And, most importantly, thanks for having the trust and faith in me

needed to put MySQL into some of the most important and visible parts of Yahoo's business.

Adam Goodman, the publisher and owner of *Linux Magazine*, helped me ease into the world of writing for a technical audience by publishing my first feature-length MySQL

articles back in 2001. Since then, he's taught me more than he realizes about editing and publishing and has encouraged me to continue on this road with my own monthly column in the magazine. Thanks, Adam.

Thanks to Monty and David for sharing MySQL with the world. Speaking of MySQL AB, thanks to all the other great folks there who have encouraged me in writing this: Kerry, Larry, Joe, Marten, Brian, Paul, Jeremy, Mark, Harrison, Matt, and the rest of the team there.

You guys rock.

Finally, thanks to all my weblog readers for encouraging me to write informally about MySQL and other technical topics on a daily basis.

And, last but not least, thanks to the Goon Squad.

From Derek

Like Jeremy, I've got to thank my family, for much the same reasons. I want to thank my parents for their constant goading that I should write a book, even if this isn't anywhere near what they had in mind. My grandparents helped me learn two valuable lessons, the meaning of the dollar and how much I would fall in love with computers, as she loaned me the money to buy my first Commodore VIC-20.

I can't thank Jeremy enough for inviting me to join him on the whirlwind book-writing roller coaster.

It's been a great experience and I look forward to working with him again in the future.

A special thanks goes out to Raymond De Roo, Brian Wohlgemuth, David Calafrancesco, Tera Doty, Jay Rubin, Bill Catlan, Anthony Howe, Mark O'Neal, George Montgomery, George Barber, and the myriad other people who patiently listened to me gripe about things, let me bounce ideas off them to see whether an outsider could understand what I was trying to say, or just managed to bring a smile to my face when I needed it most. Without you, this book might still have been written, but I almost certainly would have gone crazy in the process.

Chapter 1. Back To Basics

Many MySQL users and administrators slide into using MySQL. They hear its benefits, find that it's easy to install on

their systems (or better yet, comes pre-installed), and read a quick book on how to attach simple SQL operations to web sites or other applications.

It may take several months for the dragons to raise their heads.

Perhaps one particular web page seems to take forever, or a system failure corrupts a database and makes recovery difficult.

Real-life use of MySQL requires forethought and careand a little benchmarking and testing. This book is for the MySQL

administrator who has the basics down but realizes the need to go further. It's a good book to read after

you've installed and learned how to use MySQL but before your site starts to get a lot of traffic, and the dragons are breathing down your neck. (When problems occur during a critical service, your fellow workers and friendly manager start to take on decidedly dragon-like appearances.)

The techniques we teach are valuable in many different situations, and sometimes to solve different problems.

Replication, for instance, may be a matter of reliability for youan essential guarantee that your site will still be up if one or two systems fail.

But replication can also improve performance; we show you architectures and techniques that solve multiple problems.

We also take optimization far beyond the simple use of indexes and diagnostic (`EXPLAIN`) statements: this book tells you what the factors in good performance are, where bottlenecks occur, how to benchmark MySQL, and other advanced performance topics.

We ask for a little more patience and time commitment than the average introductory computer book. Our approach involves a learning cycle, and experience convinces us that it's

ultimately the fastest and most efficient way to get where you want.

After describing the problems we're trying to solve in a given chapter, we start with some background explanation. In other words, we give you a mental model for understanding what MySQL

is doing. Then we describe the options you have to solve the problem, and only after all that do we describe particular tools and techniques.

This book is clearly not the end of the line in terms of information.

Knowing that, we've started a web site, http://www.highperformancemysql.com, where we put useful scripts and new topics. See the Preface for more information.

Before we dig into how to tune your MySQL system to optimum performance, it's best if we go over a couple of

ground rules and make sure everyone is on the same page.

1.1 Binary Versus Compiled-From-Source Installations

There are two ways you can install MySQL.

As a novice administrator, you may have simply installed a binary package that had precompiled executables, libraries, and configuration files, and placed those files wherever the maker of the binary package decided they should go.

> It's exceedingly rare for a Windows user to compile his own copy of MySQL.
>
> If you're running MySQL on Windows, feel free to download your copy from the MySQL web site and skip this discussion.

Alternatively, for any number of reasons, you might have decided to compile the MySQL binaries on your own, by downloading a source tarball and configuring the installation to best meet your needs.

However, don't do so lightly. Compiling from source has led to countless hours of pain for some users, mostly due to subtle bugs in their compilers or thread libraries. For this very reason, the standard binaries provided by MySQL AB are statically linked. That means they are immune to any bugs in your locally installed libraries.

There aren't too many places where the issue of "binary versus

compiled-from-source" will come into play in the average MySQL tuning regimen, but they do happen. For example, in Chapter 10, our advice on chrooting your installation can be used only if every file MySQL needs is brought into a single directory tree, which might not be the case in a binary installation.

For a novice administrator on a simple installation, we recommend using a binary package (such as an RPM) to set up your system.

However, once you progress to the point of really needing to tinker with the "guts" of MySQL, you will

probably want to quickly go back, change a

`configure` flag, and recompile.

1.1.1 MySQL.com Binary Versus Distribution Binary

One thing to keep in mind is that there are a number of sources for binary packages, and nearly all of them set up the system differently.

For example, you can download the binary installation from the MySQL.com web site. You can also nstall the binary distribution included by your

Linux distribution vendor, or the one you grabbed from the

FreeBSD `ports`

collection. Finally, you can downloaded a binary for a platform that isn't officially supported, but on which someone is keeping a MySQL version current, such as the Amiga architecture.[1] In any of these cases, you will end up with different directory layouts, compilation options, etc.

[1] At the time that sentence was written, it was entirely theoretical: the thinking was "I'm not aware of anything, but

surely someone *will* do that!"

In researching it, we found that MySQL for Amiga was, indeed, happening. For those who read German, there's an article from *Amiga Magazine* at http://www.amiga-magazin.de/magazin/a08-01/mysql/

that describes how to do it, and a mailing list at http://groups.yahoo.com/group/Amiga_MySql/

for people working on it as well.

If you use the binary distributions from anyone other than MySQL AB, your support options may be significantly decreased, simply by virtue of having limited yourself to seeking help from those who use that particular distribution. Even a question as simple as, "Where is the

my.cnf file located on the FreeBSD port of MySQL?" is going to limit those who can respond to two groups: those who have run MySQL using the FreeBSD port, and those on the mailing list or newsgroup, etc. who have encountered that question before. On the plus side, if your distribution has automated security announcements and updates, you probably never need to worry about patching MySQL if a security flaw is discovered.

Many binary distributors of MySQL mold it to fit "their" layout. For example, the

Debian distribution places the config files in */etc/mysql/*, some language-specific files in */usr/share/mysql/*, the executables directly into */usr/bin/*, etc. It's not

"the Debian way" to segregate an

application's binaries; it incorporates them into the system as a whole. Likewise, in those places it does incorporate them, it does so in what may seem like an odd manner. For instance, you might expect config files to go directly into */etc/*, but instead they get put in

/etc/mysql/. It can be confusing if

you're trying to find everything you need to modify, or if you're trying to later convert from one type of installation to the other.

The MySQL.com-supplied tarball binary packages, however, behave more like the source-compilation process. All the filesconfiguration files, libraries, executables, and the database files themselvesend up in a single directory tree, created specifically for the MySQL install. This is typically */usr/local/mysql*, but it can be altered as needed at installation time. Because this behavior is much the same as a source-compiled installation, the available support from the MySQL community is much greater. It also makes things easier if you decide later to instead use a MySQL installation you compile from source.

On the other hand, the MySQL-supplied binary packages that are distributed using package-management formats such as RPM are laid out similarly to the format of the system they are designed for. For example, the RPM installation you get from MySQL.com will have its files laid out similarly to the Red Hat-supplied RPM. This is so because it's not uncommon for a Linux distribution to ship an RPM that hasn't been thoroughly tested and is broken in fairly serious ways. The RPM files MySQL.com distributes are intended as upgrade paths for users with such a problem so they can have "just what they have now, but it works."

Because of that, if you're going to install a binary you download from MySQL.com, we highly recommend using the tarball formatted files. They will yield the familiar directory structure the average MySQL administrator is used to seeing.

```
default-character-set=latin1

port=3306

socket=/var/lib/mysql/english.sock

default-character-set=latin1_de port=3307

socket=/var/lib/mysql/german.sock

default-character-set=euc_kr

port=3308

socket=/var/lib/mysql/korean.sock
```

```
$ <i>mysqld_safe --defaults-extra-file=/etc/my.german.cnf</i> $ <i>mysqld_safe --defaults-extra-file=/etc/my.english.cnf</i> $ <i>mysqld_safe --defaults-extra-file=/etc/my.korean.cnf</i>
```

```
# this is a comment

; so is this
```

```
key_buffer=128M # a comment can't go here

user = mysql

port = 3306

set-variable = key_buffer=384M

set-variable = tmp_table_size=32M

set-variable = key_buffer = 384M

set-variable=tmp_table_size=32M

set-variable = key_buffer=384M

key_buffer=384M

skip-bdb

# This is for a system with little memory (32M -
64M) where MySQL plays # a important part and
systems up to 128M very MySQL is used together
with # other programs (like a web server)

# Example mysql config file for medium systems.
```

```
#

# This is for a system with little memory (32M -
64M) where MySQL plays # a important part and
systems up to 128M very MySQL is used together
with # other programs (like a web server)
```

```
#

# You can copy this file to

# /etc/mf.cnf to set global options, # mysql-data-
dir/my.cnf to set server-specific options (in this #
installation this directory is /usr/local/mysq/var) or #
~/.my.cnf to set user-specific options.
```

```
#

# One can in this file use all long options that the
program supports.

# If you want to know which options a program
support, run the program # with --help option.

# The following options will be passed to all MySQL
clients [client]

#password = your_password

port = 3306

socket = /tmp/mysql.sock

# Here follows entries for some specific programs #
The MySQL server

[mysqld]

port = 3306

socket = /tmp/mysql.sock
```

```
skip-locking

set-variable = key_buffer=16M

set-variable = max_allowed_packet=1M

set-variable = table_cache=64

set-variable = sort_buffer=512K

set-variable = net_buffer_length=8K

set-variable = myisam_sort_buffer_size=8M

log-bin

server-id = 1

# Point the following paths to different dedicated
disks #tmpdir = /tmp/

#log-update = /path-to-dedicated-directory/hostname

# Uncomment the following if you are using BDB
tables #set-variable = bdb_cache_size=4M

#set-variable = bdb_max_lock=10000
```

```
# Uncomment the following if you are using
Innobase tables #innodb_data_file_path =
ibdata1:400M

#innodb_data_home_dir = /usr/local/mysql/var/

#innodb_log_group_home_dir =
/usr/local/mysql/var/

#innodb_log_arch_dir = /usr/local/mysql/var/

#set-variable = innodb_mirrored_log_groups=1

#set-variable = innodb_log_files_in_group=3

#set-variable = innodb_log_file_size=5M

#set-variable = innodb_log_buffer_size=8M

#innodb_flush_log_at_trx_commit=1

#innodb_log_archive=0

#set-variable = innodb_buffer_pool_size=16M

#set-variable =
innodb_additional_mem_pool_size=2M
```

```
#set-variable = innodb_file_io_threads=4

#set-variable = innodb_lock_wait_timeout=50

[mysqldump]

quick

set-variable = max_allowed_packet=16M

[mysql]

no-auto-rehash

# Remove the next comment character if you are not
familiar with SQL

#safe-updates

[isamchk]

set-variable = key_buffer=20M

set-variable = sort_buffer=20M

set-variable = read_buffer=2M

set-variable = write_buffer=2M
```

```
[myisamchk]

set-variable = key_buffer=20M

set-variable = sort_buffer=20M

set-variable = read_buffer=2M

set-variable = write_buffer=2M

[mysqlhotcopy]

interactive-timeout
```

```
mysql> <b>SET GLOBAL key_buffer=50M;</b>

mysql> <b>SET SESSION sort_buffer_size=50M;
</b>
```

```
$ <b>mysqld_safe -O key_buffer=50M</b>

$ <b>mysqld_safe --key_buffer=50M</b>
```

Command-line argument changes made in the *mysql.server* startup script will, obviously, survive from server restart to server restart, as long as that startup script is used to disable and reenable the server. It's important to point out, though, that it's usually better to have all your configuration declarations in a single place, so that maintenance doesn't become a game of hide-and-seek with the configuration options, trying to remember where you set which values.

1.3 The SHOW Commands

MySQL users often wonder how to find out what their server is actually doing at any point in timeusually when things start to slow down or behave strangely. You can look at operating system statistics to figure out how busy the server is, but that really doesn't reveal much. Knowing that the CPU is at 100% utilization or that there's a lot of disk I/O occurring provides a high-level picture of what is going on, but MySQL can tell far more.

Several `SHOW` commands provide a window into what's going on inside MySQL. They provide access to MySQL's configuration variables, ongoing statistics, and counters, as well as a description of what each client is doing.

1.3.1 SHOW VARIABLES

The easiest way to verify that configuration changes have taken effect is to ask MySQL for its current variable settings. The `SHOW VARIABLES` command does just that. Executing it produces quite a bit of output, which looks something like this:

```
mysql> SHOW VARIABLES;



+---------------------------------+---------------
----------------------------+

| Variable_name                   | Value
|

+---------------------------------+---------------
```

```
---------------------------+

| back_log                          | 20
|

| basedir                           | mysql
|

| binlog_cache_size                 | 32768
|

| character_set                     | latin1
|

| concurrent_insert                 | ON
|

| connect_timeout                   | 5
|

| datadir                           |
/home/mysql/data/                          |
```

The output continues from there, covering over 120 variables in total. The variables are listed in alphabetical order, which is convenient for reading, but sometimes related variables aren't anywhere near each other in the output. The reason for this is because as MySQL evolves, new variables are added with more descriptive names, but the older variable names aren't changed; it would break compatibility for any program that expects them.[3]

[3] In the rare event they do change, MySQL retains the old names as aliases for the new ones.

Many of the variables in the list may be adjusted by a `set-variable` entry in any of MySQL's configuration files. Some of them are compiled-in values that can not be changed. They're really constants (not variables), but they still show up in the output of `SHOW VARIABLES`. Still others are boolean flags.

Notice that the output of `SHOW VARIABLES` (and all of the `SHOW` commands, for that matter) looks just like the output of any SQL query. It's tabular data. MySQL returns the output in a structured format, making it easy to write tools that can summarize and act on the output of these commands. We'll put that to good use in later chapters.

1.3.2 SHOW PROCESSLIST

The other `SHOW` command we'll look at is `SHOW PROCESSLIST`. It outputs a list of what each thread is doing at the time you execute the command.[4] It's roughly equivalent to the `ps` or `top` commands in Unix or the Task Manager in Windows.

[4] Not all threads appear in the `SHOW PROCESSLIST` output. The thread that handles incoming network connections, for example, is never listed.

Executing it produces a process list in tabular form:

```
mysql> SHOW PROCESSLIST;



+----+---------+-----------+------+-------------+-
-----+-------+-----------------+

| Id | User    | Host      | db   | Command     |
Time | State | Info            |

+----+---------+-----------+------+-------------+-
-----+-------+-----------------+
| 17 | jzawodn | localhost | NULL | Query        |
0    | NULL  | show processlist |

+----+---------+-----------+------+-------------+-
-----+-------+------------------+
```

It's common for the `State` and `Info` columns to contain more information that produces lines long enough to wrap onscreen. So it's a good idea to use the `\G` escape in the *mysql* command interpreter to produce vertical output rather than horizontal output:

```
mysql> SHOW PROCESSLIST \G

*************************** 1. row
***************************

     Id: 17

   User: jzawodn

   Host: localhost

     db: NULL

Command: Query

   Time: 0

  State: NULL

   Info: show processlist
```

No matter which way you look at it, the same fields are included:

Id

The number that uniquely identifies this process. Since MySQL is a multi-threaded server, it really identifies the thread (or connection) and is unrelated to process IDs the operating system may use. As the operating system does with processes, MySQL starts numbering the threads at 1 and gives each new thread an ID one higher than the previous thread.

User

The name of the MySQL user connected to this thread.

Host

The name of the host or IP address from which the user is connected.

db

The database currently selected. This may be NULL if the user didn't specify a database.

Command

This shows the command state (from MySQL's internal point of view) that the thread is currently in. Table 1-1 lists each command with a description of when you are likely to see it. The commands roughly correspond to various function calls in MySQL's C API. Many commands represent very short-lived actions. Two of those that don't, `Sleep` and `Query`, appear frequently in day-to- day usage.

Table 1-1. Commands in SHOW PROCESSLIST output

Command	Meaning
`Binlog Dump`	The slave thread is reading queries from the master's binary log.
`Change user`	The client is logging in as a different user.
`Connect`	A new client is connecting.

Command	Meaning
`Connect Out`	The slave thread is connecting to the master to read queries from its binary log.
`Create DB`	A new database is being created.
`Debug`	The thread is producing debugging output. This is very uncommon.
`Delayed_insert`	The thread is processing delayed inserts.
`Drop DB`	A database is being dropped.
`Field List`	The client has requested a list of fields in a table.
`Init DB`	The thread is changing to a different database, typically as the result of a `USE` command.
`Kill`	The thread is executing a `KILL` command.
`Ping`	The client is pinging the server to see if it's still connected.
`Processlist`	The client is running `SHOW PROCESSLIST`.

Command	Meaning
`Query`	The thread is currently executing a typical SQL query: `SELECT`, `INSERT`, `UPDATE`, `DELETE`. This is the most common state other than `Sleep`.
`Quit`	The thread is being terminated as part of the server shutdown process.
`Refresh`	The thread is issuing the `FLUSH PRIVILEGES` command.
`Register Slave`	A slave has connected and is registering itself with the master.
`Shutdown`	The server is being shut down.
`Sleep`	The thread is idle. No query is being run.
`Statistics`	Table and index statistics are being gathered for the query optimizer.

Time

The number of seconds that the process has been running the current command. A process with a `Time` of 90 and `Command` of `Sleep` has been idle for a minute and a half.

State

Additional human-readable information about the state of this thread. Here's an example:

```
Slave connection: waiting for binlog update
```

This appears on the master server when a slave is actively replicating from it.

Info

This is the actual SQL currently being executed, if any. Only the first 100 characters are displayed in the output of `SHOW PROCESSLIST`. To get the full SQL, use `SHOW FULL PROCESSLIST`.

1.3.3 SHOW STATUS

In addition to all the variable information we can query, MySQL also keeps track of many useful counters and statistics. These numbers track how often various events occur. The `SHOW STATUS` command produces a tabular listing of all the statistics and their names.

To confuse matters a bit, MySQL refers to these counters as variables too. In a sense, they are variables, but they're not variables you can set. They change as the server runs and handles traffic; you simply read them and reset them using the `FLUSH STATUS` command.

The `SHOW STATUS` command, though, offers a lot of insight into your server's performance. It's covered in much greater depth in Appendix A.

1.3.4 SHOW INNODB STATUS

The `SHOW INNODB STATUS` status command provides a number of InnoDB-specific statistics. As we said earlier, InnoDB is one of MySQL's storage engines; look for more on storage engines in Chapter 2.

The output of `SHOW INNODB STATUS` is different from that of `SHOW STATUS` in that it reads more as a textual report, with section headings and such. There are different sections of the report that provide information on semaphores, transaction statistics, buffer information, transaction logs, and so forth.

`SHOW INNODB STATUS` is covered in greater detail along with `SHOW STATUS` in Appendix A. Also, note that in a future version of MySQL, this command will be replaced with a more generic `SHOW ENGINE STATUS` command.

Chapter 2. Storage Engines (Table Types)

One powerful aspect of MySQL that sets it apart from nearly every other database server is that it offers users many choices and options depending upon the user's

environment. From the server point of view, its default configuration can be changed to run well on a wide range of hardware. At the application development level, you have a variety of data types to choose from when creating tables to store records. But what's even more unusual is that you can choose the

type of table in which the records will be stored. You can even mix and match tables of different types in the same database!

Storage engines used to be called table types. From time to time we refer to them as table types when it's less awkward

to do so.

In this chapter, we'll show the major differences between the storage engines and why those differences are important.

We'll begin with a look at locking and concurrency

as well as transactionstwo concepts that are critical to understanding some of the major differences between the various engines. Then we'll discuss the process of selecting

the right one for your applications. Finally, we'll

look deeper into each of the storage engines and get a feel for their features, storage formats, strengths and weaknesses, limitations, and so on.

Before drilling down into the details, there are a few general concepts we need to cover because they apply across all the storage engines. Some aren't even specific to MySQL at all;

they're classic computer science problems that just

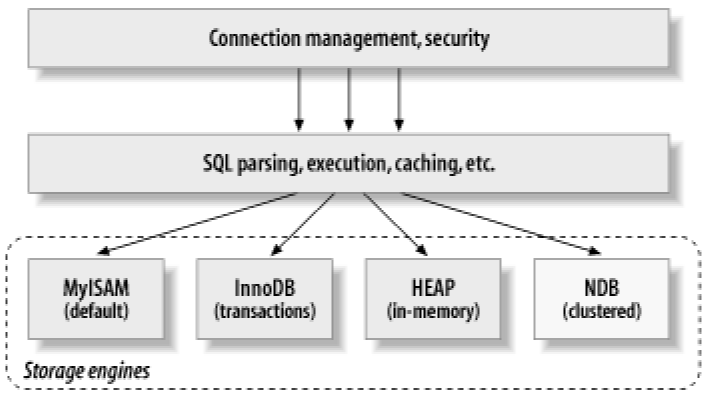
happen to occur frequently in the world of multiuser database servers.

2.1 MySQL Architecture

It will greatly aid your thinking about storage engines and the capabilities they bring to MySQL if you have a good mental picture of where they fit. Figure 2-1 provides a logical view of MySQL. It doesn't necessarily reflect the

low-level implementation, which is bound to be more complicated and less clear cut. However, it does serve as a guide that will help you understand how storage engines fit in to MySQL. (The NDB storage engine was added to MySQL just before this book was printed. Watch for it in the second edition.)

Figure 2-1. A logical view of MySQL's architecture



The topmost layer is composed of the services that aren't unique to MySQL. They're

services most network-based client/server tools or servers need: connection handling, authentication, security, etc.

The second layer is where things get interesting. Much of the brains inside MySQL live here, including query parsing, analysis, optimization, caching, and all the built-in functions (dates, times, math, encryption, etc.). Any functionality provided across storage engines lives at this level. Stored procedures, which will arrive in MySQL 5.0, also reside in this layer.

The third layer is made up of storage engines.

They're responsible for the storage and retrieval of all data stored "in" MySQL. Like

the various filesystems available for Linux, each storage engine has its own benefits and drawbacks. The good news is that many of the differences are transparent at the query layer.

The interface between the second and third layers is a single API not specific to any given storage engine. This API is made up of roughly 20 low-level functions that perform operations such as "begin a transaction" or

"fetch the row that has this primary

key" and so on. The storage engines

don't deal with SQL or communicate with each other; they simply respond to requests from the higher levels within MySQL.

2.2 Locking and Concurrency

The first of those problems is how to deal with concurrency and locking. In any data repository you have to be careful when more than one person, process, or client needs to change data at the same time. Consider, for example, a classic email box on a Unix system. The popular *mbox* file format is incredibly simple. Email messages are simply concatenated together, one after another. This simple format makes it very easy to read and parse mail messages. It also makes mail delivery easy: just append a new message to the end of the file.

But what happens when two processes try to deliver messages at the same time to the same mailbox? Clearly that can corrupt the mailbox, leaving two interleaved messages at the end of the mailbox file. To prevent corruption, all well-behaved mail delivery systems implement a form of locking to prevent simultaneous delivery from occurring. If a second delivery is attempted while the mailbox is locked, the second process must wait until it can acquire the lock before delivering the message.

This scheme works reasonably well in practice, but it provides rather poor concurrency. Since only a single program may make any changes to the mailbox at any given time, it becomes problematic with a high-volume mailbox, one that receives thousands of messages per minute. This exclusive locking makes it difficult for mail delivery not to become backlogged if someone attempts to read, respond to, and delete messages in that same mailbox. Luckily, few mailboxes are actually that busy.

2.2.1 Read/Write Locks

Reading from the mailbox isn't as troublesome.

There's nothing wrong with multiple clients reading the same mailbox simultaneously. Since they aren't making changes, nothing is likely to go wrong. But what happens if someone tries to delete message number 25 while programs are reading the mailbox? It depends. A reader could come away with a corrupted or inconsistent view of the mailbox. So to be safe, even reading from a mailbox requires special care.

Database tables are no different. If you think of each mail message as a record and the mailbox itself as a table, it's easy to see that the problem is the same. In many ways, a mailbox is really just a simple database table. Modifying records in a database table is very similar to removing or changing the content of messages in a mailbox file.

The solution to this classic problem is rather simple. Systems that deal with concurrent read/write access typically implement a locking system that consists of two lock types. These locks are usually known as *shared locks* and *exclusive locks*, or *read locks*

and *write locks*.

Without worrying about the actual locking technology, we can describe the concept as follows. Read locks on a resource are shared: many clients may read from the resource at the same time and not interfere with each other. Write locks, on the other hand, are exclusive, because it is safe to have only one client writing to the resource at given time and to prevent all reads when a client is writing. Why?

Because the single writer is free to make any changes to the resourceeven deleting it entirely.

In the database world, locking happens all the time. MySQL has to prevent one client from reading a piece of data while another is changing it. It performs this lock management internally in a way that is transparent much of the time.

2.2.2 Lock Granularity

One way to improve the concurrency of a shared resource is to be more selective about what is locked.

Rather than locking the entire resource, lock only the part that contains the data you need to change. Better yet, lock only the exact piece of data you plan to change. By decreasing the amount of data that is locked at any one time, more changes can occur simultaneouslyas long as they don't conflict with each other.

The downside of this is that locks aren't free.

There is overhead involved in obtaining a lock, checking to see whether a lock is free, releasing a lock, and so on. All this business of lock management can really start to eat away at performance because the system is spending its time performing lock management instead of actually storing and retrieving data. (Similar things happen when too many managers get involved in a software project.)

To achieve the best performance overall, some sort of balance is needed. Most commercial database servers don't give you much choice: you get what is known as row-level locking in your tables. MySQL, on the other hand, offers a choice in the matter.

Among the storage engines you can choose from in MySQL, you'll find three different granularities of locking. Let's have a look at them.

2.2.2.1 Table locks

The most basic and low-overhead locking strategy available is a *table lock*, which is analogous to the mailbox locks described earlier. The table as a whole is locked on an all-or-nothing basis. When a client wishes to write to a table (insert, delete, or update, etc.), it obtains a write lock that keeps all other read or write operations at bay for the duration of the operation. Once the write has completed, the table is unlocked to allow those waiting operations to continue. When nobody is writing, readers obtain read locks that allow other readers to do the same.

For a long time, MySQL provided only table locks, and this caused a great deal of concern among database geeks. They warned that MySQL

would never scale up beyond toy projects and work in the real world.

However, MySQL is so much faster than most commercial databases that table locking doesn't get in the way nearly as much as the naysayers predicted it would.

Part of the reason MySQL doesn't suffer as much as expected is because the majority of applications for which it is used consist primarily of read queries. In fact, the MyISAM engine (MySQL's default) was built assuming that 90% of all queries run against it will be reads. As it turns out, MyISAM tables perform very well as long as the ratio of reads to writes is very high or very low.

2.2.2.2 Page locks

A slightly more expensive form of locking that offers greater concurrency than table locking, a *page* *lock* is a lock applied to a portion of a table known as a page. All the records that reside on the same page in the table are affected by the lock. Using this scheme, the main factor influencing concurrency is the page size; if the pages in the table are large, concurrency will be worse than with smaller pages.

MySQL's BDB (Berkeley DB) tables use page-level locking on 8-KB pages.

The only hot spot in page locking is the last page in the table. If records are inserted there at regular intervals, the last page will be locked frequently.

2.2.2.3 Row locks

The locking style that offers the greatest concurrency (and carries the greatest overhead) is the *row*

lock. In most applications, it's relatively rare for several clients to need to update the exact same row at the same time. Row-level locking, as it's commonly known, is available in

MySQL's InnoDB tables. InnoDB

doesn't use a simple row locking mechanism, however.

Instead it uses row-level locking in conjunction with a multiversioning scheme, so let's have a look at that.

2.2.3 Multi-Version Concurrency Control

There is a final technique for increasing concurrency: Multi-Version Concurrency Control (MVCC). Often referred to simply as *versioning*, MVCC is used by Oracle, by PostgreSQL, and by MySQL's InnoDB storage engine. MVCC can be thought of as a new twist on row-level locking. It has the added benefit of allowing nonlocking reads while still locking the necessary records only during write operations. Some of MVCC's other properties will be of particular interest when we look at transactions in the next section.

So how does this scheme work? Conceptually, any query against a table will actually see a snapshot of the data as it existed at the time the query beganno matter how long it takes to execute. If you've never experienced this before, it may sound a little crazy. But give it a chance.

In a versioning system, each row has two additional, hidden values associated with it. These values represent when the row was created and when it was expired (or deleted). Rather than storing the actual time at which these events occur, the database stores the version number at the time each event occurred. The *database version* (or *system version*) is a number that increments each time a query[1] begins. We'll call these two values the *creation* *id*

and the *deletion*

id.

[1] That's not quite true. As you'll see when we start talking about transactions later, the version number is incremented for each transaction rather than each query.

Under MVCC, a final duty of the database server is to keep track of all the running queries (with their associated version numbers).

Let's see how this applies to particular operations:

SELECT

When records are selected from a table, the server must examine each row to ensure that it meets several criteria:

- Its creation id must be less than or equal to the system version number. This ensures that the row was created before the current query began.
- Its deletion id, if not null, must be greater than the current system version. This ensures that the row wasn't deleted before the current query began.

- Its creation id can't be in the list of running queries. This ensures that the row wasn't added or changed by a query that is still running.
- Rows that pass all of these tests may be returned as the result of the query.

INSERT

> When a row is added to a table, the database server records the current version number along with the new row, using it as the row's creation id.

DELETE

> To delete a row, the database server records the current version number as the row's deletion id.

UPDATE

> When a row is modified, the database server writes a new copy of the row, using the version number as the new row's creation id. It also writes the version number as the old row's deletion id.

The result of all this extra record keeping is that read queries never lock tables, pages, or rows. They simply read data as fast as they can, making sure to select only rows that meet the criteria laid out earlier. The drawbacks are that the server has to store a bit more data with each row and do a bit more work when examining rows.

Table 2-1 summarizes the various locking models and concurrency in MySQL.

Table 2-1. Locking models and concurrency in MySQL

Locking strategy	Concurrency	Overhead	Engines
Table socks	Lowest	Lowest	MyISAM, Heap, Merge
Page locks	Modest	Modest	BDB
Multiversioning	Highest	High	InnoDB

2.3 Transactions

You can't examine the more advanced features of a database system for very long before transactions enter the mix. A transaction is a group of SQL queries that are treated *atomically*, as a single unit of work. Either the entire group of queries is applied to a database, or none of them are. Little of this section is specific to MySQL. If you're already familiar with ACID transactions, feel free to skip ahead to the section "Transactions in MySQL."

A banking application is the classic example of why transactions are necessary. Imagine a bank's database with a two tables: checking and savings. To move $200 from Jane's checking account to her savings account, you need to perform at least three steps:

1. Make sure her checking account balance is greater than $200.
2. Subtract $200 from her checking account balance.
3. Add $200 to her savings account balance.

The entire operation should be wrapped in a transaction so that if any one of the steps fails, they can all be rolled back.

A transaction is initiated (or opened) with the `BEGIN` statement and applied with `COMMIT` or *rolled back* (undone) with `ROLLBACK`. So the SQL for the transaction might look like this:

```
BEGIN;

[step 1] SELECT balance FROM checking WHERE
customer_id = 10233276;

[step 2] UPDATE checking SET balance = balance -
200.00 WHERE customer_id = 10233276;
```

```
[step 3] UPDATE savings  SET balance = balance +
200.00 WHERE customer_id = 10233276;

         COMMIT;
```

But transactions alone aren't the whole story. What happens if the database server crashes while performing step 3? Who knows? The customer probably just lost $200. What if another process comes along between Steps 2 and 3 and removes the entire checking account balance? The bank has given the customer a $200 credit without even knowing it.

Simply having transactions isn't sufficient unless the database server passes what is known as the *ACID test*. ACID is an acronym for Atomicity, Consistency, Isolation, and Durabilityfour tightly related criteria that are required in a well-behaved transaction processing system. Transactions that meet those four criteria are often referred to as *ACID transactions*.

Atomicity

> Transactions must function as a single indivisible unit of work. The entire transaction is either applied or rolled back. When transactions are atomic, there is no such thing as a partially completed transaction: it's all or nothing.

Consistency

The database should always move from one consistent state to the next. Consistency ensures that a crash between Steps 2 and 3 doesn't result in $200 missing from the checking account. Because the transaction is never committed, none of the transaction's changes are ever reflected in the database.

Isolation

The results of a transaction are usually invisible to other transactions until the transaction is complete. This ensures that if a bank account summary runs after Step 2, but before Step 3, it still sees the $200 in the checking account. When we discuss isolation levels, you'll understand why we said *usually* invisible.

Durability

Once committed, the results of a transaction are permanent. This means that the changes must be recorded in such a way that system crashes won't lose the data. Of course, if the database server's disks fail, all bets are off. That's a hardware problem. We'll talk more about how you can minimize the effects of hardware failures in Chapter 6.

2.3.1 Benefits and Drawbacks

ACID transactions ensure that banks don't lose your money. By wrapping arbitrarily complex logic into single units of work, the database server takes some of the burden off application developers. The database server's ACID properties offer guarantees that reduce the need for code guarding against race conditions and handling crash recovery.

The downside of this extra security is that the database server has to do more work. It also means that a database server with ACID transactions will generally require more CPU power, memory, and disk space than one without them. As mentioned earlier, this is where MySQL's modularity comes into play. Because you can decide on a per-table basis if you need ACID transactions or not, you don't need to pay the performance penalty on a table that really won't benefit from transactions.

2.3.2 Isolation Levels

The previous description of isolation was a bit simplistic. Isolation is more complex than it might first appear because of some peculiar cases that can occur. The SQL standard defines four isolation levels with specific rules for which changes are and aren't visible inside and outside a transaction. Let's look at each isolation level and the type of problems that can occur.

2.3.2.1 Read uncommitted

In the *read uncommitted* isolation level, transactions can view the results of uncommitted transactions. At this level, many problems can occur unless you really, really know what you are doing and have a good reason for doing it. Read uncommitted is rarely used in practice. Reading uncommitted data is also known as a *dirty read*.

2.3.2.2 Read committed

The default isolation level for most database systems is *read committed*. It satisfies the simple definition of isolation used earlier. A transaction will see the results only of transactions that were already committed when it began, and its changes won't be visible to others until it's committed.

However, there are problems that can occur using that definition. To visualize the problems, refer to the sample data for the `Stock` and `StockPrice` tables as shown in Table 2-2 and Table 2-3.

Table 2-2. The Stock table

id	Ticker	Name
1	MSFT	Microsoft
2	EBAY	eBay
3	YHOO	Yahoo!
4	AMZN	Amazon

Table 2-3. The StockPrice table

stock_id	date	open	high	low	close
1	2002-05-01	21.25	22.30	20.18	21.30
2	2002-05-01	10.01	10.20	10.01	10.18
3	2002-05-01	18.23	19.12	18.10	19.00
4	2002-05-01	45.55	46.99	44.87	45.71
1	2002-05-02	21.30	21.45	20.02	20.21
2	2002-05-02	10.18	10.55	10.10	10.35
3	2002-05-02	19.01	19.88	19.01	19.22
4	2002-05-02	45.69	45.69	44.03	44.30

Imagine you have a Perl script that runs nightly to fetch price data about your favorite stocks. For each stock, it fetches the data and adds a record to the `StockPrice` table with the day's numbers. So to update the information for Amazon.com, the transaction might look like this:

```
BEGIN;

SELECT @id := id FROM Stock WHERE ticker = 'AMZN';
```

```
INSERT INTO StockPrice VALUES (@id, '2002-05-03',
20.50, 21.10, 20.08, 21.02);

COMMIT;
```

But what if, between the select and insert, Amazon's `id` changes from 4 to 17 and a new stock is added with `id` 4? Or what if Amazon is removed entirely? You'll end up inserting a record with the wrong `id` in the first case. And in the second case, you've inserted a record for which there is no longer a corresponding row in the `Stock` table. Neither of these is what you intended.

The problem is that you have a *nonrepeatable read* in the query. That is, the data you read in the `SELECT` becomes invalid by the time you execute the `INSERT`. The repeatable read isolation level exists to solve this problem.

2.3.2.3 Repeatable read

At the *repeatable read* isolation level, any rows that are read during a transaction are locked so that they can't be changed until the transaction finishes. This provides the perfect solution to the problem mentioned in the previous section, in which Amazon's `id` can change or vanish entirely. However, this isolation level still leaves the door open to another tricky problem: phantom reads.

Using the same data, imagine that you have a script that performs some analysis based on the data in the `StockPrice` table. And let's assume it does this while the nightly update is also running.

The analysis script does something like this:

```
BEGIN;

SELECT * FROM StockPrice WHERE close BETWEEN 10
and 20;
```

```
// think for a bit

SELECT * FROM StockPrice WHERE close BETWEEN 10
and 20;

COMMIT;
```

But the nightly update script inserts between those two queries new rows that happen to match the `close BETWEEN 10 and 20` condition. The second query will find more rows that the first one! These additional rows are known as *phantom rows* (or simply phantoms). They weren't locked the first time because they didn't exist when the query ran.

Having said all that, we need to point out that this is a bit more academic than you might think. Phantom rows are such a common problem that InnoDB's locking (known as *next-key locking*) prevents this from happening. Rather than locking only the rows you've touched in a query, InnoDB actually locks the slot following them in the index structure as well.

2.3.2.4 Serializable

The highest level of isolation, *serializable*, solves the phantom read problem by ordering transactions so that they can't conflict. At this level, a lot of timeouts and lock contention may occur, but the needs of your application may bring you to accept the decreased performance in favor of the data stability that results.

Table 2-4 summarizes the various isolation levels and the drawbacks associated with each one. Keep in mind that as you move down the list, you're sacrificing concurrency and performance for increased safety.

Table 2-4. ANSI SQL isolation levels

Isolation level	Dirty reads possible	Non-repeatable reads possible	Phantom reads possible
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

2.3.3 Deadlocks

Whenever multiple transactions obtain locks, there is the danger of encountering a deadlock condition. Deadlocks occur when two transactions attempt to obtain conflicting locks in a different order.

For example, consider these two transactions running against the `StockPrice` table:

Transaction #1:

```
BEGIN;
```

```
UPDATE StockPrice SET close = 45.50 WHERE stock_id
= 4 and date = '2002-05-01';

UPDATE StockPrice SET close = 19.80 WHERE stock_id
= 3 and date = '2002-05-02';

COMMIT;
```

Transaction #2:

```
BEGIN;

UPDATE StockPrice SET high  = 20.12 WHERE stock_id
= 3 and date = '2002-05-02';

UPDATE StockPrice SET high  = 47.20 WHERE stock_id
= 4 and date = '2002-05-01';

COMMIT;
```

If you're unlucky, each transaction will execute its first query and update a row of data, locking it in the process. Each transaction will then attempt to update its second row only to find that it is already locked. Left unchecked, the two transactions will wait for each other to completeforever.

To combat this problem, database systems implement various forms of deadlock detection and timeouts. The more sophisticated systems, such as InnoDB, will notice circular dependencies like the previous example and return an error. Others will give up after the query exceeds a timeout while waiting for a lock. InnoDB's default timeout is 50 seconds. In either case, applications that use transactions need to be able to handle deadlocks and possibly retry transactions.

2.3.4 Transaction Logging

Some of the overhead involved with transactions can be mitigated through the use of a transaction log. Rather than directly updating the tables on disk each time a change occurs, the system can update the in-memory copy of the data (which is very fast) and write a record of the change to a *transaction log* on disk. Then, at some later time, a process (or thread) can actually apply the changes that the transaction log recorded. The serial disk I/O required to append events to the log is much faster than the random seeks required to update data in various places on disk.

As long as events are written to the transaction log before a transaction is considered committed, having the changes in a log will not affect the durability of the system. If the database server crashes before all changes have been applied from the transaction log, the database will continue applying changes from the transaction log when it is restarted and before it accepts new connections.

2.3.5 Transactions in MySQL

MySQL provides two transaction-safe storage engines: Berkeley DB (BDB) and InnoDB. Their specific properties are discussed in next section. Each one offers the basic `BEGIN`/`COMMIT`/`ROLLBACK` functionality. They differ in their supported isolation levels, locking characteristics, deadlock detection, and other features.

2.3.5.1 AUTOCOMMIT

By default MySQL operates in `AUTOCOMMIT` mode. This means that unless you've explicitly begun a transaction, it automatically executes each query in a separate transaction. You can enable `AUTOCOMMIT` for the current connection by running:

```
SET AUTOCOMMIT = 1;
```

Disable it by executing:

```
SET AUTOCOMMIT = 0;
```

Changing the value of `AUTOCOMMIT` has no effect on non-transaction-safe tables such as MyISAM or HEAP.

2.3.5.2 Implicit commits

Certain commands, when issued during an open transaction, cause MySQL to commit the transaction before they execute. Typically these are commands that make significant changes, such as removing or renaming a table.

Here is the list of commands for which MySQL implicitly commits a transaction:

- `ALTER TABLE`
- `BEGIN`
- `CREATE INDEX`
- `DROP DATABASE`
- `DROP TABLE`
- `RENAME TABLE`
- `TRUNCATE`
- `LOCK TABLES`

- `UNLOCK TABLES`

As additional features are added to MySQL, it is possible that other commands will be added to the list, so be sure to check the latest available documentation.

2.3.5.3 Isolation levels

MySQL allows you to set the isolation level using the `SET TRANSACTION ISOLATION LEVEL` command. Unless otherwise specified, the isolation level is changed beginning with the next transaction.

To set the level for the whole session (connection), use: SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED

Here's how to set the global level:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL
SERIALIZABLE
```

MySQL recognizes all four ANSI standard isolation levels, and as of Version 4.0.5 of MySQL, InnoDB supports all of them:

- `READ UNCOMMITTED`
- `READ COMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

The default isolation level can also be set using the `--transaction-isolation` command-line option when starting the server or set via *my.cnf*.

2.3.5.4 Mixing storage engines in transactions

Transaction management in MySQL is currently handled by the underlying storage engines, not at a higher level. Thus, you can't reliably mix tables stored in transactional engines (such as InnoDB and BDB) in a single transaction. A higher-level transaction management service may someday be added to MySQL, making it safe to mix and match transaction-safe tables in a transaction. Until then, don't expect it to work.

If you mix transaction-safe and non-transaction-safe tables (such as InnoDB and MyISAM) in a transaction, the transaction will work properly if all goes well. However, if a rollback is required, the changes to the non-transaction-safe table won't be undone. This leaves the database in an inconsistent state that may be difficult to recover from (and renders the entire point of transactions moot).

2.3.5.5 Simulating transactions

At times you may need the behavior of transactions when you aren't using a transaction-safe table. You can achieve something like transactions using MySQL's `LOCK TABLES` and `UNLOCK TABLES` commands. If you lock the tables that will be involved in the transaction and keep track of any changes that you make (in case you need to simulate a rollback), you'll have something equivalent to running at the serializable isolation level. But the process is kludgy and error prone, so if you really need transactions, we recommend using a transactional storage engine.

```
SELECT COUNT(*) FROM mytable

SELECT COUNT(*) FROM table WHERE ...

ALTER TABLE mytable TYPE = BDB;

BEGIN;



INSERT INTO innodb_table SELECT * FROM
myisam_table WHERE id BETWEEN x AND y;

COMMIT;

CREATE TABLE newtable LIKE mytable;

INSERT INTO newtable SELECT * FROM mytable;
```

Whichever method you use, if you're dealing with a large volume of data, it's often more efficient to copy the data before adding indexes to the new table.

2.5 The Storage Engines

Now it's time to look at each of MySQL's storage engines in more detail. Table 2-5 summarizes some of the high-level characteristics of the handlers. The following sections provide some basic highlights and background about each table handler as well as any unusual characteristics and interesting features.

Before going further, it's worth noting that this isn't an exhaustive discussion of MySQL's storage engines. We assume that you've read (or at least know where to find) the information in the *MySQL Reference Manual*.

Table 2-5. Storage engine features in MySQL

Attribute	MyISAM	Heap	BDB	InnoDB
Transactions	No	No	Yes	Yes
Lock granularity	Table	Table	Page (8 KB)	Row
Storage	Split files	In-memory	Single file per table	Tablespace(s)
Isolation levels	None	None	Read committed	All

Attribute	MyISAM	Heap	BDB	InnoDB
Portable format	Yes	N/A	No	Yes
Referential integrity	No	No	No	Yes
Primary key with data	No	No	Yes	Yes
MySQL caches data records	No	Yes	Yes	Yes
Availability	All versions	All versions	MySQL-Max	All Versions[3]

[3] Prior to MySQL 4.0, InnoDB was available in MySQL-Max only.

Most of MySQL's disk-based tables have some basic things in common. Each database in MySQL is simply a subdirectory of MySQL's data directory in the underlying filesystem.[4] Whenever you create a table, MySQL stores the table definition in a *.frm* file with the same name as the table. Thus, when you create a table named `MyTable`, MySQL stores the table definition in *MyTable.frm*.

[4] In MySQL 5.0, the term "database" will likely morph into "schema."

To determine the type of a table, use the `SHOW TABLE STATUS` command. For example, to examine the user table in the `mysql` database, you execute the following:

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
*************************** 1. row
***************************
           Name: user
           Type: MyISAM
     Row_format: Dynamic
           Rows: 6
 Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
   Index_length: 2048
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
    Update_time: 2002-01-24 21:56:29
     Check_time: NULL
 Create_options:
        Comment: Users and global privileges
1 row in set (0.06 sec)
```

Notice that it's a MyISAM table. You might also notice a lot of other information and statistics in the output. Let's briefly look at what each line means:

Name

> The table's name.

Type

> The table's type. Again, in some versions of MySQL, this may say "Engine" rather than "Type."

Row_format

> Dynamic, Fixed, or Compressed. Dynamic rows vary in length because they contain variable-length fields such as `VARCHAR` or `BLOB`. Fixed rows, which are always the same size, are made up of fields that don't vary in length, such as `CHAR` and `INTEGER`. Compressed rows exist only in compressed tables (see the later section "Compressed MyISAM").

Rows

The number of rows in the table. For non-transactional tables, this number is always accurate. For transactional tables, it is usually an estimate.

Avg_row_length

How many bytes the average row contains.

Data_length

How much data (in bytes) the entire table contains.

Max_data_length

The maximum amount of data this table can hold. In a MyISAM table with dynamic (variable length) rows, the index file for a table (*tablename.MYI*) stores row locations using 32-bit pointers into the data file (*tablename.MYD*). That means it can address only up to 4 GB of space by default. See Section 2.5.1 for more details. For MyISAM tables with fixed-length rows, the limit is just under 4.3 billion rows.

Index_length

How much space is consumed by index data.

Data_free

The amount of space that has been allocated but is currently unused.

Auto_increment

The next `AUTO_INCREMENT` value.

Create_time

When the table was first created.

Update_time

When data in the table last changed.

Check_time

When the table was last checked using `CHECK TABLE` or *myisamchk*.

Create_options

Any other options that were specified when the table was created.

Comment

The comments, if any, that were set when the table was created.

2.5.1 MyISAM Tables

As MySQL's default storage engine, MyISAM provides a good compromise between performance and useful features. Versions of MySQL prior to 3.23 used the Index Sequential Access Method (ISAM) table format. In Version 3.23, ISAM tables were deprecated in favor of MyISAM, an enhanced ISAM format.[5] MyISAM tables don't provide transactions or a very granular locking model, but they do have full-text indexing (see Chapter 4), compression, and more.

[5] ISAM tables may be used in MySQL 4.0 and 4.1. Presumably they'll vanish sometime in the 5.x release cycle. If you're still using ISAM tables, it's time to upgrade to MyISAM!

2.5.1.1 Storage

In MyISAM storage, there are typically two files: a data file and an index file. The two files bear *.MYD* and *.MYI* extensions, respectively. The MyISAM format is platform-neutral, meaning you can copy the data and index files from an Intel-based server to a Macintosh PowerBook or Sun SPARC without any trouble.

MyISAM tables can contain either dynamic or static (fixed-length) rows. MySQL decides which format to use based on the table definition. The number of rows a MyISAM table can hold is limited primarily by the available disk space on your database server and the largest file your operating system will let you create. Some (mostly older) operating systems have been known to cut you off at 2 GB, so check your local documentation.

However, MyISAM files with variable-length rows, are set up by default to handle only 4 GB of data, mainly for efficiency. The index uses 32-bit pointers to the data records. To create a MyISAM table that can hold more than 4 GB, you must specify values for the `MAX_ROWS` and `AVG_ROW_LENGTH` options that represent ballpark figures for the amount of space you need:

```
CREATE TABLE mytable (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

) MAX_ROWS = 1000000000 AVG_ROW_LENGTH = 32;
```

In the example, we've told MySQL to be prepared to store at least 32 GB of data in the table. To find out what MySQL decided to do, simply ask for the table status:

```
mysql> SHOW TABLE STATUS LIKE 'mytable' \G

*************************** 1. row
***************************

           Name: mytable

           Type: MyISAM

     Row_format: Fixed
```

```
           Rows: 0

 Avg_row_length: 0

    Data_length: 0

Max_data_length: 98784247807

   Index_length: 1024

      Data_free: 0

 Auto_increment: NULL

    Create_time: 2002-02-24 17:36:57

    Update_time: 2002-02-24 17:36:57

     Check_time: NULL

 Create_options: max_rows=1000000000
avg_row_length=32

        Comment:

1 row in set (0.05 sec)
```

As you can see, MySQL remembers the create options exactly as specified. And it chose a representation capable of holding 91 GB of data!

2.5.1.2 Other stuff

As one of the oldest storage engines included in MySQL, MyISAM tables have a number of features that have been developed over time specifically to fill niche needs uncovered through years of use:

Locking and concurrency

Locking in MyISAM tables is performed at the table level. Readers obtain shared (read) locks on all tables they need to read. Writers obtain exclusive (write) locks.

Automatic repair

If MySQL is started with the `--myisam-recover` option, the first time it opens a MyISAM table, it examines the table to determine whether it was closed properly. If it was not (probably because of a hardware problem or power outage), MySQL scans the table for problems and repairs them. The downside, of course, is that your application must wait while a table it needs is being repaired.

Manual repair

You can use the `CHECK TABLE mytable` and `REPAIR TABLE mytable` commands to check a table for errors and repair them. The *myisamchk* command-line tool can also be used to check and repair tables when the server is offline.

Concurrency improvements

If a MyISAM table has no deleted rows, you can insert rows into the table while select queries are running against it.

Index features

`BLOB` and `TEXT` columns in a MyISAM table can be indexed. MyISAM tables have a limit of 500 bytes on each key, however, so the index uses only the first few hundred bytes of a `BLOB` or `TEXT` field. MyISAM tables also allow you to index columns that may contain NULL values. You can find more information on MyISAM indexes in Chapter 4.

Delayed key writes

MyISAM tables marked with the `DELAY_KEY_WRITE` create option don't have index changes written to disk as they are made. Instead, the changes are made to the in-memory key buffer only and flushed to disk when the associated blocks are pruned from the key buffer or when the table is closed. This can yield quite a performance boost on heavily used tables that change frequently.

2.5.2 Compressed MyISAM Tables

For circumstances in which the data never changes, such as CD-ROM- or DVD-ROM-based applications, or in some embedded environments, MyISAM tables can be compressed (or packed) using the *myisampack* utility. Compressed tables can't be modified, but they generally take far less space and are faster as a result. Having smaller tables means fewer disk seeks are required to find records.

On relatively modern hardware, the overhead involved in decompressing the data is insignificant for most applications. The individual rows are compressed, so MySQL doesn't need to unpack an entire table (or even a page) just to fetch a single row.

2.5.3 RAID MyISAM Tables

While they're not really a separate table type, MyISAM RAID tables do serve a particular niche. To use them, you need to compile your own copy of MySQL from source or use the MySQL-Max package. RAID tables are just like MyISAM tables except that the data file is split into several data files. Despite the reference to RAID in the name, these data files don't have to be stored on separate disks, although it is easy to do so. Writes to the table are striped across the data files, much like RAID-0 would do across physical disks. This can be helpful in two circumstances. If you have an operating system that limits file sizes to 2 or 4 GB but you need larger tables, using RAID will get you past the limit. If you're have an I/O bound table that is read from and written to very frequently, you might achieve better performance by storing each of the RAID files on a separate physical disk.

To create a RAID table, you must supply some additional options at table-creation time:

```
CREATE TABLE mytable (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

) RAID_TYPE = STRIPED RAID_CHUNKS = 4
RAID_CHUNKSIZE = 16;
```

The `RAID_TYPE` option, while required, must be `STRIPED` or `RAID0`, which are synonymous. No other RAID algorithms are available. The `RAID_CHUNKS` parameter tells MySQL how many data files to break the table into. The `RAID_CHUNKSIZE` option specifies how many kilobytes of data MySQL will write in each file before moving to the next.

In the previous example, MySQL would create four subdirectories named *00*, *01*, *02*, and *03* in which it would store a file named *mytable.MYD*. When writing data to the table, it would write 16 KB of data to one file and then move to the next one. Once created, RAID tables are transparent. You can use them just as you would normal MyISAM tables.

With the availability of inexpensive RAID controllers and the software RAID features of some operating systems, there isn't much need for using RAID tables in MySQL. Also, it's important to realize that RAID tables split only the data file, not the indexes. If you're trying to overcome file size limits, keep an eye on the size of your index files.

2.5.4 MyISAM Merge Tables

Merge tables are the final variation of MyISAM tables that MySQL provides. Where a RAID table is a single table split into smaller pieces, a Merge table is the combination of several similar tables into one virtual table.

This is particularly useful when MySQL is used in logging applications. Imagine you store web server logs in MySQL. For ease of management, you might create a table for each month. However, when it comes time to generate annual statistics, it would be easier if all the records were in a single table. Using Merge tables, that's possible. You can create 12 normal MyISAM tables, `log_2004_01`, `log_2004_02`, ... `log_2004_12`, and then a Merge table named `log_2004`.

Queries for a particular month can be run against the specific table that holds the data. But queries that may need to cross month boundaries can be run against the Merge table `log_2004` as if it was a table that contained all the data in the underlying twelve tables.

The requirements for a Merge table are that the underlying tables must:

- Have exactly the same definition
- Be MyISAM tables
- Exist in the same database (this limitation is removed in MySQL Versions 4.1.1 and higher, however)

Interestingly, it's possible for some underlying tables to be compressed MyISAM tables. That means you can compress tables as they get old (since they're no longer being written to anyway), but still use them as part of a Merge table. Just make sure to remove the table from the Merge table before compressing it, then re-add it after it has been compressed.

Using the example table from earlier, let's create several identical tables and a Merge table that aggregates them:

```
CREATE TABLE mytable0 (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

);



CREATE TABLE mytable1 (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

);



CREATE TABLE mytable2 (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

);
```

```
CREATE TABLE mytable (

  a    INTEGER  NOT NULL PRIMARY KEY,

  b    CHAR(18) NOT NULL

) TYPE = MERGE UNION = (mytable0, mytable1,
mytable2) INSERT_METHOD = LAST;
```

The only difference between the Merge table and the underlying tables is that it has a few extra options set at creation time. The type, of course, is `MERGE`. The `UNION` option specifies the tables that make up the Merge table. Order is important if you plan to insert into the Merge table rather than the underlying tables. The `INSERT_METHOD` option, which can be `NO`, `FIRST`, or `LAST`, tells MySQL how to handle inserts to the Merge table. If the method is `NO`, inserts aren't allowed. Otherwise, inserts will always go to either the first or last of the underlying tables based on the value of `INSERT_METHOD`.

The order of the tables is also important for unique-key lookups because as soon as the record is found, MySQL stops looking. Thus, the earlier in the list the table is, the better. In most logging applications where you'll be doing searches on the Merge table, it might make sense to put the tables in reverse chronological order. The order is also important for making `ORDER BY` as fast as possible because the required merge-sort will be faster when the rows are nearly in order already. If you don't specify `INSERT_METHOD`, the default is `NO`.

As with other tables, you can use `SHOW TABLE STATUS` to get information about a Merge table:

```
mysql> SHOW TABLE STATUS LIKE 'mytable' \G

*************************** 1. row
***************************
```

```
           Name: mytable
           Type: MRG_MyISAM
     Row_format: Fixed
           Rows: 2
 Avg_row_length: 23
    Data_length: 46
Max_data_length: NULL
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: NULL
    Update_time: NULL
     Check_time: NULL
 Create_options:
        Comment:
1 row in set (0.01 sec)
```

Not all of the data is available. MySQL doesn't keep track of the creation, update, and check times for merge tables. It also doesn't store the create options that you might expect. However, you can retrieve that information using `SHOW CREATE TABLE`:

```
mysql> SHOW CREATE TABLE mytable \G

*************************** 1. row
***************************

       Table: mytable

Create Table: CREATE TABLE `mytable` (

  `a` int(11) NOT NULL default '0',

  `b` char(18) NOT NULL default '',

  PRIMARY KEY  (`a`)

) TYPE=MRG_MyISAM INSERT_METHOD=LAST UNION=
(mytable0,mytable1,mytable2)

1 row in set (0.00 sec)
```

This demonstrates that Merge tables really aren't full-fledged tables. In fact, Merge tables have some important limitations and surprising behavior:

- `REPLACE` queries don't work on them.
- `AUTO_INCREMENT` columns aren't updated on insert. They are updated if you insert directly into one of the underlying tables.
- `DROP TABLE mytable` will drop only the virtual table, not the underlying tables. This may or may not be what you'd expect.

2.5.5 InnoDB Tables

The InnoDB table handler is the newest addition to the MySQL family. Developed by Heikki Tuuri of Innobase Oy in Helsinki, Finland, InnoDB was designed with transaction processing in mind and modeled largely after Oracle.

2.5.5.1 Storage

The InnoDB table handler breaks from MySQL tradition and stores all its data in a series of one or more data files that are collectively known as a *tablespace*. A tablespace is essentially a black box that is completely managed by InnoDB. If a tablespace if composed of several underlying files, you can't choose or influence which of the underlying files will contain the data for any particular database or table.

InnoDB can also use raw disk partitions in building its tablespace, but that's not very common. Using disk partitions makes it more difficult to back up InnoDB's data, and the resulting performance boost is on the order of a few percent on most operating systems.

As of MySQL 4.1, you have the option of slightly more MyISAM-like storage for InnoDB. You can enable multiple tablespace support by adding `innodb_file_per_table` to *my.cnf*; this makes InnoDB create one tablespace file per newly created InnoDB table. The filename will be of the form *tablename.ibd*. In all other respects, they're simply dynamically sized InnoDB tablespace files. Each one just happens to contain data for only one specific table.

2.5.5.2 Locking and concurrency

InnoDB uses MVCC to achieve very high concurrency. InnoDB defaults to the repeatable read isolation level, and as of MySQL Version 4.0.5, it implements all four levels: read uncommitted, read committed, repeatable read, and serializable.

In an InnoDB transaction, You may explicitly obtain either exclusive or shared locks on rows using the MySQL statements: `SELECT ... FOR UPDATE` and `SELECT ... LOCK IN SHARE MODE`.

2.5.5.3 Special features

Besides its excellent concurrency, InnoDB's next most popular feature is *referential integrity* in the form of foreign key constraints. This means that given the following schema:

```
CREATE TABLE master (

  id          INTEGER NOT NULL PRIMARY KEY,

  stuff       TEXT    NOT NULL

) TYPE = InnoDB;


CREATE TABLE detail (

  master_id   INTEGER     NOT NULL,

  detail1     VARCHAR(80) NOT NULL,

  detail2     VARCHAR(20) NOT NULL,

  INDEX       master_idx (master_id),

  FOREIGN KEY (master_id) REFERENCES master(id)

) TYPE = InnoDB;
```

InnoDB doesn't allow you to insert add records to the detail table until there is a corresponding record in the master table. Attempting to do so yields an error:

```
mysql> INSERT INTO detail VALUES (10, 'blah',
'blah');

ERROR 1216: Cannot add a child row: a foreign key
constraint fails
```

InnoDB also provides lightning fast record lookups for queries that use a primary key. Its clustered index system (described in more detail in Chapter 4) explains how it works.

2.5.6 Heap (In-Memory) Tables

MySQL provides in-memory Heap tables for applications in which you need incredibly fast access to data that either never changes or doesn't need to persist after a restart. Using a Heap table means that a query can complete without even waiting for disk I/O. This makes sense for lookup or mapping tables, such as area code to city/state name, or for caching the results of periodically aggregated data.

2.5.6.1 Limitations

While Heap tables are very fast, they often don't work well as replacements for disk-based tables. Until MySQL Version 4.1, Heap tables used only hash-based indexes rather than B-tree indexes (which MyISAM uses). Hash indexes are suited to only a subset of queries. Section 4.3.2 in Chapter 4 covers this in more detail.

2.5.7 Berkeley DB (BDB) Tables

MySQL's first transaction-safe storage engine, BDB is built on top of the Berkeley DB database library, which is now maintained and developed by Sleepycat Software. In fact, the original work to integrate the Berkeley DB technology with MySQL was performed jointly by MySQL AB and Sleepycat Software. Other than transactions, the BDB table handler's other main feature is that it uses page-level locking to achieve higher concurrency than MyISAM tables.

Though BDB tables have been available in MySQL since Version 3.23, they haven't proven very popular among users. Many users looking for transactions in MySQL were also looking for row-level locking or MVCC. Further dampening interest in BDB, by the time the BDB code had stabilized, word of InnoDB began to circulate. This prompted many users to hold out for the real thing and use MyISAM tables a bit longer.

If nothing else, the inclusion of BDB tables in MySQL served as a stepping stone in many ways. It prompted the MySQL developers to put the transaction-handling infrastructure into MySQL, while at the same time proving to the skeptics that MySQL wasn't a toy.

Chapter 3. Benchmarking

We decided to cover benchmarking very early in this book because it's a critically important skill. Much of this book focuses on information and techniques you need to keep MySQL fast or make it run even faster. You need a good performance testing framework to judge the difference between one configuration and another, one query and another, or even one server and another. You also need a lot of patience and a willingness to experiment. This chapter can't give you all the answers, but we try to provide some tools that will help you find them.

If you care about database performance in your applications (and if you're reading this book, you probably do),

benchmarking needs to become part of your development testing process. When you're testing an upgrade to MySQL or some MySQL configuration changes, run the benchmark tests you developed while building the application. Look at the results. Make sure they don't surprise you.

This chapter isn't long, but it contains essential material that we'll refer back to and apply in

future chapters. If you're planning to skip around in the book, be sure to read this chapter first.

We begin with a look at the importance of benchmarking in database applications, then continue with a look at benchmarking strategiesthings you need to think about in the planning process. Finally we get our hands dirty with a look at benchmarking tools.

We'll build on the strategies and tools presented in this chapter in those that follow. When considering performance questions, we'll consider the factors involved and present a benchmark test that can assist in the decision-making process. Take some time now to experiment with the tools and examples presented here. The skills you build now will benefit you in later chapters and in your own projects.

3.1 The Importance of Benchmarking

Benchmarking is fundamentally a "what if" game. By setting up a simple test, you can quickly answer questions such as the following:

- What if I increase the number of rows by a factor of 10? Will my queries still be fast?
- Will a RAM upgrade really help? If so, how much?
- Is the new server really twice as fast as the old one?
- What if I disable the query cache?
- Which is faster, using a subquery or two shorter queries?
- What happens when this query is run multiple times or is run with other queries?

Benchmarking is often about comparisons. When deciding to make an important change, you'll want first to test the alternative(s) and then decide what to do based on the results of the test.

Our goal is to make benchmarking MySQL easy. Anytime you catch yourself wondering if A is faster than B, or whether A or B uses more memory, just pull out your favorite benchmarking tool and find out. Sometimes you'll be surprised by the results. To achieve the goal of easy MySQL benchmarking, we've tried to document how to use the available tools.

Beyond answering what-if questions, benchmarking is especially important in database-driven applications because it can highlight problems that are otherwise difficult to pinpoint. When an application slows down, the database may not be the first suspect. After spending a lot of time testing the application code, you'll eventually need to isolate the database to see whether it is a significant bottleneck. Having a prebuilt benchmark makes that task trivial.

3.2 Benchmarking Strategies

We'll look at the mechanics of benchmarking shortly. First it's important to convey some of strategies and ideas that make up the philosophy behind benchmarking.

To start with, it's important to make a distinction between

performance testing and stress testing. Both processes use the tools we'll look at in this chapter, but the goals are very different. When doing performance testing, you're usually comparing two alternativesmost

often in isolation from everything else. For instance, would it be faster to use a `UNION` or run two separate queries?

Stress testing, on the other hand, is about finding limits: what's the maximum number of requests I can handle with this configuration?

If the two types of benchmarking still sound similar, look at it this way: in performance testing, the numbers you get aren't as important as the difference between them.

You may see that alternative #1 usually runs in 0.01 seconds (or 100

queries/second), while alternative #2 runs in 0.20 seconds (or 5

queries/second). That tells you the first alternative is 20 times faster than the second one. However, knowing that you can handle 100

queries per second doesn't tell you how your

application *as a whole* will perform unless, of course, your application always runs the same query. In contrast, stress testing can help in situations such as: "We expect the promotion we just offered to bring in 30% more hits than we have now. What will the effects on our server be?"

To make benchmarking as realistic and hassle-free as possible, here are several suggestions to consider:

Change one thing at a time

In science this is called

isolating the variable. No matter how well you think you understand the effects your changes will have, don't make more than one change between test runs.

Otherwise you'll never know which one was

responsible for the doubling (or halving) of performance. You might be surprised to find that an adjustment you made once before to improve performance actually makes it worse in your current tests.

Test iteratively

Try not to make dramatic changes. When adjusting MySQL's buffers and caches,

you'll often be trying to find the smallest value that comfortably handles your load. Rather than increasing a value by 500%, start with a 50% or 100% increase and continue using that percentage increase on subsequent tests. You'll probably find the optimal value faster this way. Similarly, if you're working from larger values to smaller, the time-tested "divide and conquer"

technique is your best bet. Cut the current value in half, retest, and repeat the process until you've zeroed in close to the correct value.

Always repeat tests

No matter how carefully you control the environment, something can creep in and really mess up your numbers. Maybe you forgot to disable *cron*, or you have some disk-intensive script running in the background. Because the disk is already being hit, you may not notice the new process, but it sure can slow down MySQL.

By running each test several times (we recommend no fewer than four) and throwing out the first result, you minimize the chance of an outside influence getting in the way. It will be pretty clear that something was wrong with the first result when the second and third set of tests run twice as fast as the first. Also, consider restarting MySQL and even rebooting your server between test runs to factor out caching artifacts.

Use real data

It sounds like common sense, doesn't it? If

you're not testing with real data,

it's difficult to draw conclusions based on the numbers you get. As you'll see in Chapter 4, MySQL will often behave differently when presented with different sets of data. The query optimizer makes decisions based on what it knows about the data you've stored. If you're testing

with fake data, there's a chance that the

optimizer's decisions aren't the

same as they'll be when you switch to using your real data.

In a similar vein, try to use a realistic amount of data. If you plan to have 45 million rows in a table but test with only 45 thousand, you'll find that performance drops off quite a bit after the table is filled upand it has nothing to do with limits in MySQL. The simple fact is that your server probably has enough memory to keep 45 thousand rows cached, but 45 million rows aren't nearly as likely to be entirely cached.

Don't use too many clients

Try not to go crazy with benchmarking.

It's fun to see how hard you can push your server, but unless you're doing stress testing,

there's little need to run more than 40 or 50

concurrent clients in day-to-day benchmarking.[1] What you'll likely find is that performance (measured in queries/second) reaches a plateau when you try to increase the simulated clients beyond a certain number.

[1] There will always be exceptions. If your site must routinely handle 450

connections, you'll obviously need to test with numbers close to 450.

When you attempt to use too many clients, your server will refuse to accept any more connections than specified by the `max_clients` setting. Be careful not to increase this value too much; if you do, the server may start to swap wildly and grind to a halt simply because it doesn't have the resources (typically memory) to handle huge numbers of clients.

We'll come back to this in Chapter 6 when we look at service performance. But the test doesn't help you evaluate your server realistically.

You can find the optimal number of clients by using a simple iterative testing method. Start with a small number such as 20, and run the benchmark. Double the number, and run it again. Continue doubling it until the performance does not increase, meaning that the total queries per second stays the same or decreases. Another option is to use data from your logs to find out roughly how many concurrent users you handle during peak times.

Separate the clients from the server

Even if your real application runs on the same host as MySQL, it's best to run the benchmarking client on a separate machine. In this way, you need not worry about the resources required by the client interfering with MySQL's performance during the test.

```
$ <b>cd sql-bench</b>

sql-bench$ <b>./run-all-tests --server=mysql --
user=root --log --fast</b> Test finished. You can find
the result in: output/RUN-mysql_fast-
Linux_2.4.18_686_smp_i686

sql-bench/output$ <b>tail -5 select-mysql_fast-
Linux_2.4.18_686_smp_i686</b> Time for
count_distinct_group_on_key (1000:6000): 34
wallclock secs ( 0.20 usr 0.08 sys + 0.00 cusr 0.00
csys = 0.28 CPU) Time for
count_distinct_group_on_key_parts (1000:100000):
34 wallclock secs ( 0.57 usr 0.27 sys + 0.00 cusr 0.00
csys = 0.84 CPU) Time for count_distinct_group
(1000:100000): 34 wallclock secs ( 0.59 usr 0.20 sys
+ 0.00 cusr 0.00 csys = 0.79 CPU) Time for
count_distinct_big (100:1000000): 8 wallclock secs (
4.22 usr 2.20 sys + 0.00 cusr 0.00 csys = 6.42 CPU)
Total time:

  868 wallclock secs (33.24 usr 9.55 sys + 0.00 cusr
0.00 csys = 42.79 CPU)
```

```
sql-bench$ <b>./test-insert</b> Testing server
'MySQL 4.0.13 log' at 2003-05-18 11:02:39

Testing the speed of inserting data into 1 table and do
some selects on it.

The tests are done with a table that has 100000 rows.
```

Generating random keys

Creating tables

Inserting 100000 rows in order

Inserting 100000 rows in reverse order Inserting 100000 rows in random order Time for insert (300000):

  42 wallclock secs (7.91 usr 5.03 sys + 0.00 cusr 0.00 csys = 12.94 CPU) Testing insert of duplicates

Time for insert_duplicates (100000): 16 wallclock secs (2.28 usr 1.89 sys + 0.00 cusr 0.00 csys = 4.17 CPU)

/tmp$ <b>tar -zxf super-smack-1.1.tar.gz</b> /tmp$ <b>cd super-smack-1.1</b> /tmp/super-smack-1.1$ <b>./configure --with-mysql</b> ... lots of configure output ...

/tmp/super-smack-1.1$ <b>make</b> ... lots of compilation output ...

/tmp/super-smack-1.1$ <b>sudo make install</b>

/usr/share/smacks$ <b>ls -l</b> total 8

```
-rw-r--r-- 1 jzawodn jzawodn 3211 Feb 2 2004 select-
key.smack -rw-r--r-- 1 jzawodn jzawodn 3547 Feb 2
2004 update-select.smack

/usr/share/smacks$ <b>super-smack update-
select.smack 30 10000</b> Error running query
select count(*) from http_auth:Table 'test.http_auth'
doesn't exist Creating table 'http_auth'

Loading data from file '/var/smack-data/words.dat'
into table 'http_auth'

Table http_auth is now ready for the test Query
Barrel Report for client smacker connect: max=49ms
min=0ms avg= 14ms from 30 clients Query_type
num_queries max_time min_time q_per_s
select_index 300000 10 0 2726.41

update_index 300000 5 0 2726.41

/usr/share/smacks$ <b>super-smack update-
select.smack 5 10000</b> Query Barrel Report for
client smacker connect: max=2ms min=1ms avg=
1ms from 5 clients Query_type num_queries
max_time min_time q_per_s select_index 50000 1 0
3306.66
```

```
update_index 50000 1 0 3306.66
```

```
SELECT id INTO OUTFILE "/tmp/product.dat"
FROM product
```

```
SELECT id, type INTO OUTFILE
"/tmp/product.dat"

FIELDS TERMINATED BY ','

OPTIONALLY ENCLOSED BY '"'

LINES TERMINATED BY "\n"
```

```
FROM product

client "admin"

{

  user "root"; host "localhost"; db "test";

  pass "";

  socket "/var/lib/mysql/mysql.sock"; }

port "3307";

table "http_auth"

{

  client "admin"; create "create table http_auth
(username char(25) not null primary key, pass
char(25), uid integer not null, gid integer not null )";

  min_rows "90000"; data_file "words.dat";
gen_data_file "gen-data -n 90000 -f %12-
12s%n,%25-25s,%n,%d"; }
```

```
$ <b>gen-data -n 5 -f %12-12s%n,%25-
25s,%n,%d</b>
pajgyycklwiv1,qbnvqtcewpwvxpobgpcgwppkw,1,76
3719779

epqjynjbrpew2,mhvcdpmifuefqdmjblodvlset,2,34485
8293

fbntssvvmwck3,cfydxkranoqfiuvyhqvtprmpx,3,2125
632375

fcwtayvakrxr4,ldaprgacrwsbujrnlxxsxqwse,4,151305
0921

jnaixvfvktpf5,htihaukugfiurnnmxnysypsnr,5,1872952
907

dictionary "word"

{

  type "rand"; source_type "file"; source "words.dat";
delim ",";

  file_size_equiv "45000"; }

query "select_by_username"
```

```
{

  query "select * from http_auth where username =
'$word'"; type "select_index"; has_result_set "y";
parsed "y";

}

query "update_by_username"

{

  query "update http_auth set pass='$word' where
username = '$word'"; type "update_index";
has_result_set "n"; parsed "y";

}

client "smacker"

{

  user "test"; pass "";

  host "localhost"; db "test";
```

socket "/var/lib/mysql/mysql.sock"; query_barrel "1 select_by_username 1 update_by_username"; }

```
main

{

  smacker.init( ); smacker.set_num_rounds($2);
smacker.create_threads($1); smacker.connect( );
smacker.unload_query_barrel( );
smacker.collect_threads( ); smacker.disconnect( ); }

#!/usr/bin/perl -w

use strict;

use MyBench;

use Getopt::Std;

use Time::HiRes qw(gettimeofday tv_interval); use
DBI;

my %opt;

Getopt::Std::getopt('n:r:h:', \%opt); my $num_kids =
$opt{n} || 10;

my $num_runs = $opt{r} || 100;
my $db = "test";

my $user = "test";

my $pass = "";

my $port = 3306;

my $host = $opt{h} || "192.168.0.1"; my $dsn =
"DBI:mysql:$db:$host;port=$port";

my $callback = sub

{

  my $id = shift; my $dbh = DBI->connect($dsn,
$user, $pass, { RaiseError => 1 }); my $sth = $dbh-
>prepare("SELECT * FROM mytable WHERE ID =
?"); my $cnt = 0; my @times = ( ); ## wait for the
parent to HUP me local $SIG{HUP} = sub { }; sleep
600;

  while ($cnt < $num_runs) {

  my $v = int(rand(100_000)); ## time the query my
$t0 = [gettimeofday]; $sth->execute($v); my $t1 =
tv_interval($t0, [gettimeofday]); push @times, $t1;
$sth->finish( ); $cnt++;

  }

  ## cleanup

  $dbh->disconnect( ); my @r = ($id, scalar(@times),
min(@times), max(@times), avg(@times),
tot(@times)); return @r;

};

my @results =
MyBench::fork_and_work($num_kids, $callback);
MyBench::compute_results('test', @results); exit;

_ _END_ _

$ <b>./bench_select_1 -n 10 -r 100000</b> forking:
++++++++++

sleeping for 2 seconds while kids get ready waiting: -
---------

test: 1000000 7.5e-05 0.65045
0.000561082981999975 561.082981999975
```

```
17822.6756483597

  clients : 10

  queries : 1000000

  fastest : 7.5e-05

  slowest : 0.65045

  average : 0.000561082981999975

  serial : 561.082981999975

  q/sec : 17822.6756483597
```

The first three lines are merely status updates so you can tell that the test is doing something while it runs. The `test:` line produces all the statistics on a single line, suitable for processing in another script or pasting into a spreadsheet. They're followed by human readable output.

There you can see how many clients were used, the total number of queries executed, and the response times (in seconds) of fastest and slowest queries as well as the average. The `serial` value explains approximately how many seconds the queries would have taken if executed serially. Finally, the `q/sec` number tells us how many queries per second (on average) the server handled during the test.

Because the code times only the query and not the work done by the Perl script, you can add arbitrarily complex logic to the main loop. Rather than generate a random number, maybe you need to read a value from a file or from another database table. Perhaps you need to run a few special queries every 785th iteration, to simulate the behavior of your real application. Doing so with MyBench would be easy; using super-smack would be more of a challenge.

Chapter 4. Indexes

Indexes allow MySQL to quickly find and retrieve a set of records from the millions or even billions that a table may contain. If you've been using MySQL for any length of time,

you've probably created indexes in the hopes of

getting lighting-quick answers to your queries. And you've probably been surprised to find that MySQL

didn't always use the index you thought it would.

For many users, indexes are something of a black art. Sometimes they work wonders, and other times they seem just to slow down inserts and get in the way. And then there are the times when they work fine for a while, then begin to slowly degrade.

In this chapter, we'll begin by looking at some of the concepts behind indexing and the various types of indexes MySQL

provides. From there, we'll cover some of the

specifics in MySQL's implementation of indexes. The chapter concludes with recommendations for selecting columns to index and the longer term care and feeding of your indexes.

```
SELECT * FROM phone_book WHERE last_name
= 'Zawodny'

ALTER TABLE phone_book ADD INDEX
(last_name)

ALTER TABLE phone_book ADD INDEX
(last_name(4))

SELECT * FROM phone_book WHERE last_name
= 'Smith'

ALTER TABLE phone_book ADD INDEX
(last_name, first_name)

ALTER TABLE phone_book ADD INDEX
(last_name(4), first_name(4))

SELECT * FROM phone_book



WHERE last_name = 'Woodward'
```

```
   AND first_name = 'Josh'
```

```
SELECT * FROM phone_book WHERE last_name
= 'Zawodny'
```

```
ORDER BY first_name DESC
```

```
SELECT * FROM phone_book WHERE last_name
= 'Zawodny'
```

```
ORDER BY first_name ASC
```

```
ALTER TABLE phone_book ADD UNIQUE
(phone_number)
```

```
SELECT * FROM phone_book WHERE
phone_number = '555-7271'
```

```
SELECT * FROM phone_book WHERE
phone_number = '555-7271'
```

MySQL scans the `phone_number` index to find the entry for `555-7271`, which contains the primary key entry `Zawodny` because `phone_book`'s primary index is the last name. MySQL then skips to the relevant entry in the database itself.

In other words, lookups based on your primary key happen exceedingly fast, and lookups based on secondary indexes happen at essentially the same speed as MyISAM index lookups would.

But under the right (or rather, the wrong) circumstances, the clustered index can actually degrade performance. When you use one together with a secondary index, you have to consider the combined impact on storage. Secondary indexes point to the primary key rather than the row. Therefore, if you index on a very large value and have several secondary indexes, you will end up with many duplicate copies of that primary index, first as the clustered index stored alongside the records themselves, but then again for as many times as you have secondary indexes pointing to those clustered indexes. With a small value as the primary key, this may not be so bad, but if you are using something potentially long, such as a URL, this repeated storage of the primary key on disk may cause storage issues.

Another less common but equally problematic condition happens when the data is altered such that the primary key is changed on a record. This is the most costly function of clustered indexes. A number of things can happen to make this operation a more severe performance hit:

- Alter the record in question according to the query that was issued.
- Determine the new primary key for that record, based on the altered data record.
- Relocate the stored records so that the record in question is moved to the proper location in the tablespace.
- Update any secondary indexes that point to that primary key.

As you might imagine, if you're altering the primary key for a number of records, that `UPDATE` command might take quite some time to do its job, especially on larger tables. Choose your primary keys wisely. Use values that are unlikely to change, such as a Social Security account number instead of a last name, serial number instead of a product name, and so on.

4.1.1.6 Unique indexes versus primary keys

If you're coming from other relational databases, you might wonder what the difference between a primary key and a unique index is in MySQL. As usual, it depends. In MyISAM tables, there's almost no difference. The only thing special about a primary key is that it can't contain NULL values. The primary key is simply a `NOT NULL UNIQUE INDEX` named `PRIMARY`. MyISAM tables don't require that you declare a primary key.

InnoDB and BDB tables require primary keys for every table. There's no requirement that you specify one, however. If you don't, the storage engine automatically adds a hidden primary key for you. In both cases, the primary keys are simply incrementing numeric values, similar to an `AUTO-INCREMENT` column. If you decide to add your own primary key at a later time, simply use `ALTER TABLE` to add one. Both storage engines will discard their internally generated keys in favor of yours. Heap tables don't require a primary key but will create one for you. In fact, you can create Heap tables with no indexes at all.

4.1.1.7 Indexing NULLs

It is often difficult to remember that SQL uses tristate logic when performing logical operations. Unless a column is declared `NOT NULL`, there are three possible outcomes in a logical comparison. The comparison may be true because the values are equivalent; it may be false because the values aren't equivalent; or it may not match because one of the values is NULL. Whenever one of the values is NULL, the outcome is also NULL.

Programmers often think of NULL as undefined or unknown. It's a way of telling the database server "an unknown value goes here." So how do NULL values affect indexes?

NULL values may be used in normal (nonunique) indexes. This is true of all database servers. However, unlike many database servers, MySQL allows you to use NULL values in unique indexes.[6] You can store as many NULL values as you'd like in such an index. This may seem a bit counterintuitive, but that's the nature of NULL. Because NULL represents an undefined value, MySQL needs to assert that all NULL values are the same if it allowed only a single value in a unique index.

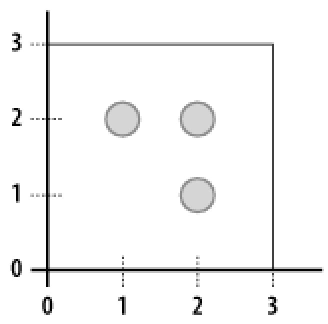
> [6] MySQL Version 3.23 and older don't allow this, Versions 4.0 and newer do.

To make things just a bit more interesting, a NULL value may appear only once as a primary key. Why? The SQL standard dictates this behavior. It is one of the few ways in which primary keys are different from unique indexes in MySQL. And, in case you're wondering, allowing NULL values in the index really doesn't impact performance.

```
SELECT * FROM phone_book WHERE last_name
BETWEEN 'Marten' and 'Mason'

SELECT COUNT(*) FROM phone_book WHERE
last_name > 'Zawodny'

mysql> <b>SELECT MD5('Smith');</b> +------------
----------------------+

| MD5('Smith') |

+----------------------------------+

| e95f770ac4fb91ac2e4873e4b2dfc0e6 |

+----------------------------------+

1 row in set (0.46 sec)

mysql> <b>SELECT MD5('Smitty');</b> +-----------
-----------------------+

| MD5('Smitty') |

+----------------------------------+
| 6d6f09a116b2eded33b9c871e6797a47 |

+----------------------------------+

1 row in set (0.00 sec)

mysql> <b>create table map_test</b> -> (

  -> <b>name varchar(100) not null primary key,</b>
-> <b>loc geometry,</b> -> <b>spatial index(loc)
</b> -> <b>);</b> Query OK, 0 rows affected (0.00
sec)

mysql> <b>insert into map_test values ('One Two',
point(1,2));</b> Query OK, 1 row affected (0.00 sec)
mysql> <b>insert into map_test values ('Two Two',
point(2,2));</b> Query OK, 1 row affected (0.00 sec)
mysql> <b>insert into map_test values ('Two One',
point(2,1));</b> Query OK, 1 row affected (0.00 sec)

mysql> select name, AsText(loc) from map_test; +---
------+------------+

| name | AsText(loc) |

+---------+------------+
| One Two | POINT(1 2) |

| Two Two | POINT(2 2) |

| Two One | POINT(2 1) |

+---------+-------------+

3 rows in set (0.00 sec)

mysql> <b>SELECT name FROM map_test
WHERE</b> ->
<b>Contains(GeomFromText('POLYGON((0 0, 0 3,
3 3, 3 0, 0 0))'), loc);</b> +---------+

| name |

+---------+

| One Two |

| Two Two |

| Two One |

+---------+
3 rows in set (0.00 sec)
```

Figure 4-1 shows the points and polygon on a graph.

Figure 4-1. 2-D points and a polygon that contains them



MySQL indexes the various shapes that can be represented (points, lines, polygons) using the shape's minimum bounding rectangle (MBR). To do so, it computes the smallest rectangle you can draw that completely contains the shape. MySQL stores the coordinates of that rectangle and uses them when trying to find shapes in a given area.

```
PACKED_KEY = 1

mysql> <b>create table heap_test (</b>

  -> name varchar(50) not null,

  -> index using btree (name)

  -> ) type = HEAP;

Query OK, 0 rows affected (0.00 sec)

mysql> <b>show keys from heap_test \G</b>

*************************** 1. row
***************************


  Table: heap_test

  Non_unique: 1



  Key_name: name
```

```
Seq_in_index: 1

Column_name: name

  Collation: A
Cardinality: NULL

  Sub_part: NULL

  Packed: NULL

  Null:

  Index_type: BTREE
```

Comment:

```
1 row in set (0.00 sec)

select * from articles where body = "%database%"

select * from articles (body) match against
('database')

select * from pages where page_text like "%buffy%"

select phone_number from phone_book where
last_name like "%son%"

select last_name from phone_book where last_name
rlike "(son|ith)$"

select last_name from phone_book where
rev_last_name like "thi%"
```

```
union

select last_name from phone_book where
rev_last_name like "nos%"

select last_name from phone_book where
rev_last_name rlike "^(thi|nos)"
```

You would be disappointed by its performance. The MySQL optimizer simply never tries to optimize regex-based queries.

4.3.6.3 Poor statistics or corruption

If MySQL's internal index statistics become corrupted or otherwise incorrect (possibly as the result of a crash or accidental server shutdown), MySQL may begin to exhibit very strange behavior. If the statistics are simply wrong, you may find that it no longer uses an index for your query. Or it may use an index only some of the time.

What's likely happened is that MySQL believes that the number of rows that match your query is so high that it would actually be more efficient to perform a full table scan. Because table scans are primarily sequential reads, they're faster than reading a large percentage of the records using an index, which requires far more disk seeks.

If this happens (or you suspect it has), try the index repair and analysis commands explained in the "Index Maintenance" section later in this chapter.

4.3.6.4 Too many matching rows

Similarly, if a table actually does have too many rows that really do match your query, performance can be quite slow. How many rows are too many for MySQL? It depends. But a good rule of thumb is that when MySQL believes more than about 30% of the rows are likely matches, it will resort to a table scan rather than using the index. There are a few exceptions to this rule. You'll find a more detailed discussion of this problem in Chapter 5.

```
mysql> <b>SHOW INDEXES FROM
access_jeremy_zawodny_com \G</b>
*************************** 1. row
***************************

  Table: access_jeremy_zawodny_com Non_unique:
1

  Key_name: time_stamp Seq_in_index: 1

Column_name: time_stamp

  Collation: A Cardinality: 9434851

  Sub_part: NULL

  Packed: NULL

  Null: YES

  Index_type: BTREE

  Comment:

1 rows in set (0.00 sec)
```

```
$ <b>cd </b><span
class="docEmphBoldItalic">database-name</span>
$ <b>myisamchk </b><span
class="docEmphBoldItalic">table-name</span>
```

Just be sure that MySQL isn't running when you try this, or you run the risk of corrupting your indexes.

BDB and InnoDB tables are less likely to need this sort of tuning. That's a good thing, because the only ways to reindex them are a bit more time consuming. You can manually drop and re-create all the indexes, or you have to dump and reload the tables. However, using `ANALYZE TABLE` on an InnoDB table causes InnoDB to re-sample the data in an attempt to collect better statistics.

Chapter 5. Query Performance

This chapter deals with an issue faced by every MySQL user sooner or later: speeding up slow queries.

MySQL is a very fast database server, but its innate speed can carry your applications only so far. Eventually you need to roll up your sleeves, get your hands dirty, and figure out why your queries are slowand ultimately figure out what needs to be done to get a response quickly.

We're frequently asked how we

"figure this stuff out."

It's really quite simple. Once you start to

understand how MySQL does what it does, you'll begin to have an intuitive feeling for it, and query optimization will start to seem really easy. It's not always that easy, but with the proper background, you should end up able to figure out most optimization problems.

This chapter aims to provide a framework for understanding how MySQL

works to resolve queries. With this foundation, you can continue through this chapter to the next, where the knowledge is applied to application design and server performance tuning.

We'll begin with an overview of how MySQL handles query processing. After that, we'll look at the optimizer's built-in features. Then

we'll discuss identifying slow queries and finish up with a look at some of the hints you can provide to MySQL's query optimizer.

```
query_cache_type = 1
```

```
SELECT * FROM table1
```

```
select * FROM table1
```

```
/* <b>GetLatestStuff</b> */ SELECT * FROM
sometable WHERE ...
```

```
SELECT SQL_NO_CACHE * FROM mytable
```

```
SELECT SQL_CACHE * FROM mytable
```

```
mysql> <b>describe Headline;</b> +-----------+-----
-------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+-----------+------------------+------+-----+---------+----
------------+

| Id | int(10) unsigned | | PRI | NULL | auto_increment
|

| Headline | varchar(255) | | | | |
| Url | varchar(255) | | UNI | | |

| Time | int(10) unsigned | | MUL | 0 | |

| ExpireTime | int(10) unsigned | | | 0 | |

| Date | varchar(6) | | | | |

| Summary | text | YES | | NULL | |

| ModTime | timestamp | YES | | NULL | |

+------------+-----------------+------+-----+---------+----
------------+

8 rows in set (0.00 sec)

mysql> <b>SELECT Headline, Url FROM Headline
WHERE Id = 13950120 \G</b>
*************************** 1. row
***************************

Headline: Midwest Cash Grain PM - Soy off, USDA
data awaited Url:
http://biz.yahoo.com/rm/030328/markets_grain_cash
_2.html 1 row in set (0.00 sec)
```

```
mysql> <b>EXPLAIN SELECT Headline, Url
FROM Headline WHERE id = 13950120 \G</b>
*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: const

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4

  ref: const

  rows: 1

  Extra:

1 row in set (0.00 sec)

mysql> <b>SELECT Url FROM Headline WHERE
id BETWEEN 13950120 AND 13950125;</b> +-----
```

```
-----------------------------------------------------+

| Url |

+----------------------------------------------------------+

|
http://biz.yahoo.com/rm/030328/markets_grain_cash
_2.html |

| http://biz.yahoo.com/prnews/030328/cgf038_1.html
|

| http://biz.yahoo.com/bw/030328/285487_1.html |

|
http://biz.yahoo.com/rc/030328/turkey_hijack_5.html
|

|
http://biz.yahoo.com/rm/030328/food_aid_iraq_1.ht
ml |

+----------------------------------------------------------+

5 rows in set (0.00 sec)
```

```
mysql> <b>EXPLAIN SELECT Url FROM
Headline WHERE id BETWEEN 13950120 AND
13950125 \G</b> ****************************
1. row ***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4

  ref: NULL

  rows: 3

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b>SELECT COUNT(*) FROM Headline
WHERE ExpireTime >= 1112201600;</b> +----------
+
| COUNT(*) |

+----------+

| 3971 |

+----------+

1 row in set (1.04 sec)

mysql> <b>EXPLAIN SELECT COUNT(*) FROM
Headline WHERE ExpireTime >= 1112201600
\G</b> *************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: ALL

possible_keys: NULL

  key: NULL

  key_len: NULL
```

```
   ref: NULL

   rows: 302116

   Extra: Using where 1 row in set (0.00 sec)
```

```
mysql> <b>ALTER TABLE Headline ADD INDEX
(ExpireTime);</b> Query OK, 302116 rows affected
(40.02 sec) Records: 302116 Duplicates: 0 Warnings:
0

mysql> <b>SELECT COUNT(*) FROM Headline
WHERE ExpireTime >= 1112201600;</b> +----------
+

| COUNT(*) |

+----------+

| 3971 |

+----------+

1 row in set (0.01 sec)

mysql> <b>EXPLAIN SELECT COUNT(*) FROM
Headline WHERE ExpireTime >= 1112201600
```

```
\G</b> *************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: ExpireTime

  key: ExpireTime key_len: 4

  ref: NULL

  rows: 12009

  Extra: Using where; Using index 1 row in set (0.00
sec)

mysql> <b>SELECT Url FROM Headline WHERE
Id IN(1531513, 10231599, 13962322);</b> +---------
------------------------------------+

| Url |

+--------------------------------------------+
| http://biz.yahoo.com/bond/010117/bf.html |

| http://biz.yahoo.com/e/021101/yhoo10-q.html |

| http://biz.yahoo.com/bw/030331/315850_1.html |

+-------------------------------------------+

3 rows in set (0.00 sec)

mysql> <b>EXPLAIN SELECT Url FROM
Headline WHERE Id IN(1531513, 10231599,
13962322) \G</b> ***************************
1. row ***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4
  ref: NULL

  rows: 3

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b> SELECT Url FROM Headline WHERE
Id = 1531513 OR Id = 10231599 OR Id = 13962322;
</b> +----------------------------------------------+

| Url |

+----------------------------------------------+

| http://biz.yahoo.com/bond/010117/bf.html |

| http://biz.yahoo.com/e/021101/yhoo10-q.html |

| http://biz.yahoo.com/bw/030331/315850_1.html |

+----------------------------------------------+

3 rows in set (0.03 sec)

mysql> <b> EXPLAIN SELECT Url FROM
Headline WHERE Id = 1531513 OR Id = 10231599
OR Id =
13962322 \G</b>

*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4

  ref: NULL

  rows: 3

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b>EXPLAIN SELECT Url FROM
Headline WHERE Id IN (SELECT MAX(Id) FROM
Headline);</b>
```

```
mysql> <b>EXPLAIN SELECT Url FROM
Headline WHERE Id IN (SELECT MAX(id) FROM
Headline) \G</b> ***************************
1. row ***************************

  id: 1

  select_type: PRIMARY

  table: Headline type: ALL

possible_keys: NULL

  key: NULL

  key_len: NULL

  ref: NULL

  rows: 302116

  Extra: Using where
*************************** 2. row
***************************

  id: 2
  select_type: DEPENDENT SUBSELECT

  table: Headline type: index

possible_keys: NULL

  key: PRIMARY

  key_len: 4

  ref: NULL

  rows: 302116

  Extra: Using index 2 rows in set (0.00 sec)

mysql> <b>SELECT Url FROM Headline WHERE
Id = (SELECT MAX(id) FROM Headline);</b> +----
------------------------------------------+

| Url |

+---------------------------------------------+

| http://biz.yahoo.com/bw/030331/315850_1.html |

+---------------------------------------------+
1 row in set (0.00 sec)

mysql> <b>EXPLAIN SELECT Url FROM
Headline WHERE Id = (SELECT MAX(id) FROM
Headline) \G</b> ***************************
1. row ***************************

  id: 1

  select_type: PRIMARY

  table: Headline type: const

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4

  ref: const

  rows: 1

  Extra:

*************************** 2. row
***************************
```

```
  id: 2

  select_type: SUBSELECT

  table: NULL

  type: NULL

possible_keys: NULL

  key: NULL

  key_len: NULL

  ref: NULL

  rows: NULL

  Extra: Select tables optimized away 2 rows in set
(0.00 sec)

mysql> SELECT @max := MAX(Id) FROM
Headline; +-----------------+

| @max := MAX(Id) |

+-----------------+
| 13962322 |

+----------------+

1 row in set (0.00 sec)

mysql> SELECT Url FROM Headline WHERE Id =
@max; +------------------------------------------+

| Url |

+------------------------------------------+

| http://biz.yahoo.com/bw/030331/315850_1.html |

+------------------------------------------+

1 row in set (0.00 sec)

SELECT Url FROM Headline ORDER BY Id DESC
LIMIT 1;

mysql> <b>SELECT COUNT(*) FROM Headline
WHERE ExpireTime >= 1112201600 AND Id <=

5000000;</b>
```

```
+----------+

| COUNT(*) |

+----------+

| 1175 |

+----------+

1 row in set (0.04 sec)

mysql> <b>EXPLAIN SELECT COUNT(*) FROM
Headline</b> -> <b>WHERE ExpireTime >=
1112201600 AND Id <= 5000000 \G</b>
*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: PRIMARY,ExpireTime

  key: ExpireTime key_len: 4
  ref: NULL

  rows: 12009

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b>EXPLAIN SELECT COUNT(*) FROM
Headline WHERE ExpireTime >= 1012201600 AND
Id <=

5000000 \G</b>

*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: range

possible_keys: PRIMARY,ExpireTime

  key: PRIMARY

  key_len: 4
  ref: NULL

  rows: 13174

  Extra: Using where 1 row in set (0.00 sec)
```

```
SELECT customer.name, order.date_placed,
region.name FROM customer, order, region

WHERE order.customer_id = customer.id AND
customer.region_id = region.id

AND customer.name = 'John Doe'
```

The rows MySQL will need to retrieve from the `order` table depend on the `customer` table. So it must read `customer` before `order`. In fact, the same is true of `region`. So in this case, MySQL has to read `customer` records first. From there it will decide to read the remaining tables in whatever order it chooses.

Unfortunately, finding the optimal join order is one of MySQL's weakest skills. Rather than being clever about this problem, the optimizer simply tries to brute-force its way through. It tries every possible combination before choosing one. That can spell disaster in a some cases. We've seen at least one case in which MySQL took 29 seconds to decide how to execute a multitable join and then 1 second to actually execute it. In this particular case, there were over 10 tables involved. Since MySQL is considering all possible combinations, performance begins to degrade quite drastically as you go beyond a handful of tables. The exact number, of course, depends on how powerful CPUs are this year.

5.1.4 Execution

There's not a lot to say about query execution. MySQL simply follows its plan, fetching rows from each table in order and joining based on the relevant columns (hopefully using indexes). Along the way, it may need to create a temporary table (in memory or on disk) to store the results. Once all the rows are available, it sends them to the client.

Along the way, MySQL gathers some information and statistics about each query it executes, including:

- Who issued the query

- How long the process took

- How many rows were returned

That information will appear in the slow query log (discussed later in this chapter) if the query time exceeds the server's threshold, and the log is enabled. If the query is issued interactively, it will also appear after the query results.

```
CREATE TABLE weather

(

   city VARCHAR(100) NOT NULL,
   high_temp TINYINT NOT NULL,
   low_temp TINYINT NOT NULL,
   the_date DATE NOT NULL,
   INDEX (city),
   INDEX (the_date),
)
SELECT AVG(high_temp) FROM weather
```

```
WHERE the_date BETWEEN '1980-01-01' AND
'1980-12-31';

SELECT * FROM weather WHERE city = 'Toledo'
ORDER BY the_date DESC

ALTER TABLE weather DROP INDEX city, ADD
INDEX (city, the_date)

SELECT * FROM mytable WHERE id < 5000 and
id > 30000

mysql> <b>SELECT * FROM mytable WHERE id <
5000 and id > 30000</b>

+-----------------------------------------------------+



| Comment |



+-----------------------------------------------------+
| Impossible WHERE noticed after reading const
tables |

+-----------------------------------------------------+

1 row in set (0.00 sec)
```

you'll know what it was thinking.

Aside from making a simple typo, it's unlikely that you'll run many queries like that. However, if you're building an application on top of MySQL and happen to make a typo or a serious logic error in the code, you can end up running lots of pointless queries before tracking down the problem. It's good to know that MySQL doesn't waste much time dealing with your illogical queries.

5.2.4 Full-Text Instead of LIKE

From Chapter 4, it's clear that full-text indexes are much faster than using a `LIKE` clause in your queries to search for a word or phrase. In the vast majority of cases, you should use a full-text index to tackle these types of problems.

However, there are times when this can be problematic. The query optimizer doesn't look very closely at full-text indexes when deciding which index to use for a table. In fact, if there's a usable full-text index, the optimizer will always prefer it regardless of how many rows it actually eliminates from the result set. Hopefully this will be fixed in a future version of MySQL.

```
# Time: 030303 0:51:27

# User@Host: user[user] @ client.example.com
[192.168.50.12]

# Query_time: 25 Lock_time: 0 Rows_sent: 3949
Rows_examined: 378036

select ArticleHtmlFiles.SourceTag,
ArticleHtmlFiles.AuxId from ArticleHtmlFiles left

join Headlines on ArticleHtmlFiles.SourceTag =
Headlines.SourceTag and

ArticleHtmlFiles.AuxId = Headlines.AuxId where
Headlines.AuxId is NULL;
```

While the log contains a lot of useful information, there's one very important bit of information missing: an idea of why the query was slow. Sure, if the log says 12,000,000 rows were examined and 1,200,000 sent to the client, you know why it was slow. But things are rarely that clear cut. Worse yet, you may find a slow query, paste it into your favorite MySQL client, and find that it executes in a fraction of a second.

You must be careful not to read too much information into the slow query log. When a query appears in the log, it doesn't mean that it's a bad queryor even a slow one. It simply means that the query took a long time then. It doesn't mean that the query will take a long time now or in the future.

There are any number of reasons why a query may be slow at one time but not at others:

- A table may have been locked, causing the query to wait. The `Lock_time` indicates how long the query waited for locks to be released.
- None of the data or indexes may have been cached in memory yet. This is common when

MySQL is first started or hasn't been well tuned. Chapter 4 covers this in more detail.

- A nightly backup process was running, making all disk I/O considerably slower.
- The server may have been handling hundreds of other unrelated queries at that same time, and there simply wasn't enough CPU power to do the job efficiently.

The list could go on. The bottom line is this: the slow query log is nothing more than a partial record of what happened. You can use it to generate a list of possible suspects, but you really need to investigate each of them in more depth. Of course, if you happen to see the same query appearing in the log over and over, there's a very good chance you have a slow query on your hands.

MySQL also comes with mysqldumpslow, a Perl script that can summarize the slow query log and provide a better idea of how often each slower query executes. That way you don't waste time trying to optimize a 30-second slow query that runs once a day, while there are five other 2-second slow queries that run thousands of time per day.

Appendix B contains information on using *mytop* to perform real-time query monitoring, including slow queries.

```
SELECT SQL_CACHE * FROM mytable ...

SELECT /*! SQL_CACHE */ * FROM mytable ...

SELECT * FROM table1 STRAIGHT_JOIN table2
WHERE ...

SELECT * FROM mytable USE INDEX (mod_time,
name) ...

SELECT * FROM mytale IGNORE INDEX
(priority) ...

SELECT * FROM mytable FORCE INDEX
(mod_time) ...
```

In doing so, you're telling MySQL to ignore any decisions it might otherwise have made about the best way to find the data you've asked for. It will disobey that request only if the index you specify can't possibly be used to resolve the query.

5.4.3 Result Sizes

A set of hints also exists to tell MySQL that you'd like the resulting rows to be handled in a particular way. Like most hints, you really shouldn't be using them unless you know they help. Overusing them will likely cause performance problems sooner or later.

When dealing with a large number of rows that may take a bit of time for the client to consume, consider using `SQL_BUFFER_RESULT`. Doing so tells MySQL to store the result in a temporary table, thus freeing up any locks much sooner.

The `SQL_BIG_RESULT` hint tells MySQL that there will be a large number of rows coming back. When MySQL sees this hint, it can make more aggressive decisions about using disk-based temporary tables. It will also be less likely to build an index on the temporary table for the purpose of sorting the results.

5.4.4 Query Cache

As noted at the beginning of this chapter, the query cache stores the results of frequently executed `SELECT` queries in memory for fast retrieval. MySQL provides opt-in and opt-out hints that can be used to control whether or not a query's results are cached.

By using `SQL_CACHE`, you ask MySQL to cache the results of this query. If the `query_cache_type` is set to 1, this hint has no affect because all `SELECT` queries are cached by default. If `query_cache_type` is set to 2, however, the cache is enabled, but queries are cached only on request. Using `SQL_CACHE` covers this case.

On the flip side, `SQL_NO_CACHE` asks MySQL not to cache the results of a query. Because this is an opt-out request, it works for `query_cache_type` 1 or 2.

```
mysql> <b>DESCRIBE PriceHistory;</b> +----------
+---------+------+-----+------------+-------+

| Field | Type | Null | Key | Default | Extra |

+----------+---------+------+-----+------------+-------+

| SymbolID | int(11) | | PRI | 0 | |

| Date | date | | PRI | 0000-00-00 | |

| Open | float | | | 0 | |

| High | float | | | 0 | |

| Low | float | | | 0 | |

| Close | float | | | 0 | |

| Volume | float | | | 0 | |

+----------+---------+------+-----+------------+-------+

8 rows in set (0.01 sec)

mysql> <b>EXPLAIN SELECT
date_format(Date,'%Y%m%d') as Day, Close</b> ->
```

```
<b>FROM Symbols, PriceHistory</b> ->
<b>WHERE Symbols.ID=PriceHistory.SymbolID
AND Symbols.Symbol = 'ibm'</b> -> <b>ORDER
BY Date DESC \G</b>
*************************** 1. row
***************************

  table: Symbols type: const

possible_keys: PRIMARY,Symbols_SymbolIDX

  key: Symbols_SymbolIDX

  key_len: 20

  ref: const

  rows: 1

  Extra: Using filesort
*************************** 2. row
***************************

  table: PriceHistory type: ref

possible_keys: PriceHistory_IDX
```

```
  key: PriceHistory_IDX

  key_len: 4

  ref: const

  rows: 471

  Extra: Using where 2 rows in set (0.01 sec)

mysql> <b>SELECT @sid := Id FROM Symbols
WHERE Symbol = 'ibm';</b> +------------+

| @sid := Id |

+------------+

| 459378 |

+------------+

1 row in set (0.02 sec)

mysql> <b>EXPLAIN SELECT
date_format(Date,'%Y%m%d') as Day, Close</b> ->
<b>FROM PriceHistory WHERE SymbolID = @sid
ORDER BY Date DESC \G</b>
```

```
*************************** 1. row
***************************

  table: PriceHistory type: ref

possible_keys: PriceHistory_IDX

  key: PriceHistory_IDX

  key_len: 4

  ref: const

  rows: 7234

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b>EXPLAIN SELECT COUNT(*) FROM
Headline</b> -> <b>WHERE ExpireTime >=
1112201600 AND Id <= 5000000 \G</b>
*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE
  table: Headline type: range

possible_keys: PRIMARY,ExpireTime

  key: ExpireTime key_len: 4

  ref: NULL

  rows: 12009

  Extra: Using where 1 row in set (0.00 sec)

mysql> <b>EXPLAIN SELECT * FROM
Headline</b> -> <b>WHERE ExpireTime >=
1012201600 OR Id <= 5000000</b> -> <b>ORDER
BY ExpireTime ASC LIMIT 10\G</b>
*************************** 1. row
***************************

  id: 1

  select_type: SIMPLE

  table: Headline type: ALL

possible_keys: PRIMARY,ExpireTime
```

```
   key: NULL

   key_len: NULL

   ref: NULL

   rows: 302116

   Extra: Using where 1 row in set (0.00 sec)
```

```
(SELECT * FROM Headline WHERE ExpireTime
>= 1081020749

ORDER BY ExpireTime ASC LIMIT 10)
```

UNION

(SELECT * FROM Headline WHERE Id <= 50000

ORDER BY ExpireTime ASC LIMIT 10)

ORDER BY ExpireTime ASC LIMIT 10

mysql> <b>EXPLAIN (SELECT * FROM Headline WHERE ExpireTime >= 1081020749</b> -> <b>ORDER BY ExpireTime ASC LIMIT 10)</b> -> <b>UNION</b> -> <b>(SELECT * FROM Headline WHERE Id <= 50000</b> -> <b>ORDER BY ExpireTime ASC LIMIT 10)</b> -> <b>ORDER BY ExpireTime ASC LIMIT 10 \G</b>
*************************** 1. row ***************************

```
  id: 1

  select_type: PRIMARY

  table: Headline type: range

possible_keys: ExpireTime
```

```
  key: ExpireTime key_len: 4

  ref: NULL

  rows: 40306

  Extra: Using where
*************************** 2. row
***************************

  id: 2

  select_type: UNION

  table: Headline type: range

possible_keys: PRIMARY

  key: PRIMARY

  key_len: 4

  ref: NULL

  rows: 1
```

```
   Extra: Using where; Using filesort 2 rows in set
(0.00 sec)
```

Not bad at all. The second query needs a file sort operation, but at least it will use an index to locate all the rows.

Chapter 6. Server Performance Tuning

The operating system your MySQL server runs on and the server's configuration can be just as important to your server's performance as the indexes, schema, or queries themselves. In this chapter, we will help you understand how to tune your server to improve performance, as opposed to tuning schema or queries. We'll be looking at changes to your hardware, operating system, and MySQL configuration to see what effects they have on overall performance.

We assume that you've already made efforts to boost the performance of your queries. If you haven't done that already, stop now and read Chapter 4 and Chapter 5 to get a handle on optimizing your queries and your application code. Only then should you worry about server settings. Hardware is often not the solution to MySQL performance problems.

Poorly optimized queries can slow you down far more than not having the latest CPU or SCSI disk. To put this in perspective, one of the MySQL

AB trainers even says that changing hardware might, in the best cases, give you a 10-fold performance increase. But tuning queries (and schemas) can *often* give you 1000-fold performance increase. Seriously.

Some topics covered in this chapter are platform-specific. The authors' knowledge of the various platforms on which MySQL runs is limited. In many cases, you'll need to consult your local documentation for various operating system tools and specifics.

We start with an overview of the factors that limit performance and then look more in depth at RAID, hardware, and operating system issues. The chapter finishes with a discussion of techniques you can use to locate, identify, and fix bottlenecks.

6.1 Performance-Limiting Factors

Before we can begin to think about what to adjust on a busy MySQL server, it's best to get an understanding of the various factors that affect performance and, most importantly, *how* they can affect it. One of the single biggest problems that most MySQL users face is simply not understanding how to go about finding bottlenecks.

6.1.1 Disks

The fundamental battle in a database server is usually between the CPU(s) and available disk I/O performance; we'll discuss memory momentarily.

The CPU in an average server is orders of magnitude faster than the hard disks. If you can't get data to the CPU fast enough, it must sit idle while the disks locate the data and transfer it to main memory.

The real problem is that a lot of the disk access is random rather than sequential: read 2 blocks from here, 10 from there, 4 from there, and so on. This means that even though your shiny new SCSI disks are rated at 80 MB/sec throughput, you'll rarely see values that high. Most of the time you'll be waiting for the disks to locate the data. The speed at which the heads move across the platter and fetch another piece of data is known as *seek* *time,* and it's often the governing factor in real-world disk performance.

The seek time consists of two factors. First is the amount of time required to move the head from one location to the next. When the head arrives at the new location, it often needs to wait for the disk platter to rotate a bit more so that it can read the desired piece of information. The disk's rotation speed, measured in RPMs, is the second factor. Generally speaking, the faster the platters rotate, the lower the disk's seek time will be. When you're shopping for your database

server's disks, it's usually better

to spend the extra cash for the 15,000-RPM model rather than saving a bit with the cheaper 10,000-RPM model. As a bonus, higher RPM drives provide greater transfer rates because they're

reading data from a faster moving platter.

This all means that the first bottleneck you're likely to encounter is disk I/O. The disks are clearly the slowest part of the system. Like the CPU's caches,

MySQL's various buffers and caches use main memory as a cache for data that's sitting on disk. If your MySQL server has sufficient disk I/O capacity, and MySQL has been configured to use the available memory efficiently, you can better use the CPU's power.

A common complaint against MySQL is that it can't handle really large tables. Assuming the people making that statement have even used MySQL, they likely encountered an I/O bottleneck they didn't know how to fix. MySQL worked great with a few hundred megabytes of data, but once loaded up with 60 GB, it became slow. The conclusion drawn was that MySQL was somehow inadequate.

Of course, there are some circumstances in which MySQL can become CPU-bound rather than I/O-bound: they're simply not as common. If you often ask MySQL to perform some computation on your data (math, string comparison, etc.), the CPU will work harder. When running a `CHECK TABLE` command, you'll likely find the CPU pegged. And, of course, queries that aren't using indexes really tax it as well.

6.1.2 Memory

To bridge the gap between blazingly fast CPUs and comparatively slow disks, we have memory. With respect to performance, it's in the middlesignificantly faster than disks but still much slower than the CPU. The underlying operating system generally uses free memory to cache data read from and written to disk. That means if you frequently query the same small MyISAM table over and over, there's a very good chance you'll

never touch the disk. Even though MySQL doesn't cache row data for MyISAM tables (only the index blocks), the entire MyISAM table is likely in the operating system's disk cache.

Modern CPUs are even substantially faster than main memory. To combat this mismatch, chip makers have designed multilevel caching systems. It's common for a CPU to contain level 1, level 2, and even level 3 caches. The caches use significantly faster and more expensive memory, so they're generally a fraction of the size of main memory; a 512-KB L2 cache is generous.

With that in mind, simply adding memory to your server will improve MySQL performance only if the operating system can make good use of it by caching even more disk blocks. If your database is 512 MB, and you already have 1 GB of memory, adding more memory probably won't help.

On the other hand, if you run more than just MySQL on the server, adding memory may help. Maybe that Java application server you've been running is eating up a lot of the memory that could otherwise cache disk access. Keep in mind that Linux, like most modern operating systems, considers caching disk I/O an optional feature. It doesn't reserve any memory for it. So when free memory is low, MySQL can really suffer because MyISAM

tables expect the OS to do some read caching.

6.1.2.1 MySQL's buffers and caches

By adjusting how much memory MySQL uses, you can often realize significant performance improvements. To do that effectively, you first need to understand how MySQL uses memory. Most of the memory MySQL allocates is used for various internal buffers and caches. These buffers fall into two major groups: *global*

buffers and *per-connection* buffers. As their name implies, global buffers are shared among all the connections (or threads) in MySQL.

The two most important global buffers are the MyISAM key buffer (`key_buffer_size`) and InnoDB's buffer pool (`innodb_buffer_pool_size`).

The MyISAM key buffer is where MySQL caches frequently used blocks of index data for MyISAM tables. The less often MySQL needs to hit the disk to scan a table's index, the faster queries will be. If possible, consider making the key buffer large enough to hold the indexes for your most actively used tablesif not all your tables. By adding up the size of the *.MYI* files for the tables, you'll have a good idea how large to set the buffer.

MySQL doesn't cache rows for MyISAM

tablesonly indexes. InnoDB, on the other hand, caches index and row data together in its buffer pool. As you'll recall from Chapter 4, InnoDB uses clustered indexes. Because it stores the index and row data together, it's only natural to cache the index and row data in memory when possible.

Buying Server Hardware

When you shop for new database hardware, either with the intention to build yourself or to buy from a big-name vendor, there are many details to consider.

What's the difference between the $4,000 server sold by a big name vendor such as IBM, HP, or Dell, and the seemingly equivalent $2,300 unit that your favorite "white box" company is selling? There are several, and some affect MySQL performance. Let's have a look.

Memory speed

The CPU can access data faster if it's stored in PC3700

memory than older PC133 memory. Be sure to get the fastest system bus you can and memory to match. The less time the CPU spends waiting for data to arrive, the more work it can get done in a given amount of time. Server-class hardware often uses Error Checking and Correcting (ECC) memory that can detect flaws in memory that result from aging and outside factors such as radiation and cosmic rays.

CPU cache

Frequently accessed memory is cached by the CPU in its level 1, 2, or 3 cache. The larger cache you can get, the better.

Multiple I/O channels

More expensive "server class" systems

often have multiple, separate I/O channels rather than a single shared bus. That means the data moving between main memory and your disk controller doesn't interfere with the data path between the CPU and your network card. Again, this means the CPU

spends less time waiting for data to arrive or depart.

Unfortunately, this difference doesn't show up until a the system is under a fair amount of stress. If you take a normal white box system and a server class system and compare them with a simple benchmark, they may score the same. The white box might even score higher. But when they are under real-world production loads, the white box could perform miserably.

Redundant power

Having multiple power supplies won't make your server any faster. It will, however, allow the server to keep running if the primary supply dies. Given the choice between good performance and no performance, choose wisely. And, if you plug them into different power sources, you're protected in case a fuse or circuit breaker dies.

Hot-swappable disks

Hot-swappable RAID disks are a valuable feature not all servers provide. Not having them means that you can survive a disk failure, but you'll eventually need to shut down the machine to swap out the bad disk. The only way around this is if there's room for a spare disk (or *hot spare*) the RAID system can bring online in the event of a failure. When running a RAID array in "degraded" mode (missing a disk),

you're either sacrificing performance, redundancy, or both. You probably don't want to do either one for very long!

On a similar note, many name-brand servers provide battery-backed RAID controllers that ensure unwritten changes do get written to disk when power is restored. This boosts performance as well, because the writes can be considered completed when they are written to the controllers memory, rather than actually waiting for the physical disk writes to complete. Unfortunately, the caches provided by most vendors are relatively small.

Gigabit network or multiple network ports

Server-class hardware typically comes with better networking options than your run-of-the-mill desktop or laptop. Specifically you'll either see gigabit

Ethernet or dual Ethernet ports (often 100 Mbit). Having multiple network ports may be useful when setting up replication, as you'll see in Chapter 7.

It can be very tempting, especially if buying a number of servers for a cluster, to consider skimping on "the little

things" like how much CPU cache is onboard, or the speed of the memory, because those little things, over the cost of a couple hundred machines, can add up. Resist that urge, when you are building a singer server or replication master. It is one of the few times that "throwing money at it"

can make your life significantly more pleasant down the road.

On the other hand, if you want to build the next Google, your goal is probably to buy the greatest number of inexpensive machines as possible and to scale by simply adding more of them later on.

6.1.3 Network

The performance of your network usually doesn't have much bearing on MySQL. In most deployments, clients are very near the serversoften connected to the same switchso latency is low, and available bandwidth is quite high. But there are less common circumstances in which the network can get in the way.

Duplex mismatch is a common network configuration problem that often goes unnoticed until load begins to increase. When it does, by all appearances MySQL is sending results very slowly to clients. But when you check the server, you find the CPU is nearly idle, and the disks aren't working very hard either. For whatever reason, there's a lot of 100-Mbit Ethernet equipment that has trouble auto-sensing the proper settings. Be sure your server and switch agree on either half or full duplex operation.

Some MySQL deployments use Network Attached Storage (NAS) devices, such as a Network Appliance filer, rather than local disks for MySQL's data. The idea is that if the server dies, you can simply swap in a new one without having to worry about copying data or dealing with synchronization issues. (See Chapter 8 for more on this topic.) While that's all true, in dealing with a configuration it's critical that your network be as uncongested as possible. Ideally, you'll want to have a fast

dedicated network path between your MySQL server and the storage server. Typically that means installing a second Network Interface Card (NIC) that is connected to a private network with your storage server.

In a replication setup consisting of a single master and many slaves, it's quite possible to saturate a single network interface on the master with all the traffic generated by the slaves. This isn't because of something MySQL does horribly wrong. It's really just a matter of scale. Imagine that you have 50 slaves replicating from the master. Under normal circumstances, each slave uses a relatively small amount of bandwidthsay 100 KB/sec. That adds up to 5 Mbit/sec of bandwidth required for 50 slaves. If you're using 100-Mbit Ethernet,

that's not a big deal. But what if your master

begins getting more inserts per second, or large inserts that contain `BLOB` fields? You may reach the point that each slave needs 800 KB/sec of bandwidth to keep up with the master's data stream. At that point,

you're looking at 40 Mbit/sec of data on your

100-MBit network.

At that point you should begin to worry. One hundred Mbit/sec is the network's theoretical maximum bandwidth. In reality its capacity is quite a bit less that. Many network engineers use 50% utilization as a rule of thumb for capacity planning. Once they consistently see utilization that high, they begin thinking about how to break up the network to better isolate the traffic. The trouble is, that doesn't help much in this case. Because

there's a single master, all slaves must read from it.

There are three possible solutions to this problem. First, you can take a load off the master by introducing a second tier of slaves that replicate from the master. They, in turn, serve as masters for the 50 slaves. See Chapter 7 for more information about multitiered replication architectures.

Another option is to add a second network card to the master and split the 50 slaves across multiple switches. Each of the master's NICs are connected to a different switch.

The problem is that you'd need to remember which server is on which switch port and adjust the slave configuration appropriately.

A final solution is to compress the data stream between the master and slaves. This assumes that the data isn't already compressed and that the master has sufficient CPU power to handle compressing 50 outbound data streams while handling a high rate of inserts. Given the rate at which CPUs are evolving, this will soon be feasible. Chapter 7 discusses options for encrypting and compressing replication.

Performance can become an issue when your network links have relatively high latency. This is typically a problem when the client and server are separated by a great distance or by an inherently high-latency link, such as dial-up or satellite. Your goal should be to keep the clients and servers as close (in network sense) to each other as possible. If you can't do this, consider setting up slaves that are close to your most distant clients.

At first glance, this may not seem like a server-performance issue, but a high-latency or low-bandwidth network can really slow things down on the server side. When a client performs a large `SELECT` on a MyISAM table, it obtains a read lock on the data. Until the `SELECT` completes, the server won't release the lock and service any

pending write requests for the table. If the client asking for the data happens to be far away or on a flaky or congested network, it will take a long time to retrieve the data and release the lock. The end result is that things get backed up on the server side even though the server has sufficient CPU and disk I/O to do the work.

6.2 RAID

Nobody likes to lose data. And since disks eventually die, often with little warning, it's wise to consider setting up a RAID (Redundant Array of Inexpensive[1] Disks) array on your database servers to prevent a disk failure from causing unplanned downtime and data loss. But there are many different types of RAID to consider: RAID 0, 1, 0+1, 5, and 10. And what about hardware RAID versus software RAID?

[1] The "I" in RAID has meant, at various times, either "Inexpensive" or "Independent." It started out as "Inexpensive," but started being referred to as "Independent" because drives weren't really all that inexpensive. By the time people actually started using "Independent," the price of disks had plummeted and they really were "Inexpensive." Murphy at work.

From a performance standpoint, some options are better than others. The faster ones will sacrifice something to gain that performanceusually price or durability. In all cases, the more disks you have, the better performance you'll get. Let's consider the benefits and drawbacks of each RAID option.[2]

[2] For a more complete treatment of this topic, consult Derek Vadala's *Managing RAID on Linux* published by O'Reilly.

RAID 0

Of all the RAID types, RAID 0, or *striping*, offers the biggest performance improvement. Writes and reads are both faster in RAID 0 than in any other configuration. Because there are no spare or mirrored disks, it's inexpensive. You're using every disk you pay for. But the performance comes at a high price.

There's no redundancy at all. Losing a single disk means that your whole array is dead.

RAID 0 should be used only when you don't care about data loss. For example, if you're building a cluster of MySQL slaves, it's entirely reasonable to use RAID 0. You'll reap all the performance benefits, and if a server does die, you can always clone the data from one of the other slaves.

RAID 1

Moving up the scale, RAID 1, or *mirroring*, isn't as fast as RAID 0, but it provides redundancy; you can lose a disk and keep on running. The performance boost applies only to reads. Since all the data is on every disk in the mirrored volume, the system may decide to read data in parallel from the disks. The result is that in the optimal case it can read the same amount of data in roughly half the time.

Write performance, however is only as good as a single disk. It can even be half as good depending on whether the RAID controller performs the writes in parallel or sequential order. Also, from a price point of view, you're paying for twice as much space as you're using. RAID 1 is a good choice when you need redundancy but have space or budget for only two diskssuch as in a 1-U rackmount case.

RAID 5

From a performance standpoint, RAID 5, which is striping (RAID 0) with distributed *parity blocks*, can be beneficial. There are two disks involved in every operation, so it's not substantially faster than RAID 1 until you have more than three disks total. Even then, its other benefit, size, shines through. Using RAID 5, you can create rather large volumes without spending a lot of cash because you sacrifice only a single disk. By using more smaller disks, such as eight 36-GB disks instead of four 72-GB disks, you increase the number of spindles in the array and therefore boost seek performance and throughput.

RAID 5 is the most commonly used RAID implementation. When funds are tight, and redundancy is clearly more important than performance, it's the best compromise available.

RAID 10 (also known as RAID 1+0)

To get the best of both worlds (the performance benefits of RAID 0 along with the redundancy of RAID 1), you need to buy twice as many disks. RAID 10 is the only way to get the highest performance on your database server without sacrificing redundancy. If you have the budget to justify it, you won't be disappointed.

JBOD

The configuration sometimes called "Just a Bunch of Disks" (JBOD) provides no added performance or redundancy. It's simply a combination of two or more smaller disks to produce a single, larger virtual disk.

Table 6-1 summarizes various RAID features.

Table 6-1. Summary of various RAID features

Level	Redundancy	Disks required	Faster reads	Faster writes
RAID 0	No	N	Yes	Yes
RAID 1	Yes	2[3]	Yes	No
RAID 5	Yes	N+1	Yes	No
RAID 10	Yes	N*2	Yes	Yes
JBOD	No	N/A	No	No

[3] Typically, RAID 1 is used with two disks. but it's possible to use more than two. Doing so will boost read performance but doesn't change write performance.

6.2.1 Mix and Match

When deciding how to configure your disks, consider the possibility of multiple RAID arrays. RAID controllers aren't that expensive, so you might benefit from using RAID 5 or RAID 10 for your databases and a separate RAID 1 array for your transaction and replication logs. Some multichannel controllers can manage multiple arrays, and some can even bind several channel controllers together into a single controller to support more disks.

Doing this isolates most of the serial disk I/O from most of the random, seek-intensive I/O. This is because transaction and replication logs are usually large files that are read from and written to in a serial manner, usually by a small number of threads. So it's not necessary to have a lot of spindles available to spread the seeks across. What's important is having sufficient bandwidth, and virtually any modern pair of disks can fill that role nicely. Meanwhile, the actual data and indexes are being read from and written to by many threads simultaneously in a fairly random manner. Having the extra spindles associated with RAID 10 will boost performance. Or, if you simply have too much data to fit on a single disk, RAID 5's ability to create large volumes works to your advantage.

6.2.1.1 Sample configuration

To make this more concrete, let's see what such a setup might look like with both InnoDB and MyISAM tables. It's entirely possible to move most of the files around and leave symlinks in the original locations (at least on Unix-based systems), but that can be a bit messy, and it's too easy to accidentally remove a symlink (or accidentally back up symlinks instead of actual data!). Instead, you can adjust the *my.cnf* file to put files where they belong.

Let's assume you have a RAID 1 volume on which the following filesystems are mounted: */*, */usr*, and *swap*. You also have a RAID 5 (or RAID 10) filesystem mounted as */data*. On this particular server, MySQL was installed from a binary tarball into */usr/local/mysql*, making */usr/local/mysql/data* the default data directory.

The goal is to keep the InnoDB logs and replication logs on the RAID-1 volume, while moving everything else to */data*. These *my.cnf* entries can accomplish that:

```
datadir = /data/myisam

log-bin = /usr/local/mysql/data/repl/bin-log

innodb_data_file_path =
ibdata1:16386M;ibdata2:16385M

innodb_data_home_dir = /data/ibdata

innodb_log_group_home_dir =
/usr/local/mysql/data/iblog

innodb_log_arch_dir = /usr/local/mysql/data/iblog
```

These entries provide two top-level directories in */data* for MySQL's data files: *ibdata* for the InnoDB data and *myisam* for the MyISAM files. All the logs remain in or below */usr/local/mysql/data* on the RAID 1 volume.

6.2.2 Hardware Versus Software

Some operating systems can perform *software RAID*. Rather than buying a dedicated RAID controller, the operating system's kernel splits the I/O among multiple disks. Many users shy away from using these features because they've long been considered slow or buggy.

In reality, software RAID is quite stable and performs rather well. The performance differences between hardware and software RAID tend not to be significant until they're under quite a bit of load. For smaller and medium-sized workloads, there's little discernible difference between them. Yes, the server's CPU must do a bit more work when using software RAID, but modern CPUs are so fast that the RAID operations consume a small fraction of the available CPU time. And, as we stressed earlier, the CPU is usually not the bottleneck in a database server anyway.

Even with software RAID, you can use multiple disk controllers to achieve redundancy at the hardware level without actually paying for a RAID controller. In fact, some would argue that having two non-RAID controllers is better than a single RAID controller. You'll have twice the available I/O bandwidth and have eliminated a single point of failure if you use RAID 1 or 10 across them.

Having said that, there is one thing that can be done with hardware RAID that simply can't be done in software: write caching. Many RAID controllers can add battery-backed RAM that caches reads and writes. Since there's a battery on the card, you don't need to worry about lost writes even when the power fails. If it does, the data stays in memory on the controller until the machine is powered back up. Most hardware RAID controllers can also read cache as well.

6.2.3 IDE or SCSI?

It's a perpetual question: do you use IDE or SCSI disks for your server? A few years ago, the answer was easy: SCSI. But the issue is further muddied by the availability of faster IDE bus speeds and IDE RAID controllers from 3Ware and other vendors. For our purposes, Serial-ATA is the same as IDE.

The traditional view is that SCSI is better than IDE in servers. While many people dismiss this argument, there's real merit to it when dealing with database servers. IDE disks handle requests in a sequential manner. If the CPU asks the disk to read four blocks from an inside track, followed by eight blocks from an outside track, then two more blocks from an inside track, the disk will do exactly what it's told; even if it's not the most efficient way to read all that data.

SCSI disks have a feature known as Tagged Command Queuing (TCQ). TCQ allows the CPU to send several read/write requests to the disk at the same time. The disk controller then tries to find the optimal read/write pattern to minimize seeks.

IDE also suffers from scaling problems; you can't use more than one drive per IDE channel without suffering a severe performance hit. Because most motherboards offer only four IDE channels at most, you're stuck with only four disks unless you add an additional controller. Worse yet, IDE has rather restrictive cable limits. With SCSI, you can typically add 7 or 14 disks before purchasing a new controller. Furthermore, the constant downward price pressure on hard disks has affected SCSI as much as IDE.

On the other hand, SCSI disks still cost more than their IDE counterparts. When you're considering four or more disks, the price difference is significant enough that you might be able to purchase IDE disks and be able to afford another controller, possibly even an IDE RAID controller. Many MySQL users are quite happy using 3Ware IDE RAID controllers with 4-12 disks on them. It costs less than a SCSI option, and the performance is reasonably close to that of a high-end SCSI RAID controller.

6.2.4 RAID on Slaves

As we mentioned in the discussion of RAID 0, if you're using replication to create a cluster of slaves for your application, it's likely that you can save money on the slaves by using a different form of RAID. That means using a higher-performance configuration that doesn't provide redundancy (RAID 0), using fewer disks (RAID 5 instead of RAID 10), or using software rather than hardware RAID, for example. If you have enough slaves, you may not necessarily need the redundancy on the slaves. In the event that one slave suffers the loss of a disk, you can always synchronize it with another nearby slave to get it started again.

```
innodb_data_home_dir=

innodb_data_file_path=/dev/sdb1:18Graw;/dev/sdc1:18Graw
```

However, you must first initialize the partitions. To do so, use `newraw` instead of `raw` the first time and start MySQL. InnoDB will the initialize the partitions. Watch the MySQL log file for completion, shut down MySQL, change `newraw` to `raw`, and start MySQL again.

From a performance standpoint, tests have shown a very small (2-5%) performance improvement using raw partitions. When you use raw partitions, you can no longer use any of your favorite command-line tools (ls, du, etc.) to investigate the storage. Furthermore, backups are more complicated when using raw disks. Your choice of backup tools is greatly reduced because most deal with filesystems rather than raw disk partitions.

6.3.2 Swap

In an ideal world, your server would never swap. Swapping is usually an indication that you don't have enough memory or that things are configured improperlymaybe MySQL's key buffer is too large, or you're starting too many unused services at boot time. Maybe it's the operating system itself. Some operating systems make a habit of swapping when there's still free memory available.

Some versions of the 2.4 Linux kernel, for example, are known for being a bit too aggressive with swapping. Linux has generally tried to use all available free memory for caching disk access. From the virtual memory subsystem's point of view, free memory is wasted memory. Early versions (2.4.0-2.4.9) were okay, as are later versions (2.4.18 onward). But the middle versions (2.4.10-2.4.17) were known for being a bit too aggressive. On a dedicated MySQL server, with a key buffer of 1 GB and 2 GB of total RAM, it was not uncommon to see Linux swap out parts of the key buffer while performing a table scan, only to swap it back in moments later. Needless to say, this had a very negative affect on performance. The only solution in such a case is to turn off swap entirely or upgrade to a newer kernel. Luckily, most other operating systems haven't suffered from this problem. Even though most systems are well behaved, some MySQL administrators advocate turning swap off as a preventative measure.

6.3.3 Threading

As a multithreaded server, MySQL is most efficient on an operating system that has a well implemented threading system. Windows and Solaris are excellent in this respect. Linux, as usual, is a bit different. Traditionally, Linux has had a slightly unusual threading implementationusing cloned processes as threads. It performs well under most circumstances, but in situations with thousands of active client connections, it imposes a bit of overhead.

More recent work on the Linux scheduler and alternative threading libraries have improved the situation. The Native POSIX Thread Library (NPTL) is shipped by default in RedHat Linux Version 9.0. Other distributions have just begun adopting it as well.

Another popular free operating system, FreeBSD, has threading problems that are much worse. Versions prior to 5.2 provide rather weak native threading. In some circumstances, I/O-intensive threads are able to get an unfair amount of CPU time, thus keeping other threads from executing as quickly as they should. Given the I/O-intensive nature of some database queries, this has a rather devastating affect on MySQL.

If upgrading isn't an option, build MySQL from the FreeBSD ports collection, and be sure to enable support for LinuxThreads. Doing so causes MySQL to use an alternative threading that's more like that available in Linux 2.4. Each thread is actually a process that, thanks to FreeBSD's `rfork( )` call, has shared access to MySQL's global buffers. The overhead of this approach may sound like an issue, but it's really quite efficient. Many of Yahoo's hundreds of MySQL servers are using LinuxThreads on FreeBSD quite effectively.

Section 6.4.4 later in this chapter discusses how MySQL's thread cache can help reduce the overhead associated with creating and destroying threads.

```
+------------------+----------------------+------+-----+-----
----+-------+

| Field | Type | Null | Key | Default | Extra |

+------------------+----------------------+------+-----+-----
----+-------+

| agent | varchar(255) | YES | MUL | NULL | |

| bytes_sent | int(10) unsigned | YES | | NULL | |

| child_pid | smallint(5) unsigned | YES | | NULL | |

| cookie | varchar(255) | YES | | NULL | |

| request_file | varchar(255) | YES | | NULL | |

| referer | varchar(255) | YES | | NULL | |

| remote_host | varchar(50) | YES | MUL | NULL | |

| remote_logname | varchar(50) | YES | | NULL | |

| remote_user | varchar(50) | YES | | NULL | |
| request_duration | smallint(5) unsigned | YES | |
NULL | |

| request_line | varchar(255) | YES | | NULL | |

| request_method | varchar(6) | YES | | NULL | |

| request_protocol | varchar(10) | YES | | NULL | |

| request_time | varchar(28) | YES | | NULL | |

| request_uri | varchar(255) | YES | MUL | NULL | |

| server_port | smallint(5) unsigned | YES | | NULL | |

| ssl_cipher | varchar(25) | YES | | NULL | |

| ssl_keysize | smallint(5) unsigned | YES | | NULL | |

| ssl_maxkeysize | smallint(5) unsigned | YES | |
NULL | |

| status | smallint(5) unsigned | YES | | NULL | |

| time_stamp | int(10) unsigned | YES | MUL | NULL
| |
| virtual_host | varchar(50) | YES | | NULL | |

+------------------+---------------------+------+-----+-----
----+-------+

select request_uri from
access_jeremy_zawodny_com where remote_host =
'24.69.255.236'

  and time_stamp >= 1056782930

  and time_stamp <= 1056869330

order by time_stamp asc

mysql> <b>explain select request_uri from
access_jeremy_zawodny_com</b> -> <b>where
remote_host = '24.69.255.236'</b> -> <b>and
time_stamp >= 1056782930</b> -> <b>and
time_stamp <= 1056869330</b> -> <b>order by
time_stamp asc \G</b>
*************************** 1. row
***************************

  table: access_jeremy_zawodny_com type: ref

possible_keys: time_stamp,remote_host
```

```
  key: remote_host

  key_len: 6

  ref: const

  rows: 4902

  Extra: Using where; Using filesort 1 row in set
(0.00 sec)

mysql> <b>explain select request_uri from
access_jeremy_zawodny_com</b> -> <b>where
remote_host = '67.121.154.34'</b> -> <b>and
time_stamp >= 1056782930</b> -> <b>and
time_stamp <= 1056869330</b> -> <b>order by
time_stamp asc \G</b>
*************************** 1. row
***************************

  table: access_jeremy_zawodny_com type: range

possible_keys: time_stamp,remote_host

  key: time_stamp

  key_len: 5
  ref: NULL

  rows: 20631

  Extra: Using where

1 row in set (0.01 sec)

SELECT ... USE_INDEX(time_stamp) ...

mysql> <b>SHOW STATUS LIKE
'Created_tmp_%';</b> +-------------------------+-------
+

| Variable_name | Value |

+-------------------------+-------+

| Created_tmp_disk_tables | 18 |

| Created_tmp_tables | 203 |

| Created_tmp_files | 0 |

+-------------------------+-------+
```

min_memory_needed = global_buffers + (thread_buffers * max_connections)

where `thread_buffers` includes the following:

```
sort_buffer
myisam_sort_buffer
read_buffer
join_buffer
read_rnd_buffer
```

and `global_buffers` includes:

```
key_buffer
innodb_buffer_pool
innodb_log_buffer
innodb_additional_mem_pool
net_buffer
```

We say that's the minimum memory required because ideally you'd like some left over for the operating system itself to use. In the case of MyISAM tables, "spare" memory will often be put to use caching records from MyISAM data (.MYD) files.

In addition to any memory the threads may allocate in the process of handling queries, the threads themselves also require a bit of memory simply to exist. The `thread_stack` variable controls this overhead. On most platforms, 192 KB is the default value.[5]

[5] If you happen to be using LinuxThreads on FreeBSD, the value is hardcoded in the LinuxThreads library. Changing MySQL's `thread_stack` setting will have no effect. You must recompile the library to change the stack size.

A likely problem is typified by an all-too-common scenario. Imagine you have a server with 1 GB of memory hosting a mix of MyISAM and InnoDB tablesmostly MyISAM. To get the most bang for your buck, you configure a 512-MB `key_buffer` after watching the key efficiency in `mytop` (see Appendix B) and a 256-MB `innodb_buffer_pool` after checking the buffer pool and memory statistics from `SHOW INNODB STATUS` (see Appendix A). That leaves 256 MB that is used to cache data files at the operating system level as well as the per-thread buffers that are allocated on an as-needed basis. The MySQL server handles a relatively small number of concurrent users, maybe 20-50 most of the time, and the per-thread buffer settings are all left at their default sizes.

Things work very well until a few new applications are built that also use this MySQL server. These new applications need a significant number of concurrent connections. Instead of 20-50 connections on average, the server is handling 300-400. When this happens, the odds of several connections needing to allocate a per-thread buffer (such as the `sort_buffer`) at the same time increase quite a bit.

This can lead to a particularly nasty series of events. If a large number of threads need to allocate additional memory, it's probably because the server is handling a heavy query load. That can cause MySQL to allocate so much memory that the operating system begins swapping, which causes performance to degrade further, which means that each query takes longer to complete. With queries running more slowly, the odds of more threads needing memory increases. It's a vicious spiral.

The only solution is to restore balance between the system's memory and MySQL's memory needs. That means doing one of the following.

- Add more memory
- Decrease `max_connections`
- Decrease some of the per-thread buffer sizes

Be proactive. Monitor memory use on your servers. Do the math to ensure that in the worst case (hitting `max_connections` and each thread allocating additional memory), you'll still have a bit of breathing room.

6.4.4 Solving Kernel Bottlenecks

Though it's not common, you may find that MySQL doesn't appear to be using an overwhelming amount of CPU time, yet the machine is rather busy. There's little idle CPU. Upon looking at it more closely, you find that quite a bit of the time is spent in "system" rather than "user" or "idle." That's likely a sign that MySQL is doing something unusual to exercise the kernelusually creating and destroying threads.

This happened at Yahoo! during the launch of a new web site. In September 2002, engineers were scrambling to create a September 11th memorial web site known as remember.yahoo.com.[6] On it, anyone could create a memorial "tile" by selecting a graphic and adding a customized message. The tile was then viewable by anyone visiting the site. To get the job done as quickly as possible, it was constructed using standard open source tools, including FreeBSD, Apache, PHP, and MySQL

[6] The entire site was conceived, designed, built, and launched in roughly two weeks using the spare time of handful of Yahoo's engineers.

The architecture was relatively straightforward, but we'll simplify it a bit to focus on the main point. A group of frontend web servers was configured to connect to a slave server by way of a hardware load balancer. Using the slave connection, the server could pull the information necessary to display the tiles. When a visitor created a tile, however, the web server needed to connect to the master to insert several records. The master was a beefier machine: dual 1.2-GHz CPUs, 2 GB of RAM, and a SCSI hardware RAID 5 disk array.

At its peak, there were roughly 25-30 web servers that needed to work with the master. Each server was configured to run roughly 30-40 Apache processes. That meant the master would need to support over 1,000 concurrent clients. Knowing that could tie up substantial resources on the master, the designers opted for a simplified approach. Unfortunately, the web application (written in PHP) was configured to use persistent connections. So, to keep connection numbers down on the master, the `wait_timeout` was set very lowto roughly 10 seconds.

By and large, it worked. Idle connections were dropped after 10 seconds. The number of connections on the master remained below 200, leaving lots of resources free. But there was a problem: the CPUs in the master were quite busy. Most of the time there was less than 10% idle time, and nearly 50% of the CPU time was being spent on system (rather than user) tasks.

After an hour or so of head-scratching, looking at system logs and the output of `SHOW STATUS`, a light finally flickered on in Jeremy's head. The value of `Threads_created` was very large and increasing at an alarming rate. The kernel was so busy creating and destroying threads that it was eating into MySQL's ability to use the CPUs effectively.

With that realization, the solution was easy. Increasing the `thread_cache` from its default value of 0 to roughly 150 resulted in an instant improvement. The system CPU time dropped to roughly 10%, thus freeing up quite a bit of CPU time for MySQL to use. As it turns out, MySQL didn't need it all, so the machine ended up with 20% idle timebreathing room.

Chapter 7. Replication

MySQL

use often grows organically. In the corporate world, a single application developer may build the company's next killer app on top of MySQL. This initial success with MySQL

development typically breeds more projects and more success. As the amount of data you manage using MySQL grows, you'll certainly appreciate its ability to handle large amounts of data efficiently. You may even find that MySQL has become the de facto standard backend storage for your applications.

At the same time, you may also begin to wish for an easy way to copy all the data from one MySQL server to another. Maybe you need to share data with a remote office in your organization, or you might just like to have a "hot spare"

available in case your server dies. Fortunately, MySQL has a built-in replication system.

You

can easily configure a second server as a

slave of your *master*,

ensuring that it has an exact copy of all your data.

In this chapter, we'll examine all aspects of MySQL

replication. We begin with an overview of how replication works, the problems it solves, and the problems it doesn't solve. We then move on to the ins and outs of configuring replication. After that we'll consider the various architectures you can construct using various numbers of masters and slaves. We'll continue with a discussion of

administrative issues, including maintenance, security, useful tools, and common problems. Finally, we'll look ahead to some planned changes and improvements for MySQL's replication.

MySQL Versions 3.23.xx and 4.0.x have slightly different replication implementations. Much of the discussion in this chapter applies to both versions.

There are sections that apply to only one, however, and they are explicitly noted.

7.1 Replication Overview

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

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

Before peering under the hood, let's look at the types of problems replication does and doesn't

solve. If you're reading this in the hopes of

deploying replication to cure a problem, this section may help you decide whether you're on the right track.

7.1.1 Problems Solved with Replication

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

7.1.1.1 Data distribution

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

In fact, it's possible to use

replication over a network connection that isn't

"always on," such as traditional

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

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

7.1.1.2 Load balancing

If you use MySQL on a large data warehousing application or a popular web site, odds are that your server is running many more read queries (`SELECT`) than write queries (`INSERT`,

`UPDATE`, and `DELETE`). If

that's the case, replication is an excellent way to support basic load balancing. By having one or more slave servers, you can spread most of the work among several servers.

The trick, of course, is coming up with an effective way to spread the queries among the available slaves so they get roughly equal workloads. One simple approach is to use round-robin DNS. Assign multiple IP addresses for a hostname such as *db-slave.example.com*, and your application will connect to one at random each time it opens a new connection to MySQL.[1]

[1] Some operating systems don't randomize this very well.

A more sophisticated approach involves the same solutions that are used in web server load balancing. Network load-balancing products from Foundry Networks, Cisco, Nortel, and others work just as well for MySQL as they do for web sites.[2] The same is true of software solutions such as the Linux Virtual Server (LVS) project (http://www.linuxvirtualserver.org/).

[2] That's not entirely true, as you'll soon see.

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

7.1.1.3 Backup and recovery

Backing up a busy MySQL server can be difficult when your clients demand access to the data 24 hours a day.

Rather than deal with the complexity of implementing a backup process that minimizes the impact on your clients, you might find it easier simply to configure a slave server and back it up instead. Because the slave will have an exact copy of the data, it's just as good as backing up the master. Plus you'll never impact the clients who are using the master. You might even decide that you don't necessarily need or want to back up the data as long as you have the "hot

spare" slave database available in the case of

problems with the master.

Chapter 9 covers backup and recovery techniques in more detail.

7.1.1.4 High availability and failover

Using replication, you can avoid making MySQL (or the system hosting it) a single point of failure in your applications. Although there's no

out-of-the-box, automated failover solution for MySQL, you can achieve a good degree of high availability using some relatively simple techniques.

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

Let's suppose you have two MySQL servers, *db1.example.com* and

db2.example.com. Rather than hardcoding the name of the master in your applications, you can set up *db.example.com* as a

CNAME (or alias) in DNS for your master. By using a very low Time To Live (TTL) on the DNS record, you can ensure that clients will not cache the information longer than necessary.

In the event your master goes down, simply update your DNS to point `db.example.com` at the new master. As soon as the TTL expires, your applications will pick up the new information and connect to the proper server. There will be some time during which the applications can't contact MySQL, but that time will be relatively brief if you use a low enough TTL.[3]

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

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

- Use an IP address for each role, such as 192.168.1.1 for the master and an address in the 192.168.1.100-192.168.120 range for each slave.

- Make sure each machine has its own primary IP address in addition to the role-based IP address.

- When the master goes down, any of the slaves can be scripted to pick up the IP address and immediately take over.

- The master should be set so that if it ever loses its master address or goes down, it doesn't automatically pick up the address again (i.e., it assumes someone else will).

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

7.1.2 Problems Not Solved with Replication

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

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

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

won't be able to get data fast enough to do so.

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

7.1.2.1 Real-time data transmission

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

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

Of course, you can also build your application in such a way that it updates all the slaves with the newest data. However, that can add a lot of complexity and may not be worth the effort.

You'd end up writing a lot of code to handle the exceptional conditions, such as when a single server falls behind or is intermittently inaccessible. Testing and debugging all those situations can be very time-consuming and difficult.

As Derek went over this, he thought, "Wouldn't it be cool if MySQL could

provide a query response that signified, `Go ask another server, I'm really busy right

now'?" This would allow clients to

automatically find willing servers in a multihost DNS rotation.

For example, the client wants to connect to *db.example.com* (which is

db1, *db2*, and

db3). It connects (randomly) to

db2, and the server answers the query with "I'm busy; go ask someone

else," whereupon the client knows enough to try

db1 or *db3*. Because the

client library would be connecting to the same virtual server, it could transparently disconnect from the busy server and connect to some other (hopefully less busy) server.

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

7.1.2.2 Online ordering

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

Transactions aren't written to the binary log and therefore sent to slaves until they first commit on the master. A slave will contain only committed transactions.

Replication can still be very useful for an order processing system.

It's reasonable to use a slave for running nightly reports and queries that don't need the most recent data.

7.1.3 Replication Performance

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

general idea of how fast replication really is?

That's exactly what we wanted to know when we first began using replicationpartly for our own sanity and partly because we knew a lot of people would soon be interested in MySQL

replication. The first question they'd ask is,

"How fast is it?" To answer that

question, we devised the following simple test to measure the practical minimum replication latency in a particular environment.

A Perl script opened two database connections, one to the master and one to the slave. The master and slave were on the same 100-Mbit switched Ethernet network. The script then inserted a record into the master and immediately attempt to retrieve it from the slave. If the record wasn't available, the script immediately

retried. We kept the records intentionally small, containing just an `auto-increment` column and a

`VARCHAR` field into which we inserted the current time.

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

```
mysql> <b>GRANT REPLICATION SLAVE,
REPLICATION CLIENT ON *.* TO

repl@"192.168.1.0/255.255.255.0" IDENTIFIED BY
'c0pyIT!';</b>

Query OK, 0 rows affected (0.00 sec)

mysql> <b>SHOW GRANTS FOR repl;</b>

+------------------------------------------------------------
-------------------+

|Grants for repl@"192.168.1.0/255.255.255.0" |

+------------------------------------------------------------
-------------------+
| GRANT REPLICATION SLAVE, REPLICATION
CLIENT ON *.* TO 'repl'@'...' IDENTIFIED BY ...|

+--------------------------------------------------------
----------+

1 row in set (0.00 sec)
```

```
log-bin
```

```
server-id = 1
log-bin = /var/db/repl/log-bin
server-id = 2
```

```
master-host = master.example.com
master-user = repl
```

```
master-password = c0pyIT!



master-port = 3306

$ <b>mysqlbinlog master-bin.001</b>

# at 4



#020922 14:59:11 server id 1 log_pos 4 \

Start: binlog v 3, server v 4.0.4-beta-log created
020922 14:59:11

021103 13:58:10 Slave I/O thread: connected to
master 'repl@master:3306',

replication started in log 'log-bin.001' at position 4

log-bin
```

```
server-id = 1

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

```
LOAD TABLE mytable FROM MASTER
```

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

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

7.2.2.4 Online copy and synchronize (MySQL 4.x only)

MySQL 4.0 introduced the `LOAD DATA FROM MASTER` command. It combines the previous two approaches by first obtaining a read lock on all the master's tables, then loading each table one by one using the `LOAD TABLE` mechanism.[6] It respects any slave-side database or table filtering. Once it completes the loading process, it releases the locks on the master and begins replicating.

> [6] This doesn't include the tables in the `mysql` database. Put another way, `LOAD DATA FROM MASTER` doesn't clone users and permissions from the master.

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

7.3 Under the Hood

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

7.3.1 Replication in 3.23

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

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

1. The client issues a query on the master.
2. The master parses and executes the query.
3. The master records the query in the binary log.
4. The slave reads the query from the master.
5. The slave parses and executes the query.
6. The slave performs a sanity check, comparing its result with the master's. If the query failed on the slave but succeeded on the master, replication stops. The reverse is also true. If the query

partially completed on the master but succeeds on the slave, the slave stops and complains.

7. The slave updates the *master.info* file to reflect the new offset at which it is reading the master's binary log.

8. The slave waits for the next query to appear in the master's binary log. When one appears, it starts over at Step 4.

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

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

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

7.3.2 Replication in 4.0

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

These two threads divide the work in an effort to make sure the slave can always be as up to date as possible. The IO thread is concerned only with replicating queries from the master's binary log. Rather than execute them, it records them into the slave's *relay log*. [7] The SQL thread reads queries from the local relay log and executes them.

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

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

1. The client issues a query on the master.
2. The master parses and executes the query.
3. The master records the query in the binary log.
4. The slave's IO thread reads the query from the master and appends it to the relay log.
5. The slave's IO thread updates the *master.info* file to reflect the new offset at which it is reading the master's binary log. It then returns to Step 4, waiting for the next query.
6. The slave's SQL thread reads the query from its relay log, parses it, and then executes it.
7.

The slave's SQL thread performs a sanity check, comparing its result with the master's. If the query failed on the slave but succeeded on the master, replication stops.

8. The slave's SQL thread updates the *relay-log.info* file to reflect the new offset at which it is reading the local relay log.

9. The slave's SQL thread waits for the next query to appear in the relay log. When one appears, it starts over at Step 6.

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

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

7.3.3 Files and Settings Related to Replication

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

7.3.3.1 Log files

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

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

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

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

7.3.3.2 Log index files

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

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

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

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

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

7.3.3.3 Status files

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

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

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

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

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

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

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

7.3.3.4 Filtering

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

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

```
binlog-do-db=dbname

binlog-ignore-db=dbname
```

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

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

Here is the second set of options:

```
replicate-do-table=dbname.tablename

replicate-ignore-table=dbname.tablename

replicate-wild-do-table=dbname.tablename

replicate-wild-ignore-table=dbname.tablename

replicate-do-db=dbname

replicate-ignore-db=dbname

replicate-rewrite-db=from_dbname->to_dbname
```

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

7.4 Replication Architectures

Though MySQL's replication system is relatively simple

compared to some commercial databases, you can use it to build arbitrarily complex architectures that solve a variety of problems.

In this section we'll look at some of the more

popular and exotic configurations. We'll also review how MySQL's replication design makes this possible.

7.4.1 The Replication Rules

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

- Every slave must have a unique server ID.
- A slave may have only one master.

- A master may have many slaves.
- Slaves can also be masters for other slaves.

The first rule isn't entirely true, but let's assume that it is for right now, and soon

enough you'll see why it isn't

always necessary. In any case, the rules aren't

terribly complex. Those four rules provide quite a bit of flexibility, as the next sections illustrate.

7.4.2 Sample Configurations

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

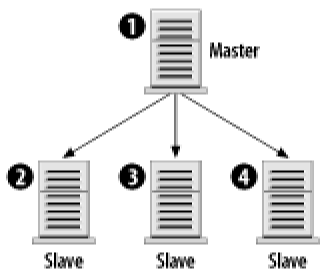
configuration as a building block for arbitrarily complex architectures.

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

7.4.2.1 Master with slaves

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

Figure 7-1. Simple master/slave replication



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

The only limit you are likely to hit is bandwidth from the master to the slaves. If you have 20 slaves, which each need to pull an average of 500 KB per second, that's a total of 10,000

KB/sec (or nearly 10 Mbits/sec) of bandwidth.

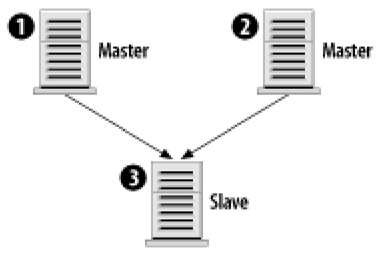
A 100-Mbit network should have little trouble with that volume, but if either the rate of updates to the master increases or you significantly increase the number of slaves, you run the risk of saturating even a 100-Mbit network. In this case, you need to consider gigabit network hardware or an alternative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

7.4.2.2 Slave with two masters

It would be nice to use a single slave to handle two unrelated masters, as seen in Figure 7-2. That allows you to minimize hardware costs and still have a backup server for each master. However, it's a violation of the second rule: a slave

can't have two masters.

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



To get around that limitation, you can run two copies of MySQL on the slave machine. Each MySQL instance is responsible for replicating a different master. In fact, there's no reason you

couldn't do this for 5 or 10 distinct MySQL masters.

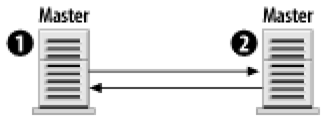
As long as the slave has sufficient disk space, I/O, and CPU power to keep up with all the masters, you shouldn't have any problems.

7.4.2.3 Dual master

Another possibility is to have a pair of masters, as pictured in Figure 7-3.

This is particularly useful when two geographically separate parts of an organization need write access to the same shared database. Using a dual-master design means that neither site has to endure the latency associated with a WAN connection.

Figure 7-3. Dual master replication



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

Of course, there are drawbacks to this setup. Section 7.7.3, later in this chapter, discusses some of the problems associated with a multi-master setup.

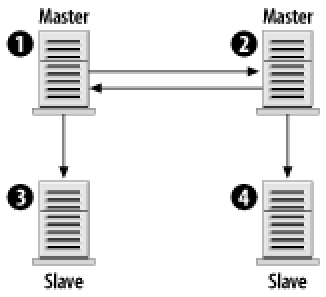
However, if responsibility for your data is relatively well partitioned (site A writes only to customer records, and site B

writes only to contract records) you may not have much to worry about.

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

data.

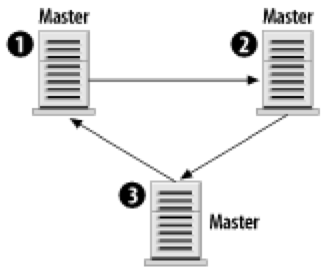
Figure 7-4. Dual master replication with slaves



7.4.2.4 Replication ring (multi-master)

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

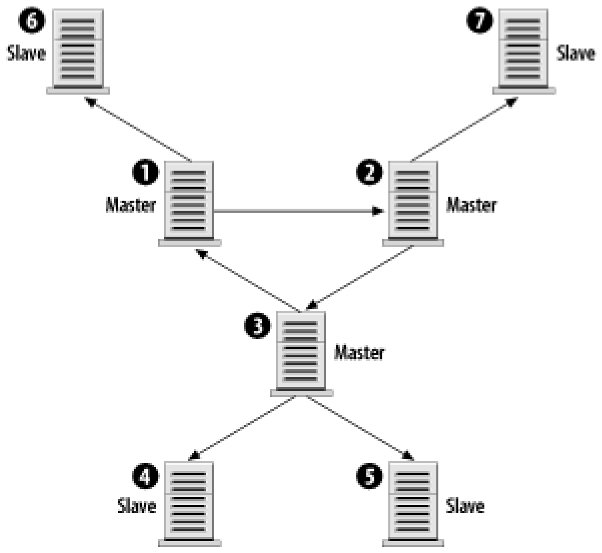
Figure 7-5. A replication ring or multi-master replication topology



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

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

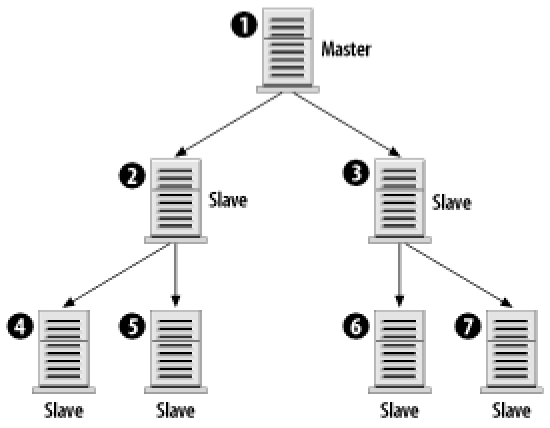
Figure 7-6. A replication ring with slaves at each site



7.4.2.5 Pyramid

In large, geographically diverse organizations, there may be a single master that must be replicated to many smaller offices. Rather than configure each slave to contact the master directly, it may be more manageable to use a pyramid design as illustrated in Figure 7-7.

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



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

7.4.2.6 Design your own

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

to: if you need to replicate your data to various locations, there's a good chance you can design a good solution using MySQL.

You can often combine aspects of the architectures we've looked at. In reality, however, the vast

majority of needs are handled with less complicated architectures. As load and traffic grows, the number of servers may increase, but the ways in which they are organized generally doesn't.

We'll return to this topic in Chapter 8.

```
mysql> <b>SHOW MASTER STATUS \G</b>
*************************** 1. row
***************************

   File: binary-log.004

   Position: 635904327

   Binlog_do_db: Binlog_ignore_db:

1 row in set (0.00 sec)

mysql> <b>SHOW MASTER LOGS;</b> +----------
------+

| Log_name |

+----------------+

| binary-log.001 |

| binary-log.002 |

| binary-log.003 |

| binary-log.004 |
+---------------+

4 rows in set (0.02 sec)

mysql> <b>SHOW SLAVE STATUS \G</b>
*************************** 1. row
***************************

  Master_Host: master.example.com Master_User:
repl Master_Port: 3306

  Connect_retry: 15

  Master_Log_File: binary-log.004

  Read_Master_Log_Pos: 635904807

  Relay_Log_File: relay-log.004

  Relay_Log_Pos: 846096118

Relay_Master_Log_File: binary-log.004

  Slave_IO_Running: Yes Slave_SQL_Running: Yes
Replicate_do_db: Replicate_ignore_db: Last_errno: 0

  Last_error: Skip_counter: 0
```

```
   Exec_master_log_pos: 635904807

   Relay_log_space: 846096122

1 row in set (0.00 sec)

mysql> <b>SHOW MASTER LOGS;</b> +----------
------+

| Log_name |

+----------------+

| binary-log.001 |

| binary-log.002 |

| binary-log.003 |

| binary-log.004 |

+----------------+

4 rows in set (0.02 sec)

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

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

$ <b>mysqlbinlog log-bin.001</b> ...

# at 683

#021103 18:36:33 server id 1 log_pos 683 Query
thread_id=288 exec_time=0

error_code=0

SET TIMESTAMP=1036377393;

insert into test1 values (8);

$ <b>mysql log-bin.001</b> ...

# at 683

#021103 18:36:33 server id 1 log_pos 683 Query
thread_id=288 exec_time=0
```

```
error_code=0

SET TIMESTAMP=1036377393;

insert into test1 values (8);

$ <b>mysqlbinlog -h slave3.example.com -u root -p -
o 35532 log-bin.032</b>

#!/usr/bin/perl -w

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

use strict;

use DBIx::DWIW;

my $conn = DBIx::DWIW->Connect(

  DB => "mysql", User => "root", Pass =>
"password", Host => "localhost", ) or exit;

my $info = $conn->Hash("SHOW SLAVE
STATUS"); if (exists $info->{Slave_SQL_Running}
and $info->{Slave_SQL_Running} eq 'No') {
  warn "slave SQL thread has stopped\n"; }

elsif (exists $info->{Slave_IO_Running} and $info->
{Slave_IO_Running} eq 'No') {

  warn "slave IO thread has stopped\n"; }

elsif (exists $info->{Slave_Running} and $info->
{Slave_Running} eq 'No') {

  warn "slave has stopped\n"; }

SET SQL_SLAVE_SKIP_COUNTER=1
```

SLAVE START

SET SQL_SLAVE_SKIP_COUNTER=1

SLAVE START SQL_THREAD

SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1

SLAVE START SQL_THREAD

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

```
#

# fix mysql replication if it encounters a problem
$|=1; # unbuffer stdout

use strict;

use DBIx::DWIW;

my $host = shift || 'localhost'; my $conn =
DBIx::DWIW->Connect(

  DB => "mysql", User => "root", Pass =>
"pa55word", Host => $host, ) or die "Couldn't
connect to database!"; print "checking $host ... ";

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

my $start_cmd;

# 3.23

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

# 4.0.0 - 4.0.2

elsif ($version =~ /^4\.0\.[012]/ and $info->
{Slave_SQL_Running} eq 'No') {

  $fix_cmd = "SET SQL_SLAVE_SKIP_COUNTER
= 1"; $start_cmd = "SLAVE START
SQL_THREAD"; }

# 4.0.3 - 4.0.xx, 4.1.xx. Don't know what 5.0 will be
like.

elsif ($version =~ /^4\.[01]\./ and $info->
{Slave_SQL_Running} eq 'No') {

  $fix_cmd = "SET GLOBAL
SQL_SLAVE_SKIP_COUNTER = 1"; $start_cmd =
"SLAVE START SQL_THREAD"; }

# things are okay or unknown version?

else

{
  print "GOOD\n"; exit;

}

print "FIXING ... ";

$conn->Execute($fix_cmd);

$conn->Execute($start_cmd);

print "DONE\n";

exit;

#!/usr/bin/perl -w

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

use strict;

use DBIx::DWIW;

my $MIN_LOGS = 4; ## keep main log plus three old binary logs around my $conn = DBIx::DWIW->Connect(
   DB => "mysql", User => "root", Pass =>
"password", Host => 'localhost', );

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

## if so, figure out what the last one we want to keep
is, then purge ## the rest

my $last_log = $logs[-$MIN_LOGS]; print "last log
is $last_log\n" unless $ENV{CRON}; $conn-
>Execute("PURGE MASTER LOGS TO
'$last_log'"); exit;

#!/usr/bin/perl -w

use strict;

use DBIx::DWIW;

$|=1; # unbuffer stdout

my $db_user = 'root';

my $db_pass = 'password';
my $db_name = 'test';

my $master = 'master.example.com'; my @slaves =
qw(

   slave1.example.com slave2.example.com
slave3.example.com );

my %master_count;

for my $server ($master)

{

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

   my $count = $conn->Scalar("SELECT COUNT(*)
FROM '$table'"); $master_count{$table} = $count; }

   print "OK\n"; }

for my $server (@slaves)
```

```
{

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

  my $count = $conn->Scalar("SELECT COUNT(*)
FROM '$table'"); if ($count !=
$master_count{$table}) {

  print "MISMATCH (got $count on $table,
expecting $master_count{$table}\n"; }

  }

  print "OK\n"; }

exit;

CREATE TABLE Heartbeat

(

  unix_time INTEGER NOT NULL, db_time
TIMESTAMP NOT NULL, INDEX
time_idx(unix_time) )

#!/usr/bin/perl -w

use strict;

use DBIx::DWIW;

my $conn = DBIx::DWIW->Connect(

  DB => "MySQL_Admin", User => "root", Pass =>
"password", Host => 'localhost', ) or die;

my $unix_time = time( );

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

exit;

#!/usr/bin/perl -w

use strict;

use DBIx::DWIW;
```

```
my $conn = DBIx::DWIW->Connect(

  DB => "MySQL_Admin", User => "root", Pass =>
"password", Host => 'localhost', ) or die;

my $sql = "SELECT unix_time, db_time FROM
Heartbeat ORDER BY unix_time DESC LIMIT 1";
my $info = $conn->Hash($sql); my $time = $info->
{unix_time}; my $delay = time( ) - $time;

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

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

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

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

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

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

You can't simply shut down the slave, edit the *my.cnf* file, and start it back up. MySQL always uses the *master.info* file if it exists, despite the settings contained in the *my.cnf* file.[8]

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

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

```
CREATE TABLE orders (

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

mysql> <b>insert into orders values (1, NULL,
'testing');</b> Query OK, 1 row affected (0.01 sec)
mysql> <b>insert into orders values (1, NULL,
'testing');</b> Query OK, 1 row affected (0.00 sec)
mysql> <b>insert into orders values (2, NULL,
'testing');</b> Query OK, 1 row affected (0.00 sec)
mysql> <b>select * from orders;</b> +-----------+----
-------+---------+

| server_id | record_id | stuff |

+-----------+-----------+---------+

| 1 | 1 | testing |

| 1 | 2 | testing |

| 2 | 1 | testing |
+-----------+-----------+---------+

3 rows in set (0.03 sec)

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

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

mysql> <b>show variables like 'server_id';</b> +----
-----------+-------+

| Variable_name | Value |

+---------------+-------+

| server_id | 102 |

+---------------+-------+

1 row in set (0.00 sec)
```

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

7.7.3.2 Partitioned auto-increment fields

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

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

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

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

It makes sense to break compatibility with existing MySQL versions (which use 32-bit server IDs) and reduce the size of the server ID to 8 or 16 bits. After all, with even 8 bits available, you can have up to 255 unique servers in a single replication setup; with 16 bits, that jumps to 65,535. It's unlikely anyone will have that many servers in a single replication setup. [9]

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

Chapter 8. Load Balancing and High Availability

After you've set up replication and have a number of MySQL slaves available to handle your needs, the next problem you're likely to face is how to route the traffic.

For the most part, the problem is quite similar to traditional HTTP

load balancing. But since the MySQL protocol isn't HTTP, there are some important differences that emerge when you get into the nitty-gritty of load balancing MySQL.

The material in this chapter assumes that your MySQL servers are on different machines from your application servers. If you've set up a local MySQL slave on each of your web or application servers, there's no need to worry about MySQL load balancing. Instead, you need a load-balancing solution for the web or application server.

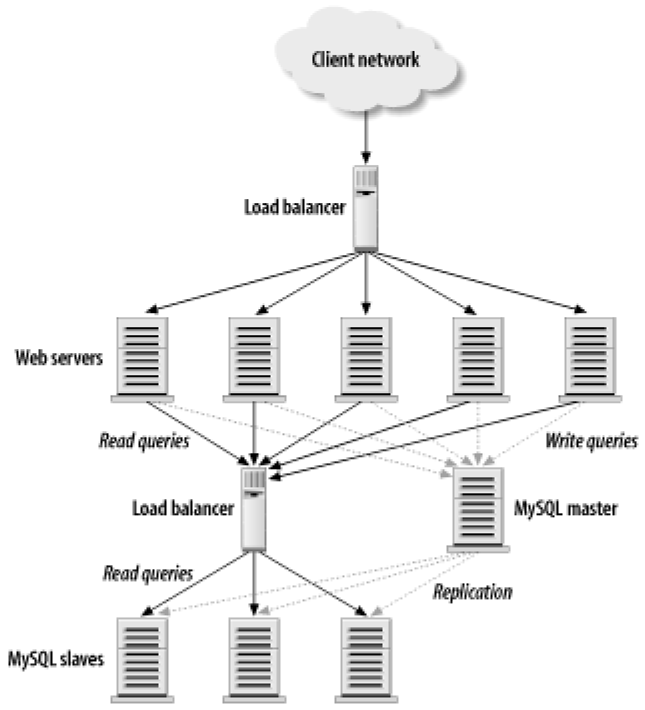
We'll start with a quick overview of load balancing from both a network and application perspective, and we'll discuss how load-balancing benefits MySQL

deployments. Then we move to some of the issues specific to load balancing MySQL in various configurations, notably health checks and balancing algorithms.

In the limited scope of this book, there's no way to cover all issues surrounding load balancers and high availability of your systems. For more information on the topic we suggest Tony Bourke's *Server Load Balancing*, also published by O'Reilly.

8.1 Load Balancing Basics

Figure 8-1. Typical load-balancing architecture for a read-intensive web site



The basic

idea behind load balancing is quite simple. You have a farm or cluster of two or more servers, and you'd like them to share the workload as evenly as possible. In addition to the backend servers, a load balancer (often a specialized piece of hardware) handles the work of routing incoming connections to the least busy of the available servers. Figure 8-1

shows a typical load-balancing implementation for a large web site.

Note that one load balancer is used for HTTP traffic and another for MySQL traffic.

There are four common goals or objectives in load balancing:

Scalability

In an ideal configuration, increasing capacity is as simple as adding more servers to the farm. By doing this properly, you can achieve linear scalability (for read-intensive applications) with relatively inexpensive equipment and guarantee good performance for your clients. Of course, not all applications scale this way, and those that do may require a more complex setup. We discuss those later in this chapter.

Efficiency

Load balancing helps to use server resources more efficiently because you get a fair amount of control over how requests are routed. This is particularly important when your cluster is composed of machines that aren't equally powerful. You can ensure that the less powerful machines aren't asked to do more than their fair share of the work.

Availability

With a cluster of MySQL slaves in place, the loss of any one server doesn't need to affect clients. They all have

identical copies of the data, so the remaining servers can shoulder the increased load. This level of redundancy is similar to using RAID

1 across multiple hard disks.

Transparency

Transparency means that clients don't need to know about the load-balancing setup. They shouldn't care how many machines are in your cluster or what their names are. As far as they're concerned, there's one big

virtual server that handles their requests.

Achieving all four goals is critical to providing the type of reliable service that many modern applications demand.

Software Solutions

If you're not interested in a hardware solution for load balancing, you might consider a language-specific load-balancing API. In the Java world, for example, Clustered JDBC (C-JDBC) provides a transparent layer on top of JDBC that handles load-balancing SELECT queries behind the scenes. For more information, see the C-JDBC web site (http://c-jdbc.objectweb.org/) Some Java application servers also support pools-of-pools specifically for clustering purposes.

Perl DBI users are encouraged to look at the `DBIx::DBCluster` module on their nearest CPAN mirror.

For a language-independent solution, the Open Source SQL Relay package (available from http://sqlrelay.sourceforge.net/) may be more appropriate. It supports most popular compiled and scripting languages, connection pooling, access controls, and more.

8.1.1 Differences Between MySQL and HTTP Load Balancing

If you're already familiar with HTTP load balancing, you may be tempted to run ahead and set up something similar for MySQL. After all, MySQL is just another TCP-based network service that happens to run on port 3306 rather that port 80, right? While that's true, there are some important differences between HTTP and MySQL's protocol as well as differences between the ways that database servers and web servers tend to be used.

8.1.1.1 Requests

To begin with, the connect-request-response cycle for MySQL is different. Most web servers and web application servers accept all connections, process a request, respond, and then disconnect almost immediately.[1]

They don't perform any fancy authentication. In fact, most don't even bother with a reverse lookup of an inbound IP address. In other words, the process of establishing the connection is very lightweight.

[1] With the increased adoption of HTTP/1.1, the disconnect may not occur right away, but the delay is still quite short in comparison to a typical MySQL server.

The actual request and response process is typically lightweight too.

In many cases, the request is for a static HTML file or an image. In that case, the web server simply reads the file from disk, responds to the client with the data, and logs the request. If the content is dynamic, the PHP, Java, or Perl code that generates it is likely to execute very quickly. The real bottlenecks tend to be the result of waiting on other backend services, such as MySQL or an LDAP server.

Sure, there can be poorly designed algorithms that cause the code to execute more slowly, but the bulk of web-based applications tend to have relatively thin business-logic layers. They also tend to push nearly all the data storage and retrieval off to MySQL.

Even when there are major differences from request to request, the differences tend to be in the amount of code executed. But that's exactly where you want the extra work to be done. The CPU is far and away the fastest part of the computer. Said another way, when you're dealing with HTTP, all requests are created equalat least compared to a database server.

As you saw early on in this book, the biggest bottleneck on a database server usually isn't CPU;

it's the disk. Disk I/O is an order of magnitude slower than the CPU, so even occasionally waiting for disk I/O can make a huge difference in performance. A query that uses an index that happens to be cached in memory may take 0.08 seconds to run, while a slightly different query that requires more disk I/O may take 3 seconds to complete.

On the database side, not all requests are created equal. Some are far more expensive than others, and the load balancer has no way of knowing which ones are expensive. That means that the load balancer may not be *balancing* the load as much as it is crudely *distributing* the load.

8.1.1.2 Partitioning

Another feature that's common in load-balanced web

application architectures is a caching system. When users first visit a web site, the web server may assign a session ID to the user, then pull quite a bit of information from a database to construct the user's preferences and profile. Since that can be an expensive operation to perform on every request, the application code caches the data locally on the web servereither on disk or in memoryand reuse it for subsequent visits until the cache expires.

To take advantage of the locally cached data, the load balancer is configured to inspect that user's session ID (visible either in the URL or in a site-wide cookie) and use it to decide which backend web server should handle the request. In this way, the load balancer works to send the same user to the same backend server, minimizing the number of times the user's profile must be looked up and cached. Of course, if the server goes offline, the load balancer will select an alternative server for the user.

Such a partitioning system eliminates the redundant caching that occurs if the load balancer sent each request to a random backend server each time. Rather than having an effective cache size equal to that of a single server, you end up with an effective cache size equal to the sum of all the backend servers.

MySQL's query cache can benefit from such a scheme.

If you've dedicated 512 MB of memory on each slave for its query cache, and you have 8 slaves, you can cache up to 4 GB

of different data across the cluster. Unfortunately, it's not that easy. MySQL's network

protocol doesn't have a way to expose any hints to the load balancer. There are no URL parameters or cookies in which to store a session ID.

A solution to this problem is to handle the partitioning of queries at the application level. You can split the 8 servers into 4 clusters of 2 servers each. Then you'd decide, in your

application, whether a given query should go to cluster 1, 2, 3, or 4. You'll see more of this shortly.

8.1.1.3 Connection pooling

Many applications use connection-pooling techniques; these techniques seem especially popular in the Java world and in PHP using persistent connections via *mysql_pconnect*( ). While connection pooling works rather well under normal conditions, it doesn't always scale well under load because it breaks one of the basic assumptions behind load balancing. With connection pooling, each client maintains a fixed or variable number of connections to one or more database servers. Rather than disconnecting and discarding the connection when a session is complete, the connection is placed back into a share pool so that it can be reused later.

Load balancing works best when clients connect and disconnect frequently. That gives the load balancer the best chance of spreading the load evenly; otherwise the transparency is lost. Imagine you have a group of 16 web servers and 4 MySQL servers. Your web site becomes very busy, and the MySQL servers begin to get bogged down, so you add 2 more servers to the cluster. But your application uses connection pooling, so the requests continue to go to the 4 overworked servers while the 2 new ones sit idle.

In effect, connection pooling (or persistent connections) work against load balancing. It's possible to compromise between the two if you have a connection-pooling system that allows the size of the pool to change as the demand increases and decreases.

Also, by setting timeouts relatively low (say, five minutes instead of six hours), you can still achieve a level of load balancing while taking advantage of persistent database connections.

You can also enforce this on the MySQL side by setting each server's `wait_timeout` to a

relatively low number. (This value tells MySQL how long a connection may remain idle before it is disconnected.) Doing so encourages sessions to be reestablished when needed, but the negative affects on the application side are minimal. Most MySQL APIs allow for automatic reconnection to the server any time you attempt to reuse a closed connection. If you make this change, consider also adjusting the `thread_cache` as described in Section 6.4.4 in Chapter 6.

We don't mean to paint connection pooling in a negative light. It certainly has its uses. Every worthwhile Java application server provides some form of connection pooling. As mentioned earlier, some provide their own load-balancing or clustering mechanisms as well. In such systems, connection pooling *combined with* load balancing is a fine solution because there's a single authority mediating the traffic to the database servers. In the PHP and *mysql_pconnect( )* world, there often is not.

Multi-Master Load Balancing

While the main focus of this chapter is on the load balancing of MySQL slaves, it's entirely possible to use a load balancer to spread the workload among several masters. Assuming you followed the advice in theSection 7.7.3

of Chapter 7, there's little difference in the setup required.

There are different reasons for using slaves and for using multiple masters. When you use multiple masters, you'll still get transparency and redundancy; however, scalability and efficiency don't really apply because in a multi-master setup, every master must still execute every write query sooner or later.

By having several masters behind a load balancer, you can better handle brief surges in traffic that can otherwise overwhelm a single server. During that time, each master fall farther and farther behind on the updates it receives from the other(s), but when the traffic returns to a normal level, the masters will catch up with each other and return to a state of equilibrium.

It's very important to realize that this model doesn't work well for all applications. In this type of setup, there is no "one true

source" of definitely correct information. That can cause subtle "bugs" in your application(s); for example, if you need to know if a record exists, you need to ask both servers.

8.2 Configuration Issues

To route the connection to a server, the load balancer must select a target server. To do this, it takes two pieces of information into account. First, it needs to know which servers are available. At any time, one or more of the backend servers can be offline (for maintenance, as the result of a crash, etc.). To keep track of the servers, the load balancer periodically checks each one's health.

Once the load balancer has a list of candidate servers, it must decide which should get the next connection. This process can take a number of factors into account, including past performance, load, client address, and so on. Let's look at both issues in more detail.

8.2.1 Health Checks

Load balancers need to perform a health check for each real server to ensure that it's still alive, well, and willing to accept new connections. When load-balancing a web server, this is often a trivial matter. The load balancer is configured to connect to TCP port 80 and request a *status file* such as */health.html*. If the server gets a 2xx response code back, it assumes the server is fine. If not, it may stop sending new requests to the server until it becomes healthy again.

A nice side benefit of asking for a specific file, rather than simply looking for any response on port 80, is that a server can be removed from the cluster without taking it offline: simply remove or rename the file.

Most load balancers provide a great deal of control over the parameters used when testing cluster hosts. Options may include the frequency of checks, the duration of check timeouts, and the number of unhealthy responses required to remove a server from the cluster.

See your load balancer's documentation for details.

8.2.1.1 Determining health

So what constitutes a good health check for MySQL? Unfortunately, there's no single answer to that question.

It depends on how sophisticated your load balancer is. Some load balancers can verify only that each server is responding on the necessary TCP port. They'll generally connect to TCP

port 3306 (or whichever port you're using) and assume the server is unhealthy if the connection is refused or if it has to wait too long for a response.

Some load balancers are more flexible. They might give you the option of scripting a complicated health check or of running the health check against a different port than normal. This provides a lot of flexibility and control. For example, you can run a web server (such as Apache) on the server and configure the load balancer to check a status file, just as you would for standard HTTP load balancing. You can exploit this indirect kind of check by making the status file a script (PHP, Perl, etc.) or Java servlet that performs arbitrarily complex logic to decide whether the server is really healthy.[2] The arbitrarily complex logic can be as simple as running a `SELECT 1`

query, or as complicated as parsing the output of `SHOW SLAVE`

`STATUS` to verify that the slave is reasonably up to date.

[2] Provided, of course, that the arbitrarily complex logic doesn't take arbitrarily long to execute. The load balancer won't wait

forever.

If your load balancer offers this degree of flexibility, we highly recommend taking advantage of it. By taking control over the decision-making process, you'll have a better idea of how your cluster will respond in various situations. And after testing, if you're not happy with the results, simply adjust the logic and try again.

What types of things might you check for? This goes back to the question we're trying to answer: what makes a healthy MySQL server, from the load balancer's point of view?

A good health check also depends on your application needs and what's most important. For example, on a nearly real-time dynamic web site like Yahoo! News, you might put more emphasis on replication. If a slave gets busy enough handling regular queries that it becomes sluggish and ends up more than 30 seconds behind on replication, your code can return an unhealthy status code.

The load balancer then removes the slave from the list of available servers until the health check passes again. Presumably the reduced demand on the server will allow it to quickly catch up and rejoin the cluster. (See the "Monitoring"

section in Chapter 7 for ideas about detecting slow slaves.)

Of course, the success of this algorithm depends on how smart your scripts are. What if the slow server doesn't get caught up? And what if the additional demand that the remaining servers must bear causes them to fall behind?

There's a very real chance that one by one,

they'll start deciding they too are unhealthy.

Before long, the problem cascades until you're left with a cluster of unhealthy servers sitting behind a load balancer that doesn't know where to send connections anymore.

At Yahoo! Finance, we've seen individual servers that try to be smart and end up creating even bigger problems because they didn't have the whole picture. Anticipating the problem mentioned in the previous paragraph, the code that performed health checks introduced yet another level of checking. Each server knew all the other members of the cluster. The health check included code to make sure that there were enough servers left. If a server determined that too many other servers were already down, it would elect to keep handling requests. After all, a slow site is better than no site at all.

But our implementation still wasn't smart enough; the servers still went down in a cascade. The reason turned out to be a simple race condition. The code performed a series of checks, but it did them in the wrong order. The code first checked to see that a sufficient number of other servers were healthy. It then went on to make sure MySQL wasn't too far behind on

replication. The problem was that several servers could be doing the health check at exactly the same time. If that happened, it was possible for all servers to believe that all other servers were healthy and proceed to declare themselves unhealthy.

There are numerous solutions to the problem. One is to add a simple sanity check. Each server can, after declaring itself unhealthy, check to make sure that the situation hasn't radically changed. Another option is to appoint a single server in each cluster as the authority for determining who is and isn't healthy. While it introduces a single point of failure (what if this server dies?), it means there are fewer chances for race conditions and similar problems.

To summarize, some load balancers provide you with a lot of flexibility and power. Be careful how you use it. If you elect to take control of the decision-making process (and add complexity to it), be sure that the code is well tested. Ask a few peers to review it for you. Consider what will happen in unusual situations.

8.2.1.2 Connection limits

In normal operations, the load balancer should distribute connections relatively evenly among your severs. If you have eight backend servers, any one of them will handle roughly one eighth of the connections at a given time. But what happens when several backend servers go down at the same time? Because the rest of the cluster must bear the load, you need to ensure that the se servers are configured to handle it.

The most important setting to check is `max_connections`.

In this circumstance, you'll find that if

`max_connections` is set too low, otherwise healthy MySQL servers start refusing connections even if they're powerful enough to handle the load. Many installations don't set the

`max_connections` option, so MySQL uses its built-in default of 100. Instead, set `max_connections` high enough that this problem can't happen. For example, if you find that each server typically handles 75 connections, a reasonable value for `max_connections` might be 150 or more. That way, even if half the backend servers failed, you're application won't fail to

connect.

8.2.2 Next-Connection Algorithms

Different load balancers implement different algorithms to decide which server should receive the next connection. Some call these scheduling algorithms. Each vendor has different terminology, but this list should provide an idea of what's available:

Random

> Each request is directed to a backend server selected at random from the pool of available servers.

Round-robin

> Requests are sent to servers in a repeating sequence: A, B, C, A, B, C, etc.

Least connections

The next connection goes to the server with the fewest active connections.

Fastest response

The server that has been handling requests the fastest receives the next connection. This tends to work well when the backend servers are a mix of fast and slow machines.

Hashed

The source IP address of the connection is hashed, thereby mapping it to one of the backend servers. Each time a connection request comes from the same client IP address, it is sent to the same backend server. The bindings change only when the number of machines in the cluster does.

Weighted

Several of the other algorithms can be weighted. For example, you may have four single-CPU machines and four dual-CPU machines. The dual-CPU machines are roughly twice as powerful as the single-CPU

machines, so you tell the load balancer to send them twice as many requestson average.

Which algorithm is right for MySQL? Again, it depends. There are several factors to consider and some pitfalls to avoid. One of the pitfalls is best illustrated with an example.

8.2.2.1 The consequences of poor algorithm choice

In September 2002, Yahoo! launched a one-week memorial site for those affected by the September 11, 2001 terrorist attacks. This site was described in Chapter 6. The *remember.yahoo.com* site was heavily promoted on the Yahoo! home page and elsewhere. The entire site was built by a small group of Yahoo! employees in the two weeks before the site's launch on September 9.

Needless to say, the site got a lot of traffic. So much, in fact, that Jeremy spent a couple of sleepless nights working to optimize the SQL queries and bring new MySQL servers online to handle the load. During that time the MySQL servers were running red hot. They weren't handling many queries per second (because they are poorly optimized) but they were either disk-bound, CPU-bound, or both. A server was slowest when it first came online because MySQL's key buffer hadn't

yet been populated, and the operating system's disk cache didn't have any of the relevant disk blocks cached. They needed several minutes to warm up before taking their full query load.

The situation was made worse by the fact that the load balancer hadn't been configured with this in mind, and nobody realized it until very late in the process. When a server was reconfigured and brought back online, it was immediately pounded with 30-50 new queries. The machine became completely saturated and needed several minutes to recover. During the recovery time, it was nearly unresponsive, with the CPU at 100%, a load average over 25, and the disk nearly maxed out.

After quite a bit of theorizing and poking around, someone thought to question the load-balancer configuration. It turned out that it was set on a least-connections scheduling algorithm. That clearly explained why a new machine was bombarded with new connections and rendered useless for several minutes. Once the load balancer was switched to a random scheduling algorithm, it became much easier to bring down a slave, adjust the configuration, and put it back online without becoming completely overwhelmed.

The moral of the story is that the connection algorithm you select may come back to bite you when you least expect it (and can least afford it). Consider how your algorithm will work in day-to-day operations as well as when you're under an unusually high load or have a higher than normal number of backend servers offline for some reason.

We can't recommend the right configuration for your needs. You need to think about what will work best for your hardware, network, and applications. Furthermore, your algorithm choices are limited by the load balancing hardware or software you're using. When in doubt, test.

8.3 Cluster Partitioning

As noted earlier, Figure 8-1 is a common setup for many web sites. While that architecture provides a good starting point, the time may come when you want to squeeze more performance out of your replication setup. Partitioning is often the next evolutionary step as the system grows. In this section, we'll look at several

related partitioning schemes that can be applied to most load-balanced MySQL clusters.

8.3.1 Role-Based Partitioning

Many applications using a MySQL

backend do so in different roles. Let's consider a

large community web site for which users register and then exchange messages and participate in discussions online. From the data storage angle, several features must be implemented for the site to function.

For example, the system must store and retrieve records for individual users (their profiles) as well as messages and message-related metadata.

At some point, you decide to add a search capability to the site.

Over the past year, you've accumulated a ton of

interesting data, and your users want to search it. So you add full-text indexes to the content and offer some basic search facilities. What you soon realize is that the search queries behave quite a bit differently from most of the other queries you run.

They're really a whole new class of queries. Rather

than retrieving a very small amount of data in a very specific way (fetching a message based on its ID or looking up a user based on username), search queries are more intensive; they take more CPU time to execute. And the full-text indexes are quite a bit larger than your typical MyISAM indexes.

In a situation like this, it may make sense to split up the responsibility for the various classes of queries

you're executing. Users often expect a search to

take a second or two to execute, but pulling up a post or a user page should always happen instantly. To keep the longer-running search queries from interfering with the "must be

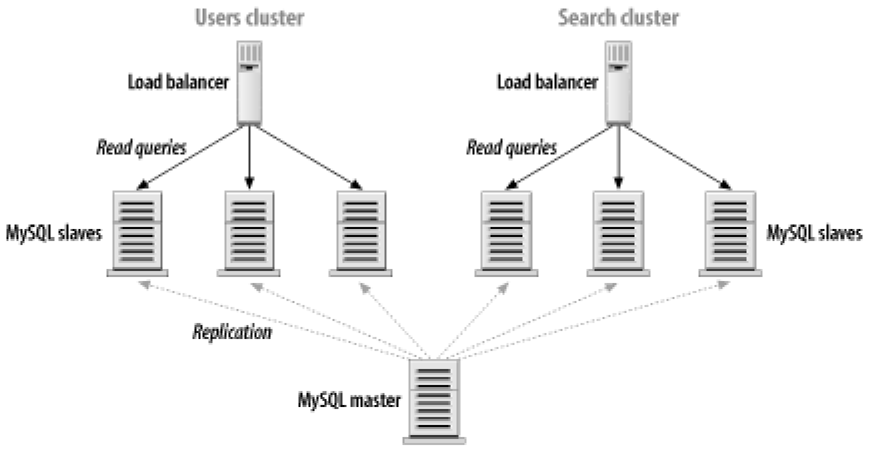
fast" queries, you can break the slaves into logical

subgroups. They'll all still be fed by the same

master (for now), but they will be serving in more narrowly focused roles.

Figure 8-2 shows a simple example of this with the top half of the diagram omitted. There need not be two physically different load balancers involved. Instead, think of those as logical boxes rather than physical. Most load-balancing hardware can handle dozens of backend clusters simultaneously.

Figure 8-2. Partitioning based on role



With this separation in place, it's much easier to match the hardware to the task at hand. Queries sent to the user cluster are likely to be I/O bound rather taxing the CPU.

They're mainly fetching a few random rows off disk

over and over. So maybe it makes sense to spend less money on the CPUs and invest a bit more in the disks and memory (for caching).

Perhaps RAID 0 is a good choice on these machines.

The search cluster, on the other hand, spends far more CPU time looking through the full-text indexes to match search terms and ranking results based on their score. The machines in this group probably need faster (or dual) CPUs and a fair amount of memory.

This architecture is versatile enough to handle workload splitting for a variety of applications. Anytime you notice an imbalance among the types of queries, consider whether it might be worthwhile to split your large cluster into a cluster made up of smaller groups based on a division of labor.

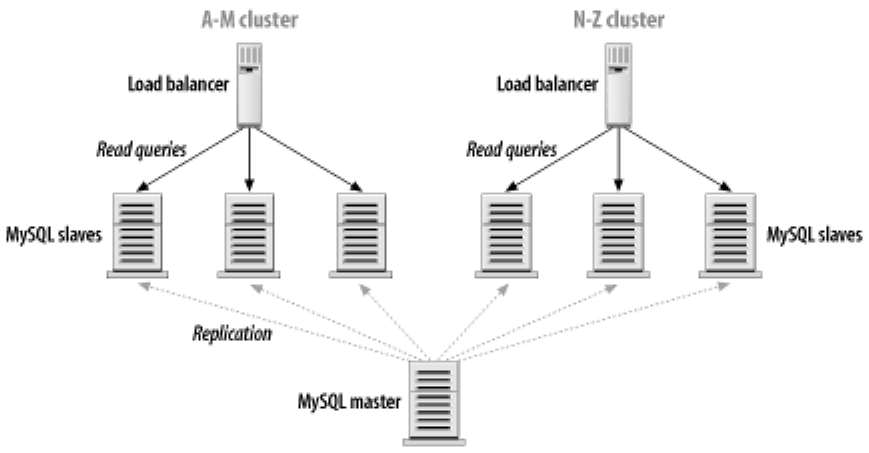
8.3.2 Data-Based Partitioning

Some high-volume applications have surprisingly little variety in the types of queries they use.

Partitioning across roles isn't effective in these

cases, so the alternative is to partition the data itself and put a bit of additional logic into the application code. Figure 8-3 illustrates this.

Figure 8-3. Partitioning based on data



In an application that deals with fetching user data from MySQL, a simple partitioning scheme is to use the first character of the username. Those beginning with letters A-M reside in the first partition. The other partition handles the second half of the alphabet. The additional application logic is simply a matter of checking the username before deciding which database connection to use when fetching the data.

The choice of splitting based on an alphabetic range is purely arbitrary. You can just as easily use a numeric representation of each username, sending all users with even numbers to one cluster and all odd numbers to the other. Volumes have been written on efficient and uniform hashing functions that can be used to group arbitrarily large volumes of data into a fixed number of buckets. Our goal isn't to recommend a particular method but to

suggest that you look at the wealth of existing information and techniques before inventing something of your own.

8.3.3 Filtering and Multicluster Partitioning

Assuming that the majority of activity is read-only (that is, on the slaves), the previous partitioning solutions scale well as the demand on a high-volume application increases. But what happens when a bottleneck develops on the master? The obvious solution is to upgrade the master. If it is CPU-bound, add more CPU power. If it's I/O bound, add faster disks and more of them.

There's a flaw in that logic, however. If the master is having trouble keeping up with the load, the slaves will be under at least as much stress. Remember that MySQL's

replication is query based. The volume of write queries handled by the slaves is usually identical to that handled by the master. If the master can no longer keep up, odds are that the slaves are struggling just as much.

8.3.3.1 Filtering

An easy solution to the problem is filtering. As described in Chapter 7, MySQL provides the ability to filter the replication stream selectively on both the master and the slave. The problem is that you can filter based only on database or table names.

Filtering is therefore not an option if you use data-based partitioning. MySQL has no facility to filter based on the queries themselves, only the names of the databases and tables involved.

Filtering may work well in a role-based partitioning setup in which the various slave clusters don't need full copies of

the master's data (for instance, where a search cluster needs two particular tables, and the user cluster needs the other four). If you use role-based partitioning,

it's probably worthwhile to set up each cluster to

replicate only the tables or databases the cluster needs to do its job. The filtering must be on the slaves themselves, as opposed to the master, so the slaves' IO thread will still copy

all the master's write queries. However, the SQL

thread will read right past queries the slaves

aren't interested in (those that are filtered out).

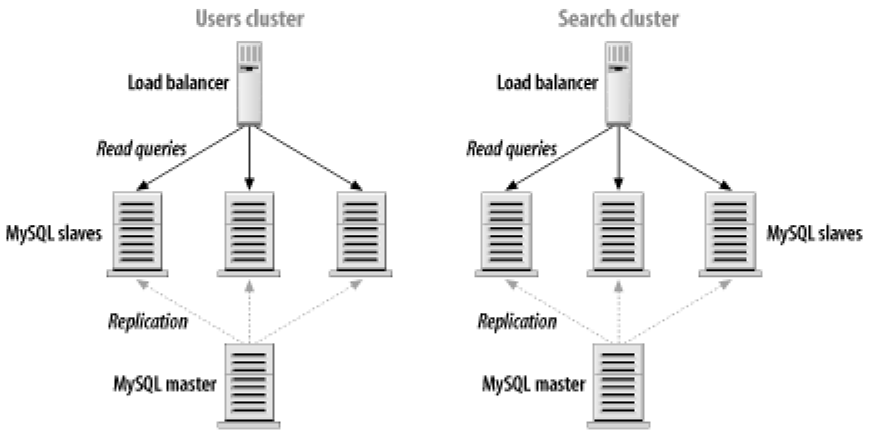
8.3.3.2 Separate clusters

Aside from Moore's Law, the only real solution to scaling the

write side with this model is to use truly separate clusters. By going from a single master with many slaves to several independent masters with their own slaves, you eliminate the bottlenecks associated with a higher volume of write activity, and you can get away with using less expensive hardware.

Figure 8-4 illustrates this logical progression. As before, there are two groups of slaves, one for search and one for user lookups, but this time each group is served by its own master.

Figure 8-4. Multicluster partitioning



8.4 High Availability

So far we've concerned ourselves with the slaves. Using a proper heartbeat setup and load balancer, you can achieve a high degree of availability and transparency for MySQL-based applications. In its current state, MySQL doesn't offer a lot in the way of high

availability support on the master, but that doesn't mean all hope is lost.

In this section, we'll look at several high-availability solutions (both commercial and free). Each of the options considered has pros and cons, which we've

done our best to document.

NDB Cluster

As we were putting the finishing touches on this book, MySQL AB was completing the initial integration work on the newest storage engine: NDB. In 2003, MySQL AB acquired Alzato, a company started by Ericsson in 2000. The company developed NDB Cluster, a clustered database system designed for both high availability and scalability.

When the integration is complete, MySQL's NDB storage engine will provide an interface to a backend NDB cluster.

For the first time, MySQL will have built-in clustering with automatic failover capabilities. See the MySQL web site and manual for more details on the NDB technology.

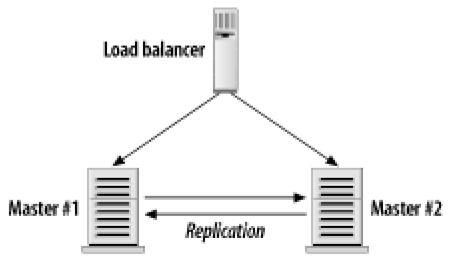
8.4.1 Dual-Master Replication

We looked at dual-master replication back in Chapter 7.

While it doesn't help in scaling an application

(both servers must handle the full write load), you can achieve much improved availability and transparency by putting a load balancer in the mix. Figure 8-5 illustrates this arrangement.

Figure 8-5. Dual-master replication for high availability



Aside from the downsides mentioned in Chapter 7 (mostly a lack of conflict resolution), there isn't

a lot that can go wrong with this setup. The worst problem is the potential for data loss, but that's really no

different from master/slave replication. After a query writes a record to *master 1*, MySQL records the query in the binary log, and the other master has a chance to read it. If *master 1* happens to crash between the time that the record is written and when the binary log is updated, however, the other master (and any slaves) will never know about the query. As far as *master 2* is concerned, the query never happened. The solution would be for MySQL to provide synchronous replication with two-phase commit, but it doesn't.

On the plus side, this solution is relatively easy to set up and understand. If you already know how to configure replication and have a working load balancer set up with good health checks, dual-master replication isn't much extra work. If you need to

perform maintenance on the masters, you can simply take *master 1* offline, do the work, bring it back online, and repeat the process on the other as soon as the first has caught up. Of course, it's best to do this

gracefully. Set the health check to fail, and wait until clients are no longer accessing the master before shutting it down. Otherwise you risk interrupting in-progress transactions.

If your load balancer is sophisticated enough, you can virtually eliminate the problem of conflict resolution. Here's how it works: rather than having both masters active, configure the load balancer so that *master 1* is active, and *master 2* is on standby. Only when *master 1* goes down should the load balancer send any traffic to *master 2*. Most load balancers provide a mechanism for doing this.

However, a wrinkle occurs when *master 1* comes back online. What should the load balancer do? If it begins sending connections to *master 1* again, you'll have a situation in which writes could be

occurring to both masters at the same time. That's a recipe for conflict. Remember, MySQL connections can be long-running, so the load balancer can't assume that clients will suddenly disconnect from *master 2*. The load balancer needs to be configured so that the notion of the "live master" changes only when the

current live master goes down.

8.4.2 Shared Storage with Standby

By increasing the cost and complexity of your infrastructure, you can eliminate the problem of lost updates described previously. Instead of two servers with their own copies of the data using replication to stay in sync, you can configure the active and standby masters to use shared storage.[3] It's very important to realize that the standby master shouldn't mount the filesystem or

start MySQL until the first is offline.

[3] The exact type of shared storage isn't terribly important. You see greater

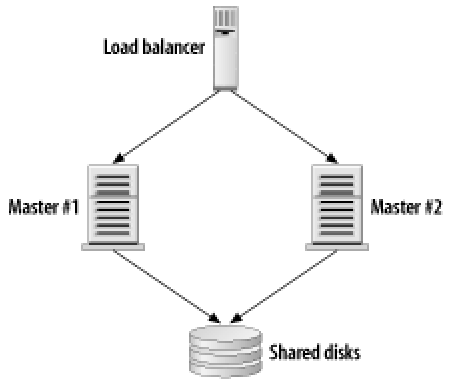
performance from directly attached systems than network attached storage, however, due mainly to the reduced latency.

Figure 8-6 shows one implementation of shared storage. It's worth pointing out that a load balancer isn't strictly necessary in this scenario.

All you really need is an agent running on each node to monitor the other. If the agent running on *master 2* finds that *master 1* is unavailable, it takes over *master 1*'s IP address and starts up MySQL with an identical configuration (same data directory, log filenames, etc.). If the configuration is truly identical, starting up MySQL on *master 2* is logically no different from fixing *master 1* and bringing MySQL up there. However, in reality there is an important difference: time. *Master 2* is already booted and ready to go. Starting up MySQL

takes a matter of seconds. The only delay is imposed by consistency checks on the data. Shared storage means the possibility of share corruption if you're not using InnoDB or BDB tables.

Figure 8-6. A live master and a warm standby master using shared storage



Writing such an agent is a tricky undertaking. We don't recommend you try it unless you have a lot of time available for testing all the possible edge cases you're likely to encounter with flaky network

equipment. Instead, spend some time looking at existing tools. There are numerous open source projects that can be adapted to do this for MySQL. The best candidate is

keepalived (http://keepalived.sourceforge.net/), a keep-alive health check facility written to work in conjunction with LVS. There are also two commercial solutions on the market today, described in the next section.

8.4.3 Commercial Solutions

As of this writing, there are two commercial products worthy of consideration for high availability.

Each takes a completely different approach to solving the problem, so different sites may find one or the other suitable, or neither. Keep an eye on this market: we expect to see a lot of new development in this area in the next year or so.

8.4.3.1 Veritas cluster a gent

Veritas has a well established reputation for providing the technology necessary to build many sorts of clusters. Their MySQL offering builds on the shared storage with standby model we just looked at.

The cluster agent runs on both the active and standby nodes, monitoring the health of the primary master. When the agent detects a problem on the master, it brings the standby instance online and takes over the primary master's functionality.

8.4.3.2 EMIC Networks

EMIC Networks provides a full-blown clustering solution for MySQL. By combining a number of relatively inexpensive servers running EMIC's version of MySQL, you can create incredibly robust MySQL clusters without needing to worry about the single point of failure most other architectures have.

Chapter 9. Backup and Recovery

Ask your favorite system administrator what the least favorite part of her job is and there's a good chance she'll mutter, "backups," with a sullen look on her face. Running backups ranks right up there with a visit to the dentist on most people's list of least fun things to do.[1]

[1] If Dr. Huntley ever reads this, Jeremy hopes he doesn't take it personally.

If you already rely on standard backup software to handle your MySQL servers, you probably have a false sense of security about backups. There aren't many popular backup tools that know how to back up MySQL properly, so that there is no corruption, half-committed transactions, or other assorted problems.

In this chapter we'll begin by considering why you need backups in the first place. Then we'll examine the issues that arise when trying to back up a running database server, including a look at why most backup software isn't well suited to MySQL backups. That leads to a discussion of the various backup-related tools for MySQL and how you can put them to use. Finally, we'll consider what's involved in creating a custom backup script.

Most of the how-to material is in the second half of the chapter. Much of the initial discussion revolves around understanding your backup options and how to go about selecting the right one.

9.1 Why Backups?

Strangely, some people never stop to consider *why* they need to back up their servers. The data is important, so we just assume that backing it up is equally important. That's good, because backups *are* important and do need to be done. But by understanding the various ways in which backups may be used, we gain some perspective on the utility of various backup strategies.

9.1.1 Disaster Recovery

Disaster recovery is the most popular motivation for running backups, but in reality it is often not as relevant as some of the other reasons we'll look at.

What is a disaster? For our purposes, a disaster is any event that causes significant portions of the data to be corrupted or unavailable. Some examples of disasters include the following:

- Hardware failure
- Software failure
- Accidental erasure of data[2]

  [2] The day after writing this section, Jeremy received a late phone call from a coworker who had accidentally mistyped the `WHERE` clause in a `DELETE` query. Luckily there was a good backup on hand.

- Stolen server
- Physically destroyed server

Any of these disasters can occur at any time. The odds of any one of them occurring are pretty low, but none of them are impossible. Having a known good copy of your data on hand will greatly minimize the pain of having to recover. It's a form of insuranceand cheap insurance at that.

Some of these disasters might be the result of a natural disaster (tornado, earthquake, mudslide, etc.). Unlike a simple disk failure, nature's catastrophes have a habit of physically damaging and even destroying entire buildings. To be truly safe, you need to have off-site backups. Something as simple as taking the tapes home with you every other week or sending a set to a remote office may prove to be invaluable if nature strikes.

9.1.2 Auditing

There are times when you'd like to be able to go back in time and see what a database, table, or even a single record looked like. Having older backups available makes this relatively easy to do. Just pull out the correct files, load them onto a test server, and run some queries. Depending on the type of data you store, there may even be legal reasons why you need to keep old copies of your data around.

Why else might you need the ability to go back in time and examine older copies of your data? You might have to:

- Look for data corruption
- Decide how to fix a newly discovered bug retroactively
- Compute the rate of growth for your databases
- Provide evidence for a lawsuit or investigation

Of course, there are countless other situations in which older data can be invaluable. The trouble is, you may not realize that until it is too late.

9.1.3 Testing

It's usually a good idea to test changes to an application before putting them into production. To do that, you'll probably have a separate database server you can load data onto to run various tests. Over the course of development, you may need to wipe the data clean and reload it several times.

If you have a recent backup of your production server available, setting up a test server can be downright trivial. Just shut down MySQL, restore the data, start MySQL, and begin testing.

9.2 Considerations and Tradeoffs

We considered calling this section "Things You Really Need To Think About" because backing up a running database is more complex than it may first appear to be. This isn't because backups are inherently difficult; it's because MySQL is a bit more complex that you might think.

When it comes to actually performing the backups, you can script the process yourself, use one of the prebuilt tools, or both. It all depends on your needs. In this section, we'll examine the major decisions you'll need to make and how they influence the backup techniques you can use. Then in the next section we'll look at the most popular tools.

9.2.1 Dump or Raw Backup?

One of the first decisions to make is the format of the backups you'd like to create. The result of a database *dump* is one or more files that contain the SQL statements (mostly `INSERT` and `CREATE TABLE`) necessary to re-create the data. Dumps are produced using *mysqldump*, described in more detail in Section 9.3, later in this chapter. You can perform dumps over the network so that your backups are created on a host other than your database server. It's possible to produce dumps of any MySQL table type.

Having the contents of the tables as SQL files provides a lot of flexibility. If you simply need to look for a few records, you can load the file in your favorite editor or use a tool such as *grep* or *less* to locate the data. The dumped data is quite readable.

Restoring a dump is easy. Because the dump file contains all the necessary information to re-create the table, you simply need to feed that file back into the *mysql* command-line tool. And if you need to restore only some of the records, you can directly edit the file directly or write a script to prefilter out the records you don't need. Raw backups don't provide this flexibility. You can't easily filter out records from a table when using a raw backup; you can operate only on whole tables.

There are some downsides to using dumps. A dump file consumes far more disk space than the table or database it represents. Not only are there a lot of `INSERT` statements in the file, all numeric data (which MySQL stores quite efficiently) becomes ASCII, using quite a bit more space. Dumps are more CPU-intensive to produce, so they'll take longer than other methods. Dump files compress rather well using tools such as *gzip* or *bzip2*. Also, reloading a dump requires that MySQL spend considerable CPU time to rebuild all the indexes.

Because there's often a fair amount of unused space and overhead in InnoDB's data files, you'll find that InnoDB tables often take far less space that you might expect when backed up.

While dumps have a lot of advantages, the extra space, time, and CPU power they require are often not worth expendingespecially as your databases get larger and larger. It's more efficient to use a *raw backup* technique rather than using dumps. A raw backup is a direct copy of MySQL's data files as they exist on disk. Because the records aren't converted from their native format to ASCII, raw backups are much faster and more efficient than dumps. For ISAM and MyISAM tables, this means copying the data, index, and table definition files. For BDB and InnoDB tables, it also involves preserving the transaction logs and the data.

Both *mysqlhotcopy* and *mysqlsnapshot*, which we describe in some detail later, can be used to produce raw backups of ISAM and MyISAM tables. They do so by locking and flushing the tables before copying the underlying files. The tables may not be written to during the backup process. The InnoDB Hot Backup tool, also discussed later in this chapter, provides a raw backup of your

InnoDB data without the need for downtime or locking. There is no equivalent tool for BDB tables.

Raw backups are most often used to back up a live server. To get a consistent backup, ISAM and MyISAM tables need to be locked so that no changes can occur until the backup completes. InnoDB tables have no such restriction.

Restoring a raw backup is relatively easy. For ISAM and MyISAM tables, you simply put the data files in MySQL's data directory. Unless you're using InnoDB's multiple-tablespace support in Version 4.1 or newer, InnoDB tables can't be restored individually from a raw backup because they are stored in shared tablespace files rather than individually. Instead, you'll need to shut down MySQL and restore the tablespace files.

If you have the luxury of shutting down MySQL to perform backups, the backup and restore processes can be greatly simplified. In fact, that's the next decision to consider.

9.2.2 Online or Offline?

Being able to shut down MySQL during backups means not having to worry about consistency problems (discussed in the next section), locking out changes from live applications, or degrading server performance. A nonrunning MySQL instance can be backed up using standard backup software. There's no danger of files changing. If MySQL isn't running, the backup process will likely be faster too; it won't be competing with MySQL for I/O and CPU cycles.

If you're planning to shut down MySQL during backups, make sure that your backup software is configured to back up all of the MySQL-related data. Ideally, you'd back up the entire system, but there may be cases when that isn't feasible. Large MySQL installations often span several filesystems. The binaries may be in one place, config files in another, and the data files elsewhere. Having them on different backup schedules could leave you with a difficult problem if you need to restore just after a major upgrade. The config files may not match the data file locations, for example.

9.2.3 Table Types and Consistency

Maintaining consistency is one of the most tricky and often overlooked issues in database backups. You need to ensure that you're getting a consistent snapshot of your data. Doing so requires an understanding of the types of tables you need to back up and how MySQL handles them.

If you're using MyISAM tables, simply making copies of the various data files isn't sufficient. You must guarantee that all changes have been flushed to disk and that MySQL won't be making changes to any of the tables during the backup process. The obvious solution is to obtain a read lock on each table before it is backed up. That will prevent anyone from making changes to the table while still allowing them to read from it.

That technique works well for a single table, but in a relational database, tables are often related to each other. Records inserted into one table depend on those in another. If that's not accounted for, you can end up with an inconsistent backuprecords may exist in one table but have no counterparts in another. It all depends on the order in which the tables were copied and the likelihood that changes were made to one while the other was backed up.

So a good backup program needs to lock groups of related tables before they are copied. Rather than deal with that complexity, the popular solutions for MySQL give you the option of either locking all tables and keeping them locked until the backup is done, or locking and backing up tables one at a time.[3] If neither option appeals to you, there's a good chance that you need to script your own solution. See Section 9.4, later in this chapter, for details.

[3] Ideally, we'd have the option to unlock each table selectively after it is copied, but MySQL doesn't allow that yet.

9.2.4 Storage Requirements

The amount of space required to store backups must factor into the decision-making process. How much room does your backup media have? Tape, CD, DVD, and hard disks all have capacity limits, costs, and lifetimes.[4]

[4] But hard disks seem to be growing in capacity without bound. It shouldn't be long before you can buy a tera-byte hard disk.

After you've determined how much space you can afford and manage effectively, you need to consider how frequently you really need to perform backups. Do you need to back up all your data every day? Can you get by with backing up only your most active tables or databases daily and performing a full backup on the weekend? That will save a lot of space if much of your data changes infrequently.

When dealing with backups, it's a good idea to consider compression. If you're backing up to a tape drive with hardware compression, it's handled for you automatically. Otherwise, you can choose any compression scheme you'd like. Most dump files and raw backups compress rather well. However, if a lot of your data is already compressed (either compressed MyISAM tables or tables with `BLOB` fields that contain compressed data), there will be little benefit in further compression attempts.

If you have more than a few compressed MyISAM tables, not only should you avoid trying to compress them further, but you should also consider backing them up less frequently. Compressed MyISAM tables are read-only; by definition, they don't change often. You'd have to uncompress the table, make changes, and recompress it. That's rare.

The final issue to think about is retention. How long do you need to keep backups around? Rather than simply throwing out backups when you begin to run out of space, it's best to plan ahead. By taking into account the amount of data you must back up, the amount of space you need, and how long you want to keep data around, you won't run into surprises.

If you find yourself running out of space, consider staggering the backups that you do save. Rather than always deleting the oldest backups, you can use an alternative approach such as removing backups that fall on odd-numbered days. That would allow you to double the age of your oldest backup.

9.2.5 Replication

If you're using MySQL's replication features (described in Chapter 7), you can be a lot more flexible in your approach to backups. In fact, you may want to set up a slave just to simplify backups.

By performing backups on a slave, you eliminate the need ever to interrupt systems that may need to make changes on the master. In a 24 x 7 x 365 operation, this is an excellent way to ensure that you always have a copy of your data on another machine (this method is commonly used at Yahoo!). And since you can switch to the slave if the master dies, it significantly reduces the downtime when something does go wrong.

When backing up a slave, it's important always to save the replication files as well. That includes the *master.info* file, relay logs, relay index, and so on. Without them, you can't easily restore a slave that has suffered a failure. The files contain information about where the slave left off in the replication process. See Chapter 7 for more information.

```
$ <b>mysqldump -u root -pPassword -x --all-databases > dump.sql</b>
```

```
$ <b>mysqldump -u root -pPassword -x --databases db1 db2 db3 > dump.sql</b>
```

```
$ <b>mysqldump -u root -pPassword -x db1 table1 table2 table3 > dump.sql</b>
```

```
$ <b>mysqldump -h db.example.com -u root -pPassword -x --all-databases > dump.sql</b>
```

```
$ <b>mysql -u root -pPassword < dump.sql</b>
```

```
INSERT INTO mytable (col1, col2, col3)



VALUES (val1, val2, val3) (val1, val2, val3) ...
```

```
$ <b>mysqldump -u root -pPassword --all-databases --result-file=dump.sql</b>
```

```
$ <b>mysqlhotcopy -u root -p Password test /tmp</b>
$ <b>ls -l /tmp/test</b>

total 108

-rw-rw---- 1 mysql users 8550 May 3 12:02
archive.frm

-rw-rw---- 1 mysql users 25 May 3 12:02
archive.MYD

-rw-rw---- 1 mysql users 2048 May 23 12:58
archive.MYI

-rw-rw---- 1 mysql users 8924 Mar 4 21:52
contacts.frm

-rw-rw---- 1 mysql users 7500 Mar 5 21:11
contacts.MYD
```

```
-rw-rw---- 1 mysql users 5120 May 23 12:58
contacts.MYI

-rw-rw---- 1 mysql users 8550 May 3 12:02 dirty.frm

-rw-rw---- 1 mysql users 25 May 3 12:02 dirty.MYD



-rw-rw---- 1 mysql users 2048 May 23 12:58
dirty.MYI

-rwxr-xr-x 1 mysql users 8558 Feb 26 2001
maybe_bug.frm*



-rwxr-xr-x 1 mysql users 45 Feb 26 2001
maybe_bug.MYD*



-rwxr-xr-x 1 mysql users 2048 May 23 12:58
maybe_bug.MYI*
```

```
-rwxr-xr-x 1 mysql users 8715 Jan 15 2001
test_more_info.frm*



-rwxr-xr-x 1 mysql users 784 Jan 16 2001
test_more_info.MYD*



-rwxr-xr-x 1 mysql users 2048 May 23 12:58
test_more_info.MYI*

$ <b>mysqlhotcopy -u root -p Password --noindices
test /tmp</b>

$ <b>mysqlhotcopy -u root -p Password --
regexp=test /tmp</b>

$ <b>cp /tmp/test/test_more_info.* </b>datadir<span
class="docEmphBold">/test</span>

$ <b>cd </b>datadir<span
class="docEmphBold">/test</span>

$ <b>myisamchk -r test_more_info</b>
```

```
mysql> <b>REPAIR TABLE test_more_info</b>

$ <b>mysqlsnapshot -u root -p Password -s
/tmp/snap --split -n</b>

checking for binary logging... ok



backing up db database... done



backing up db jzawodn... done



backing up db mysql... done



backing up db nuke... done



backing up db phplib... done
backing up db prout... done


backing up db test... done


snapshot completed in /tmp/snap/

$ <b>cd </b>datadir/test


$ <b>tar -xvf /tmp/prout.tar</b>

$ <b>ibbackup /etc/my.cnf /etc/ibbackup.cnf</b>

$ <b>ibbackup --restore /etc/ibbackup.cnf</b>

# Now, shut down MySQL before the backup begins.


mysqladmin -u root -pPassword shutdown
```

```
# And start the backup

...

# Then bring MySQL back up

/usr/local/mysql/bin/mysqld_safe &
```

If you use a prepackaged backup system, you need to ensure that MySQL is down before it starts. If the backup software is run locally on the MySQL server, that's easy. Rather than running the software directly, create a small shell script or batch file that handles the stopping and starting of MySQL around the backup processmuch like the previous example.

In larger environments, it is common to run client/server backup software. The backup server contacts a daemon running on a remote server when it is time for the backup process to begin. That daemon (running on your MySQL server) then feeds data to the backup server over the network. It is also common in such environments to let the backup software control the exact starting time of the backup.

In a case like that, you may need to find an alternative approach for backing up MySQL, or you'll need to do some digging in the backup software's manual. There's a good chance that you can find a way to make the backup software start and stop MySQL when it needs to. If not, you may be able to use one of the other backup strategies. If you have sufficient disk space, you can perform the backup directly on the MySQL server and let your normal backup process back up those files.

9.3.5.1 Restoring

Once again, MySQL makes it easy to restore data.[6] Unless you're restoring the entire MySQL installation, you need to recover the files that make up the tables and databases you need to restore. Once you have them, copy them back into MySQL's data directory and start MySQL.

[6] Your backup software may not, but there's little we can do about that here.

9.3.6 Filesystem Snapshots

Taking a snapshot of MySQL's data is the fastest and least intrusive method of backing up an online server. While the implementation details vary, a snapshot is an online copy of your datausually stored on the same filesystem or volume. In fact, most systems use a copy-on-write scheme to minimize the free space required to take a snapshot.

MySQL itself provides no support for taking snapshots, but various free and commercial filesystems and storage solutions do. In the Linux world, LVM (the Linux volume manager) has snapshot capabilities. Veritas sells a filesystem product for most versions of Unix (and Linux) that can take snapshots. FreeBSD 5.x may offer snapshot capabilities too.

In the hardware space, Network Appliance's popular "filers" can be used to take filesystem snapshots. EMC has two ways of doing this: snapshots, which are just like the snapshots described above, and

BCVs (business continuance volumes). They are, in effect, additional mirrors of a volume that can be broken off and mounted on other systems. They require double the amount of storage and are therefore expensive.

Snapshots are best used with a more traditional backup solution. By itself, a snapshot doesn't do much to guard against hardware failures. Sure, you can use a snapshot to quickly restore an accidentally dropped table, but all the snapshots in the world won't help if the disk controller catches fire.

Be sure that you have sufficient space reserved on your volume for the number of snapshots you plan to keep online. Most snapshot-capable filesystems require that you reserve a minimum amount of disk space for snapshot data. If your server processes a lot of write queries, you can easily exceed the reserved space. Check your filesystem documentation for complete details.

Just as with the other approach to online backups, you must be careful to flush and obtain a read lock on all ISAM and MyISAM tables before initiating a snapshot. The easiest way to do this is to use

MySQL's `FLUSH TABLES WITH READ LOCK` command. It will hold the lock until you disconnect from MySQL or issue an `UNLOCK TABLES` command. We'll discuss this in the next section.

```
MAILTO=backup-admin@example.com 00 */12 * * * /usr/local/bin/mysnap.pl
```

```
00 0,12 * * * /usr/local/bin/mysnap.pl | mail backup-admin@example.com
```

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

```
#

# mysnap.pl - snapshot mysql and mail stats to
backup admins use strict;

use DBIx::DWIW;

$|=1; # unbuffer output

my $db_user = 'backup_user'; my $db_pass =
'backup_pass'; my $db_name = 'mysql';

my $db_host = 'localhost';

my $command = '/usr/local/bin/snapshot'; my $conn
= DBIx::DWIW->Connect(DB => $db_name, User
=> $db_user, Pass => $db_pass, Host => $db_host);
my @table_sizes;

# flush and lock all tables $conn->Execute("FLUSH
TABLES WITH READ LOCK"); # gather stats on
the tables my @db_list = $conn->FlatArray("SHOW
DATABASES"); for my $db (@db_list)

{
  $conn->Execute("USE $db") or die "$!"; my
@table_info = $conn->Hashes("SHOW TABLE
STATUS"); for my $table (@table_info) {

  my $name = $table->{Name}; my $size = $table->
{Data_length}; push @table_sizes, ["$db.$name",
$size]; }

}

# run the snapshot

system($command);

# unlock the tables

$conn->Execute("UNLOCK TABLES"); $conn-
>Disconnect;

# sort by size and print

for my $info (sort { $b->[1] cmp $a->[1] }
@table_sizes) {

  printf "%-10s %s\n", $info->[1], $info->[0]; }

exit;
```

```
_ _END_ _

mysql> <b>SHOW TABLE STATUS \G</b>
*************************** 1. row
***************************

  Name: journal Type: MyISAM

  Row_format: Dynamic Rows: 417

Avg_row_length: 553

  Data_length: 230848

Max_data_length: 4294967295

  Index_length: 5120

  Data_free: 0

Auto_increment: NULL

  Create_time: 2001-12-09 23:18:06

  Update_time: 2002-06-16 22:20:13

  Check_time: 2002-05-19 17:03:35
```

Create_options:

  Comment:

$ <b>mysnap.pl</b> 9300388448 Datascope.SymbolHistory 1458868716 Chart.SymbolHistory 773481608 logs.pfs

749644404 IDX.LinkLog

457454228 SEC.SEC_Filings

442951712 IDX.BusinessWireArticles 343099968 Datascope.Symbols 208388096 IDX.Headlines

...

As expected, the largest tables are listed firstregardless of which databases they reside in.

There are many ways *mysnap.pl* can be improved or enhanced. It could:

- Perform more error checking.
- Compare the current table sizes with those from the previous run.

- Notice whether a table has grown beyond a preset threshold.
- Ignore Heap tables, since they don't reside on disk.

None of those enhancements are particularly difficult. With even a basic grasp of Perl and a bit of time, you can transform that script to something custom-tailored for your needs.

Chapter 10. Security

Keeping MySQL secure is critical to maintaining the integrity and privacy of your data. Just as you have to protect Unix or Windows login accounts, you need to ensure that MySQL accounts have good passwords and only the privileges they need. Because MySQL is often used on a network, you also need to consider the security of the host that runs MySQL, who has access to it, and what someone could learn by sniffing traffic on your network.

In this chapter we'll look at how MySQL's permissions work and how you can keep

control of who has access to the data. We'll also consider some of the basic operating system and network security measures you can employ to keep the bad guys out of your databases.

Finally, we'll discuss encryption and running MySQL

in a highly restricted environment.

10.1 Account Basics

Consider first the example of a typical Unix login. You have a username and a password, along with, possibly, some other information such as the login owner's full name, telephone number, or other information. There is no distinction between the user *dredd* coming from *foo.example.com* and *dredd* coming from *bar.example.com.* To Unix, they are one and the same.

Each account in MySQL is composed of a username, password, and location (usually hostname, IP address, or wildcard). As we'll see, having a location associated with the username adds a bit of complexity to an otherwise simple system. The user *joe* who logs in from *joe.example.com* may or may not be the same as the *joe* who logs in from *sally.example.com*. From MySQL's point of view, they are completely different. They may even have different passwords and privileges.

Database-Specific Passwords

We indicated that users are stored as username/password/location. It's important to note that one qualifier not included is the database. For instance:

```
mysql> GRANT SELECT ON Foo.* to
'nobody'@'localhost' IDENTIFIED BY
'FooPass';

mysql> GRANT SELECT ON Bar.* to
'nobody'@'localhost' IDENTIFIED BY
'BarPass';
```

You might think, to look at that, that user *nobody* connects to `Foo` using *FooPass* as his password and to `Bar` using *BarPass* as his password. That's not the case. What actually happens is that *nobody* has his password changed in the `users` table to *BarPass*, and any connections to the `Bar` database using *FooPass* will fail to authenticate.

This is especially important because it means that if you want to limit access for an application to one database and not another, your codebase may have the password to "its" database encoded into it. If someone sees that source code, and you use the same MySQL user for some other application that accesses a different database, the person who sees one set of source code will now know how to gain access to the other database.

MySQL uses a series of *grant tables* to keep track of users and the various privileges they can have. The tables are ordinary MyISAM tables[1] that live in the `mysql` database. Storing the security information itself in MySQL makes a lot of sense. It allows you to use standard SQL queries to make any security changes. There are no additional configuration files for MySQL to process. But, this also means that if the server is improperly configured, any user could make security changes!

[1] And they must remain ordinary MyISAM tables. Don't change their type.

Over the lifetime of a typical database connection, MySQL may perform three different types of security checks:

Authentication

> Who are you? For each incoming connection, MySQL checks your username, the password you supplied, and the host from which you are connecting. Once it knows who you are, the information is used to determine your privileges.

Authorization

> What are you allowed to do? Shutting down the server, for example, requires that you have the shutdown privilege.

Access control

> What data are you allowed to see and/or manipulate? When you try to read or modify data, MySQL checks to see that you've been granted permission to see or change the columns you are selecting.

As you'll see, authorization and access control can be a bit difficult to distinguish in MySQL. Just remember that authorization applies to global privileges (discussed shortly), while access control applies to typical queries (`SELECT`, `UPDATE`, and so on).

10.1.1 Privileges

Access control is made up of several *privileges* that control how you may use and manipulate the various objects in MySQL: databases, tables, columns, and indexes. For any combination of objects, the privileges are all booleaneither you have them or you don't. These per-object privileges are named after the SQL queries you use to trigger their checks. For example, you need the select privilege on a table to `SELECT` data from it.

Here's the full list of per-object privileges:

- Select
- Insert
- Update
- Index
- Alter
- Create
- Grant
- References

Not all privileges apply to each type of object in MySQL. The insert privilege is checked for all of them, but the alter privilege applies only to databases and tables. That makes perfect sense, because you insert data into columns all the time, but there's no `ALTER`

COLUMN command in SQL. Table 10-1 lists which privileges apply to each type of object in MySQL.

Table 10-1. Access control privileges

Privilege	Databases	Tables	Columns
Select	✓	✓	✓
Insert	✓	✓	✓
Update	✓	✓	✓
Delete	✓	✓	
Index	✓	✓	
Alter	✓	✓	
Create	✓	✓	
Drop	✓	✓	
Grant	✓	✓	

Privilege	Databases	Tables	Columns
References	✓	✓	✓

While most of those privileges are rather straightforward, a few deserve some additional explanation:

Select

The select privilege is required for `SELECT` queries that access data stored in MySQL. No privilege is needed to perform simple math (`SELECT 2*5`), date/time conversions (`SELECT Unix_TIMESTAMP(NOW( ))`) and formatting, or various utility functions (`SELECT MD5('hello world')`).

Index

This single privilege allows you to create and drop indexes. Even though index changes are made via `ALTER TABLE` commands, the index privilege is what matters.

Grant

When using the `GRANT` command (described later), you may specify `WITH GRANT OPTION` to give the user the grant privilege on a table. This privilege allows the user to grant any rights you have granted him to other users. In other words, he can share his privileges with another user.

References

The references privilege controls whether or not you may reference a column in a given table as part of a foreign key constraint.

10.1.1.1 Global privileges

In addition to the per-object privileges, there is a group of privileges that are concerned with the functioning of MySQL itself and are applied server-wide. These are the authorization checks mentioned earlier:

Reload

The reload privilege is the least harmful of the server-wide privileges. It allows you to execute the various `FLUSH` commands, such as `FLUSH TABLES`, `FLUSH STATUS`, and so on.

Shutdown

This privilege allows you to shut down MySQL.

Process

The process privilege allows you to execute the `SHOW PROCESSLIST` and `KILL` commands. By watching the processlist in MySQL, you can capture raw SQL queries as they are being executedincluding the queries that set passwords.

File

This privilege controls whether you can execute a `LOAD DATA INFILE...` command. The danger in allowing this is that a user can use the command to read an arbitrary file into a table, as long as it is readable by the *mysqld* process.

Super

This privilege allows you to `KILL` any query on the server. Without it, you're limited to only those queries that belong to you.

Each server-wide privilege has far-reaching security implications, so be very cautious when granting any of them!

10.2 The Grant Tables

MySQL's grant tables are the heart of its security system. The information in these tables determines the privileges of every user and host that connects to MySQL. By correctly manipulating the records, you can give users exactly the permissions they need (and no more). Incorrectly manipulating them can open up your server to the possibility of abuse and damage.

Let's take a brief look at the five grant tables before really digging in. We've included them here in the order that MySQL consults them. You'll see why that becomes important in a minute.

user

The `user` table contains the global privileges and encrypted passwords. It is responsible for determining which hosts and users may connect to the server.

host

The `host` table assigns privileges on a per-host basis, regardless of the user. When deciding to accept or reject a connection, MySQL consults the `user` table as noted earlier. Though we list it as a grant table, the `host` is never modified through use of the `GRANT` or `REVOKE` commands. You can add and remove entries manually, however.

db

The `db` table sets database-level privileges.

tables_priv

The `tables_priv` table controls table-specific privileges.

columns_priv

Records in the `columns_priv` table specify a user's privileges for a single column of a single table in a particular database.

10.2.1 Privilege Checks

For each query issued, MySQL checks to make sure the user has the required privileges to perform the query. In doing so, it consults each of the tables in a specific order. Privileges set in one table may be overridden by a table checked later.

In other words, the privilege system works through inheritance. Privileges granted in the `user` table are passed down through all the other checks. If there are no matching records in any of the other tables, the original privileges set forth in the `user` table apply.

MySQL uses different criteria when checking each grant table. Records in the `host` table, for example, are matched based on the host from which the user has connected and the name of the database that the query will read from or write to. Records in the `db` table, on the other hand, match based on the host, database, and username. Table 10-2 summarizes the fields used for matching records in each of the grant tables.

Table 10-2. Fields used for matching grant table records

Table	Password	User	Host	Db	Table	Column
`user`	✓	✓	✓			
`host`			✓	✓		
`db`		✓	✓	✓		
`tables_priv`		✓	✓	✓	✓	
`columns_priv`		✓	✓	✓	✓	✓

Let's look at the schema for each table as well as the privileges each affects.

10.2.2 The user Table

MySQL's `user` table contains authentication information about users as well as their global privileges. It contains fields for the username, hostname, and password. The remainder of the fields represent each of the privileges, which are all off by default. As you'll see, many of the other tables also contain the `Host` and `User` fields as well as a subset of the privilege fields that are present in the `user` table, but only the `user` table contains passwords. In a way, it is the */etc/passwd* of MySQL.

Even if a user has no global privileges at all, there must be a record in the `user` table for her, if she is to issue a command successfully. See the Section 10.3.1, later in this chapter, for an example.

In the meantime, let's have a look at the fields in the `user` table:

```
mysql> DESCRIBE user;

+-----------------------+--------------------------+------+-
----+---------+-------+

| Field                 | Type                     | Null |
Key | Default | Extra |

+-----------------------+--------------------------+------+-
----+---------+-------+

| Host                  | varchar(60)              |      |
PRI |         |       |

| User                  | varchar(16)              |      |
PRI |         |       |

| Password              | varchar(45)              |      |
|         |       |

| Select_priv           | enum('N','Y')            |      |
| N       |       |

| Insert_priv           | enum('N','Y')            |      |
| N       |       |

| Update_priv           | enum('N','Y')            |      |
| N       |       |

| Delete_priv           | enum('N','Y')            |      |
| N       |       |

| Create_priv           | enum('N','Y')            |      |
| N       |       |

| Drop_priv             | enum('N','Y')            |      |
| N       |       |

| Reload_priv           | enum('N','Y')            |      |
| N       |       |

| Shutdown_priv         | enum('N','Y')            |      |
| N       |       |
| Process_priv          | enum('N','Y')           |      |
| N        |        |

| File_priv             | enum('N','Y')           |      |
| N        |        |

| Grant_priv            | enum('N','Y')           |      |
| N        |        |

| References_priv       | enum('N','Y')           |      |
| N        |        |

| Index_priv            | enum('N','Y')           |      |
| N        |        |

| Alter_priv            | enum('N','Y')           |      |
| N        |        |

| Show_db_priv          | enum('N','Y')           |      |
| N        |        |

| Super_priv            | enum('N','Y')           |      |
| N        |        |

| Create_tmp_table_priv | enum('N','Y')           |      |
| N        |        |

| Lock_tables_priv      | enum('N','Y')           |      |
| N        |        |

| Execute_priv          | enum('N','Y')           |      |
| N        |        |

| Repl_slave_priv       | enum('N','Y')           |      |
| N        |        |

| Repl_client_priv      | enum('N','Y')           |      |
| N        |        |

| ssl_type                  |
enum('','ANY','X509','SPECIFIED')    |         |       |
| ssl_cipher            | blob                      |       |
|         |       |

| x509_issuer           | blob                      |       |
|         |       |

| x509_subject          | blob                      |       |
|         |       |

| max_questions         | int(11) unsigned          |       |
| 0       |       |

| max_updates           | int(11) unsigned          |       |
| 0       |       |

| max_connections       | int(11) unsigned          |       |
| 0       |       |

+-----------------------+-------------------------+------+-
----+---------+-------+
```

Password Security

Just because MySQL passwords aren't stored in plain text, you shouldn't be lax about password selection. Anyone with the ability connect to your MySQL server can run a brute-force attack against your server in an attempt to discover passwords. A password such as *fred* or *database* is worthless; either can be easily guessed by automated software. It is best to invent a password that isn't a real word.

Because choosing strong passwords is an important part of giving users access to MySQL, here are a few guidelines for selecting and keeping good passwords:

Have a minimum length

The longer a password is, the more difficult it will be to guess.

Require special characters

A password that includes nonalphanumeric characters such as `!@#$%^&*` is more difficult to guess than one composed of numbers and letters only. Substitute the at sign (`@`) for the letter `a`. Add punctuation. Be creative.

Change passwords

Once a password is set, many people have a tendency never to change it. Often, a password may be assigned to an account that doesn't even correspond to a real person. It might belong to an application such as a web server, or middleware application. Because of this, MySQL has no built-in password aging mechanism, so you'll need to put a note on your calendar or somehow automate the process of aging passwords.

It's important to note, though, that MySQL doesn't provide any way for an administrator to enforce good password standards. You can't link MySQL against *libcrack* and demand that passwords meet that criteria, no matter how cool that idea may be. Luckily, most users can't change their own MySQL passwords, so you don't have to worry about them switching to a weak password at a later date, and as long as you (as the administrator) choose a strong password for them, they should be all right.

When a user first connects to MySQL, it checks the `user` table to decide if the user is allowed to connect and is who she says she is (the password check). But how exactly does MySQL make those decisions?

Matching a username is a simple test of equality. If the username exists in the table, it's a match. The same is true of the password. Because all MySQL passwords are hashed using the built-in `PASSWORD( )` function, expect MySQL to do something like this:

```
SELECT *

  FROM user

 WHERE User = 'username'

   AND Password = PASSWORD('password')
```

However, this query could return multiple records. The `user` table's primary key is composed of the fields `User` and `Host`, not just `User`, which means a single user can have multiple entries in the tableespecially if she is allowed to connect from several specifically named hosts. MySQL must check all those records to see which one matches.

Things get more interesting when you realize that the `Host` field may contain any of the standard SQL wildcard characters: `_` (matches a single character) and `%` (matches any number of characters). What does MySQL do if the user *jane* attempts to connect from the host *jane.example.com*, and the user table contains records for *jane@jane.example.com* as well as *jane@%.example.com*?

10.2.2.1 Host matching

The first rule you need to know about MySQL's privilege system is this: the most specific match always wins. MySQL will always prefer an exact match over one that uses a wildcard of any sort.

MySQL accomplishes this by internally sorting the records in the `user` table based on the `Host` and `User` fieldsin that order. Hostnames and IP addresses without wildcards come before those that contain them.

Given a list of host entries such as this:

- *%*
- *localhost*
- *jane.example.com*
- *%.example.com*
- *192.168.1.50*
- *joe.example.com*
- *192.168.2.0/255.255.255.0*

MySQL sorts them in this order:

- *localhost*
- *192.168.1.50*
- *jane.example.com*
- *joe.example.com*
- *192.168.2.0/255.255.255.0*
- *%.example.com*
- *%*

To clarify what "most specific" means to MySQL, let's consider how MySQL will match several username and hostname combinations. Assuming that the user *jane* and the "any user" (represented here as the absence of a username) can connect from some of the various hosts listed earlier, MySQL sorts the entries like this:

- *jane@jane.example.com*

- *jane@joe.example.com*
- *@localhost*
- *@192.168.1.50*
- *@jane.example.com*
- *@joe.example.com*
- *@%.example.com*
- *jane@%.example.com*
- *jane@%*

When *jane* connects from *jane.example.com*, she may have a different set of privileges from when she connects from *joe.example.com*. Other users connecting from *web.example.com* will match the *%@%.example.com* record and receive whatever privileges have been granted in that row. When *jane* connects from *web.example.com*, she'll receive the privileges granted to *jane@%.example.com*.

10.2.3 The host Table

The `host` table assigns database-level privileges for users connecting from specific hosts (or groups of hosts). Let's look at the table:

```
mysql> DESCRIBE host;

+-----------------------+---------------+------+-----+-----
----+-------+

| Field                 | Type          | Null | Key |
Default | Extra |

+-----------------------+---------------+------+-----+-----
----+-------+
| Host                  | char(60)      |       | PRI |
|       |
| Db                    | char(64)      |       | PRI |
|       |
| Select_priv           | enum('N','Y') |       |     | N
|       |
| Insert_priv           | enum('N','Y') |       |     | N
|       |
| Update_priv           | enum('N','Y') |       |     | N
|       |
| Delete_priv           | enum('N','Y') |       |     | N
|       |
| Create_priv           | enum('N','Y') |       |     | N
|       |
| Drop_priv             | enum('N','Y') |       |     | N
|       |
| Grant_priv            | enum('N','Y') |       |     | N
|       |
| References_priv       | enum('N','Y') |       |     | N
|       |
| Index_priv            | enum('N','Y') |       |     | N
|       |
| Alter_priv            | enum('N','Y') |       |     | N
|       |
| Create_tmp_table_priv | enum('N','Y') |       |     | N
|       |
| Lock_tables_priv      | enum('N','Y') |       |     | N
|       |
+-----------------------+----------------+-------+------+-----
----+--------+
```

With the exception of the `Db` field, this table is a subset of the `user` table. It is missing all the global privileges (such as the shutdown privilege), but all the privileges that can be applied to a database objects are there. As expected, they all default to No.

Records might appear in this table to enforce a rule that all connections from hosts in the *public.example.com* domain are forbidden from changing any data. You can also allow anyone connecting from *secure.example.com* to have full privileges on tables in the `security` database.

10.2.4 The db Table

The `db` table specifies database-level privileges for a particular user and database:

```
mysql> DESCRIBE db;

+-----------------------+----------------+-------+------+-----
----+--------+

| Field                 | Type           | Null | Key |
Default | Extra |

+-----------------------+----------------+-------+------+-----
----+--------+

| Host                  | char(60)       |      | PRI |
|       |

| Db                    | char(64)       |      | PRI |
|       |

| User                  | char(16)       |      | PRI |
|       |

| Select_priv           | enum('N','Y') |      |     | N
|       |
| Insert_priv           | enum('N','Y') |      |     | N
|       |

| Update_priv           | enum('N','Y') |      |     | N
|       |

| Delete_priv           | enum('N','Y') |      |     | N
|       |

| Create_priv           | enum('N','Y') |      |     | N
|       |

| Drop_priv             | enum('N','Y') |      |     | N
|       |

| Grant_priv            | enum('N','Y') |      |     | N
|       |

| References_priv       | enum('N','Y') |      |     | N
|       |

| Index_priv            | enum('N','Y') |      |     | N
|       |

| Alter_priv            | enum('N','Y') |      |     | N
|       |

| Create_tmp_table_priv | enum('N','Y') |      |     | N
|       |

| Lock_tables_priv      | enum('N','Y') |      |     | N
|       |

+-----------------------+---------------+------+-----+-----
----+-------+
```

This table is virtually identical to the `host` table. The only difference is the addition of the `User` field, which is needed in order to create per-user privileges.

By making the appropriate entries in this table, you could ensure that *joe* has full privileges on the `sales` database when connecting from either *accounting.example.com* or *cfo.example.com.*

10.2.5 The tables_priv Table

Going a level deeper, the `tables_priv` table controls table-level privileges (those applied to all columns in a table) for a particular user:

```
mysql> DESCRIBE tables_priv;

+-------------+---------------------+------+-----+---------
+-------+

| Field       | Type                | Null | Key | Default
| Extra |

+-------------+---------------------+------+-----+---------
+-------+

| Host        | char(60) binary     |      | PRI |
|       |

| Db          | char(64) binary     |      | PRI |
|       |

| User        | char(16) binary     |      | PRI |
|       |

| Table_name  | char(60) binary     |      | PRI |
|       |

| Grantor     | char(77)            |      | MUL |
|       |

| Timestamp   | timestamp(14)       | YES  |     | NULL
|       |

| Table_priv  | set(...)            |      |     |
|       |

| Column_priv | set(...)            |      |     |
|       |

+-------------+---------------------+------+-----+---------
+-------+
```

This table probably looks a bit odd. The creators of MySQL decided to use a `SET( )` function to represent privileges in both the `tables_priv` and `columns_priv` tables. In doing so, they made it difficult for authors to present a nice clean listing of all the grant tables in their books (we're sure that wasn't their intent).

The `...` in the `Table_priv` field should actually read:

```
'Select','Insert','Update','Delete','Create','Drop','Grant'
```

and the `...` in the `Column_priv` field really contains:

```
'Select','Insert','Update','References'
```

Both are new fields not seen in previous tables. As their names imply, they control table and column privileges. There's another new field in the table: `Grantor`. This 77-character field records the identity of the user who granted these privileges. It is 77 characters in size because it is intended to hold a username (up to 16 characters), an `@` symbol, and a hostname (up to 60 characters).

The `Timestamp` field also makes its first appearance in this table. As you'd expect, it simply records the time when the record was created or modified.

Using table-level privileges isn't very common in MySQL, so don't be surprised if your server has no records in its `tables_priv` table. If you've installed the popular *phpMyAdmin* utility (discussed in Appendix C), however, you might see something like this:

```
mysql> SELECT * FROM tables_priv \G

*************************** 1. row
***************************

       Host: localhost

         Db: mysql

       User: phpmyadmin

 Table_name: user

    Grantor: root@localhost
```

```
  Timestamp: 20020308185823

 Table_priv:

Column_priv: Select
```

This entry grants the `phpmyadmin` user access to the database, with the `Select` privileges he needs to obtain information from MySQL. This table doesn't grant privileges on any particular data; that has to be done in another table, as you'll see in the next section.

10.2.6 The columns_priv Table

The final table, `columns_priv`, is similar to the `tables_priv` table. It specifies individual column privileges in a particular table:

```
mysql> DESCRIBE columns_priv;

+-------------+---------------------+------+-----+---------
+-------+

| Field       | Type                | Null | Key | Default
| Extra |

+-------------+---------------------+------+-----+---------
+-------+

| Host        | char(60) binary     |      | PRI |
|       |

| Db          | char(64) binary     |      | PRI |
|       |

| User        | char(16) binary     |      | PRI |
|       |

| Table_name  | char(64) binary     |      | PRI |
|       |

| Column_name | char(64) binary     |      | PRI |
|       |
| Timestamp   | timestamp(14)       | YES  |     | NULL
|       |

| Column_priv | set(...)            |      |     |
|       |

+------------+--------------------+------+-----+---------
+-------+
```

Just as in the previous table, the `...` in the `Column_priv` field really contains:

```
'Select','Insert','Update','References'
```

Column-level privileges also aren't very common in MySQL. But there are cases when you're likely to encounter them. Again, *phpMyAdmin* is a great example:

```
mysql> SELECT * FROM columns_priv LIMIT 1 \G

*************************** 1. row
***************************

       Host: localhost

         Db: mysql

       User: phpmyadmin

 Table_name: tables_priv

Column_name: Column_priv

  Timestamp: 20020308185830

Column_priv: Select
```

This record allows the `phpmyadmin` user to select data from the `Column_priv` column of the `tables_priv` table in the `mysql` database.

Confused yet? Can't blame you. The grant tables can be quite confusing at first. Until you spend some time working with them, you won't really appreciate the flexibility this design provides.

We wouldn't recommend spending that time unless absolutely necessary. Instead, read the next section. It reviews the `GRANT` and `REVOKE` commands and then looks at how they interact with the grant tables so that you don't have to. It's only worth delving deeply into the grant tables if you find a situation that can't be set up (or is too complex) using the `GRANT` command.

```
mysql> <b>REVOKE ALL PRIVILEGES ON *.*
FROM 'root'@'localhost';</b>

mysql> <b>GRANT ALL PRIVILEGES ON *.* TO
'raymond'@'%' IDENTIFIED BY '27skuw!'</b> ->
<b>WITH GRANT OPTION;</b>

mysql> <b>SELECT * FROM user WHERE User =
'raymond' \G</b> ***************************
1. row ***************************

  Host: %

  User: raymond

  Password: 11417e201753de4b

  Select_priv: Y

  Insert_priv: Y

  Update_priv: Y

  Delete_priv: Y

  Create_priv: Y
  Drop_priv: Y

  Reload_priv: Y

  Shutdown_priv: Y

  Process_priv: Y

  File_priv: Y

  Grant_priv: Y

References_priv: Y

  Index_priv: Y

  Alter_priv: Y
```

```
mysql> <b>GRANT ALL PRIVILEGES ON *.* TO
'diana'@'%.widgets.example.dom' IDENTIFIED BY
</b> -> <b>'yu-gi-oh' WITH GRANT OPTION;</b>

mysql> <b>GRANT INSERT,UPDATE
PRIVILEGES ON widgets.orders</b> -> <b>TO
'tera'@'%.widgets.example.com'</b> ->
<b>IDENTIFIED BY 'rachel!94';</b>
```

```
mysql> <b>GRANT INSERT ON logs.* TO
'logger'@'%.widgets.example.com' </b> <b> </b>->
<b>IDENTIFIED BY 'blah0halb';</b>

mysql> <b>SELECT * FROM user WHERE User =
'logger' \G</b> *************************** 1.
row ***************************

 Host: %.widgets.example.com

 User: logger

 Password: 2d502d346553f4f3

 Select_priv: N

 Insert_priv: N

 Update_priv: N

 Delete_priv: N

 Create_priv: N

 Drop_priv: N

 Reload_priv: N
```

```
  Shutdown_priv: N

  Process_priv: N

  File_priv: N

  Grant_priv: N

References_priv: N

  Index_priv: N

  Alter_priv: N

mysql> <b>SELECT * FROM db WHERE User =
'logger' \G</b> *************************** 1.
row ***************************

  Host: %.widgets.example.com

  Db: logs

  User: logger

  Select_priv: N

  Insert_priv: Y
  Update_priv: N

  Delete_priv: N

  Create_priv: N

  Drop_priv: N

  Grant_priv: N

References_priv: N

  Index_priv: N

  Alter_priv: N
```

```
mysql> <b>GRANT PROCESS, SHUTDOWN on
*.* </b> <b> </b>-> <b>TO
'noc'@'monitorserver.noc.widgets.example.com' </b>
<b> </b> -> <b>IDENTIFIED BY 'q!w@e#r$t%';
</b>

mysql> <b>SELECT * FROM user WHERE User =
'noc' \G</b> *************************** 1. row
***************************
```

```
  Host: monitorserver.noc.widgets.example.com
User: noc

  Password: 7abf52ce38207ca0

  Select_priv: N

  Insert_priv: N

  Update_priv: N

  Delete_priv: N

  Create_priv: N

  Drop_priv: N

  Reload_priv: N

  Shutdown_priv: Y

  Process_priv: Y

  File_priv: N

  Grant_priv: N

References_priv: N
```

```
   Index_priv: N

   Alter_priv: N

mysql> <b>REVOKE SELECT ON payroll.* FROM
raymond;</b> ERROR 1141: There is no such grant
defined for user 'raymond' on host '%'

mysql> <b>SHOW GRANTS FOR raymond \G</b>
*************************** 1. row
***************************

Grants for raymond@%: GRANT ALL
PRIVILEGES ON *.* TO 'raymond'@'%'

IDENTIFIED BY PASSWORD '11417e201753de4b'
WITH GRANT OPTION

mysql> <b>GRANT ALL PRIVILEGES ON db1.*
TO raymond WITH GRANT OPTION;</b> mysql>
<b>GRANT ALL PRIVILEGES ON db2.* TO
raymond WITH GRANT OPTION;</b> mysql>
<b>GRANT ALL PRIVILEGES ON db3.* TO
raymond WITH GRANT OPTION;</b>

mysql> <b>GRANT ALL PRIVILEGES ON *.*
EXCEPT payroll.* TO raymond;</b>
```

```
mysql> <b>GRANT ALL PRIVILEGES ON *.* TO raymond@"%"</b> -> <b>EXCEPT raymond@insecure.example.com;</b>
```

```
mysql> <b>GRANT ALL PRIVILEGES ON my_db.* TO raymond;</b>
```

```
$ <b>mysqladmin drop my_db</b>
```

```
mysql> <b>DELETE FROM db where Db='my_db';</b> mysql> <b>DELETE FROM tables_priv where Db='my_db';</b> mysql> <b>DELETE FROM columns_priv where Db='my_db';</b> mysql> <b>FLUSH PRIVILEGES;</b>
```

```
mysql> <b>DELETE FROM tables_priv where Db='my_db' AND Table_name='my_table';</b> mysql> <b>DELETE FROM columns_priv where Db='my_db' AND Table_name='my_table';</b> mysql> <b>FLUSH PRIVILEGES;</b>
```

Obviously, no `DELETE` is needed against the `db` table because it isn't a database-wide privilege that needs to be revoked.

In some cases, you might find this useful. For example, if you're dropping a table just to reload it again from backup, it's much more convenient not to have to worry about revoking and regranting privileges.[4]

[4] An argument can be made that if you're restoring from a backup and leaving the existing privileges in place, you're not necessarily restoring to the backed-up state and might be leaving any security holes that were created afterwards still in place.

In an ideal world, this would be an option to commands like `ALTER TABLE` or `DROP DATABASE`, to allow the system to hunt down and destroy granted privileges automatically. Alternatively, MySQL could default to a theoretically "secure" methodology of destroying stale privileges but offer the option to leave the privileges intact.

10.4 Operating System Security

Even the most well thought out and secure grant tables will do you little good if any random cracker can get root access to your server. With unlimited access, someone could simply copy all your data files to another machine running MySQL.[5]

Doing so would effectively give the cracker an identical copy of your database.

[5] Remember: MyISAM data files are portable across operating systems and CPU architectures.

Data theft isn't the only threat to guard against. A creative cracker may decide that it's more fun to

make subtle changes to your data over the course of weeks or even months. Depending on how long you keep backups around and when the data corruption is noticed, such an attack could be quite devastating.

10.4.1 Guidelines

The general guidelines discussed here aren't a comprehensive guide to system security. If you are serious about securityand you should bewe recommend a copy of

O'Reilly's *Practical*

Unix and Internet Security by Simson Garfinkel, Gene Spafford, and Alan Schwartz. That said, here are some ideas for maintaining good security on your database servers:

Don't run MySQL from a privileged account

> The root user on Unix and the system (Administrator) user on Windows possess ultimate control over the system. If a security bug is discovered in MySQL, and you're running it as a
>
> privileged user, a hacker can gain extensive access to your server.
>
> The installation instructions are quite clear about this, but it bears repeating. Create a separate account, usually *mysql*, for the purpose of running MySQL.

Keep your operating system up to date

> All operating system vendors (Microsoft, Sun, RedHat, SUSE, etc.) provide notifications when a security-related update is available.

Find your vendor's mailing list and subscribe to it.

Pay special attention to the security list for MySQL itself, obviously, as well as anything that may interact directly with the database, such as PHP or Perl.

Restrict logins on the database host

Does every developer building a MySQL-based application need an account on the server? Certainly not; only system and database administrators need accounts on the machine. All the developers need to be able to do is issue queries against the database remotely using TCP/IP.

Have your server audited

Many larger organizations have internal auditors who can assess the security of a server and make recommendations for improving it. If you aren't lucky enough to have access to auditors, you can hire a security consultant to perform the audit.

Backups are important here as well. If your server is broken into, you'll need to reinstall the operating system from

an untainted source. Once that's done,

you'll be faced with the task of having to restore

all the data. If you have the luxury of time, you might compare the hacked server to a known good backup in an effort to determine how the hacker was able to get in. Chapter 9 is devoted to backup and recovery issues.

```
$ <b>mysqld_safe --skip-networking</b>

[mysqld]



skip-networking

mysql> <b>SHOW VARIABLES LIKE
'have_openssl';</b>

+---------------+-------+



| Variable_name | Value |



+---------------+-------+



| have_openssl | NO |
+---------------+-------+

1 row in set (0.00 sec)

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to 'raymond'@'%'</b>

   -> <b>IDENTIFIED BY "FooBar!" REQUIRE
SSL;</b>

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to raymond@%</b>

   -> <b>IDENTIFIED BY "FooBar!" REQUIRE
x509;</b>

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to 'raymond'@'%'</b>

   -> <b>IDENTIFIED BY "FooBar!"</b>

   -> <b>REQUIRE SUBJECT "/C=US/ST=New
York/L=Albany/O=Widgets Inc./CN=client-ray.
```

```
example.com/emailAddress=raymond@example.com
";</b>

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to 'raymond'@'%'</b>

   -> <b>IDENTIFIED BY "FooBar!"</b>

   -> <b>REQUIRE ISSUER
"/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.



com/emailAddress=admin@example.com";</b>

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to 'raymond'@'%'</b>

   -> <b>IDENTIFIED BY "FooBar!"</b>

   -> <b>REQUIRE SUBJECT "/C=US/ST=New
York/L=Albany/O=Widgets Inc./CN=client-ray.
```

```
example.com/emailAddress=raymond@example.com
"</b>

   -> <b>AND ISSUER
"/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/



emailAddress=admin@example.com";</b>

mysql> <b>GRANT ALL PRIVILEGES ON
ssl_only_db.* to 'raymond'@'%'</b>

   -> <b>IDENTIFIED BY "FooBar!"</b>

   -> <b>REQUIRE CIPHER "EDH-RSA-DES-
CBC3-SHA";</b>

$ <b>ssh -N -f -L 4406:db.example.com:3306</b>

$ <b>mysql -h 127.0.0.1 -P 4406</b>

$ <b>./configure --with-
libwrap=/usr/local/tcp_wrappers</b>

# deny all connections
```

```
ALL: ALL
```

```
# allow mysql connections from hosts on the local network
```

```
mysqld: 192.168.1.0/255.255.0.0 : allow
```

```
mysql 3306/tcp # MySQL Server
```

```
Host 'host.badguy.com' blocked because of many connection errors.


Unblock with 'mysqladmin flush-hosts'
```

```
$ <b>mysqld_safe -O max_connection_errors=999999999</b>
```

It's currently not possible to set `max_connection_errors` to zero and disable the check entirely. The only way to do that is to remove the check from the source code.

```
mysql> <b>SELECT MD5('pa55word');</b> +-------
---------------------------+

| MD5('pa55word') |

+----------------------------------+

| a17a41337551d6542fd005e18b43afd4 |

+----------------------------------+

1 row in set (0.13 sec) mysql> <b>SELECT
PASSWORD('pa55word');</b> +--------------------+

| PASSWORD('pa55word') |

+--------------------+

| 1d35c6556b8cab45 |

+--------------------+

1 row in set (0.00 sec) mysql> <b>SELECT
ENCRYPT('pa55word');</b> +--------------------+

| ENCRYPT('pa55word') |
+---------------------+

| up2Ecb0Hdj25A |

+---------------------+

1 row in set (0.17 sec)
```

```
INSERT INTO user_table (user, pass) VALUE
('jzawodn', MD5('pa55word') )
```

```
SELECT *

   FROM user_table WHERE user = '$username'

   AND pass = MD5('$password')
```

```
SELECT MIN(balance), MAX(balance),
AVG(balance) FROM account GROUP BY type
```

```
SELECT *
  FROM account WHERE balance > 100000
```

If there is an index on the `balance` field, MySQL will probably locate the records in a split second. But if the data is encrypted, you have to get all the records in your application and filter them after they're decrypted. There's just no way for MySQL to help you out.

10.6.4 Source Code Modification

If you're looking for a more flexible approach than either encrypted filesystems or application-based encryption, you can always build a custom solution. The source code for MySQL is freely available under the GNU General Public License.

This sort of work requires that you either know C++ or hire someone who does. Beyond that, you'll be looking to create your own table handler with native encryption support, or you might find it easier to extend an existing table handler (the MyISAM and

BDB handlers are easiest to understand) with encryption.

You'll find the relevant files in the *sql* directory of the MySQL source code. Each table handler is composed of at least two C++ files. The MyISAM handler code, for example, is in *ha_myisam.h* and *ha_myisam.cc*.

```
$ <b>./configure --prefix=/chroot/mysql</b>

chroot /chroot/mysql

$ <b>cd /chroot/mysql</b>

$ <b>mkdir chroot</b>

$ <b>cd chroot</b>

$ <b>ln -s /chroot/mysql mysql</b>

$ <b>mysqld_safe --chroot=/chroot/mysql --
user=1001</b>
```

You'll notice we used the Unix UID of the MySQL user, instead of `--user=mysql`. This is because in the chrooted environment, the MySQL server may no longer be able to query your authentication backend to do username-to-UID lookups.[14]

[14] From our experience in testing this, it might be as simple as copying `libnss*` to your MySQL library directory in the chrooted environment, but from a practical standpoint, it's probably best not to worry about such things,

and just enter the UID directly in your startup script.

There are some caveats when using a chrooted MySQL server. `LOAD DATA INFILE` and other commands that directly access filenames may behave significantly differently than you expect because the server no longer considers `/` to be the filesystem root. So, if you tell it to load data from */tmp/filename*, you should be sure that the file is actually */chroot/mysql/tmp/filename*, or MySQL won't be able to find it.

Appendix A. The SHOW STATUS and SHOW INNODB STATUS Commands

A.1 SHOW STATUS

The SHOW STATUS command allows you to view a snapshot of the many (over 120) internal status counters that MySQL maintains. These counters track particular events in MySQL. For example, every time you issue a SELECT query, MySQL increments the Com_select counter.

This command is valuable because early signs of performance problems often appear first in the SHOW STATUS outputbut you have to be looking for them. By learning which counters are most important to server performance and how to interpret them, you'll be well prepared to head off problems before they become an issue for your users.

This appendix is designed to do just that. Here you'll find a brief summary of the more important counters MySQL provides, as well as some discussion of what to watch out for and how you might correct some of the problems highlighted here. We've attempted to group related items together rather than simply using an alphabetical list. And we've omitted the counters that have little relevance to MySQL performance. See the *MySQL Reference Manual* for a full list of the counters available in your version of MySQL.

Running the SHOW STATUS command repeatedly and examining the results is a very tedious process. To make life a bit easier, *mytop* automates much of the process. See Appendix B for more about *mytop*.

Note that these counters are stored as unsigned integers. On a 32-bit platform such as Intel x86, that means the counters will wrap just over the 4.2 billion mark. This can lead to very confusing numbers and wildly incorrect conclusions. So be sure to check how long your server has been online (`Uptime`) before jumping to conclusions. The odds of a counter wrapping increase as time goes on.

As you read the descriptions in this appendix, consider how you might add some of these counters to your monitoring infrastructure. Third-party MySQL modules already exist for most of the freely available *rrdtool*-based systems (Cricket, Cacti, etc.). If none are available for your system, consider using one of the free plug-ins as a starting point for building your own. They're not very complicated.

A.1.1 Thread and Connection Statistics

Just because connections to MySQL are very lightweight doesn't excuse applications that poorly use their connections. A rapid-fire connect/disconnect cycle will slow down a MySQL server. It may not be noticeable under most circumstances, but when things get busy you don't want it getting in the way.

Using information in the following counters, you can get a high-level picture of what's going on with MySQL's connections and the threads that service them.

Aborted_clients

This is the number of connections to the server that were aborted because the client disconnected without properly closing the session. This might happen if the client program dies abruptly from a runtime error or is killed.

Aborted_connects

This counter contains the number of connection attempts that failed. These failures may be because of user privilege issues, such as an incorrect password, or communications issues such as malformed connection packets or `connect_timeout` being exceededoften as the result of a network or firewall problem.

Bytes_received

Number of bytes received from all clients, including other MySQL servers involved in replication.

Bytes_sent

Number of bytes sent to all clients, including other MySQL servers.

Connections

Total number of connection attempts, both successful and failed, to the MySQL server.

Max_used_connections

The peak number of simultaneous connections.

Slow_launch_threads

Number of threads that have taken longer than `slow_launch_time` to be created. A nonzero value here is a often a sign of excessive CPU load on the server.

Threads_cached

Number of threads in the thread cache. See Chapter 6 for more about MySQL's thread cache.

Threads_connected

Number of currently open connections.

Threads_created

Total number of threads that have been created to handle connections.

Threads_running

Number of threads that are doing work (not sleeping).

Uptime

How long (in seconds) the MySQL server has been up and running.

A.1.2 Command Counters

A large percentage of MySQL's counters are devoted to counting the various commands or queries that you issue to a MySQL server. Everything from a `SELECT` to a `RESET MASTER` is counted.

Com_`*`

> The number of times each `*` command has been executed. Most names map directly to SQL queries or related commands. Some are derived from function names in the MySQL C API. For example, `Com_select` counts `SELECT` queries, while `Com_change_db` is incremented any time you issue a `USE` command to switch databases. `Com_change_db` can also count the number of times you change databases programmatically using the `mysql_change_db( )` function from the C API or a language such as PHP.

Questions

> The total of number of queries and commands sent to the server. It should be the same as summing all the `Com_*` values.

A.1.3 Temporary Files and Tables

During normal operations, MySQL may need to create temporary tables and files from time to time. It's completely normal. If this happens excessively, however, performance may degrade as a result of the additional disk I/O required.

Created_tmp_disk_tables

> The number of temporary tables created while executing statements that were stored on disk. The decision to put a temporary table on disk rather than in memory is controlled by the `tmp_table_size` variable. Tables larger than the value of this variable will be created on disk, while those smaller will be created in memory. But temporary tables created explicitly with

`CREATE TEMPORARY TABLE` aren't governed by this. They always reside on disk.

Created_tmp_tables

Similar to `Created_tmp_disk_tables` except that it counts the number of implicit temporary tables created in memory and on disk.

Created_tmp_files

How many temporary files `mysqld` has created.

Comparing `Created_tmp_tables` to `Created_tmp_disk_tables` will tell you the percentage of your temporary tables that are being constructed on the much slower disk as opposed to being created in much faster memory. Obviously, you will never be able to completely eliminate the use of on-disk temporary tables, but if too many of your tables are being created on disk, you may want to increase your `tmp_table_size`.

A.1.4 Data Access Patterns

The handler counters track the various ways that rows are read from tables at the lower level. MySQL communicates with each storage engine through a common API. Because storage engines used to be known as table handlers, the counters still refer to handler operations.

Studying these values will tell you how often MySQL can fetch the exact records it needs as opposed to fetching lots of records and checking field values to see if it really wanted the records. Generally, the counters help to highlight when MySQL is or isn't effectively using your indexes. For example, if the

`Handler_read_first` is too high, the server is doing a lot of full index scans, which is probably not what you want it to do.

On the other hand, if the `Handler_read_key` value is high, MySQL is using the indexes to optimum effect and going right after the row it needs quite often without having to dig around and look for it, and your queries and tables are using indexes to optimum effect.

Handler_commit

> Number of internal `COMMIT` commands.

Handler_delete

> Number of times MySQL has deleted a row from a table.

Handler_read_first

> Number of times the first entry was read from an index.

Handler_read_key

> Number of times a row was requested based on a key. The higher this value is, the better. It means that MySQL is effectively using your indexes.

Handler_read_next

> Number of requests to read next row using the key order. This is incremented if you are querying an index column with a range constraint or doing an index scan.

Handler_read_prev

Number of requests to read previous row in key order. This is mainly used when you have a query using `ORDER BY ... DESC`.

Handler_read_rnd

Number of requests to read a row based on a fixed position. If you do a lot of queries that require sorting of the result, this figure will likely be quite high.

Handler_read_rnd_next

How many times MySQL has read the next row in a datafile. This figure will be high if you are doing a lot of table scans. If that is the case, it's likely that either your tables need to be indexed, or the queries you are submitting need to be changed to take better advantage of the indexes that do exist.

Handler_rollback

Number of internal `ROLLBACK` commands.

Handler_update

Number of requests to update a table row.

Handler_write

Number of table rows that have been inserted.

A.1.5 MyISAM Key Buffer

As described in Chapter 4, the key buffer is where MySQL caches index blocks for MyISAM tables. Generally speaking, a large key buffer means hitting a disk less frequently, so queries will run more efficiently. Increasing the size of the key buffer is often the single biggest "bang for your buck" adjustment you can make on a server that uses mostly MyISAM tables.

Key_blocks_used

The number of 1024-byte blocks contained in the key cache.

Key_read_requests

The number of times a block is requested to be read. It might be found in cache, or it might be read from disk (in which case `Key_reads` are also incremented).

Key_reads

The number of physical reads during which a key block was read from disk.

Key_write_requests

The number of requests for a key block to be written.

Key_writes

The number of physical writes during which key blocks were written to the disk.

These last four counters tell you how often MySQL needed to read/write a key block. Each time a "request" occurs, there may or may not be an actual read or write to match it. If there's not, that's good, because it means the data was already in memory, and the request never hit the disk.

As a general rule of thumb, you want the request numbers to be roughly 50-100 times higher than the corresponding read or write numbers. Higher is better! If they're smaller than that, increasing the size of the key buffer is likely in order.

A.1.6 File Descriptors

On a MySQL server that handles hundreds or thousands of simultaneous queries, you need to keep an eye on the number of open file descriptors MySQL is using. The `table_cache` setting has the largest impact on MySQL's file descriptor usage if you're mainly using MyISAM tables. For MyISAM tables, the numbers work out like this: each *.frm* file is opened once when the table is first accessed. The contents are cached, and it is immediately closed. The index file (*.MYI*) is opened once and is shared among all clients accessing it. The data file (*.MYD*) is opened by each client using the table. The table cache may reduce the number of times that the *.frm* file is reopened on a system with many active tables.

The following counters help keep track of MySQL's file descriptor usage:

Open_tables

> The total number of tables that are currently open.

Open_files

The total number of open files.

Open_streams

Number of streams that are open. (These are mostly used for logging.)

Opened_tables

Number of tables that have been opened since the server started. If `Opened_tables` is significantly higher than `Open_tables`, you should increase the size of `table_cache`.

A.1.7 Query Cache

As described in Chapter 5, the query cache can provide an impressive performance boost to applications that issue identical queries in a repetitive manner. The following counters will help you understand how effective the query cache is and whether you can safely increase or decrease its size.

Qcache_queries_in_cache

How many query results are in the query cache.

Qcache_inserts

How many times MySQL has inserted the results of a query into the cache.

Qcache_hits

The number of times MySQL has found a query in the cache instead of having to actually execute the query.

Qcache_lowmem_prunes

Each time MySQL needs to prune the query cache (remove some entries) because it has run out of memory, it increments this counter. Ideally this counter should be 0. If the number increases with any regularity, consider increasing the `query_cache_size`.

Qcache_not_cached

This is the number of queries that aren't cachable, either because the query explicitly opted out of the cache, or the result was larger than `query_cache_limit`.

Qcache_free_memory

Free space (in bytes) remaining in the cache.

Qcache_free_blocks

How many free (unused) blocks exist in the cache.

Qcache_total_blocks

This is the total number of blocks in the cache. By subtracting `Qcache_free_blocks` from this value, you can derive the number of nonempty blocks. Because the query cache blocks are allocated on an as-needed basis, this information isn't terribly useful for anything other than impressing your coworkers.

A.1.8 SELECTs

This group of counters tracks `SELECT` queries that may be problematic. Typically they're queries that might have been run more efficiently if MySQL had been able to find an appropriate index to use. If any of these are nonzero and growing at even a moderate rate, go back to Chapter 4 to refresh your memory on how MySQL's indexes workyou probably need to add at least one.

Select_full_join

> Number of joins without keys. If this figure isn't 0, you should check your indexes carefully.

Select_full_range_join

> Number of joins that used a range search on reference table.

Select_range

> Number of joins that used ranges on the first table. It's normally not critical even if this number is big.

Select_scan

> Number of joins that did a full scan of the first table.

Select_range_check

> Number of joins that check for key usage after each row. If this isn't 0, you should check your indexes.

Slow_queries

> Number of queries that have taken more than `long_query_time`.

Unfortunately, there is no easy way to find out which query triggered a particular counter increase. By enabling the slow query log, however, you can at least capture all queries that take more than a predefined number of seconds to complete. Sometimes you'll find that those slow queries are also suffering from one of the problems listed above. See Chapter 5 for more about MySQL's query cache.

A.1.9 Sorts

Queries with `ORDER BY` clauses are commonplace, but sorting a nontrivial number of rows can become a burden if done frequently. The Section 4.1.1.2 in Chapter 4 discusses some of the index-based sorting optimizations present in MySQL 4.0 and beyond. If MySQL can't use an index for sorting, however, it must resort to old-fashioned sorting techniques.

Sort_merge_passes

> Number of merge-passes the sort algorithms have performed. If this value gets too high, you may wish to increase `sort_buffer`.

Sort_range

> Number of sorts done on ranges. This is better than sorting an entire table.

Sort_rows

> The total number of rows that have been sorted.

Sort_scan

> Number of sorts that were done using a table scan. Ideally, this shouldn't happen often. If it does, you probably need to add an index somewhere.

A.1.10 Table Locking

Any time MySQL waits for a table lock, it is a bad thing. How much of a bad thing is often a function of the application and usage patterns, but there's no way around the fact that a MySQL thread waiting for a lock is getting absolutely no work done. To help track locks and lock contention on tables, MySQL provides the following two counters.

Table_locks_immediate

> Number of times the server acquired a table lock immediately.

Table_locks_waited

> Number of times the server had to wait on a table lock.

The goal is to have `Table_locks_immediate` as high as possible and `Table_locks_waited` as close to zero as possible. Realistically, there has to be a middle ground, but those are the ideals we would hope for in a perfect world. For lower-volume or single user applications, table locks are often a nonissue. However, on large multiuser systems or high-volume web sites, table locks can be a very serious problem.

A high percentage of `Table_locks_waited` is a sign either that you need to make queries more efficient (so that they hold locks for a short period of time) or that you may need to consider an alternative table type. Moving from MyISAM to InnoDB tables will often greatly reduce lock contentionbut not always. See Chapter 2 for more details about table locking.

A.2 SHOW INNODB STATUS

As noted in Chapter 1, the `SHOW INNODB STATUS` command produces detailed statistics about what's going on inside the InnoDB storage engine (far more detailed than anything in MyISAM). A detailed understanding of all the statistics InnoDB provides is beyond the scope of what most database administrators will ever need. Much of the information InnoDB presents is useful only in rare and very specific diagnostic activities, so we'll keep the discussion fairly basic here and focus on the more commonly used values.

Sample output from `SHOW INNODB STATUS` command is included at the end of this section. The output is broken up into several labeled groups. For most day to day use, the most informative sections are Transactions, Buffer Pool and Memory, and Row Operations.

Semaphores

This section details the various locks used inside InnoDB. Higher values here generally indicate a busy server with frequent contention inside InnoDB. They are cumulative statistics, however, so the longer your server has been up, the higher you can expect them to be.

Transactions

Each of the active or pending transactions is listed in this section. For each, InnoDB lists the MySQL thread ID as well as the IP address and MySQL username responsible for initiating the transaction. You may see indications of transactions waiting on locks here. If so, there's a good chance your application is encountering deadlocks.

File I/O

Here InnoDB lists the state of each file I/O thread and provides counts of other I/O-related activity.

Insert Buffer and Adaptive Hash Index

When records are added to InnoDB, they are first put into the insert buffer. From there InnoDB merges records into the tablespace. This section provides a few metrics generated during those operations.

Log

The transaction log statistics are presented here, including the current sequence number and the highest sequence numbers from the most recent log flush and checkpoint operations. InnoDB also provides average values for the number of log-related I/O operations per second.

Buffer Pool and Memory

This section tells you how well InnoDB is using the memory you've given it via the `innodb_buffer_pool` setting. The "buffer pool size" and "free buffers" values give you an idea of how much of that memory is in use. InnoDB also provides read/create/write-per-second statistics that indicate how quickly the database pages are changing.

Row Operations

Here you'll find some very useful high-level numbers that track the frequency of `INSERT`s, `UPDATE`s, `DELETE`s, and `SELECT`s as well as counting the number of rows affected by each.

Here's some sample output from a *SHOW INNODB STATUS* command:

```
mysql> SHOW INNODB STATUS \G
*************************** 1. row
***************************
Status:
= = = = = = = = = = = = = = = = = = = = = = = = =
= = = = =
= = = = = = =
031218  8:29:53 INNODB MONITOR OUTPUT
= = = = = = = = = = = = = = = = = = = = = = = = =
= = = = =
= = = = = = =
Per second averages calculated from the last 3
seconds
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 5, signal
count 5
Mutex spin waits 0, rounds 0, OS waits 0
RW-shared spins 6, OS waits 3; RW-excl spins 2, OS
waits 2
------------
TRANSACTIONS
------------
Trx id counter 0 1039616
Purge done for trx's n:o < 0 454662 undo n:o < 0 0
Total number of lock structs in row lock hash
table 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, OS thread id
49162
MySQL thread id 16, query id 112 216.145.52.107
```

```
jzawodn
show innodb status
--------
FILE I/O
--------
I/O thread 0 state: waiting for i/o request
(insert buffer thread)
I/O thread 1 state: waiting for i/o request (log
thread)
I/O thread 2 state: waiting for i/o request (read
thread)
I/O thread 3 state: waiting for i/o request (write
thread)
Pending normal aio reads: 0, aio writes: 0,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
155 OS file reads, 4 OS file writes, 4 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s,
0.00 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf for space 0: size 1, free list len 314, seg
size 316,
0 inserts, 0 merged recs, 0 merges
Hash table size 138401, used cells 0, node heap
has 0 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number 0 900654168
Log flushed up to   0 900654168
Last checkpoint at  0 900654168
0 pending log writes, 0 pending chkp writes
9 log i/o's done, 0.00 log i/o's/second
----------------------
```

```
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 54384729; in additional
pool allocated 1167488
Buffer pool size   2048
Free buffers       1983
Database pages     65
Modified db pages  0
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 65, created 0, written 0
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
Main thread id 14344, state: waiting for server
activity
Number of rows inserted 0, updated 0, deleted 0,
read 0
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s,
0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
= = = = = = = = = = = = = = = = = = = = = = = = =
= = =

1 row in set (0.09 sec)
```

Appendix B. mytop

This appendix is a basic reference for Version 1.5 of *mytop*, a tool you can use to monitor various aspects of MySQL. *mytop* began as a simple Perl script that Jeremy wrote back in 2000 after getting sick of repeatedly running `SHOW FULL PROCESSLIST` and `SHOW STATUS` in an attempt to get a handle on what a MySQL was doing. After a bit of hacking on it, he realized that it would be useful it the tool felt a bit like the Unix *top* utility. Since then it has evolved to become quite a bit more popular and powerful. It is especially useful when tracking down problematic queries or trying to figure out what's keeping your server so busy.

mytop is an evolving piece of software. Be sure to check the

mytop web site (http://jeremy.zawodny.com/mysql/mytop/) for *mytop* news, downloads, and information about the mailing list. It's likely that new features have been added since Version 1.5.

Note that when discussing "queries"

in this chapter (and many other places in the book), we're doing so in a general sense:

`SELECT`, `INSERT`,

`UPDATE`, and `DELETE`.

B.1 Overview

mytop does much of the hard work involved in summarizing MySQL performance data. There are three primary display modes in *mytop*. The default, *thread view* (or *top view*), closely

resembles the Unix *top* command, as seen in Figure B-1. It produces a multiline summary at the top of the screen followed by a listing of threads in MySQL. The *command view* aggregates the data from

MySQL's `Com_*` command counters

(see Appendix A), as seen in Figure B-2. Finally, Figure B-3 illustrates *status view*, which tracks all the

other values in the output of `SHOW STATUS`. Like *top*, *mytop* refreshes the display periodically. The default *refresh interval* is five seconds, but that's easily adjusted.

Let's take a closer look at

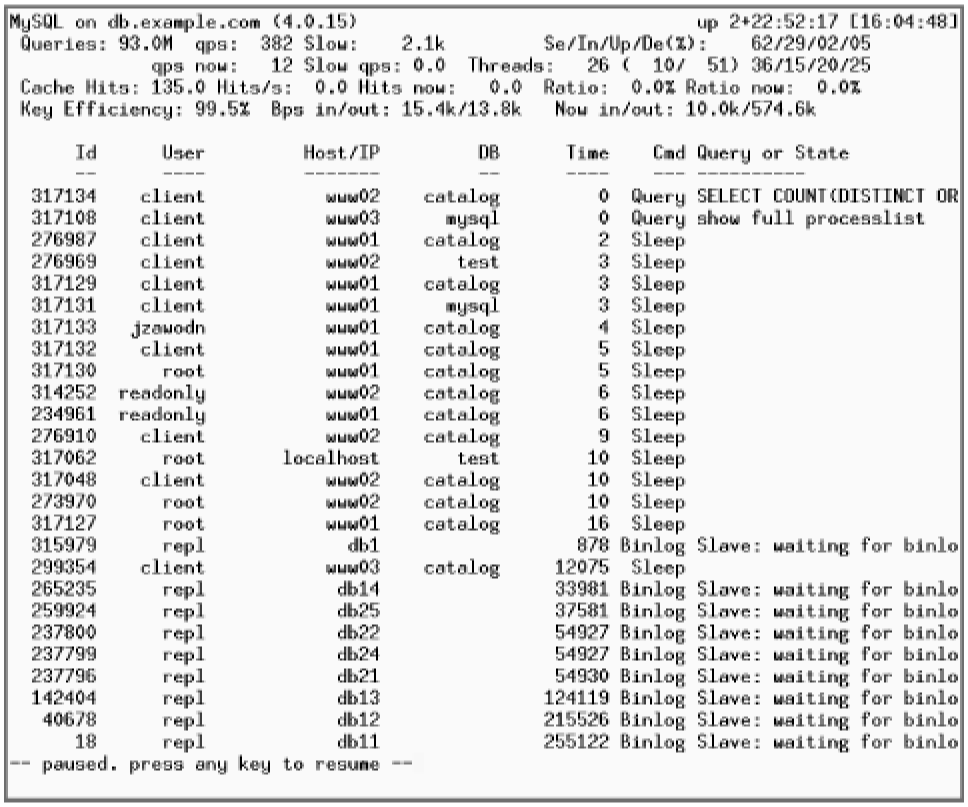
mytop's display modes.

B.1.1 Thread View

In *mytop*'s default display (Figure B-1), the first several lines of the screen are consumed by the *mytop* header. The first line identifies the hostname of the MySQL server as well as its version number. On the right side it displays the server's

uptime in Days+HH:MM:SS form followed by the current time.

Figure B-1. mytop thread view



```
MySQL on db.example.com (4.0.15)                              up 2+22:52:17 [16:04:48]
 Queries: 93.0M  qps:  382 Slow:    2.1k         Se/In/Up/De(%):    62/29/02/05
             qps now:   12 Slow qps: 0.0  Threads:   26 (  10/  51) 36/15/20/25
 Cache Hits: 135.0 Hits/s:  0.0 Hits now:   0.0  Ratio:  0.0% Ratio now:  0.0%
 Key Efficiency: 99.5%  Bps in/out: 15.4k/13.8k   Now in/out: 10.0k/574.6k

      Id      User         Host/IP         DB      Time    Cmd Query or State
      --      ----         -------         --      ----    --- ----------
  317134    client           www02    catalog         0  Query SELECT COUNT(DISTINCT OR
  317108    client           www03      mysql         0  Query show full processlist
  276987    client           www01    catalog         2  Sleep
  276969    client           www02       test         3  Sleep
  317129    client           www01    catalog         3  Sleep
  317131    client           www01      mysql         3  Sleep
  317133   jzawodn           www01    catalog         4  Sleep
  317132    client           www01    catalog         5  Sleep
  317130      root           www01    catalog         5  Sleep
  314252  readonly           www02    catalog         6  Sleep
  234961  readonly           www01    catalog         6  Sleep
  276910    client           www02    catalog         9  Sleep
  317062      root       localhost       test        10  Sleep
  317048    client           www02    catalog        10  Sleep
  273970      root           www02    catalog        10  Sleep
  317127      root           www01    catalog        16  Sleep
  315979      repl             db1                  878 Binlog Slave: waiting for binlo
  299354    client           www03    catalog     12075  Sleep
  265235      repl            db14                33981 Binlog Slave: waiting for binlo
  259924      repl            db25                37581 Binlog Slave: waiting for binlo
  237800      repl            db22                54927 Binlog Slave: waiting for binlo
  237799      repl            db24                54927 Binlog Slave: waiting for binlo
  237796      repl            db21                54930 Binlog Slave: waiting for binlo
  142404      repl            db13               124119 Binlog Slave: waiting for binlo
   40678      repl            db12               215526 Binlog Slave: waiting for binlo
      18      repl            db11               255122 Binlog Slave: waiting for binlo
-- paused. press any key to resume --
```

The next two lines display statistics about queries and threads. The first provides a cumulative count of the number of queries executed, followed by the average number of queries executed per second, then the total number of slow queries. Finally, the

`Se/In/Up/De(%)`: section displays the relative percentage of `SELECT`, `INSERT`,

`UPDATE`, and `DELETE` queries that

the server has executed.

The third line also displays queries per second (qps) and slow queries, but they reflect only queries executed during the last refresh interval. This line provides a count of the connected, running, and cached threads followed by

`Se/In/Up/De(%)`:. Again, those numbers reflect only the most recent queries.

The rest of the screen is used to display connected threads. Each thread listing shows the thread ID, username, database, hostname or IP address, time, and information about what state the thread is in.

Generally, threads are either idle (Sleep) or executing a query or command (Query). When a thread is executing a query, you see the beginning of the query in the rightmost column of the display. The time represents how long a thread has been in the same state. So if you see a thread with a time of 10 running a

`SELECT` query, that means the query has been running for 10 seconds. The display is also color-coded by default.

Idle threads are the default color, while yellow indicates a thread running a query, and green indicates a thread that is in the process of handing a new connection.

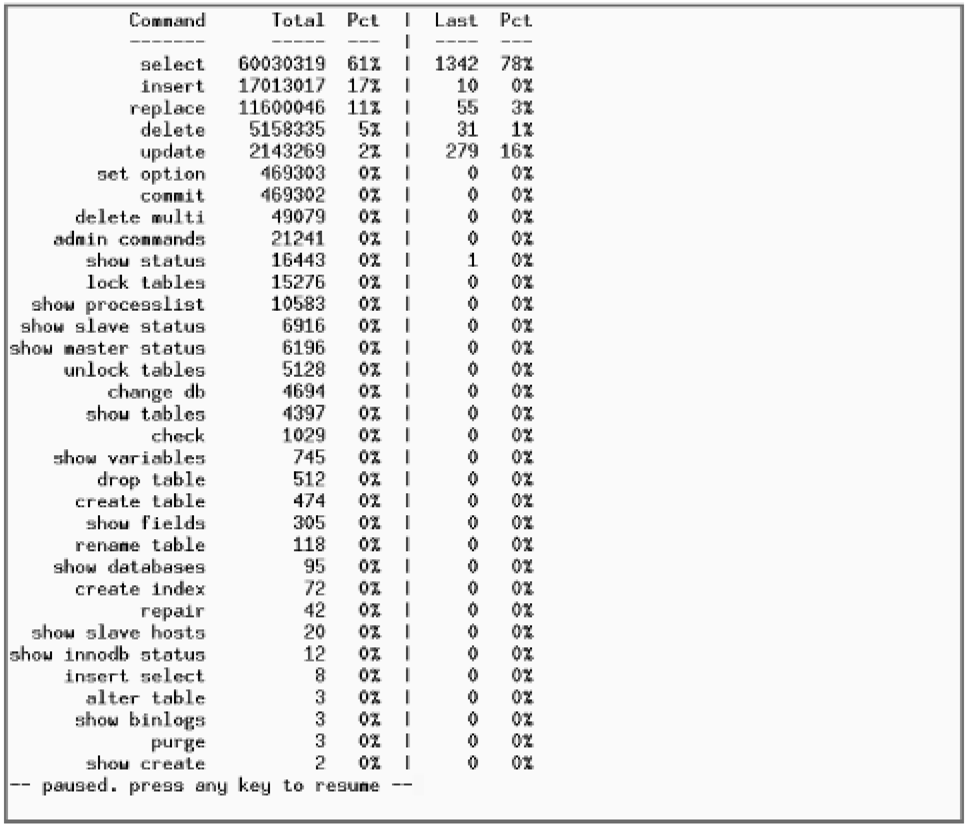
From here you can use *mytop*'s runtime keystrokes (see Table B-1 later in this chapter) to control its behavior and appearance. For example, by pressing f and entering the ID number of a thread, you can make *mytop* display the full query.

B.1.2 Command View

This view provides some insight into the relative number of times the server is asked to execute various queries or commands: `SELECT`, `INSERT`,

`UPDATE`, and so on. Figure B-2 shows an example of *mytop*'s command view.

Figure B-2. mytop command view



```
           Command       Total  Pct  |  Last  Pct
           -------       -----  ---  |  ----  ---
            select    60030319  61%  |  1342  78%
            insert    17013017  17%  |    10   0%
           replace    11600046  11%  |    55   3%
            delete     5158335   5%  |    31   1%
            update     2143269   2%  |   279  16%
        set option      469303   0%  |     0   0%
            commit      469302   0%  |     0   0%
      delete multi       49079   0%  |     0   0%
    admin commands       21241   0%  |     0   0%
       show status       16443   0%  |     1   0%
       lock tables       15276   0%  |     0   0%
  show processlist       10583   0%  |     0   0%
 show slave status        6916   0%  |     0   0%
show master status        6196   0%  |     0   0%
     unlock tables        5128   0%  |     0   0%
         change db        4694   0%  |     0   0%
       show tables        4397   0%  |     0   0%
             check        1029   0%  |     0   0%
    show variables         745   0%  |     0   0%
        drop table         512   0%  |     0   0%
      create table         474   0%  |     0   0%
       show fields         305   0%  |     0   0%
      rename table         118   0%  |     0   0%
    show databases          95   0%  |     0   0%
      create index          72   0%  |     0   0%
            repair          42   0%  |     0   0%
  show slave hosts          20   0%  |     0   0%
show innodb status          12   0%  |     0   0%
     insert select           8   0%  |     0   0%
       alter table           3   0%  |     0   0%
      show binlogs           3   0%  |     0   0%
             purge           3   0%  |     0   0%
       show create           2   0%  |     0   0%
-- paused. press any key to resume --
```

The first column lists the command being counted. Sometimes the commands map directly to queries (`SELECT`,

`INSERT`, etc.), while others represent a calls of commands. For example, `set option` covers all `SET` commands,

such as `SET AUTOCOMMIT=1`,

`SET GLOBAL.wait_timeout`, etc. Still others represent components of the MySQL C API. The best examples of this are `admin commands`, which represents the `ping` command (and a few others), and `change db`, which represents

`mysql_select_db( )` calls, including those

generated by the `USE` command.

The remaining columns measure the number of times each command has been executed, both in absolute and relative terms. The `Total` and `Pct` columns represent

the total number of command executions and the relative percentages of each. The second set of numbers, `Last` and `Pct`, do the same thing but consider only commands executed in the last refresh cycle

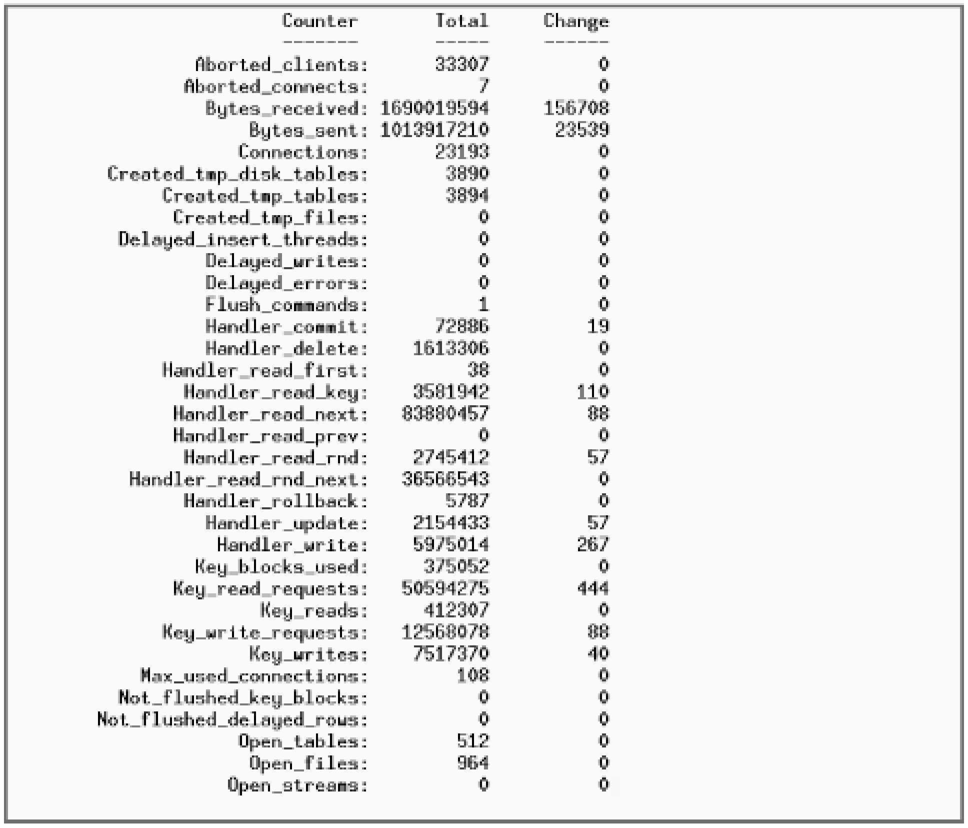
B.1.3 Status View

The newest view in *mytop* complements command view.

Status view summarizes the noncommand-related counters in `SHOW STATUS` output. Even without the command counters listed, there are quite a number of values (over 60 as of MySQL 4.0). To see them all,

you'll need a tall window. Figure B-3 shows an example on a moderately busy server.

Figure B-3. mytop status view



```
                  Counter       Total     Change
                  -------       -----     ------
         Aborted_clients:       33307          0
        Aborted_connects:           7          0
          Bytes_received:  1690019594     156708
              Bytes_sent:  1013917210      23539
             Connections:       23193          0
  Created_tmp_disk_tables:       3890          0
      Created_tmp_tables:        3894          0
       Created_tmp_files:           0          0
  Delayed_insert_threads:           0          0
          Delayed_writes:           0          0
          Delayed_errors:           0          0
          Flush_commands:           1          0
          Handler_commit:       72886         19
          Handler_delete:     1613306          0
      Handler_read_first:          38          0
        Handler_read_key:     3581942        110
       Handler_read_next:    83880457         88
       Handler_read_prev:           0          0
        Handler_read_rnd:     2745412         57
   Handler_read_rnd_next:    36566543          0
        Handler_rollback:        5787          0
          Handler_update:     2154433         57
           Handler_write:     5975014        267
         Key_blocks_used:      375052          0
       Key_read_requests:    50594275        444
               Key_reads:      412307          0
      Key_write_requests:    12568078         88
              Key_writes:     7517370         40
    Max_used_connections:         108          0
   Not_flushed_key_blocks:          0          0
 Not_flushed_delayed_rows:          0          0
             Open_tables:         512          0
              Open_files:         964          0
            Open_streams:           0          0
```

The `Total` column displays the current value of each counter, while the `Change` column contains the delta from the last refresh interval.

On a color display, positive changes are reported in yellow and negative in red. Unchanging values are shown in the default color. The i keystroke can be used in this view to filter out unchanging values from the display. This can be quite useful on small displays because many of the values aren't changing frequently.

```
$ <b>wget
http://jeremy.zawodny.com/mysql/mytop/mytop-
1.5.tar.gz</b> $ <b>tar zxvf mytop-1.5.tar.gz</b>

$ <b>cd mytop-1.5</b> mytop-1.5$ <b>perl
Makefile.PL</b> Checking if your kit is complete...
```

Looks good

```
Writing Makefile for mytop

mytop-1.5$ <b>make install</b> Installing
/usr/local/man/man1/mytop.1p Installing
/usr/bin/mytop

Writing /usr/local/lib/perl/5.8.0/auto/mytop/.packlist
Appending installation info to
/usr/local/lib/perl/5.8.0/perllocal.pod
```

Finally, try executing mytop to make sure it's installed properly along with all the prerequisites.

B.3 Configuration and Usage

mytop's behavior is controlled by a combination of command-line arguments, configuration file options, and runtime keystrokes. Most command-line arguments appear in single letter (`-p`) and longer GNU-style (`--password`) forms. Table B-1 lists the keystrokes, command-line arguments, configuration file directives, and the actions they perform.

Upon startup, *mytop* looks for a *~/.mytop*. If it finds one, it reads in the settings and uses them as defaults, which are then overridden by any command-line arguments. The configuration file format is composed of key/value pairs, one per line. A sample file might look like this: user=jzawodn pass=blah!db host=localhost

Most of the command-line arguments have a counterpart option in the configuration file. Future versions of *mytop* are expected to read MySQL's */etc/my.cnf* and *~/.my.cnf* as well, possibly deprecating *~/.mytop* at some point.

Table B-1. mytop configuration and control

Key	Argument(s)	Config file	Action
?			Display help screen
	`--batch` or `--batchmode`	`batchmode=1`	Run in batch (noninteractive) mode. Useful when called from *cron* or another script.
c	`-m=cmd` or `--mode=cmd`	`mode=cmd`	Command summary view.

Key	Argument(s)	Config file	Action
C	`--color` or `--nocolor`	color=[0|1]	Use colors in the display. (Requires the `Term::ANSIColor` module.) The key toggles color on/off.
d		filter_db=*`dbname`*	Show threads using one specific database.
	`-d` or `--database`	db=*`dbname`*	Connect to this database.
e			Explain the query a thread is running.
f			Show the full query a thread is executing.
F			Unfilter the display; return to defaults.
	`-h` or `--host`	`host=`*`hostname`*	Specify the host on which MySQL is running; default is localhost.
h			Show only connections from a particular host.

Key	Argument(s)	Config file	Action
H	`--header` or`--noheader`		Display the header *mytop*'s display (key toggles the header display).
i	`-i` or `--idle`	`idle=[0|1]`	Filter idle (sleeping) threads from the display. Key toggles this.
I	`-m`= or `--mode=innodb`	`mode=innodb`	Show InnoDB status.
k			Kill a thread.
m	`-m`= or `--mode=[qps|top|cmd|innodb]`	`mode=[qps|top|cmd|innodb]`	Mode switch. Cycle between thread view, queries per second, and command summary.
o	`--sort=[0|1]`	`sort=[0|1]`	Reverse the sort order. Default is ascending based on time.
p			Pause the display. Any key resumes.
	`-p` or `--password`	`pass=`*`password`*	Connect using this password.

Key	Argument(s)	Config file	Action
	`--prompt`	`prompt=[0|1]`	Prompt for password interactively.
q			Quit *mytop*.
r			Reset status counters (via `FLUSH STATUS`).
R	`-r` or `--resolve`	`resolve=[0|1]`	Resolve IP addresses into hostnames. This is useful when MySQL is configured with skip-name-resolve.
s	`-s` or `--delay`	`delay=`*`number`*	Adjust the refresh interval.
S	`-m= --mode=status`	`mode=status`	Switch to `SHOW STATUS` mode.
	`-S` or `--socket`	`socket=`*`/path/to/socket`*	Specify the socket to use when connecting to localhost.
t	`-m=` or `--mode=top`	`mode=top`	Switch to thread view (the default).

Key	Argument(s)	Config file	Action
u		`filter_user=`*`username`*	Show only a particular user's threads.
	-`u` or --`user`	`user=`*`username`*	Connect as this user.
V			Switch to `SHOW VARIABLES` mode.
:			Enter a complex command.

```
Se/In/Up/De(%): 61/30/02/05

   ... 63/07/12/10
```

The first line means that, overall, 61% of the server's queries are `SELECT`s, 30% are `INSERT`s, 2% are `UPDATE`s, and 5% are `DELETE`s. The second line displays values that apply to the last refresh interval (5 seconds by default) only. The two together can give a quick feel for what your server has been doing recently and how that compares to the longer term average.

If you want more detail, press c to switch mytop into command view. There you'll find detailed counts and percentages for each type of command or query executed. The first column of numbers summarizes overall counts (since the server was started or counters reset), while the second set of numbers reflects the last refresh interval only.

Kill a group of queries

Use mytop's "super-kill" feature by pressing K. You'll be prompted for a username, and mytop will then kill all of that user's threads. In the future this may be extended to evaluate more complex expressions, such as killing all nonidle threads from a given hostname or IP address.

Limit the display to a particular user or host

You can ask mytop to filter out all threads except those from a given host or those owned by a given user. If you press u, mytop prompts for a username to filter on. Similarly, pressing h allows you to provide a hostname or IP address which is used to filter the display. If you supply both, mytop restricts the display based on both criteria.

To clear the filtering, you can press F to remove all filters at once. Otherwise, you can use the u or h keys to remove either of the filters manually.

Appendix C. phpMyAdmin

There are a number of third-party user interfaces to MySQL that make it easier to access and alter the data stored in your MySQL databases. The most popular of these, by far, is *phpMyAdmin*, a web-based application written in PHP.

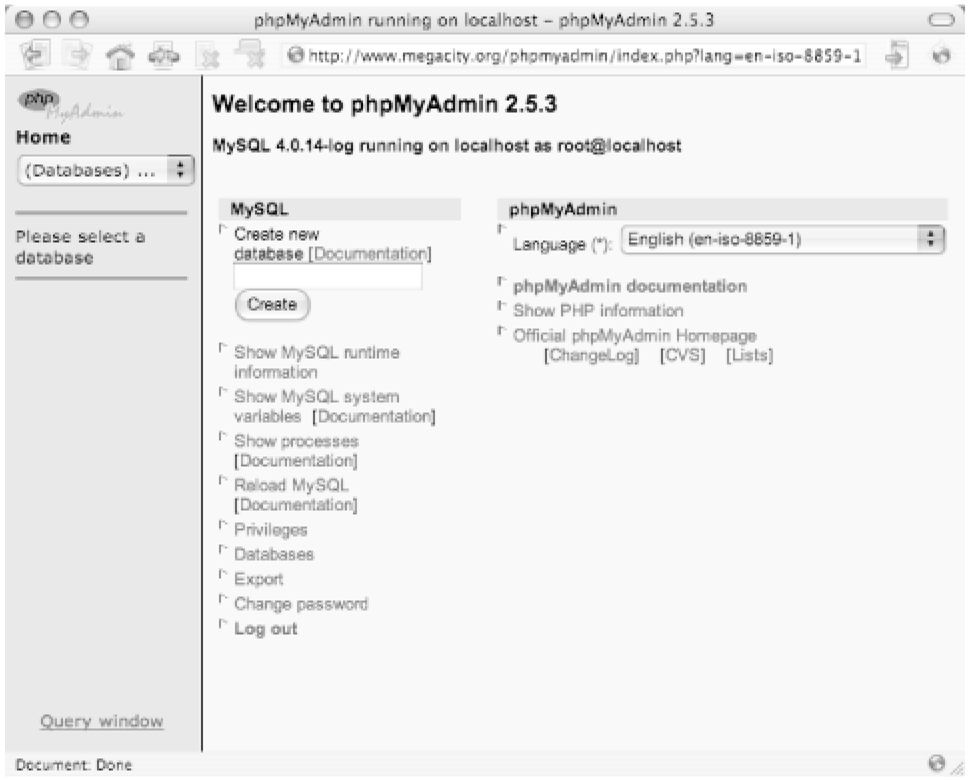
To install *phpMyAdmin*, you need first make sure you have a web server running PHP 4.x or later that either includes or has been configured to include MySQL database support. You will also need network connectivity to a MySQL server, even if that MySQL server happens to be on the same host as the web server running *phpMyAdmin*. The *phpMyAdmin* package can be downloaded from http://www.phpmyadmin.net/, or your Unix/Linux distribution might make a binary package available through its native package management system. Debian Linux users, for example, can simply run `apt-get install phpmyadmin`.

C.1 The Basics

To use *phpMyAdmin* to access your database, you need a username and password that are valid for connections from your web server. Your web server might be on the same machine as your MySQL server, in which case, obviously, the user only needs to be able to access the server from *localhost*.

Once you have logged in using a valid user account, you will see something that looks like Figure C-1.

Figure C-1. phpMyAdmin start page


phpMyAdmin running on localhost – phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/index.php?lang=en-iso-8859-1
Home
(Databases) ...
Please select a database
Query window
Document: Done
MySQL
Create new database [Documentation]
Create
Show MySQL runtime information
Show MySQL system variables [Documentation]
Show processes [Documentation]
Reload MySQL [Documentation]
Privileges
Databases
Export
Change password
Log out
phpMyAdmin
Language (*): English (en-iso-8859-1)
phpMyAdmin documentation
Show PHP information
Official phpMyAdmin Homepage
[ChangeLog] [CVS] [Lists]

Welcome to phpMyAdmin 2.5.3

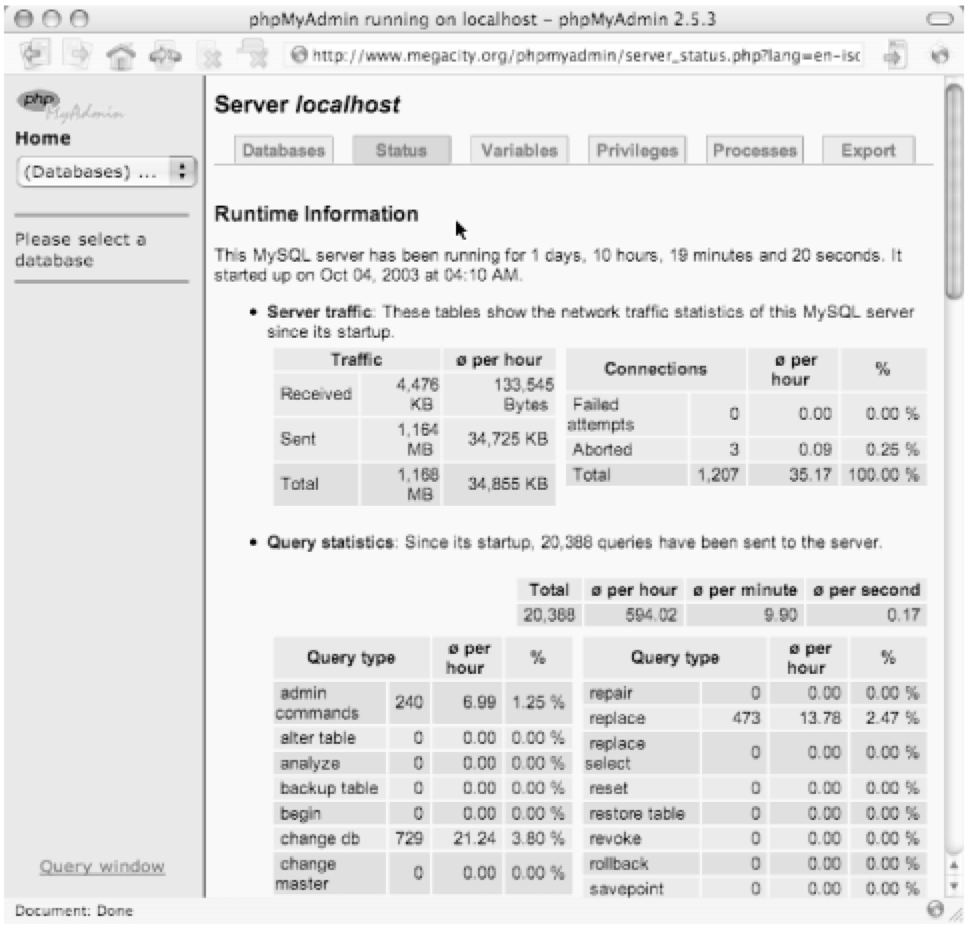
MySQL 4.0.14-log running on localhost as root@localhost

As you can see in Figure C-2, there are some links to basic server information. Via the Status link,

phpMyAdmin provides a way to see the status of

your server without logging into it and issuing commands via a command-line interface.

Figure C-2. phpMyAdmin Runtime Information screen



Server *localhost*

Runtime Information

This MySQL server has been running for 1 days, 10 hours, 19 minutes and 20 seconds. It started up on Oct 04, 2003 at 04:10 AM.

- **Server traffic**: These tables show the network traffic statistics of this MySQL server since its startup.

Traffic		ø per hour
Received	4,476 KB	133,545 Bytes
Sent	1,164 MB	34,725 KB
Total	1,168 MB	34,855 KB

Connections		ø per hour	%
Failed attempts	0	0.00	0.00 %
Aborted	3	0.09	0.25 %
Total	1,207	35.17	100.00 %

- **Query statistics**: Since its startup, 20,388 queries have been sent to the server.

Total	ø per hour	ø per minute	ø per second
20,388	594.02	9.90	0.17

Query type		ø per hour	%
admin commands	240	6.99	1.25 %
alter table	0	0.00	0.00 %
analyze	0	0.00	0.00 %
backup table	0	0.00	0.00 %
begin	0	0.00	0.00 %
change db	729	21.24	3.80 %
change master	0	0.00	0.00 %

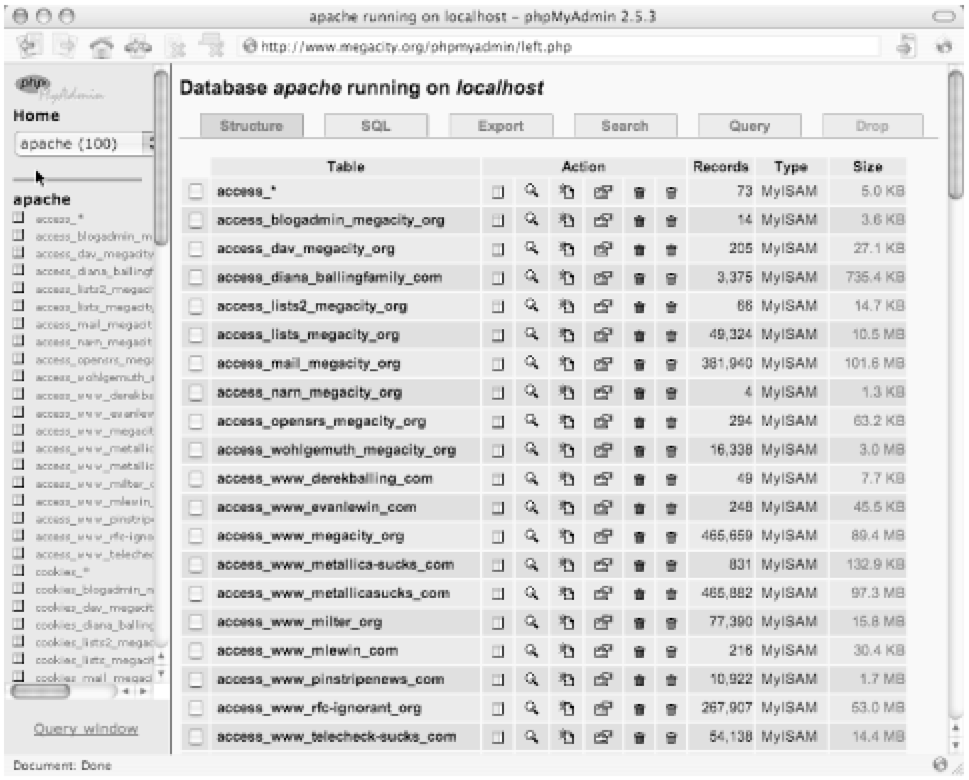
Query type		ø per hour	%
repair	0	0.00	0.00 %
replace	473	13.78	2.47 %
replace select	0	0.00	0.00 %
reset	0	0.00	0.00 %
restore table	0	0.00	0.00 %
revoke	0	0.00	0.00 %
rollback	0	0.00	0.00 %
savepoint	0	0.00	0.00 %

To drill into a specific database, the first step is to select the database name from the pull-down menu on the left menu bar.

phpMyAdmin then displays all the tables within

that database, as shown in Figure C-3. This page is extremely useful at a quick glance for checking the relative sizes of your tables, which storage engine is used for each table, and the number of records contained in each.

Figure C-3. phpMyAdmin after selecting a database


apache running on localhost – phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/left.php
Home
apache (100)
apache
Query window
Document: Done
Structure
SQL
Export
Search
Query
Drop

Database *apache* running on *localhost*

Table	Records	Type	Size
access_*	73	MyISAM	5.0 KB
access_blogadmin_megacity_org	14	MyISAM	3.6 KB
access_dav_megacity_org	205	MyISAM	27.1 KB
access_diana_ballingfamily_com	3,375	MyISAM	735.4 KB
access_lists2_megacity_org	66	MyISAM	14.7 KB
access_lists_megacity_org	49,324	MyISAM	10.5 MB
access_mail_megacity_org	381,940	MyISAM	101.6 MB
access_narn_megacity_org	4	MyISAM	1.3 KB
access_opensrs_megacity_org	294	MyISAM	63.2 KB
access_wohlgemuth_megacity_org	16,338	MyISAM	3.0 MB
access_www_derekballing_com	49	MyISAM	7.7 KB
access_www_evanlewin_com	248	MyISAM	45.5 KB
access_www_megacity_org	465,659	MyISAM	89.4 MB
access_www_metallica-sucks_com	831	MyISAM	132.9 KB
access_www_metallicasucks_com	465,882	MyISAM	97.3 MB
access_www_milter_org	77,390	MyISAM	15.8 MB
access_www_mlewin_com	216	MyISAM	30.4 KB
access_www_pinstripenews_com	10,922	MyISAM	1.7 MB
access_www_rfc-ignorant_org	267,907	MyISAM	53.0 MB
access_www_telecheck-sucks_com	54,138	MyISAM	14.4 MB

A step-by-step tutorial in how to use *phpMyAdmin* is outside the scope of this book, but we'd like to

show you some common examples of where you might find it useful to have *phpMyAdmin* installed because it can make

your job as the database administrator significantly easier, or at least faster. It can also allow you to grant people access to issue raw SQL commands and perform maintenance

without actually giving them a login on the
machine or requiring them to use the MySQL

command-line interface.

C.2 Practical Examples

In the rest of this appendix we'll describe how to use *phpMyAdmin* to accomplish some common tasks.

C.2.1 User Maintenance

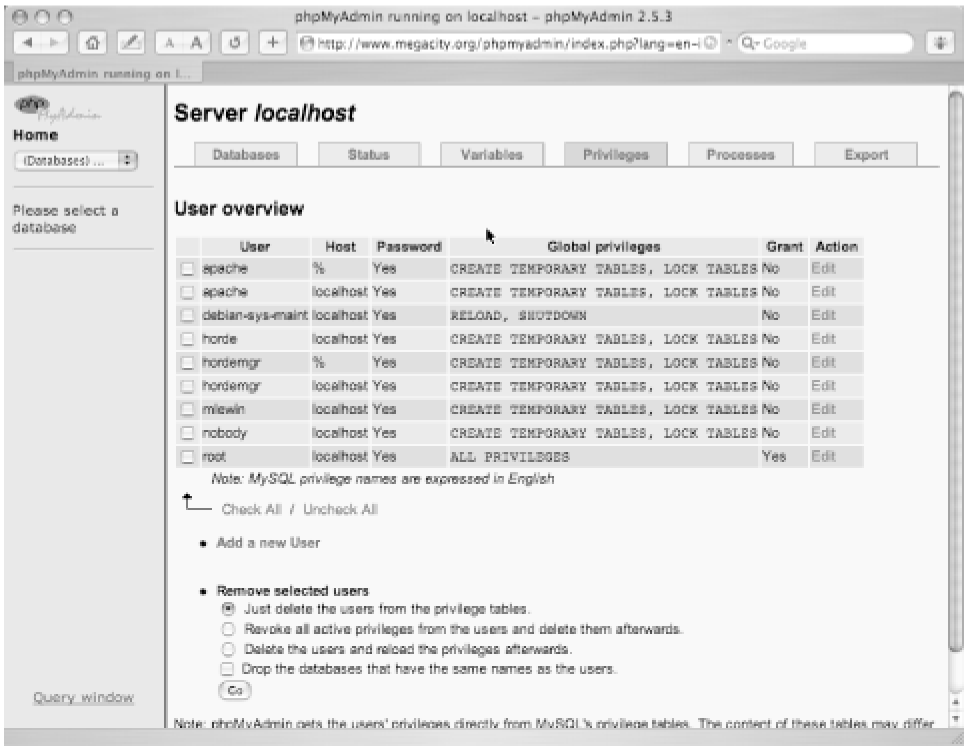
User maintenance in *phpMyAdmin* functions much as it does in

command-line MySQL operation. The administrator can deal with global as well as database and table-level privileges.

To administer global privileges, select Privileges from the start page, which then displays something similar to Figure C-4. Once you have chosen a user to add or remove privileges from, click the "edit"

link, which will present you with a user editing interface, as shown in Figure C-5.

Figure C-4. phpMyAdmin global privileges interface



Server *localhost*

User overview

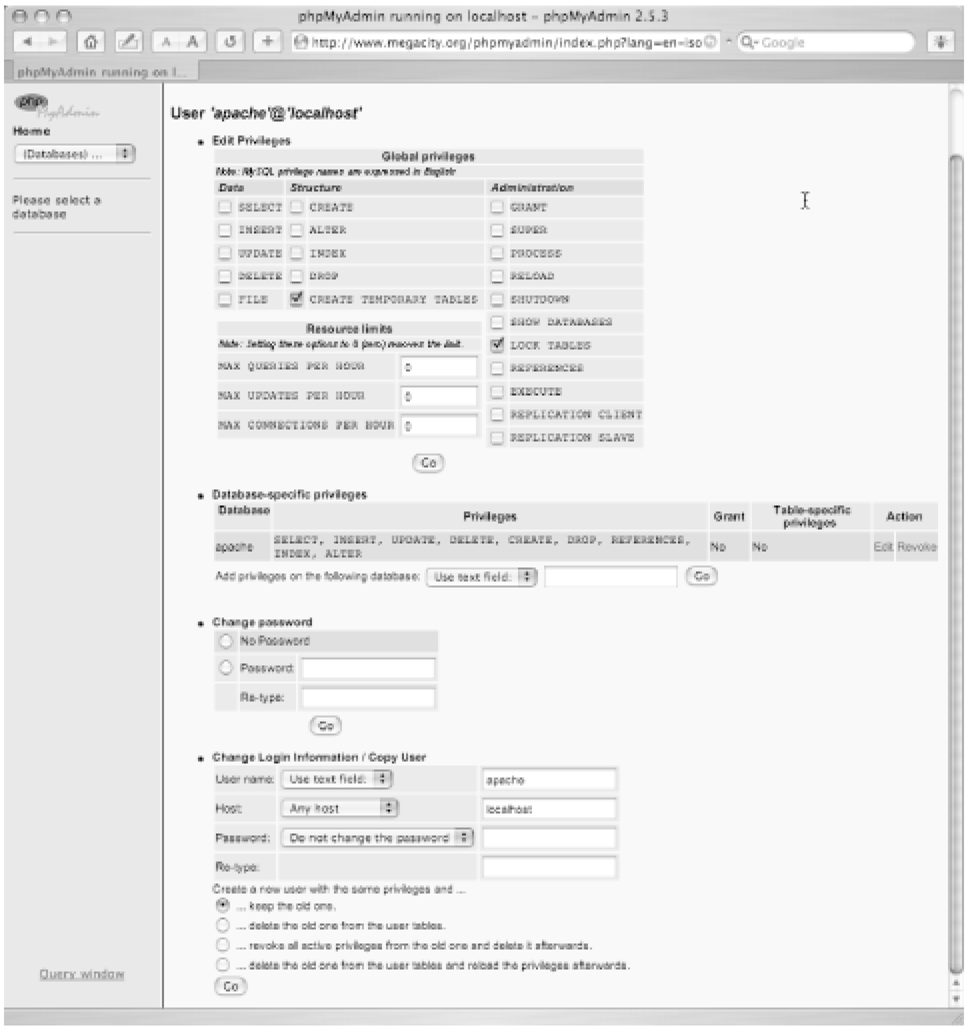
	User	Host	Password	Global privileges	Grant	Action
☐	apache	%	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	apache	localhost	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	debian-sys-maint	localhost	Yes	RELOAD, SHUTDOWN	No	Edit
☐	horde	localhost	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	hordemgr	%	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	hordemgr	localhost	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	mlewin	localhost	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	nobody	localhost	Yes	CREATE TEMPORARY TABLES, LOCK TABLES	No	Edit
☐	root	localhost	Yes	ALL PRIVILEGES	Yes	Edit

Note: MySQL privilege names are expressed in English

- **Remove selected users**
  - [✓ Just delete the users from the privilege tables.]
  - Revoke all active privileges from the users and delete them afterwards.
  - Delete the users and reload the privileges afterwards.
  - ☐ Drop the databases that have the same names as the users.

Note: phpMyAdmin gets the users' privileges directly from MySQL's privilege tables. The content of these tables may differ

Figure C-5. phpMyAdmin user edit interface


User 'apache'@'localhost'

- Edit Privileges

Global privileges		
Note: MySQL privilege names are expressed in English		
Data	*Structure*	*Administration*
☐ SELECT	☐ CREATE	☐ GRANT
☐ INSERT	☐ ALTER	☐ SUPER
☐ UPDATE	☐ INDEX	☐ PROCESS
☐ DELETE	☐ DROP	☐ RELOAD
☐ FILE	✓ CREATE TEMPORARY TABLES	☐ SHUTDOWN
		☐ SHOW DATABASES
		✓ LOCK TABLES
		☐ REFERENCES
		☐ EXECUTE
		☐ REPLICATION CLIENT
		☐ REPLICATION SLAVE

Resource limits	
Note: Setting these options to 0 (zero) removes the limit.	
MAX QUERIES PER HOUR	0
MAX UPDATES PER HOUR	0
MAX CONNECTIONS PER HOUR	0

Go

- Database-specific privileges

Database	Privileges	Grant	Table-specific privileges	Action	
apache	SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, REFERENCES, INDEX, ALTER	No	No	Edit	Revoke

Add privileges on the following database: Use text field: Go

- Change password

○	No Password	
○	Password:	
	Re-type:	

Go

- Change Login Information / Copy User

User name:	Use text field:	apache
Host:	Any host	localhost
Password:	Do not change the password	
Re-type:		

Create a new user with the same privileges and ...

● ... keep the old one.
○ ... delete the old one from the user tables.
○ ... revoke all active privileges from the old one and delete it afterwards.
○ ... delete the old one from the user tables and reload the privileges afterwards.

Go

From here, most of the functions are self-explanatory; they allow administrators to add or remove global privileges, edit any

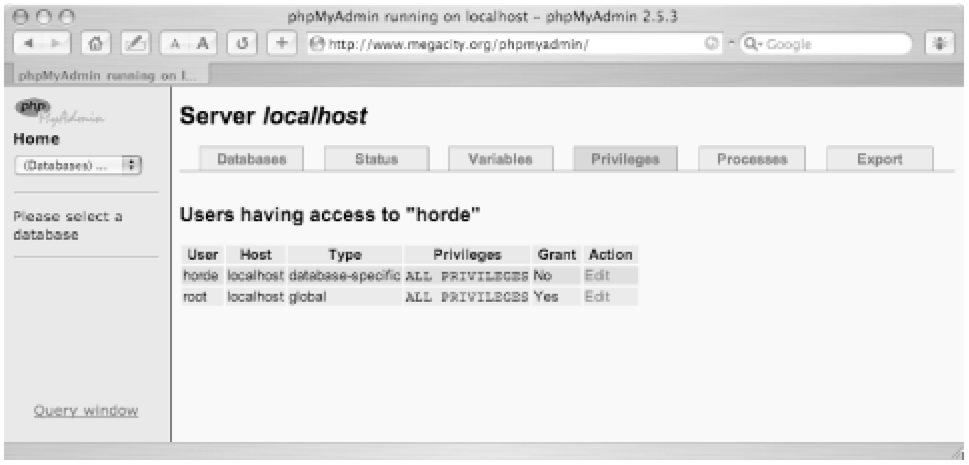
table-specific privileges the user may have, or change the

user's password. You can also clone the user,

creating one with the same privileges as the original user, but a different username. This can be handy for adding new database administrators or other users who have complicated privileges.

Perhaps you wish to see which users have access to a particular database. From the databases list, select Check Privileges next to the database you want to check for access on. A list of all users and the privileges granted to them will be displayed (along with links for editing those privileges), as shown in Figure C-6.

Figure C-6. phpMyAdmin database level privileges interface


phpMyAdmin running on localhost – phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/
Google
phpMyAdmin running on l...
Home
(Databases) ...
Please select a database
Query window
Databases
Status
Variables
Privileges
Processes
Export

Server *localhost*

Users having access to "horde"

User	Host	Type	Privileges	Grant	Action
horde	localhost	database-specific	ALL PRIVILEGES	No	Edit
root	localhost	global	ALL PRIVILEGES	Yes	Edit

C.2.2 Simple SQL Commands

Often, as an administrator, you will want to give users the ability to issue simple SQL commands against the database, but you don't necessarily want to open

up the server to login accounts in general or to give a

"non-sysadmin" user the ability to

get a login shell. In these types of situations, allowing the user in question to use the *phpMyAdmin* interface may be

the ideal solution.

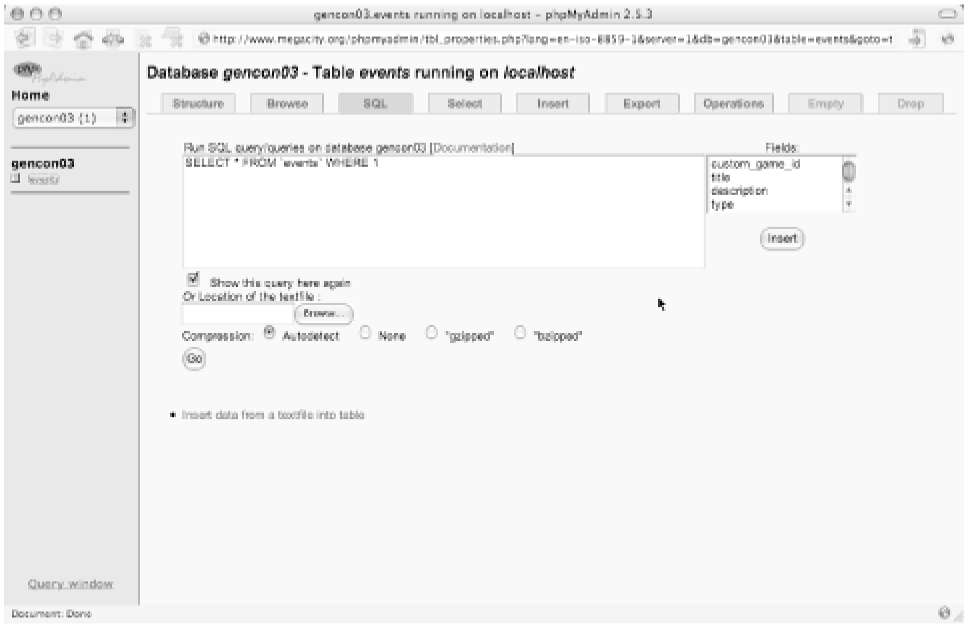
There are two basic interfaces available to the user for issuing SQL

commands. One of these is a very simple, raw text area that allows the users to type in the SQL command they wish to execute. Simply click on the SQL tab after selecting a table to work with.

There is some helpful JavaScript magic on this page that allows the user to select column names from a `select` element

to the right of the free-form text area, so that the user can minimize his typing, as shown in Figure C-7.

Figure C-7. phpMyAdmin SQL interface


gencon03.events running on localhost - phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/tbl_properties.php?lang=en-iso-8859-1&server=1&db=gencon03&table=events&goto=t
Home
gencon03 (1)
gencon03
events
Database *gencon03* - Table *events* running on *localhost*
Structure
Browse
SQL
Select
Insert
Export
Operations
Empty
Drop
Fields:
custom_game_id
title
description
type
Insert
Browse...
Go
Insert data from a textfile into table
Query window
Document: Done

Run SQL query/queries on database gencon03 [Documentation]

```
SELECT * FROM `events` WHERE 1
```

[✓] Show this query here again

Or Location of the textfile :

Compression: [✓ Autodetect] None "gzipped" "bzipped"

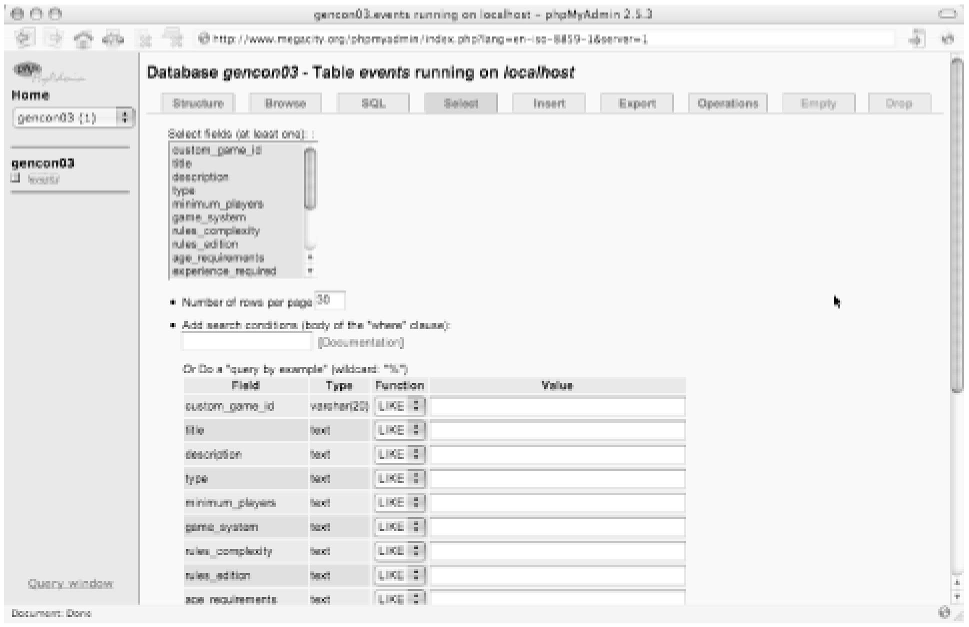
Once executed, the resulting records is displayed as returned by the user's query. It is possible to edit, or delete an

entry by clicking the note-pad button or the trash-can button, respectively, next to the record you wish to edit or delete.

The other basic record selection method is the Select tab. This is designed for simple queries and allows you to impose simple

restrictions on the query being performed, as shown in Figure C-8. The results of that query are displayed in the same format as the results of the SQL query, and likewise allow the user to edit or delete specific entries returned by the query.

Figure C-8. phpMyAdmin select interface


gencon03.events running on localhost – phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/index.php?lang=en-iso-8859-1&server=1
Home
gencon03 (1)
gencon03
events
Query window
Document: Done
Structure
Browse
SQL
Select
Insert
Export
Operations
Empty
Drop

Database *gencon03* - Table *events* running on *localhost*

Select fields (at least one): :

custom_game_id
title
description
type
minimum_players
game_system
rules_complexity
rules_edition
age_requirements
experience_required

- Number of rows per page 30
- Add search conditions (body of the "where" clause):
  [Documentation]

Or Do a "query by example" (wildcard: "%")

Field	Type	Function	Value
custom_game_id	varchar(20)	LIKE	
title	text	LIKE	
description	text	LIKE	
type	text	LIKE	
minimum_players	text	LIKE	
game_system	text	LIKE	
rules_complexity	text	LIKE	
rules_edition	text	LIKE	
age_requirements	text	LIKE	

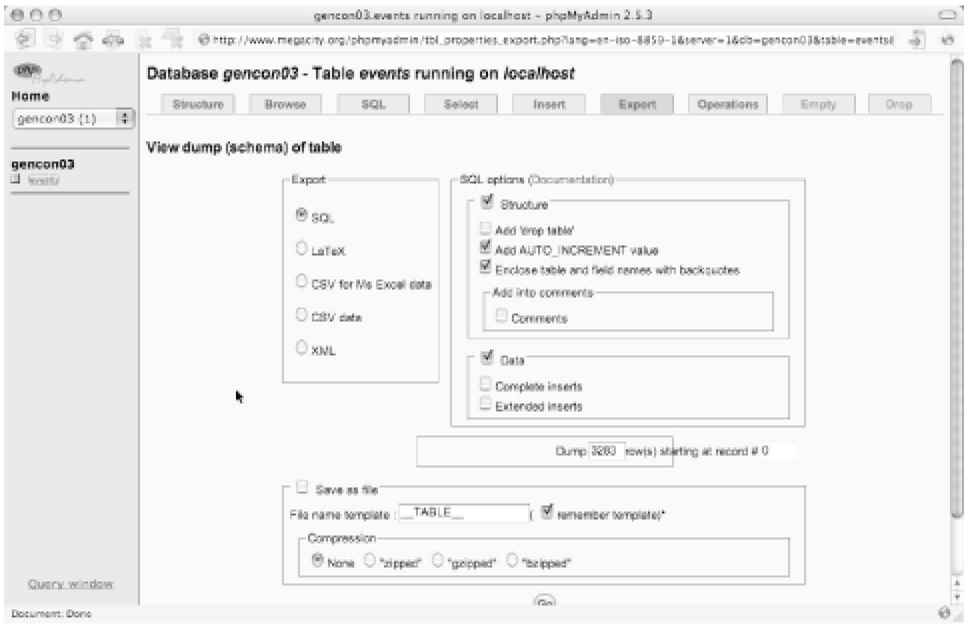
C.2.3 Exporting and Downloading Data

The

phpMyAdmin interface makes retrieving remote dumps of the database as easy as clicking some buttons on a web form. There are two different Export tabs, one if you are viewing the database as a whole, the other if you are looking at a selected table. They are virtually identical except that the database-wide version also includes a `select`

item for which tables in the database you wish to export. The table-wide interface can be seen in Figure C-9.

Figure C-9. phpMyAdmin export interface


gencon03.events running on localhost - phpMyAdmin 2.5.3
http://www.megacity.org/phpmyadmin/tbl_properties_export.php?lang=en-iso-8859-1&server=1&db=gencon03&table=events&
Home
gencon03 (1)
gencon03
events
Structure
Browse
SQL
Select
Insert
Export
Operations
Empty
Drop
Query window
Document: Done
Go

Database *gencon03* - Table *events* running on *localhost*

View dump (schema) of table

Export

[✓ SQL] LaTeX CSV for Ms Excel data CSV data XML

SQL options (Documentation)

- [✓] Structure
  - [ ] Add 'drop table'
  - [✓] Add AUTO_INCREMENT value
  - [✓] Enclose table and field names with backquotes
  - Add into comments
    - [ ] Comments
- [✓] Data
  - [ ] Complete inserts
  - [ ] Extended inserts

Dump 3283 row(s) starting at record # 0

- [ ] Save as file
  - File name template : __TABLE__ ([✓] remember template)*
  - Compression: [✓ None] "zipped" "gzipped" "bzipped"

As you can see, there are several export options available to the user. If you are looking for a *mysqldump*-style

export for possible import into another MySQL installation, you can select that option. There are CSV options for

"normal" use as well as customized

CSV output to make Microsoft Excel happy.

The different export options each enable different options in the panel just to the right, specific to the export style in question.

Once you select Go, the data will be formatted per your selection and output to your browser via the Web, where you can copy and paste it or save it to your local disk. Alternatively, you may check the "Save as file" checkbox and simply

save the downloaded file to disk. Note that you might see slightly odd behavior, though, if you use this feature to export as XML. Your browser of choice may decide to try to

"handle" the XML by displaying it,

instead of allowing you to simply save it to disk.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *High Performance MySQL: Optimization, Backups, Replication, and Load Balancing,* is a sparrow hawk (*Accipiter nisus*), a small woodland member of the falcon family found in Eurasia and North Africa. Sparrow hawks have a long tail and short wings; males are bluish-grey with a light brown breast, and females are more brown-grey and have an almost fully white breast. Males are normally somewhat smaller (11 inches) than females (15 inches).

Sparrow hawks live in coniferous woods and feed on small mammals, insects, and birds. They nest in trees and sometimes on cliff ledges. At the beginning of the summer, the female lays 4 to 6 white eggs, blotched red and brown, in a nest made in the boughs of the tallest tree available. The male feeds the female and their young.

Like all hawks, the sparrow hawk is capable of bursts of high speed in flight. Whether soaring or gliding, the sparrow hawk has a characteristic flap-flap-glide action; its large tail enables the hawk to twist and turn effortlessly in and out of cover.

Mary Anne Weeks Mayo was the production editor and proofreader, and Leanne Soylemez was the copyeditor for *High Performance MySQL: Optimization, Backups, Replication, and Load Balancing* . Emily Quill and Claire Cloutier provided quality control. Jamie

Peppard and Mary Agner provided production assistance. John Bickelhaupt wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Joe Wizda to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was compiled by Mary Anne Weeks Mayo.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.

automatic host blocking

average employee account privileges

[I]

[J]

[K]

[L]

[M]

[N]

[O]

[P]

[Q]

[R]

CREATE and SELECT table conversion method

creation id

[I]

[J]

[K]

[L]

[M]

[N]

[O]

[P]

[Q]

[R]

[I]

[J]

[K]

[L]

[M]

[N]

[O]

[P]

[Q]

[R]

NULLs

primary keys and

[I]

[J]

[K]

[L]

[M]

[N]

[**O**]

[P]

[Q]

[R]

unions versus ORs

system versions