WILEY

TIMELY. PRACTICAL. RELIABLE.

MySQL™ Enterprise Solutions

Alexander Sasha Pachev

MySQL Enterprise Solutions

Alexander “Sasha” Pachev

Wiley Publishing, Inc.

MySQL Enterprise Solutions

Alexander “Sasha” Pachev

Wiley Publishing, Inc.

Publisher: Robert Ipsen
Editor: Robert M. Elliott
Managing Editor: Vincent Kunkemueller
Book Producer: Ryan Publishing Group, Inc.

Copyeditors: Elizabeth Welch and Tiffany Taylor
Proofreader: Nancy Sixsmith
Compositor: Gina Rexrode

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:
Pachev, Alexander, 1973-
MySQL enterprise solutions / Alexander Pachev.
p. cm.
"Wiley Computer Publishing."
Includes index.
ISBN 0-471-26922-0
1. SQL (Computer program language) I. Title.
QA76.3.S67 P33 2003
005.75'65—dc21
2002153143

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Part II Developing MySQL Applications

Dedication

This book is dedicated to my wife Sarah, and my children Benjamin, Jenny, and Julia.

Acknowlegments

This book could not have been written without the participation of several parties who I would like to thank. The acknowledgments are not given in any particular order of importance as I regard everyone as an equal participant, a part of the big whole without whom the whole would not be the same, or not even exist in the first place.

Tim Ryan saw a potential author in me, approached me with the idea to write this book, and guided me through refining the rough edges of my writing.

Special thanks to Monty Widenius for creating MySQL and to David Axmark for convincing Monty to make it available to everyone, which eventually permitted me to get involved with it.

MySQL AB allowed me to take some time off to write the book. Special thanks to Marten Mickos, CEO of MySQL AB, Tom Basil, the Director of Customer Support, Larry Stefonic, the VP of Sales, and other employees of MySQL AB for their support.

Special acknowledgments to Joe Gradecki, who reviewed several chapters in the book and contributed some additional content.

My wife and children have given me a lot of support and deserve more credit than I could possibly express in writing. My wife Sarah was pregnant with our third child Julia throughout most of the work, and had to take care of Benjamin (3), and Jenny (2) on her own while I was locked up in the office writing. She went to great lengths to make sure the little things that took me a long time to do got done so I could have enough time to write the book. Every night, Benjamin and Jenny patiently waited for Daddy to finish his work for the day before he could get out of the office and play with them.

ABOUT THE AUTHOR

Alexander "Sasha" Pachev has been working with MySQL AB since 1999. His most significant contribution was the development of the master-slave replication functionality for MySQL server. He has added several other features to the server; and provided high-level e-mail, login, and phone support for MySQL customers, presales support, and on-site consulting. During his work with MySQL, he has established a reputation for being able to quickly track down and fix difficult bugs. He enjoys solving difficult problems, and his motto is "Let's make it work!"

Prior to his work at MySQL AB, Sasha was employed by several Internet start-ups. There, he developed his love and appreciation for MySQL as he used it to create solutions that allowed the company to meet its needs without spending a lot of money in software licensing fees.

Raised in Moscow, Russia, Sasha came to Provo, Utah in 1993 to study at the Brigham Young University. After taking a two year break to serve a mission for the LDS church between 1994–96, Sasha graduated from BYU with a Bachelor's degree in Computer Science in 1998. While at BYU, Sasha married his wife, Sarah Matthews, and they are parents of three children: Benjamin, Jenny, and Julia. Sasha and his family live in Provo, Utah.

Sasha is an avid distance runner, and his favorite distance is the full-length marathon (26.2 miles). He has won the Boise Marathon twice, placed in the top in many other races, and has a personal best of 2:33:20. His next goal is to qualify for the U.S. Olympic Trials, which would give him a shot at making the U.S. Olympic Team.

Introduction

Several months ago, Tim Ryan approached me with an invitation to write a book for experienced database programmers and administrators who were in the process of adopting MySQL. I initially hesitated, as my hands were already full with my job at MySQL AB, family responsibilities, serving in my church, and training in an attempt to qualify for the U.S. Olympic trials in the marathon. Nevertheless, after some prayer followed by a discussion with my wife, Sarah, I knew that writing a book would be the right thing to do. I accepted the offer, and made the attempt to communicate in writing the knowledge and experience I have accumulated working with MySQL as a user and client programmer, and later on as a server programmer and support engineer.

This book is meant to be a practical guide for anyone deploying a mission-critical application that uses MySQL as a database backend. No book can answer every question on a subject, but it is my hope that for the questions it does not answer, it will give you enough background and direction to enable you to find the answer quickly through your own research. As such, this book is not a substitute—rather it is a companion—for the MySQL online manual (www.mysql.com/doc).

Although I am an employee of MySQL AB, which undoubtedly affects my perception of the subject, the views and opinions stated in this book are my own and do not necessarily represent the official position of the company. The advice in this book is based on my own experience working with MySQL and with other MySQL users. I have made every effort to stick as close as possible to the facts available to me. It is my hope that you will be able to see the issues as they really are, so that you can decide whether to adopt and deploy MySQL, and if so, the best way to go about it.

Prominent Users of MySQL

MySQL has penetrated the enterprise in a way that perhaps would be odd for a typical proprietary application, but is very common for an open-source package. Developers brought it in through the back door, and sometimes by the time management found out, developers had a working solution based on MySQL with zero database licensing costs. This often won acceptance for MySQL at both the management and development levels.

Some prominent users of MySQL have publicly released the fact that they are using MySQL and somewhat elaborated on their use, which allows us to talk about them in this book. Others are using it in production—frequently under heavy load—but treat this information as a trade secret. The organizations I mention in this section have made their use of MySQL public, so we can discuss a few details about their implementations. The companies listed here are the tip of the iceberg.

MySQL AB collects user stories and publishes them at www.mysql.com/press/user_stories/, so you might want to check there for more information.

Yahoo! Finance

Yahoo! Finance (finance.yahoo.com) uses MySQL to power a portion of the Web site. The database contains a total of 25GB, with the largest table containing over 274 million records and 8GB of data. The platform is x86 Linux and FreeBSD. The setup is replicated: one master and three slaves. The master is the most heavily loaded, and at peak times processes over 1,200 queries per second with the read/write ratio of 70/30.

NASA

NASA is using MySQL as a backend for the NASA Acquisition Internet Service (NAIS) site (nais.nasa.gov). This system has been reported to handle several thousand users and is receiving 300,000 hits per month. The database runs on Sparc Solaris. While the load and the database size is far below the top capacity of MySQL, NASA has been very pleased with the cost reduction and improved performance since it migrated from Oracle.

U.S. Census Bureau

The U.S. Census Bureau provides access to census information through three sites: www.fedstats.gov, www.mapstats.gov, and www.quickfacts.gov. These sites use MySQL as their backend database solution. The load on the sites is

approximately 120,000 pages per day. Although the U.S. Census Bureau could have used Oracle for no additional cost (it has an Oracle site license), it chose MySQL for "its ease of installation, maintainability, configuration and speed," according to Rachael LaPorte Talor, the Senior Web Technology Architect for FedStats.gov. The database runs on x86 Linux.

Texas Instruments

Texas Instruments uses MySQL to store regression test results for its semiconductor products. The database contains over 13 million records, filling up 5GB of data. Additionally, MySQL is used for a bug-tracking database that keeps track of 70 projects with 1,000 users. The platform is Sparc Solaris.

SS8 Networks

SS8 uses MySQL in its Local Number Portability (LPN) product for persistent storage of information pertaining to phone customers who have moved and switched their carriers. The supported capacity is up to 50 million records. MySQL was chosen for its performance, low resource requirements, and low licensing costs.

Moble.de

Moble.de runs an online car dealership with 315 million pageviews per month and a MySQL database containing records for 600,000 used vehicles. Additionally, its banner server delivers over 150 million impressions per month. It is using MySQL's replication functionality and propagating its data from a master server to 50 slaves. All systems run x86 Linux. Moble.de initially tried to set up one of the "big names" as its backend, but had a hard time getting the replication to work. MySQL replication worked flawlessly with very little configuration effort, which greatly influenced its database choice.

Who This Book is For

Chapters 1 – 3 in particular are written for technically oriented IT managers and other decision-makers with some background in information technology. System administrators and database developers will also find this information useful. In the remainder of the book, I assume that you have worked with a database before. Also, I discuss running MySQL on a variety of operating systems, including Windows, but you will get more out of this book if you are at home with a Unix shell, or at least willing to learn the basics.

The client API chapters assume that you are already familiar with the basics of the languages discussed. You will need to have a thorough background in C/C++ to fully use the information in Chapter 18.

The Book's Structure

Currently, there are two branches of MySQL available: stable 3.23 and development 4.0. It is quite possible that by the time you read this book, 4.0 will have entered stable status. Although the 4.0 branch does implement a number of new features, when viewed in the context of the core code base and functionality, the changes are not revolutionary, and most of the 3.23 concepts and techniques still apply. Therefore, I have tried to not assume much about the version of MySQL you are working with in the book and focus on the core concepts of MySQL that will always be there. However, I do discuss 4.0 features and point out critical differences in the new version.

Now, let us take a brief tour of the book. The first three chapters provide the information to help you make some basic system architecture decisions and lay a foundation for the work. Chapter 1 gives an introduction to MySQL, including an overview of its history, capabilities, licensing terms, and channels of support. Chapter 2 is an overview of hardware and operating system options for a MySQL server. Chapter 3 shows you how to install MySQL, create users, perform some basic database operations, and get MySQL up and running.

The following two chapters help you ensure that your database foundation is firm. Chapter 4 discusses testing and benchmarking the installation, which will allow you to discover operating-system issues early in the game and also estimate what kind of performance you can expect from your platform. Chapter 5 gives a brief overview of MySQL access control systems and provides general guidelines for securing a MySQL server.

The next seven chapters focus on building the client. In Chapter 6, I discuss various client language and network architecture options. Chapter 7 is dedicated to the specific needs of a Web client. Chapters 8 – 11 cover the basic client APIs in C, PHP, Perl, and Java, respectively. The API chapters contain fully functional code examples you can download from this book's Web site (www.wiley.com/compbooks/pachev), run on your system, modify and experiment with, and use to jump start your own code. Chapter 12 addresses the issue of writing efficient client code. One of the highlights of Chapter 12 is an overview of the optimizer, along with guidelines on how to test its behavior with the help of a special tool called *query-wizard* that I have written for this purpose (query-wizard is also available on the book's Web site).

The rest of the book focuses on the server. The health of the server is affected in great degree by efficient table design, which is discussed in Chapter 13. Chapter 14 explains server configuration issues. The highlight of the chapter is the discussion of each server variable and how it affects performance. Chapter 15 provides information about how to understand how the server functions, and how it can do the job better. This chapter is especially helpful if you are in charge of a MySQL server that is performing below the expected standards. The highlights of the chapter are a tutorial on how to read the output of *EXPLAIN* and a discussion of each SHOW STATUS parameter with a particular focus on diagnosing optimization bottlenecks.

Chapter 16 is a detailed discussion of replication; Chapter 17 discusses backup functions and strategies. Chapter 18 is written for those who need to work directly with MySQL source code, or simply would like to understand it better. We take a brief tour of the source by following the execution path when SELECT queries are run; the chapter also explains how to add SQL functions to MySQL, and it provides an overview of writing your own table handler.

The appendixes offer some additional information about migrating to MySQL, troubleshooting, solving SQL problems, and finding additional sources of information online.

Although this book is not perfect, I hope that you find it a solid investment and an indispensable resource for working with MySQL.

CHAPTER 1

Overview of MySQL

Data storage and retrieval is a core element of most applications today. In the early days of software development, programmers wrote their own low-level code to accomplish this. However, they quickly realized that in each application they were essentially reinventing the wheel. Through the usual cycle of trial, error, and subsequent refinement a solution was developed: the data storage and retrieval engine was abstracted into a stand-alone database server with the clients connecting to it and sending requests in a custom language called SQL (Structured Query Language).

Today, developers can choose from many data storage and retrieval products that use SQL. These products are usually referred to as SQL database servers, or sometimes relational database management systems (RDBMSs). Strictly speaking, an RDBMS system must comply with a set of formal requirements. It does not necessarily implement the SQL language, and vice versa—an SQL server may comply only partially with a set of formal RDBMS requirements. However, for practical purposes, the terms are frequently used interchangeably; most RDBMS products implement the SQL standard, and an SQL server that complies only partially with the formal requirements of an RDMBS will still be regarded by many IT specialists to be in the RDBMS league.

Products such as Oracle, DB2, Informix, and Microsoft SQL Server implement the SQL standard and are widely used in the industry. Even if you know nothing about SQL and relational databases, you have no doubt heard of these products—they are the well-known giants in the world of SQL servers. This book, however, is not about the giants of the SQL world. It is about MySQL—a feisty,

lightning-fast underdog of Scandinavian origin that has surprised many developers with its capability to outperform many of the giants.

Unlike most database servers, MySQL is an open source product: its source code is freely available for download to anyone. Programmers can modify the source code to tailor MySQL to their needs. One of the values of open source products is that a wide range of professional developers and users contribute their experience to the software, making it better. As a prominent open source project, MySQL has a large community of loyal supporters. MySQL has benefited in many ways from the contributions of the community, making it more than just a piece of software.

Decision makers in the IT industry are sometimes wary of open source products. The most common concern is that open source products do not have a commercial entity behind them that will take responsibility for supporting the software. In this respect, MySQL is different from most open source products. MySQL AB is a full-fledged company that, at the time of this writing, employs some 50 people all over the world who are responsible for development, support, sales, consulting, training, documentation, and other business functions. It is not the purpose of this book to discuss MySQL AB's business model, but if you are interested in knowing how a basically free open source product can be commercially viable, go to www.mysql.com for more information.

How Is MySQL Most Commonly Used in the Enterprise?

MySQL has several million users, among them many corporate users. In this section, I discuss the most common uses of MySQL in a corporate environment. This summary is based on my own observations while handling commercial support requests at MySQL AB.

Database Backend for a Web Site

Many Web sites have to provide dynamic content (e.g., a news site) and/or collect some data from visitors (e.g., an online store). Thus, there arises the need to have some data storage/retrieval functionality in the Web application. As many Web developers have discovered, MySQL is a perfect tool for this kind of job.

Free to obtain, easy to install and configure, and providing excellent performance and stability, MySQL has been a lifesaver for more than one CTO floundering in the perilous waters of the dot-com world. Some often hesitate to bypass a more expensive alternative, somehow thinking that if MySQL is free it

cannot be good. Nevertheless, when they finally make the decision they are often surprised to discover that MySQL is not only able to handle the load, but can often handle a load that none of the database "giants" they've tested has been able to.

Usage Logger

Another common problem in the IT industry is logging events of various types for the purpose of subsequent statistical analysis or simply for record retrieval in the future. This could be, for example, a network traffic monitor, an ISP keeping track of dial-up users, a cell phone provider logging calls, or a Web-usage counter.

MySQL's speed on insert and select queries makes it an attractive choice for this kind of application. And, of course, the other advantages of MySQL mentioned earlier make it only more attractive.

Data Warehousing

Various technologies today enable the accumulation of large collections of data. For example, a business could have a list of purchase records accumulated over the years, or a computer chip manufacturer could have collected a large dataset of test results. It could be very useful for various purposes to drill through the data and produce a number of statistical reports.

MySQL's speed on select queries makes it an excellent choice for many such problems. In fact, MySQL was originally written for the specific purpose of solving a particular data-warehousing problem more efficiently than what the market could offer at the time.

Integrated Database

More and more often, software vendors are finding it necessary to integrate a database into their commercial products. For example, a desktop phone book application with various search capabilities will be much easier to write if a lightweight SQL server has been integrated into the system.

The main considerations for a database server in this situation are the cost and the resource requirements. MySQL makes the grade in both aspects. Although not free in this case, the license cost per copy could very well be below $10 if the volume is large enough. And, of course, MySQL is very frugal about the resource utilization, the binary itself being small in size and the server configuration options allowing it to use no more than a few kilobytes of system memory while still maintaining a decent performance.

Embedded Database

Sometimes an application must process large amounts of data. A low-cost, lightweight database server is the ideal solution for an application programmer working under the restrictions of the embedded environment.

In addition to the advantages mentioned in the previous section, all of which apply here, the portability of MySQL makes it an attractive choice. MySQL can already run on a large number of architectures. Even if it has not yet been ported to the target architecture, the high coding standards that diligently address potential portability issues make it very likely that the port could be done with minimal effort.

Strengths and Weakness of MySQL

The primary question I will try to answer for you in this book is: "Will MySQL solve my needs, and if so, how?" The first step in answering this question is to examine MySQL's strengths and weaknesses; with a more complete understanding of MySQL's functionality, you will be able to decide whether MySQL is a good solution for your needs.

Strengths:

- Speed
- Reliability
- Low system resource requirements
- Scalability
- Platform diversity
- Support for a large number of host languages
- ODBC support
- Free or low-cost licensing
- Inexpensive commercial support
- Strong user community backing
- Availability of the source code

Weaknesses:

- Lack of certain SQL features
- Lack of thorough testing on certain platforms
- Difficulty of working with the source code

This list is not comprehensive. I have selected the most common factors in the decision making based on my experiences in working with MySQL users. Let's now discuss each of the strengths and weaknesses in more detail to help you understand its implications for your particular problem.

Strengths

Following is a discussion of MySQL's strengths.

Speed

The core MySQL code was written from the ground up with excellent performance as a primary goal. In fact, Monty Widenius, the original creator of MySQL, was frustrated with the relatively slow speed of databases available on the market at the time, so he wrote MySQL. In my role of enterprise support at MySQL AB, I have often had to deal with the concern on behalf of a potential user that MySQL might not be fast enough for what the user needs, or it will not be able to handle the load. The MySQL support team response is always, "Why don't you write a benchmark that will simulate a part of your application and see it for yourself?" The results of these benchmarks often stun customers and make them converts on the spot.

Many production database systems run a load of 1000 to 2000 queries per second on commodity x86 hardware (dual Pentium 800 with 1–2GB of RAM). I have taken MySQL "for a spin" on several occasions and was able to get 13,000 queries per second on a quad Pentium 700 selecting one record on a key from a table with one million records. So the answer to the question "Will MySQL be fast enough for me?" in most cases is "Yes, and probably much faster than you will ever need it to go." Benchmarking MySQL performance is discussed in Chapter 4, where you can learn how to see for yourself what MySQL is capable of.

A word of caution: Like any database product, MySQL can be slow in some cases if you are not careful when writing your queries. You can avoid this problem by understanding how the server works. One of the goals of this book is to give you the information you need to write efficient queries and keep MySQL running at top speed (see Chapters 12 through 15 for tips and techniques).

Reliability

MySQL has earned a reputation for being able to run unattended for days—even months—after initial setup. Here and there, of course, various issues arise and various bugs are discovered, just like in any other database server, but

overall it is very uncommon for MySQL to go down—and when it does, it is usually able to recover gracefully from the crash. This reputation for reliability got MySQL noticed by a number of enterprise users, who decided it was a great product for their needs. The list includes Yahoo! Finance, Cisco, Texas Instruments, the United States Census Bureau, NASA, Novell, Blue World Communications, Motorola, and many others.

The development team members are extremely focused on making MySQL reliable; they are obsessed (at least by industry standards) with ridding betas of bugs. I have seen MySQL releases postponed in numerous instances just because a single and rather insignificant bug had not yet been resolved. The discovery of one serious bug is reason to build a whole new release and issue a public apology.

Low System Resource Requirements

MySQL is able to make the best of the resources you give it. Of course, the greater the resources, the better the performance you can expect, but minimal resources will not put MySQL out of commission as it does some other database servers. I have successfully run MySQL on a 32MB RAM, 166MHz Pentium system that is not fully dedicated to MySQL. There have been reports of running MySQL under even smaller configurations. The footprint of a MySQL process with a "lowfat diet" configuration is 2MB to 3MB, so it is theoretically possible to run MySQL with as little as 4MB of RAM on the system. (For more information on server configuration, see Chapter 14.)

Scalability

Practical experience shows that MySQL scales well on systems with up to four CPUs, and up to 4GB of RAM, fully taking advantage of the system resources. It is known to work well with tables containing several billion records and has been reported to handle up to 1500 concurrent users without a notable performance degradation.

It is very possible that the true limits of MySQL scalability have not yet been fully explored. At the time of this writing, the MySQL development team has not yet been able to find the time or the hardware to create tests that will do so. They do rely to a great extent on user reports to learn how MySQL performs under load.

If your scalability needs ever exceed the capabilities of a single server, you can use the internal replication capabilities of MySQL to create a cluster of systems and distribute the load by directing the writes to the master host and sending the reads to the slaves. (Replication is discussed in Chapter 16.)

Platform Diversity

MySQL runs on a wide variety of architectures and operating systems. Among the most frequently used are Linux, Windows, Solaris, and FreeBSD. MySQL also runs on Irix, HP-UX, AIX, SCO, Tru64, OpenBSD, NetBSD, and Mac OS X.

Support for a Large Number of Host Languages

When you're developing a database application, one of your primary concerns is the ability to interface with the database server using a particular programming language, which is often referred to as the *host language*. This is another area of strength for MySQL; programmers can communicate with MySQL using C/C++, PHP, Perl, Java, Python, TCL, Ruby, and Eiffel.

ODBC Support

In addition to multiple host language support, MySQL includes an ODBC driver. This gives the programmer the ability to write vendor-independent database applications using the Open Database Connectivity (ODBC) standard.

ODBC connectivity support also allows MySQL to be used with a large number of data management ODBC-capable applications, such as Microsoft Access, Microsoft Excel, Crystal Reports, and many others. ODBC support allows MySQL to be used in Visual Basic and Delphi applications; in ASP (Active Server Pages); as well as with ColdFusion, Borland Builder, and many other development tools and environments.

Free or Low-Cost Licensing

MySQL is distributed under the terms of the General Public License (GPL) created by the Free Software Foundation (FSF). This license allows you to use the software free of charge for both commercial and noncommercial purposes on the condition that any derived product must be distributed with its entire source code under the terms of the same license. More information about GPL, including the full text of the license agreement can be found at www.gnu.org/licenses/gpl.html. For MySQL, the terms of the license mean that in most cases—except when MySQL is included as part of a proprietary product that the vendor is distributing to its customers—MySQL can be used free of charge.

In the case when the license is required, or when the organization policies do not permit the use of a GPL-licensed product, a license can be purchased from MySQL AB at reasonably low cost. At the time of this writing, the price for a

single license is $200, but drops dramatically as the number of licenses increases, all the way to $20 per copy for 10,000 or more licenses. The license is issued per server, and does not restrict the number of users.

You can find more information on MySQL licensing at www.mysql.com/support/arrangements.html.

Inexpensive Commercial Support

For those planning to run MySQL in a mission-critical environment, the issue of high-quality commercial support is very important. Even if no problems develop, the CTO needs to know that there is a competent source to turn to if any questions or problems arise. Not having this kind of support available could be a serious concern in making the decision to adopt the use of a product.

MySQL AB provides a wide range of commercial support at a reasonable price, including 24x7 telephone support. The prices range from $1500 per year for entry level to $48,000 per year for deluxe. The core developers participate in handling support requests—this means you don't have to jump through several hoops before you start talking with the person who wrote the code that has something to do with your issue.

Strong User Community Backing

As mentioned earlier, MySQL is more than just a database. MySQL's founders have always focused on giving the community more than they take from it. The community has responded with loyalty, hard work, and camaraderie.

How does this strong user community affect a potential enterprise user? The most obvious effect is that it is possible to get free support from the community in addition to the support provided by MySQL. If you post to a newsgroup or a mailing list, or get on IRC and ask a question, it is likely that you will get an answer. Unlike with commercial support, the answer is not guaranteed by MySQL AB, of course, but there are many experts on these lists who give excellent advice. Consider this support option as going fishing—if you know how to fish, you can get free fish from a lake. If you do not know, or do not feel like driving out to the lake, you can simply go to the store and buy some.

Another aspect of a strong user community is that it is relatively easy to find a dedicated MySQL expert who will work for you. What does this mean for an enterprise manager? In addition to the natural objective strengths of MySQL, you will get what one might call "the self-fulfilling prophecy effect." We usually think of self-fulfilling prophecies as something negative, but in this case it is a positive force and is exactly what an enterprise manager would want. The developer who knows the capabilities of MySQL and likes to work with it

predicts that MySQL will do the job and will then go to work, and for the exact same amount of pay, will put forth the kind of effort it takes to make it happen. You rarely find this kind of enthusiasm and commitment to other database products.

One other aspect of community support worth mentioning here is the "community insurance." A typical concern about the products of small software companies is the future of the product in case the company itself fails. Even if MySQL AB stopped selling support tomorrow and went back to being solely a group of dedicated developers, the community of users, the source code, and the open source process that built MySQL would still be alive and well.

Availability of the Source Code

Access to MySQL source code is an important advantage for businesses that employ experienced C/C++ programmers. It provides an opportunity for various customizations, improvements, extensions, and bug fixes to be done without having to wait for the vendor to do it.

Another advantage of having the source code available is increased peer scrutiny, which tends to lead to higher code quality. The driving factor in this process is the MySQL developer's sense of professional honor and reputation. When the source is going to be seen only by few coworkers, there is a temptation to start cutting corners—for example, failing to check for rather uncommon error conditions, avoiding security issues, or writing inefficient code in some places, hoping that the customer hardware is fast enough anyway. However, if the source is going to be seen by a large number of competent programmers across the globe, the attitude is totally different.

Weaknesses

Following is a discussion of MySQL's known weaknesses.

Lack of Certain SQL Features

The most serious weakness of MySQL is that it currently does not support subqueries, views, stored procedures, triggers, and foreign-key enforcement. This presents a number of issues, perhaps the most important of which is porting existing applications to MySQL. If your database application contains any of the features not supported by MySQL, you will need to rewrite those portions before porting it to MySQL. In some cases, this can be a daunting task.

There exists a strong school of thought in the IT world that some of the features not yet supported by MySQL are an absolute must. Many programmers have learned to depend on those features, and it has become an essential part of their

programming repertoire. It is little surprise, therefore, that this group is somewhat wary of using MySQL, and in some cases might argue quite strongly against it. Their reaction is perhaps somewhat similar to that of a person who has used automatic transmission his entire life and who is now asked to learn to drive a stick-shift.

Avid MySQL users, on the other hand, have learned to live without those features and even enjoy the challenge of having to get around them. Pushed by the challenge, they manage to find elegant solutions that they would have otherwise missed. They learn to emphasize MySQL strengths in their code and work around the weakness when necessary, and in their hands MySQL is able to perform just about any job they set out to accomplish. They tend to argue strongly for the use of MySQL. Using the car metaphor again, their skill allows them to shift gears to get the car to top speed, which they could not have done using automatic transmission.

Because of this difference of opinion, you will frequently encounter a division among computer professionals on whether MySQL would be an appropriate choice for a particular application. If you are a decision maker who has to trust the opinion of your experts and they disagree, then you are in a difficult situation. Can MySQL really meet your needs?

The reality is that in most cases (with the exception of porting some existing applications) MySQL will meet your needs for building database applications, given a good combination of skill, creativity, and motivation on the part of your developers. The problem is that if your database programmers are strongly affiliated with the "true RDBMS religion," they would be reluctant to use MySQL. Because of this reluctance, they will not put forth as much effort in providing a MySQL-based solution as they would have if you chose another database that they find more suitable; as a result, it will cost you more. If this is the case in your organization, forcing people to use MySQL before you get them to like it would not be a good idea from the pure economic standpoint. However, if you have some folks who are excited about working with MySQL, they would quickly find a way to make things work without the "absolutely necessary" features of the enterprise-level RDBMS.

It should also be mentioned that although MySQL lacks the above-mentioned features, they are currently in development. The goal of MySQL AB for the next two years is to implement all of the missing features and become fully SQL-compliant.

Lack of Thorough Testing on Some Platforms

To understand this weakness, you need to first understand a quality assurance phenomenon that is probably unique to MySQL. MySQL AB has strict coding

standards and a set of thorough testing procedures, but this level of quality assurance can go only so far. The next stage of testing happens when a new version is downloaded from the Internet at the rate of about 20,000 a day, gets installed on a very large number of systems, and is exposed to various combinations and sequences of queries on a rich variety of systems. This process will expose various bugs and system issues that could not have possibly been discovered even in the most rigorous in-house testing. Bugs are reported, and usually quickly fixed. MySQL AB depends on the field testing done by the users for quality assurance.

As a result of this process, it is apparent that the stability and performance of MySQL on a particular platform will be greatly influenced by the size of the install base. The larger the install base, the less of a chance that a critical bug could hide for long periods of time.

Although MySQL AB puts forth a valiant effort to be as cross-platform as possible and not favor one platform above another, the skew in the install base distribution causes some platforms to be much better tested than others. Although it is difficult to know the precise MySQL usage patterns, based on the download statistics, survey results, and a feel of commercial support and public mailing list traffic, I estimate that about 40 percent of installations are on x86 Linux, 25 percent on Microsoft Windows, 15 percent on FreeBSD, 15 percent on SPARC Solaris, and 5 percent on all other platforms.

Due to the quality assurance phenomenon, the remaining 5 percent will always be in the role of poor cousin compared to the dominant platforms, unless they manage to get a larger share of the market in terms of the number of units sold. This is not to say that MySQL is somehow broken on those platforms. It runs very well, and quite a number of people report success in using them, which is due in a great measure to the high focus on portability issues in the code: well-written code has a higher chance of running problem-free on a not-so-well-tested platform. However, a smaller install base will translate into a higher chance of running into a surprise, and you must be aware of that when choosing the platform to run MySQL on.

This situation will change in the future as MySQL AB continues to grow and is able to obtain resources to perform more in-depth internal testing of the platforms that are not extremely popular in the community, such as AIX and HP-UX. This disadvantage may very well be gone in a year or so from the time of this writing.

Difficulty of Working with Server Source Code

We have already discussed the value of having MySQL source code available. Having the source code in your hands definitely gives you a lot of flexibility. The

caveat of working with somebody else's source is that you have to understand it in order for it to be of any practical value. The MySQL server (not client) source is fairly difficult to get into, even for a skilled and experienced C/C++ programmer. Relatively few people dare.

I think there are two major reasons for this difficulty. First, it is a database server, which means it has to have code to efficiently organize data on the disk, cache it in memory to minimize disk access, parse queries, and select a strategy for resolving a particular query. This kind of code by its very nature would be complex, regardless of how clearly it is written and how well it is documented.

The second reason is what one could describe as the "genius code effect." The core code has been written by Monty Widenius, who in my opinion deserves to be called "the Mozart of computer programming." A programmer with a sense of taste will have an experience similar to listening to a beautiful piece of music while studying Monty's code. While this beauty is nice and wonderful, the challenge is that even when that beauty is documented, it requires the kind of inspiration that Monty had when he wrote it to understand it and to be able to add on to it without breaking the server. Experience shows that even the best programmers will find this challenging.

The challenge, though, is manageable. MySQL development team members somehow managed to master it, as well as several users who have contributed various patches to the code. Chapter 18 discusses server internals, partially in an attempt to encourage brave souls to learn MySQL server code and contribute to the code base.

MySQL from the Application Developer's Perspective

What options are available to a developer who wants to write a client application interfacing with a MySQL server? Client API libraries exist for C/C++, PHP, Perl, Python, TCL, Ruby, and Eiffel. Java is supported through a JDBC driver. You can also connect to MySQL using the ODBC standard, which opens up MySQL for use with Visual Basic, ADO, ASP, Delphi, ColdFusion, and any other ODBC-capable language, protocol, or development environment. If you are wondering about the use of a particular tool with MySQL and there is no explicit mention of MySQL support anywhere in this book, in the MySQL online documentation, or in the documentation of the tool itself, check to see whether the tool is ODBC-capable. If it is, it can be used with MySQL.

If a certain language or tool currently does not have native or ODBC support to connect to MySQL, there is still hope. If the language or tool is capable of accessing routines in an external library, you can invoke the code from the

MySQL C API library to get the job done. Even in the case of not being able to interface with external libraries at all, you can implement the MySQL client-server protocol on the application level, thus creating a low-level MySQL driver for that language or tool. In practice, the latter has happened only once in the history of MySQL, when Java support was created. APIs for other languages were implemented by simply calling routines from the MySQL C API.

What about the option of extending the server? Unlike many other database servers, MySQL has its source code available, so this is actually an option. As we mentioned earlier, work on MySQL server source can be a challenge. Luckily, the flexibility of MySQL allows you to solve most problems without ever having to consider extending the server. However, in some cases adding code to the server could be beneficial. We discuss this option in more detail in Chapter 18.

Overview of MySQL Integration with Other Industry-Standard Software

How well does MySQL coexist with other industry-standard software? Instead of merely rattling off a long list of products it can work well with, let's examine the principles of integration that determine how well application *X* will integrate with MySQL.

The key word to remember and focus on is *standards*. What standards does the application support? Does it support ODBC? If so, MySQL will integrate with it rather smoothly. This means MySQL will work with Access, Excel, Visual Basic, ASP, Crystal Reports, Delphi, ColdFusion, and many other applications, languages, and development tools.

Although ODBC is a great portability standard, the price to pay for portability is performance. For a desktop application connecting to a remote MySQL server, the performance reduction can be neglected for all practical purposes. However, if ODBC connections are established from a busy application server—in which case performance might be more important than portability—it would be advisable to consider an alternative that would use the native MySQL API.

How well will Web server *X* integrate with MySQL? Again, we go back to the concept of a standard. Most Web servers support the CGI standard, which permits an application executable residing on the Web server to be invoked through a Web request. Thus, the problem is reduced to being able to create an executable on the Web server that can talk to MySQL, which can be done using your language of choice. MySQL, therefore, can be integrated with any Web server that supports CGI, which includes Apache, Roxen, IIS, iPlanet, WebSphere, and a multitude of others.

Although CGI will do the job of integration for most Web servers—thus making it easy to move from one server to another—as you would expect there is a price to pay, just as in the case of ODBC: reduced performance. If the Web server is a busy one, this can be a serious concern, to the point of outweighing the issues of portability. An alternative solution using MySQL's internal server API (such as PHP for Apache and ASP for IIS) might be advisable.

Web applications making the use of Java (JSP, servlets, or applets) can use MySQL using the MySQL Connector/J JDBC driver. The portability of JDBC facilitates porting to MySQL from other databases, and allows you to write code supporting multiple databases.

When making a decision between portability and performance, first perform a set of benchmarks simulating the application to get an idea of what kind of performance difference solution paths can provide and compare it with the performance requirements of the application. Realistic analysis of capabilities versus requirements will help save a lot of development time and other resources.

Getting Help with MySQL

One of the stated goals of MySQL AB is to make MySQL easy to use. To a great extent, it is reaching this goal. Many questions can be resolved easily by a quick look at the documentation. Regardless of how intuitive and well documented a product might be, some users will run into issues that they need extra help to resolve.

So what should you do when you're stuck? One solution (which I mention with a grin on my face) is to read this book. Many other resources are available—some of the most useful ones are listed as follows.

Online Documentation

A searchable online manual is available at www.mysql.com/doc/. This resource is ideal if you have the intention of browsing through the manual section hierarchy. If you would rather search the documentation, a quicker way is to go to www.mysql.com/doc/home.html.

I recommend that you first spend some time looking for the answer in the manual before turning to other sources for help. Even if you are not able to find the solution itself, reading the manual will help you become more familiar with the issues associated with your problem. This increased level of understanding will help you get a better idea of what questions you need to answer to solve the problem, and thus make your use of other resources more efficient.

Mailing List

MySQL AB maintains a public mailing list dedicated to the discussion of MySQL. The posts are open to anyone. Unlike a number of other mailing lists, it is not necessary to subscribe in order to post. To post to the list, simply send a mail to mysql@lists.mysql.com.

The posts are read by about 4000 subscribers, and some of them are very knowledgeable MySQL users. Most questions are answered, although there is no guarantee. The questions that are not answered typically fall into three categories: very frequently asked questions that are answered already in the manual, vague questions, and very difficult questions that need a MySQL developer to answer.

The following rules of etiquette will help you maximize your chances of getting an answer and enhance the quality of your experience with the MySQL community:

- Be sure to spend some time reading the manual so you are at least somewhat familiar with the issue before you ask. In a couple of sentences in your post, state that you have done so and demonstrate that you have some knowledge of the issue.
- Search through list archives first to see if your question has already been answered. You can find a list of searchable archives at www.mysql.com/documentation/searchlists.html.
- On average, you can expect an answer within 24 hours. However, you must remember that you are, figuratively speaking, fishing in a lake. So if the fish are not hungry or simply do not like your bait, you may not catch any. If that happens, try a different bait; in other words, post again by restating your question in a different way so that somebody knowledgeable might find it appealing enough to answer.
- Be as specific as possible, stating all the relevant information, such as the MySQL version you are using, your operating system, the commands you have typed, and the output produced exactly as it appears on the screen. At the same time, try not to overwhelm the potential helper with unnecessary details. Attempt to find the fine balance between too little and too much detail.
- Be polite. Even if you do have an urgent problem, avoid talking about its urgency in the message. Concentrate on asking the question nicely—do not demand an answer. Avoid angry tones. Remember that you are asking for help from the people who are not in any way obligated to give it to you and will be doing you a favor.

- Study the posts you find in the mail archives that have received a good answer in the past and learn from them.
- Never send a personal e-mail to somebody active on the mailing list asking for free assistance with your problem. Just imagine what would happen to that person's mailbox if everyone did that.
- Be creative in your post. Proper use of humor; a short, interesting story about yourself or your company; a brief description of what you are doing with MySQL; or something else that will make your post stand out could make a difference between getting an answer and not getting one, or between having some MySQL guru spend only three minutes thinking about your problem and then giving you his first semi-educated guess and having him work on your problem for as long as it takes to solve it.

In addition to the general list, MySQL AB maintains similar lists for more specialized discussions. Those lists do not have as many subscribers, and the traffic on them is not so heavy. Similar rules of etiquette apply for the posts. In addition, the posts have to stay within the topic of the list. For example, you should not ask for help with installation on the MySQL Internals list, which is dedicated to the discussion of the internals of the MySQL server. More information about those lists can be obtained from www.mysql.com/documentation/lists.html.

Local Linux User Groups

In addition to the general mailing list, it is possible to obtain help with MySQL from a local Linux user group. Many Linux professionals have a good understanding of MySQL. The two skills tend to go together because MySQL is a popular database for Linux.

Such users groups usually have a mailing list of their own. Many require that you subscribe before you can post, which means that you will receive all the posts to that list for the duration of your subscription. It is, of course, possible to unsubscribe at any time.

To find your local Linux user group, visit www.linux.org/groups/.

Commercial Support from MySQL AB

While many issues can be successfully resolved with the help of the user community, there are times when you may need to talk to a developer at MySQL AB. Some of the advantages of this are as follows:

- If you have purchased support, response time is guaranteed.

- The quality standard of the response is high because it comes from somebody working closely with the MySQL development team. If you need to, you will be able to communicate with the developer who wrote the code that you are having issues with or questions about.
- MySQL AB is committed to working with you until the problem is solved.

MySQL AB provides commercial support. Various support options are available, from the entry level to the 24x7 support, at reasonable prices. More information about commercial support contracts is available at www.mysql.com/support/.

CHAPTER 2

Selecting a Platform for MySQL Server

Selecting appropriate hardware and operating systems is a controversial decision in almost any situation. Discussions can become very heated and subjective. In this chapter, I will attempt to provide an objective overview of the major operating systems, highlighting advantages and disadvantages of each as it pertains to running MySQL server. I do have to make a disclaimer that I am biased in the direction of Linux and open source solutions.

Platform Criteria

Here is a list of the primary criteria for selecting a high-performance platform for MySQL:

- The size of the user base
- The amount of usage under high load on mission-critical servers
- The maturity of the C/C++ compiler
- The number of MySQL AB developers regularly using the platform
- The degree of standard compliance in the system libraries
- The maturity of the thread library available on the system

Let's discuss each of these criteria in a bit more detail and explain its importance.

Size of the User Base

Each end user will cause the server to execute its code in patterns that will be unique to the user's application. As more users try the code, increasingly different ways to execute the code appear. This results in a greater opportunity to uncover obscure bugs and performance bottlenecks. This leads to more bug reports, which in turn leads to getting those bugs fixed, and eventually leads to a high standard of performance and reliability.

Amount of Usage Under High Load on Mission-Critical Servers

Even with a high number of users, certain code in specific combinations is unlikely to be executed unless the system is under very high load. Therefore, certain bugs will never be discovered unless the system is put under high-concurrency execution of a wide range of queries—something that would be virtually impossible to simulate in a test environment. Again, as bugs are reported, they get fixed.

Maturity of the C/C++ Compiler

Parts of MySQL Server are written in C and other parts in C++. Ideally, programmers like to assume that their compiler is perfect and will solve most of life's problems for them. The reality of working on a project that involves a few hundred thousand lines of code—many of them written by Monty, who does not quite think like compiler test case writers—is that eventually an ugly internal bug on the MySQL code base will generate a broken executable. Additionally, some compilers may produce correctly functioning but suboptimal code, which reduces the quality of the application running on that platform.

Number of MySQL AB Developers Regularly Using the Platform

As most of you already know, the development process includes frequently repeated code plus compile plus test cycles. Therefore, the platform that a project is actually developed on (as opposed to being ported to just before release) will inherently be more stable than its "port-to" counterparts at release time because it will have undergone more testing.

Degree of Standard Compliance in the System Libraries

MySQL assumes that system routines on Unix platforms follow the POSIX standard for the most part. Unfortunately, as experience and the abundance of

#ifdef HAVE_BROKEN_... directives in the source code show, this is by far not the case. Fortunately, the magic #ifdef solves the problem once it is discovered. But no one knows how many problems we are yet to discover on the multitude of systems MySQL AB supports. The large user base here is the key to discovering such problems.

Maturity of the Thread Library Available on the System

MySQL Server heavily uses POSIX threads. Therefore, the stability and performance of the thread library is a critical factor in determining how stable and how fast MySQL will execute on any given platform.

Platform Comparison

For practical purposes, you can consider a large user base and its willingness to put MySQL in a mission-critical environment on a certain platform as a seal of community approval. Judging by this standard, the most "voted" platforms are x86 Linux, Windows, FreeBSD, and SPARC Solaris. Therefore, next we discuss those systems one at a time, highlighting the strengths and the weaknesses of each, and then have a brief overview of other systems.

Linux

MySQL's success on Linux is not a coincidence; many factors contribute to that success. First, Linux is a popular platform; this alone is a driving factor because it establishes a large user base, thus creating a large test environment for MySQL. Additionally, two aspects of the Linux kernel make it a fertile ground for MySQL: solid thread support and aggressive file caching. Also, MySQL developers tend to run Linux on their desktop machines, which double as personal development servers.

Because of the popularity of Linux, the GCC compiler on this platform has received a baptism of fire through brutal testing by a horde of merciless users who write all kinds of variants of C/C++ code. This makes it likely that the GCC compiler will compile MySQL sources without compiler-specific problems.

Some issues have emerged in the thread library (LinuxThreads) that negatively affect MySQL performance and stability under high load, and these issues have generated quite a bit of inaccurate publicity with claims that "MySQL does not scale." Fortunately, LinuxThreads is an open source library, and with the help of the developer community, MySQL AB has been able to provide a patch to address these issues.

Although in my opinion Linux is the best platform for running MySQL in most situations, Linux is far from perfect. The current state of virtual memory implementation leaves much to be desired, and the lack of unity among Linux developers has been a great impediment to progress. There exists a large variety of kernel flavors: the "virgin" kernel, kernels patched by Linux distribution vendors (e.g., RedHat and SuSE), and numerous special patches maintained by various groups and individuals that have not been included for one reason or another into the main kernel tree.

None of these flavors is perfect; all have bugs or performance quirks of one kind or another. Some will be stable when running MySQL, and others will not. To make matters worse, some usage patterns of MySQL may expose a bug, while others will not. As of this writing, the MySQL development team concurs that the most stable kernel for MySQL Server is the one from SuSE 7.3.

MySQL is not as stable or fast on non-x86 Linux flavors. Two factors play an important role. The non-x86 Linux user install base is not nearly as large as the x86 one, which in practice means that a lot of architecture-specific kernel bugs that would have been quickly discovered on x86 will not be discovered on a non-x86 architecture by the time you decide to put your system into production. Similarly, the MySQL user base on the non-x86 platform is also small. This means that platform-specific MySQL bugs are not as likely to be discovered and fixed.

Having said that, I believe that running MySQL on a non-x86 Linux machine is still worth a try. Both MySQL and Linux have been written with great consideration for portability, and the code for both is of very high quality. MySQL may run better on Linux than on the native operating system provided by the hardware vendor.

Windows

Microsoft Windows and MySQL have a very interesting relationship. MySQL was originally written for Unix, but one of the goals for MySQL is to make it a superior cross-platform database. Windows is a popular platform on both the desktop and server, but it took some pleading from the Windows users to get MySQL ported to Windows; it was done more out of necessity than desire.

Naturally, this kind of relationship is not conducive to success on a platform. However, the sheer size of the Windows user base has had a powerful effect on the progress of MySQL on Windows. The bugs kept getting fixed, and eventually the Windows port became quite robust.

On a benchmark conducted by *eWeek* magazine in February 2002, MySQL version 4.0.1-alpha running on Windows outperformed DB2, SQL Server, and

Sybase; and tied with Oracle. MySQL and Oracle were the only databases that could run the unmodified *eWeek* test for eight hours. Despite the success that MySQL has enjoyed on this platform, Windows has a number of factors that negatively affect the performance and stability. Although the Windows user base is large, many Windows installations of MySQL are run on a desktop with only one or two concurrent connections and hardly ever running two queries at the same time. There are not as many high-concurrency Windows users as there are on Linux.

The typical Windows installation does not have the bug-tracking tools normally available on a Unix platform (such as a debugger or system call tracer) that will help gather important diagnostic information. Such tools usually have to be purchased and installed separately. Because of that, the quality of a typical Windows bug report is much lower than the typical Unix report.

These comments should not lead you to conclude that MySQL does not run well on Windows. In a family of millionaires, even the poor cousin is quite well off—he just does not have as much as the rest of the folks. If you have to run MySQL on Windows, you will experience a measure of success. However, if you have a choice in the matter, a different operating system is likely to bring you better results.

Solaris

SPARC Solaris is the most prominent "big-iron" platform for MySQL. Part of the secret of MySQL's success on Solaris is that Monty had been developing on this platform for quite a while until he decided to switch to Linux for his primary development server. Another factor is that Solaris has a MySQL-friendly thread library. Sun Microsystems's overall commitment to supporting industry standards also plays an important role.

Part of the standard prerelease quality assurance process involves running MySQL on Solaris under Purify (a commercial runtime analysis tool). Purify can detect common programming errors that might be hard to detect through regular testing.

Solaris MySQL users tend to have a high level of satisfaction with their experience; MySQL generally runs without problems. When problems do arise, standard tools are available to diagnose them. Database and system administrators working with Solaris tend to be skilled, well-rounded, and creative professionals. They tend to produce excellent bug reports that make it easy to track down the problem without ever having to log into the system in question.

The main drawback of running MySQL on SPARC Solaris is the low "bang-for-the-buck" value. Sun hardware is much more expensive than x86 systems of

equivalent capacity. In two years of working for MySQL support, I have never logged in to a Sun machine that could outperform my desktop (a dual Pentium 500 with 256MB of RAM running Linux) on MySQL tests. This is not to say that such machines do not exist—this is simply to point out that they are so expensive that none of the MySQL customers I've worked with could afford one.

FreeBSD

Many MySQL users are having success with FreeBSD in mission-critical, high-concurrency applications. The most prominent example is Yahoo! Finance. Much of what FreeBSD contributes to MySQL success is a solid, rigorously tested kernel code; a stable and efficient virtual memory and file system; and a reasonably sized install base for the operating system itself, which in turn results in a decent install base for MySQL.

If only FreeBSD could borrow the thread capabilities and the user base from Linux, it would probably become the most recommended platform for MySQL. Unfortunately, though, as of this writing that has not happened. FreeBSD threads are still on the user level, which reduces any threaded application, including MySQL, to running on only one processor. Additionally, the thread library does have a number of performance and stability quirks that can end up having a "fly in the ointment" effect on the user experience with MySQL.

Other Systems

Although MySQL will compile and run on a multitude of other systems (AIX, HP-UX, Irix, SCO Unix, Tru64, OpenBSD, NetBSD, BSDi, Mac OS X), because of the relatively small-size install base, it is hard to comment on its stability and performance on those platforms. MySQL AB's proactive efforts at porting to those platforms are limited to compiling binaries for some of them at release time. The rest of the effort is reactive. None of the developers have used those systems for their development server, so development is done only when somebody reports a problem.

Despite the lack of proactive development effort, the solid design of the MySQL code base bears fruit. It is not uncommon for MySQL to run with a great degree of success on a poorly tested platform. If you already have an idle machine with one of the above-mentioned operating systems, or if you are considering a solution that absolutely has to run on such a system, it is definitely worth a try to see how well MySQL will perform. However, if you are building a new system and deciding on the ideal MySQL platform, these systems would not be the best option.

Operating System Tuning Tips

Next, we discuss the general principles of tuning an operating system on a MySQL server. Because specifics vary widely from system to system, and because systems keep changing at a rate that is impossible to fully track even on a Web site, I will not list specific instructions for each tuning operation; rather, I refer you to the operating systems manual for such details.

To maximize MySQL performance and stability:

- Apply the latest vendor patches to the kernel and to the system libraries. In case of Linux, make sure you are running a kernel with good reputation (e.g., the one from SuSE 7.3).
- Some operating systems have a tunable file cache. If this is the case with your system, make sure that enough memory is allocated for it. You may want to run benchmarks on your data as you play with the file cache settings.
- MySQL depends extensively on the underlying file system performance. Experiments with file system settings as well as trying out different file systems may produce significant differences in the overall performance of MySQL.
- If the data does not fit entirely into RAM, your application to a certain degree will be disk-bound. If this is the case, you may try to tune various parameters in the operating system that affect your disk performance.
- Make sure that enough file descriptors are allocated for the use of the MySQL process.
- On some systems (e.g., Linux), a thread is accounted for as a separate process. On such systems, you need to ensure that the MySQL process is allowed to create the necessary number of child processes.
- Avoid putting your data directory on an NFS (Network File System) volume and put it on a local disk instead. If you absolutely have to, use InnoDB tables instead of MyISAM—the InnoDB table handler caches both keys and data in RAM, whereas MyISAM caches only keys relying on the file system disk cache to cache the data. File system caching is not very effective with NFS because the disk cache is flushed on every system write.
- If you are not sure about a certain setting, do not be afraid to benchmark before you spend too much time researching. Your research will be more meaningful after you have some benchmark results. Remember that modern-day hardware and software is so complex that it is easy to make a mistake in trying to theoretically predict how it is going to behave under certain circumstances. It is much easier and much more productive to

simply simulate those circumstances and see what happens. Luckily, computer science is different from chemistry in that you will not blow up your lab when something goes wrong with your experiment.

Hardware Tips

In this section, we address general principles of hardware selection and configuration as it pertains to MySQL; we cannot discuss the specifics because they are changing at a pace that is impossible to follow. We advise you to study the specifics and apply these principles when configuring a system for MySQL Server:

- If the data fits entirely (or at least mostly) into RAM, the CPU cache size becomes an important factor in MySQL performance. On one test, the cache increase from 512KB to 2MB without increasing the CPU speed improved MySQL performance by 60 percent.
- Do whatever it takes to fit your data into RAM. In terms of hardware, this means buying more RAM. Buy as much RAM as you can reasonably afford.
- Buy the fastest RAM possible.
- Before you consider buying a faster disk, see if you can buy more RAM with that money. If your application becomes disk-bound, a faster disk will give a speed-up of the factor of 2–3 or so. Keeping the data in RAM can speed up the application 100 or more times.
- Will adding more processors help? It depends on the system. On Linux and Solaris, it will. On FreeBSD, it will not. On Windows, it should. On other systems, it might if you luck out with the vendor thread library—if it is tuned for frequent short critical regions, adding processors will help; otherwise, it will make things worse. (No joking here—I have seen cases where getting rid of an extra processor improved MySQL performance.)
- How many processors can MySQL take advantage of? MySQL scales well on x86 Linux, SPARC Solaris, and HP-UX 11 with up to four processors. It does not scale well on x86 Linux with eight processors: the performance is virtually the same as on a four-processor machine. I am not aware of any benchmarks on non-x86 systems with more than four processors. It is recommended that you consider using a cluster of low-cost machines with replication when performance requirements exceed the capability of one server.

CHAPTER 3

Installing MySQL

In this chapter, we will step you through the basics of installing MySQL. Before you install, you need to decide whether to install from source or binary distribution, whether you need transactional table support, and whether you will use the stable or the development version. We will discuss the issues associated with making those decisions. Afterward, we will provide a sequence of steps for different methods of installation, followed by brief troubleshooting guidelines.

Method of Installation

There are basically two methods for installing MySQL. The first is to download a binary supplied by MySQL AB; the second is to compile your own binary from the provided source. If you do not want to struggle for too long with this decision and want a quick approach, try the binary first and worry about the source only if the binary does not work on your system or is not available.

These two methods of installation have both advantages and disadvantages. MySQL AB currently provides binaries for the following platforms:

- Linux (x86, Alpha, SPARC, and IA64)
- FreeBSD
- Windows
- SPARC Solaris 7 and 8
- HP-UX 11

- Mac OS X
- AIX 4.3
- Irix 6.5
- Dec OSF 5.1

This list of platforms may change somewhat from time to time, but you can always count on having current x86 Linux, SPARC Solaris, Windows, and FreeBSD binaries available.

MySQL AB's goal is to build binaries with the maximum degree of portability, performance, and stability. Each binary is compiled with a special set of compiler flags, and in the case of x86, Linux is linked against specially patched system libraries to achieve that goal. Although MySQL AB always tries its best when building a binary, it is important to understand that the degree of expertise varies from platform to platform, which in turn will affect the quality of the binary.

Therefore, although on some platforms it might be worthwhile to try to build your own binary that will run better than the one supplied by MySQL AB, on others it is not likely that even an expert user will succeed in doing so. I recommend using MySQL AB's binaries on x86 Linux, FreeBSD, Solaris, and Windows unless you have a good reason to build your own (such as a need to extend the server).

In some instances, you may be restricted to building from source. The most obvious case is when the binary for your platform is not available, but you may also need to build from source when the system libraries are not compatible with the MySQL AB binary. This happens quite often on commercial Unix systems such as Solaris and HP-UX. And of course, if you want to extend MySQL server, you will need the source code.

Even when suitable binaries are available, some people prefer to install from source, for several reasons. Some like the security of knowing that if something goes wrong with MySQL, they have the source they can fix. Some want to experiment with different compilers and/or compiler options. For others, it's a matter of principle: I know many system administrators who install everything from source, partially so that they know exactly what is being installed on their system, and partially for the sense of satisfaction from knowing that they have personally compiled every package on their system.

The Need for Transactional Table Support

If you are in a hurry to get MySQL up and running and do not want to spend too much time thinking about whether you will need transactional tables, assume

you will, make the decision to install MySQL-Max, and skip the rest of this section.

One unique feature of MySQL is the support for multiple table handlers or types. A *table handler* can be described in simple terms as the low-level data and index storage/retrieval implementation. In other words, the MySQL query optimizer abstracts itself from the low-level storage and retrieval and makes calls to the table handler routines when such operations are required, leaving it up to them to do the "dirty job." This design naturally allows for hooks to get the "dirty job" done in several different ways. The hook—in other words, the table handler—takes full control of all operations associated with the low-level storage and retrieval of data, and is free to implement them in a variety of ways.

Currently, the binaries from MySQL AB support the following table handlers: MyISAM, ISAM, HEAP, MERGE, INNODB, and BDB. MySQL AB supplies two kinds of binaries: regular and Max. Support for BDB and INNODB is included only in Max; otherwise, both binaries support every other table handler. Therefore, it may become important to know prior to the installation if the benefits of having support for BDB and INNODB in the binary are worth the overhead.

For most enterprise users, the answer will be yes. Unlike all other table types, BDB and INNODB tables have transactional capabilities. With a non-transactional table, if you perform an update the change is irreversible. With a transactional table, an update might be reversed (or *rolled back*) if you change your mind prior to committing. There is a good chance that eventually you will need that functionality in your application. The main disadvantage of having that capability supported in the binary when you are not using it is the increase in the memory footprint of the server somewhere in the order of a magnitude of 5MB, depending on the platform plus whatever you decide to allocate for the transaction-specific buffers, just in case you might decide to have a transactional table someday. On most modern systems, this overhead is not worth worrying about; however, there is no need to incur it if your data is updated very infrequently and the majority of your queries only read the data.

Version Issue

At any given time, two version branches of MySQL are available: stable and development. The *stable* branch contains more mature and tested code. The only modifications made in the stable branch are critical bug fixes. No major new features are added so as not to disrupt the core code. Communication protocols and storage formats do not change as the version updates are released.

The *development* branch keeps changing quite frequently. Portions of the core code are rewritten. Major new features are constantly added. Communication

protocols change. Incompatibility with the old version protocols and formats may be introduced. Although development branch versions are tested with the same degree of rigor internally prior to a release, the lack of field testing of the new code means that there might be some subtle bugs in those areas.

The decision on which branch to use depends on whether you need the new features available only in the development branch bad enough to sacrifice the possible difference in stability. If you are a new user trying to learn a bit about MySQL, you should probably use the stable version. If you are planning an enterprise application and you already know that the stable branch can do everything you need, experimenting with the development branch may not be worth the risk. However, do not be afraid to try out the development branch if you need a certain feature, or if you simply would like to have the latest feature set at your fingertips. MySQL development branches have a history of being much more stable than the "release" version software of many commercial vendors.

As of this writing, the development branch is version 4.0, and the stable branch is 3.23. Version 4.0 has officially entered the feature freeze, but has not yet been declared stable. If you plan to use replication in production, I would suggest sticking with 3.23 and waiting until about 4.0.8 release or later. If you are not using replication, using 4.0.5 (the current version as of this writing) is not a big stability risk and might be worthwhile if you need the new functionality. Some of the feature highlights of version 4.0 are

- **Query cache:** greatly improves performance of the applications that run the same query over and over while the tables involved in the query do not change nearly as frequently as the query being run. See www.mysql.com/doc/en/Query_Cache.html.
- **Multi-table UPDATE and DELETE:** allows you to delete or update records in several tables in one query that match the criteria of a join. This feature, for example, permits deleting records in one table that have a counterpart in some others in one query. See www.mysql.com/doc/en/DELETE.html and www.mysql.com/doc/en/UPDATE.html.
- **UNION:** allows you to combine the results of different select queries, thus avoiding the need for temporary tables in many cases. See www.mysql.com/doc/en/UNION.html.
- **HANDLER:** allows the application to have complete control of index traversal by issuing direct commands bypassing the optimizer. See www.mysql.com/doc/en/HANDLER.html.
- A number of performance enhancements.
- Rewrite of the replication slave to use a two-threaded model—one thread reading the updates from the master and storing them in a temporary relay

log, while the other is applying them on the slave. This reduces the data loss to the minimum when the slave server is behind the master on updates and the master server goes down and never comes back.

Installation Process

By the time you get to this point, I assume you have already decided if you want to use the source or binary installation, whether you need transactions, and whether you will use the stable or the development branch. In the examples that follow, I assume that you are using the Max distribution, which supports transactions in all cases. If you do not want transactions at all, simply drop the -Max command out of all the examples.

I also assume that you are installing the current stable branch—3.23. If you would like to install 4.0, or if 4.0 becomes stable by the time you are reading this book, visit www.mysql.com/downloads/ and follow the appropriate links from there to get the correct distribution. Because there is no fundamental difference in installing 3.23 and 4.0, once you understand how to install 3.23, you will be able to install 4.0 without much difficulty.

I would like to emphasize the importance of understanding the principle of how to install MySQL rather than simply following the installation instructions I provide here in hope that the guru magic will work. Although I have tried to ensure that the majority of users will be able to install painlessly by following these instructions, I realize that systems vary to a degree that makes it impossible to cover every potential issue. If you experience trouble during the installation, I recommend that you spend some time understanding the installation instructions and the concepts they are based on. For example, you may want to study Unix file system permissions, read the manual page on RPM, familiarize yourself with basic Unix commands, investigate the meaning of standard Unix error numbers and messages, or learn about Unix sockets. You may also find it beneficial to examine the source of the shell scripts referenced in the installation instructions to help you understand what is actually going on.

The time will be well spent. When you have a basic understanding of the installation process and your system, or if something out of the ordinary happens that I have not covered in the troubleshooting section, you still will be able to diagnose and solve it yourself. This will give you a fulfilling sense of competence and add a stripe to your sysadmin karate belt.

Binary Installation

Binary installation is essentially the same for all platforms, with the exceptions of Windows and Linux. On Linux, MySQL AB provides RPM packages in

addition to the regular binary installation. RPM is the recommended installation method for systems that have RPM (e.g., RedHat, SuSE, Mandrake, and Caldera). If you have a Linux system without RPM (such as Debian), or if you are installing as a non-root user, you will need to use the standard binary distribution as opposed to RPM. On Windows, the installation process simply consists of unzipping the distribution archive and running the setup program. We first discuss the Linux RPM installation, then Windows, and then talk about the generic Unix installation.

Linux RPM

Follow these steps to install MySQL on any version of Linux with RPM:

1. Check to see if MySQL is already installed on your system. Some distributions, such as RedHat and SuSE, may preinstall MySQL if told to do so during system installation. There are several ways to check; one is ls -l /usr/sbin/mysqld. If this command shows that /usr/sbin/mysqld exists, then MySQL is already installed. If the file does not exist, although it is technically possible to install it in other locations, I do not know of a distribution using RPM that would. So you can quite safely assume that it is not installed.
2. If MySQL is already installed, you may consider uninstalling it and replacing it with the RPM from MySQL AB. This way, you get an updated version and the binary built for a high-load setup. Additionally, the preinstalled version may not have been built with transactional table support. The following should work on all RPM-based distributions for the preinstalled MySQL package removal:

   ```
   rpm -qa | grep -i mysql | xargs rpm -e --nodeps
   ```

   Alternatively, you may use rpm -qa | grep -i mysql and then remove each MySQL package manually one at a time, addressing the issue of dependencies as you go. However, you may decide that the version that is already installed is good enough for your needs and stop here.
3. Go to http://www.mysql.com/downloads/mysql-3.23.html. Scroll down to RedHat packages, and click on the link that says Server.
4. Jump through the mirror selection hoops, picking a mirror that is closest to you, and download the rpm file—we assume you have saved it as /tmp/MySQL.i386.rpm on your Linux server.
5. Repeat the process and download at least the client and the development (libraries and headers) RPMs. You may also want to get the client shared libraries and the benchmark RPMs.
6. Go to www.mysql.com/downloads/mysql-max-3.23.html.

7. Locate and download the server RPM the same way you did on the non-Max page. The reason you have to go to non-Max page first is that the Max RPM provides the binary for the server, but not the error message files, documentation, and wrapper scripts for starting the server (these are found only in the regular server package). We assume you saved the Max RPM in /tmp/MySQL-Max.i386.rpm.
8. Run this command as root: rpm -i /tmp/MySQL.i386.rpm**.** This will install the MySQL package.
9. Run this command as root: rpm -i /tmp/MySQL-Max.i386.rpm**.** This will install the MySQL-Max package.
10. Run rpm -i for each additional package (client, development, shared libraries, and benchmarks) that you would like to install. You may skip benchmarks and shared libraries, but I strongly recommend you install client and development RPMs.
11. By now you will have the regular (non-Max) server running. Stop it with /etc/rc.d/init.d/mysql stop.
12. Execute /etc/rc.d/init.d/mysql start to start the Max binary. If you want to understand how this piece of magic operates, study /usr/bin/safe_mysqld. Here's what happens: if mysqld-max is not present, mysqld is started, but if mysqld-max is present, mysqld-max is started.
13. Now MySQL server is installed and running

One difference in version 4.0 is that you will get transactional table support without having to download the Max package. You can simply skip step 9 if the only reason you are getting Max is for transactional table support.

Windows

Here are installation instructions for Windows:

1. Go to www.mysql.com/downloads/mysql-max-3.23.html.
2. Scroll down to the Windows section and download the zip file.
3. Create a temporary folder and extract the contents of the zip file into that folder.
4. Execute *setup.exe* from the new folder.
5. MySQL server now should be installed and running.

Standard Unix Binary (Root User)

The following are installation instructions for installing MySQL as a root user on a standard Unix binary:

1. Go to www.mysql.com/downloads/mysql-max-3.23.html.
2. Scroll down to your platform section.
3. Download the binary.
4. Check if you have GNU tar installed. You can verify this by typing gtar. If you get the "Command not found" message, it is either not installed or is not in your path, in which case you can run find / -name gtar -print and see if it will turn up something. (Be careful with the previous command if you are not the only user on the system—it might be very resource-intensive, and other users may not appreciate the slowdown.) If it is not installed, you either use the native system tar or you can get it from ftp://ftp.gnu.org/pub/gnu/tar/. GNU tar is recommended for all systems and required for Solaris.
5. Let's suppose you have saved the distribution archive in /tmp/mysql-max.tar.gz. Proceed with cd /usr/local; gtar zxvf mysql-max.tar.gz (or if you do not have gtar, gunzip -c mysql-max.tar.gz | tar xvf -).
6. Run ln -s mysql-*version* mysql, replacing *version* with the MySQL version identifier; for example, ln -s mysql-3.23.49-sun-solaris2.8-sparc mysql.
7. Run the following commands:

```
cd mysql.
groupadd mysql
useradd -g mysql mysql
scripts/mysql_install_db
chown -R root .
chown -R mysql data
chgrp -R mysql .
bin/safe_mysqld --user=mysql & for 3.23, or bin/mysqld_safe
  --user=mysql & in version 4.0
```

Standard Unix Binary (Non-Root User)

Here are installation instructions for installing MySQL as a non-root user on a standard Unix binary:

1. Obtain the distribution for your platform as described in the root user installation section.
2. Check if you have gtar installed. See the Unix binary tar discussion for the tar issues.
3. Run gtar zxvf mysql-max.tar.gz or gunzip -c mysql-max.tar.gz | tar xvf - if you do not have gtar.
4. Run ln -s mysql-*version* mysql, replacing *version* with the actual version string of the distribution.

5. Run the following commands:

```
cd mysql
scripts/mysql_install_db
bin/safe_mysqld --datadir=$HOME/mysql/data --
socket=$HOME/mysql/data/mysql.sock & for 3.23 or bin/mysqld_safe --
datadir=$HOME/mysql/data --socket=$HOME/mysql/data/mysql.sock & for
  4.0
```

Source Installation

In this section, we describe how to compile your own version of MySQL from source files.

Unix

Follow these steps for compiling MySQL source on a Unix platform.

1. Visit www.mysql.com/downloads/mysql-3.23.html.
2. Scroll down to the very bottom and download the tarball distribution.
3. Make sure you have a C++ compiler installed. For most platforms, it is recommended that you install GCC version 2.95. There are platforms, though, where it is better to compile with gcc 3.1 (HP-UX 11, AIX 4.3, and possibly other Unix platforms). However, the recommended version of the compiler may change as the newer GCC versions stabilize. Check the operating system notes for your OS at www.mysql.com/doc/en/Which_OS.html for the most recent recommendation.
4. Install GNU make.
5. Assuming you have saved the downloaded file in mysql.tar.gz, you execute gunzip -c mysql.tar.gz | tar xvf -.
6. Run cd mysql-*.
7. Now the most important and most challenging part: to run the configure command that will examine your system and will prepare the source tree for the compilation. This step varies from platform to platform and some tweaking may be necessary. Usually the following works: CFLAGS="-O3" CXXFLAGS="-O3 -felide-constructors -fno-expections -fno-rtti" CXX=gcc ./configure –prefix=/usr/local/mysql –with-innodb . If you are using a different compiler than GCC, the flags would need to be different. Check the compiler manual to see how to disable exceptions and RTTI (run-time type information). If you have less than 128MB of RAM, you may need to add --with-low-memory argument to ./configure.
8. Run gmake (to make sure GNU make, not the native one, is executed) —this will take approximately 5 minutes on a dual Pentium 500 with 512MB of RAM.

9. Optionally, you can execute gmake test once make completes. Ideally, all tests should pass. However, if you are on a platform that MySQL has never been compiled on by the MySQL team, chances are some tests might fail. It is usually the replication tests that do, because they are more prone to be affected by various quirks in the TCP/IP implementation and in the thread library on a particular platform. The binary might still be usable for some purposes even if that happens, though.
10. If everything is fine up to this point, su to become root, and type make install.
11. Run cd /usr/local/mysql; bin/safe_mysqld &.

Windows

Windows installation is a little more straightforward, but for the purposes of completeness and for your convenience, I am providing the instructions found in the MySQL manual at www.mysql.com/doc/en/Windows_source_build.html. You will need the following:

- C++ 6.0 compiler (updated with 4 or 5 SP and Pre-processor package). The Pre-processor package is necessary for the macro assembler. More details at http://msdn.microsoft.com/vstudio/sp/vs6sp5/faq.asp.
- The MySQL source distribution for Windows, which can be downloaded from http://www.mysql.com/downloads/.

To build MySQL, follow these steps:

1. Create a work directory (e.g., workdir).
2. Unpack the source distribution in the aforementioned directory.
3. Start the VC++ 6.0 compiler.
4. In the File menu, select Open Workspace.
5. Open the mysql.dsw workspace you find on the work directory.
6. From the Build menu, select the Set Active Configuration menu.
7. Click over the screen selecting mysqld - Win32 Debug and click OK.
8. Press F7 to begin the build of the debug server, libs, and some client applications.
9. When the compilation finishes, copy the libs and the executables to a separate directory.
10. Compile the release versions that you want, in the same way.
11. Create the directory for the MySQL stuff: e.g., c:\mysql.

12. From the workdir directory copy for the c:\mysql directory the following directories:

    Data

    Docs

    Share

13. Create the directory c:\mysql\bin and copy all the servers and clients that you compiled previously.
14. If you want, also create the lib directory and copy the libs that you compiled previously.
15. Do a clean using Visual Studio.

Basic Post-Installation Checks

Now we are entering the exciting part of the journey. We are going to see for ourselves if the installation process was successful. I personally find it very fulfilling to verify that an installation was successful even in simple cases, and especially when I have had some trouble down the road.

The first check is to see if you can connect with the command-line MySQL client. If you installed from RPM, it will be in /usr/bin/mysql, which is most likely in your path. If you installed as Unix root, execute PATH=/usr/local/mysql/bin:$PATH; export PATH if you are in the Bourne shell, and setenv PATH /usr/local/mysql/bin:$PATH if you are using the C shell. If you installed as a non-root Unix user, execute PATH=$HOME/mysql/bin ; export PATH for the Bourne shell, and setenv PATH $HOME/mysql/bin:$PATH for the C shell. If you are not sure which shell you are running, or if your shell is neither Bourne nor C, try it both ways—one of them will work, and in the worst case the other will simply produce an error.

Once your path has been set, you can try to connect to the running MySQL server with the command-line client. If you have installed as a root user, either with RPM or with the non-RPM binary distribution, simply type mysql. If you have installed as non-root user, you need to give it a socket argument, so type mysql –socket=$HOME/mysql/data/mysql.sock. If everything works, you'll see something like this:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.43-Max-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Unfortunately, sometimes there are problems during MySQL installation, and you may get an error message instead. If you see one, read the troubleshooting section in this chapter.

To exit the command-line client, type quit or exit.

Post-Installation Setup

When moving into a new home or apartment, it does not quite feel like home until you unload the truck, arrange the furniture, unpack the boxes, and put your stuff where it belongs. In a similar way, after installing MySQL there are things to do to make your installation feel like home.

Just like installing a good lock would be a must to feel secure in your home, the very first thing to do with your newly installed MySQL is setting the root password. This can be done in one shell command:

```
mysqladmin -uroot password tops3cret
```

replacing *tops3cret* with the actual value of the password. If you type your intended root password incorrectly, or simply do not like your initial root password, you can change it in the following way:

```
mysqladmin -uroot -poldtops3cret password newtops3cret
```

Be sure to type your old password exactly the same way after -p as you did when you set the password initially.

An alternative way to set the root password is to connect to the server using the command-line client and execute a special command:

```
mysql -uroot
SET PASSWORD=PASSWORD('tops3cret');
```

The next move will be similar to securing your garage door. Access to the garage is not the same as access to the house, but you still do not want strangers walking in there. MySQL by default installs with an open garage door—a local test user with full privileges on the test database. While it makes things easier for the test and benchmark scripts, this is perhaps not what you want on your system. A malicious local user can cause a lot of problems even with this rather restricted test account. Let's close this garage door by removing the test account. Connect to your MySQL server with mysql -uroot -p*tops3cret* and type

```
DELETE FROM mysql.user WHERE user='';
FLUSH PRIVILEGES;
```

Now it is time to create some users and allow them to operate on some databases. Although there are a multitude of ways to set up users and grant them privileges, there are some common user configuration patterns. Let's talk about the most common ones and provide some examples.

Proxy Database Access

This configuration pattern applies when users are not given direct access to database tables. The only way data can be read from or written to the database is by going through some database client application that performs strict checks on user input before issuing any kind of query. This would be the case, for example, if you have a Web-based shopping cart with the MySQL backend. The users of the shopping cart do not even know that a database is involved at all. All interaction with MySQL occurs through the shopping cart Web application, which plays the role of a proxy or a mediator between the end user and the database.

Although ideally the proxy database application will ensure the integrity of the queries being sent to the database, the reality often differs from the ideal. Even the most thoroughly written, reviewed, and tested applications might have hidden security holes that will allow a malicious user to execute an arbitrary query in the database. Therefore, it is a good idea to give the proxy application the minimum rights it needs to do the job.

Just as there are many ways to arrange furniture in a room for any given furniture collection, and there are a large number of furniture collections to add to the variety, there are many possible ways to set up access for a proxy-style database application, and it would not be possible to cover the issue comprehensively. However, let's consider a few commonly used examples.

Suppose the proxy-style database application runs on the same machine as the database server, operates only in one database called *sales*, and needs to be able to perform schema modifications, such as creating, dropping, or altering tables. We can solve the problem with the following command in the MySQL command-line client after connecting to MySQL as root (mysql -uroot -p):

```
CREATE DATABASE sales;
GRANT ALL ON sales.* TO 'salesuser'@'localhost' IDENTIFIED BY
  'trades3cret';
```

This creates a database called sales; then creates a user salesuser that is allowed to connect from the local host and can perform any table operation as long as it happens in the sales database with trades3cret as the password.

Now let's consider a more complex situation. Our application can be broken into two parts—one that operates on the data itself—inserting, selecting,

deleting or updating rows—and the one that administers the schema, creating, dropping, or altering tables. We solve the problem by creating separate users for each one of the functions:

```
CREATE DATABASE sales;
GRANT SELECT,INSERT,DELETE,UPDATE ON sales.* TO
  'salesuser'@'localhost' IDENTIFIED BY 'trades3cret';
GRANT CREATE,DROP,ALTER,INDEX ON sales.* TO 'salesadmin'@'localhost'
  IDENTIFIED BY 'admins3cret';
```

Now let's consider a slightly more complex scenario. Everything is the same as in the example above, except we have a farm of servers that are running different instances of our application. Schema modification code runs only on server www1.mycompany.com, while the code that operates on data is on 10 servers named www1.mycomany.com through www10.mycomany.com. We proceed with the following:

```
CREATE DATABASE sales;
GRANT SELECT,INSERT,DELETE,UPDATE ON sales.* TO
  'salesuser'@'www%.mycompany.com' IDENTIFIED BY 'trades3cret';
GRANT CREATE,DROP,ALTER,INDEX ON sales.* TO
  'salesadmin'@'www1.mycompany.com' IDENTIFIED BY 'admins3cret';
```

The first GRANT actually accomplishes a bit more than what we wanted. It allows connections for salesuser from any host that resolves into something that starts with www and ends with .mycompany.com. If you do not take proper measures, this could be a security problem, although not a very big one. Without going into a deep discussion of security, placing the database server and your Web servers behind a firewall that blocks the MySQL port (3306 by default) for connections coming from the outside is sufficient to take care of it.

Hosting Provider

If you are running a hosting service and would like to provide access to MySQL for your users, a standard solution is to create a database for each user and grant that user full rights on it:

```
CREATE DATABASE username;
GRANT ALL ON username.* TO 'username'@'localhost' IDENTIFIED BY
  's3cretk3y';
```

Single User

Some installations may require having only one user. For example, you may have installed MySQL on your desktop so that you can experiment with it, perhaps to do some data mining, or to run some application that needs a database

and supports MySQL as one of the options. In this case, the following should take care of your user needs:

```
GRANT ALL ON *.* TO 'me'@'localhost' IDENTIFIED BY 's3cretpwd';
```

Direct Multiple-User Database Access

In some cases, you may have multiple users accessing the same data, and you would like to give them enough flexibility to execute arbitrary SQL queries. For example, you could have an employee roll. Suppose you want Larry to be able to have full rights in the *employee* database, but you want Kerry and Joe only to be able to select from it. Larry's computer is called larry.mycompany.com; Kerry's is called kerry.mycompany.com, and Joe's is called joe.mycompany.com. You do not want them to be able to access MySQL from other computers. Here is how we can solve the problem:

```
CREATE DATABASE employee;
GRANT ALL ON employee.* TO 'larry'@'larry.mycompany.com';
GRANT SELECT ON employee.* TO 'kerry'@'kerry.mycompany.com';
GRANT SELECT ON employee.* TO 'joe'@'joe.mycompany.com';
```

I hope the previous examples have given you enough of a jump-start to decide how you want to initially configure the MySQL privilege system on your server and actually do it. For a more in-depth discussion of access control, see Chapter 5.

At some point in the future you might want to modify the default configuration parameters of your MySQL server. Therefore, we recommend you set up to be able to do so right away. Although MySQL server will look for a default configuration file in several locations, the most common one is /etc/my.cnf on Unix and C:\my.cnf on Windows. While it is possible to handcraft your own my.cnf file from scratch if you are up for a challenge, it is much easier to copy one of the sample distribution files into /etc/my.cnf or C:\my.cnf if you are on Windows, and either use it as is or edit it to make it more suitable for your needs. The sample my.cnf files are placed in /usr/doc/MySQL-version/ in the RPM distribution, /usr/local/mysql/support-files in the binary or source distribution on Unix, and in C:\mysql on Windows.

On a Unix system, to make your life a bit easier you might want to create a file in your home directory called .my.cnf (do not miss the . symbol at the beginning), and put the following into it:

```
[client]

user=user_you_normally_connect_as
password=the_password_of_the_above_user
```

You should replace the placeholders with the actual values of *user* and *password*. This may seem obvious, but some very capable and intelligent people at

times take instructions too literally. It is very important to execute chmod 600 .my.cnf so that this file will not be readable to others since it contains your password. This will allow you to use the command-line client mysql and other MySQL command-line utilities without having to type your MySQL username and password if you are logged into your Unix shell account.

One thing many experienced users like to do after installing MySQL is to perform some simple set of operations to convince themselves that it actually works and to give themselves a sense of satisfaction of actually having accomplished something. Let's try a few simple commands:

```
mysql> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> create table t1 (name char(20), phone char(15));
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t1 values ('Larry','123-4567'), ('Kerry','234-
  5678'), ('Joe','345-6789');
Query OK, 3 rows affected (0.02 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> select * from t1;
+--------+---------+
| name   | phone   |
+--------+---------+
| Larry  | 123-4567|
| Kerry  | 234-5678|
| Joe    | 345-6789|
+--------+---------+
3 rows in set (0.00 sec)
```

It looks like our installation is actually working. However, suppose you encountered problems that you were not able to resolve. In this case, you need to read the upcoming troubleshooting section.

Troubleshooting

Unfortunately, things are not always as rosy as described in the previous section. Problems can arise. There are so many things that could go wrong with even such a simple process as installing MySQL that it is not possible to discuss all of them even if we were to dedicate the entire book to that topic. In this section, we cover some of the common problems, and then provide some guidance on what to do in case your problem is not listed.

mysqld ended

This message comes from the safe_mysqld (in 4.0 mysqld_safe) script on Unix, which is a shell script wrapper around the server binary mysqld. The message means that there was an attempt to start mysqld but that attempt failed. There can be numerous reasons why. The key to diagnosing the problem is reading the error log of mysqld. The error log is located in the data directory and is called `hostname`.err (backticks (``) meaning that to find out the name of your host, you should execute the hostname command). On an RPM installation, the data directory by default is /var/lib/mysql, on a binary installation it is /usr/local/mysql/data, and on a source installation it is /usr/local/mysql/var.

There are four common causes of this onerous mysqld ended message. One is a typo in /etc/my.cnf. If that is the case, you'll see a usage message with a list of all possible mysqld options in the error log. Amidst this mess, somewhere toward the top there will be a message that will say "Invalid option". This message tells you what it is exactly that mysqld is not happy about, and you can correct the typo.

Another cause is trying to run mysqld implicitly as root. This is not allowed for security reasons. In some cases, though, running it as root is worth the security risk because you get better performance—the system will allocate more resources to a root-owned process, but if that is the case, mysqld wants you to tell it so explicitly. To determine if this is the problem, look for a message in the error log telling you that you should not run mysqld as root. The problem can be solved by adding user=root if that was your intention; otherwise, add user=mysql to /etc/my.cnf.

The third problem is permissions. You'll see a message that will mention error 13 in the error log. The solution is to use ls -l first to figure out which ownership/permissions are wrong. Then, use the chown and chmod commands to adjust permissions on the files and subdirectories in the data directory to make all of them accessible both for reading and for writing to the mysql user or the user that mysqld runs as.

The fourth problem comes from forgetting to run scripts/mysql_install_db in the Unix binary installation. When this happens, mysqld will complain about not being able to find the ./mysql/host.frm file. The fix is to just run scripts/mysql_install_db and try to start the server again.

Installation of grant tables failed!

This message is most frequently encountered during the binary installation on Unix. The most likely cause of it is binary incompatibility of the MySQL server

binary and the system libraries or even the architecture itself. You should first try to apply all the vendor patches to your system libraries. If that does not solve the problem, in most cases the only available alternative is to build from source. If you get that message, you should brace yourself for a serious challenge. A system that has a problem running the standard MySQL binary likely will also present some challenges in installing a properly functioning compiler and actually compiling a stable MySQL binary. However, you might succeed in that, and if this system is your only alternative for running MySQL, it is worth a try.

Occasionally, you may get this ugly message during the make install stage of the source installation. This means there is something wrong with the system libraries—something that is fundamentally incompatible with MySQL code. The first thing to try is to apply the latest vendor patches to the system libraries. If that fails, your only hope is to try to debug MySQL server and see if you can patch it.

ERROR 1045: Access denied

This message usually comes from a MySQL command-line client. It means that you have not provided proper authentication credentials (username or password). This could, of course, be simply a typo on your part, or it could be that the access privilege system is not configured quite like you meant to have it—for example, because of a typo in the GRANT command. The first thing to do in this case is to make sure that the username and the password you are supplying are correct. If that is the case, the problem is in the server. If you can connect as root, you can do the following to fix it:

```
USE mysql;
DELETE FROM user WHERE user <> 'root';
FLUSH PRIVILEGES;
```

Then retype the GRANT command(s), double-checking the spelling and the syntax. If there is only one user that is giving you problems, you can run DELETE FROM user WHERE user = *troubleuser* instead of the DELETE command.

If you cannot even log in as root to MySQL (note that we are not talking about the root account in Unix), things are a bit more difficult. You basically shut down the running server, restart it with a special option that ignores the privilege tables and allows anyone to connect, edit the privilege tables, and then either activate them with FLUSH PRIVILEGES or simply shut down and restart the server one more time with regular options. Here is the procedure for the 3.23 Linux RPM distribution:

Log in as root to the system, and at the shell prompt execute the following commands:

```
kill `cat /var/lib/mysql/mysqld.pid`
safe_mysqld --user=mysql --skip-grant --skip-net &
  mysql
```

Now you are connected to a MySQL server running in the skip-grant mode, which resembles the single-user mode in Unix to a certain extent, although there are some fundamental differences, (one of them, ironically, is that still more than one user can connect to the server at the same time). However, any username or password will be accepted as valid, and all connections will have full rights to modify any table or perform any other server operation. Therefore, for security reasons, we explicitly tell the server to disable network connections and allow only local users to connect with the --skip-net option. Accessing the server in this way now will allow us to fix the privilege tables:

```
USE mysql;
UPDATE user SET password=PASSWORD('newns3cret') WHERE user = 'root';
FLUSH PRIVILEGES;
```

Instead of or in addition to running FLUSH PRIVILEGES, you can simply shut down and restart the server by typing the following Unix shell commands (as root):

```
mysqladmin shutdown
safe_mysqld --user=mysql &
```

On a Unix binary installation, the method of root password recovery is basically identical except the paths to different files are different. The pid file (mysqld.pid) is located in the data directory—/usr/local/mysql/data—and safe_mysqld is in /usr/local/mysql/bin. safe_mysqld has been renamed to mysqld_safe in MySQL 4.0. Otherwise, root password recovery procedures will be the same with the 3.23 and 4.0 versions.

ERROR 2002: Can't connect to local MySQL server through socket '/tmp/mysql.sock' (111)

This error usually comes from a MySQL command-line client. It indicates that the client failed to connect to the server. There are basically two possible causes of this error: the server is not running at all, or the server is running but is listening on a different socket. First, determine whether the MySQL server (mysqld) is running at all by using the ps command. On Linux and FreeBSD, the magic incantation is

```
ps auxw | grep mysqld
```

On Solaris, it would be

```
ps -ef | grep mysqld
```

On other Unix systems, the command would follow either the former (Linux/FreeBSD) style or the latter (Solaris) . If the command produces no output, mysqld is not running. If it is running, you'll see a line or several lines from the process list containing mysqld. If the server is running, it is simply listening on a different socket than the one the command client thinks it is listening on. The discrepancy can happen because the command-line client was compiled with different defaults, the server was told to listen on a different socket, or possibly there is an option file (my.cnf) that is telling the client to use the wrong socket. There are many ways to correct the discrepancy, and one is usually not any better than the other. One way is to force a TCP/IP connection to the server through the local interface (instead of using a Unix socket); once the connection is established, ask the server which socket it is listening on, and then tell the command-line client either on the command line or in the .my.cnf option file in the home directory to use the correct socket. Here is an example:

```
sasha@mysql:~ > mysql —host=127.0.0.1
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.23.43-Max-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show variables like 'socket';
+---------------+---------------------------+
| Variable_name | Value                     |
+---------------+---------------------------+
| socket        | /var/lib/mysql/mysql.sock |
+---------------+---------------------------+
1 row in set (0.08 sec)

mysql> exit
Bye
sasha@mysql:~ > mysql —socket=/var/lib/mysql/mysql.sock
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 3.23.43-Max-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> exit
Bye
```

Other Problems

If the symptoms you are getting do not fit into any of the above categories, you should begin by examining the MySQL error log in the data directory. It usually

will have a diagnostic message that you can attempt to decipher. In many cases, the problem and the solution become quite obvious from the message. In other cases, there might be no message, or the message is not directly related to the actual problem. Strange behavior sometimes is the result of binary incompatibility. If you were using a binary distribution, try compiling from source to see if you have better luck.

If compiling from source does not help, the first thing I recommend that you do is write to mysql@lists.mysql.com in hopes that some experienced user might be able to help you, or purchase a MySQL support contract from https://order.mysql.com/ and have a MySQL support team member work with you to resolve the case.

If you are experienced in C++, you might try debugging the problem. Add –-with-debug to the ./configure options, and add –debug to the mysql_install_db and/or safe_mysqld arguments. This will create a trace file, /tmp/mysqld.trace, containing a sequence of function calls and various intermediate values that you can examine and compare against the source to try to understand what is going on. Usually the last few lines of the trace file are the most interesting, although sometimes the trouble happens earlier. You may also add fprintf(stderr, ...) calls. Your debugging messages will be redirected to the error log.

Usually the problem is that some supposedly standard API call does not behave quite the way MySQL team thought it should on your system. For example, fcntl() call may return the wrong value, accept() might fail some unexpected error requiring, or perhaps pthread_mutex_trylock() might have its return values inverted returning 1 on success and 0 when the mutex is locked, instead of the other way around. The problems can usually be tracked down and fixed quite easily by a person familiar with C/C++ and MySQL source code.

CHAPTER 4

Testing Your MySQL Installation

After installing MySQL and performing a number of basic connectivity and functionality tests described in chapter 3, you may want to dig a little deeper and probe the limits of your system by running a number of more thorough tests. In this chapter, we describe five types of tests:

- The standard MySQL test suite (mysql-test-run)
- The server limit test (crash-me)
- The one-threaded standard MySQL benchmark
- The basic multithreaded benchmark (mysqlsyseval)
- Your own tests

The Standard MySQL Test Suite (mysql-test-run)

This test suite is employed frequently during the development process, and MySQL binaries are checked with it prior to each release. Currently, it is available only on a Unix platform, although an effort is in progress to port it to Windows.

The purpose of this test suite is to validate the frequently used functionality in MySQL. It serves as a regression test to ensure that new code has not broken what used to work before. You will want to run this test if you built the binary yourself, or if you are installing it on a system where you suspect binary compatibility problems.

If you are using a binary installation, you can find mysql-test-run in /usr/local/mysql/mysql-test. To run the test, change to the directory where the test is located and execute ./mysql-test-run. You will see output similar to that shown in Listing 4.1.

NOTE

If you installed from source, you should simply do make test before you do make install. If you installed from RPM, you should install the MySQL-Bench package and run the tests from /usr/share/mysql-test.

```
Installing Test Databases
Removing Stale Files
Installing Master Databases
020607  6:42:09  ../sql/mysqld: Shutdown Complete

Installing Slave Databases
020607  6:42:09  ../sql/mysqld: Shutdown Complete

Starting MySQL daemon
Loading Standard Test Databases
Starting Tests

 TEST                           USER   SYSTEM  ELAPSED        RESULT
-------------------------------------------
alias                           0.00     0.01     0.19       [ pass ]
alter_table                     0.00     0.02     0.30       [ pass ]
analyse                         0.02     0.01     0.11       [ pass ]
auto_increment                  0.02     0.02     0.28       [ pass ]
backup                          0.05     0.01     0.30       [ pass ]
bdb-crash                       0.00     0.01     0.08       [ pass ]
bdb-deadlock                    ....     ....     ....       [ skipped ]
bdb                             ....     ....     ....       [ skipped ]
bench_count_distinct            0.09     0.01     0.46       [ pass ]
bigint                          0.00     0.01     0.11       [ pass ]
binary                          0.00     0.01     0.16       [ pass ]
case                            0.02     0.01     0.20       [ pass ]
check                          47.55     0.38   123.52       [ pass ]
comments                        0.02     0.01     0.15       [ pass ]
compare                         0.01     0.01     0.17       [ pass ]
count_distinct                  0.02     0.01     0.28       [ pass ]
create                          0.04     0.01     0.42       [ pass ]
delayed                         0.01     0.02     3.21       [ pass ]
delete                          0.01     0.01     0.28       [ pass ]
dirty-close                     0.02     0.01     0.33       [ pass ]
distinct                        0.07     0.01     1.09       [ pass ]
```

Listing 4.1 Common results generated by mysql-test-run. (continues)

```
drop                          0.04     0.00     0.32       [ pass ]
empty_table                   0.02     0.02     0.17       [ pass ]
err000001                     0.01     0.00     0.04       [ pass ]
explain                       0.02     0.01     0.17       [ pass ]
flush                         0.03     0.01     0.87       [ pass ]
flush_table                   0.03     0.00     0.15       [ pass ]
foreign_key                   0.02     0.00     0.13       [ pass ]
fulltext                      0.03     0.01     0.33       [ pass ]
fulltext_cache                0.01     0.01     0.11       [ pass ]
fulltext_left_join            0.03     0.00     0.16       [ pass ]
fulltext_multi                0.02     0.00     0.25       [ pass ]
fulltext_order_by             0.03     0.02     0.15       [ pass ]
fulltext_update               0.02     0.00     0.24       [ pass ]
func_concat                   0.01     0.02     0.22       [ pass ]
func_crypt                    0.02     0.00     0.16       [ pass ]
func_date_add                 0.03     0.01     0.14       [ pass ]
func_equal                    0.01     0.00     0.19       [ pass ]
func_group                    0.05     0.01     0.41       [ pass ]
func_if                       0.05     0.00     0.17       [ pass ]
func_in                       0.01     0.00     0.13       [ pass ]
func_isnull                   0.00     0.03     0.10       [ pass ]
func_like                     0.03     0.00     0.16       [ pass ]
func_math                     0.00     0.02     0.18       [ pass ]
func_misc                     0.01     0.01     0.08       [ pass ]
func_op                       0.03     0.01     0.16       [ pass ]
func_regexp                   0.01     0.01     0.26       [ pass ]
func_set                      0.01     0.00     0.10       [ pass ]
func_str                      0.03     0.02     0.94       [ pass ]
func_system                   0.01     0.01     0.02       [ pass ]
func_test                     0.03     0.01     0.15       [ pass ]
func_time                     0.06     0.03     0.47       [ pass ]
func_timestamp                0.03     0.00     0.17       [ pass ]
gcc296                        0.01     0.00     0.09       [ pass ]
gemini                        ....     ....     ....       [ skipped ]
group_by                      0.06     0.00     0.47       [ pass ]
having                        0.01     0.01     0.22       [ pass ]
heap                          0.01     0.02     0.56       [ pass ]
innodb                        ....     ....     ....       [ skipped ]
ins000001                     0.02     0.00     0.04       [ pass ]
insert                        0.01     0.01     0.14       [ pass ]
insert_select                 0.01     0.01     0.20       [ pass ]
isam                          0.07     0.02     6.18       [ pass ]
isolation                     ....     ....     ....       [ skipped ]
join                          0.05     0.01     0.77       [ pass ]
join_crash                    0.03     0.01     0.25       [ pass ]
join_outer                    0.11     0.02     1.41       [ pass ]
key                           0.04     0.01     0.36       [ pass ]
```

Listing 4.1 Common results generated by mysql-test-run. (continues)

```
key_diff                      0.01     0.01     0.17       [ pass ]
key_primary                   0.01     0.00     0.15       [ pass ]
keywords                      0.01     0.01     0.15       [ pass ]
kill                          0.02     0.01     5.18       [ pass ]
limit                         0.02     0.00     0.12       [ pass ]
lock                          0.02     0.01     8.36       [ pass ]
merge                         0.05     0.02     0.64       [ pass ]
myisam                        0.06     0.02     6.12       [ pass ]
null                          0.03     0.01     0.14       [ pass ]
null_key                      0.03     0.00     0.29       [ pass ]
odbc                          0.02     0.00     0.14       [ pass ]
order_by                      0.06     0.00     0.68       [ pass ]
outfile                       0.02     0.01     0.14       [ pass ]
overflow                      0.03     0.01     0.14       [ pass ]
raid                          ....     ....     ....       [ skipped ]
range                         0.02     0.01     0.70       [ pass ]
rename                        0.02     0.01     0.21       [ pass ]
replace                       0.01     0.03     0.18       [ pass ]
rollback                      0.02     0.00     0.14       [ pass ]
rpl000001                     0.06     0.01     3.35       [ pass ]
rpl000002                     0.04     0.00     0.23       [ pass ]
rpl000003                     0.04     0.00     0.25       [ pass ]
rpl000004                     0.02     0.00     0.27       [ pass ]
rpl000005                     0.03     0.02     0.25       [ pass ]
rpl000006                     0.03     0.01     0.28       [ pass ]
rpl000007                     0.04     0.01     0.29       [ pass ]
rpl000008                     0.04     0.01     0.34       [ pass ]
rpl000009                     0.02     0.02     0.25       [ pass ]
rpl000010                     0.02     0.03     4.36       [ pass ]
rpl000011                     0.04     0.02     0.31       [ pass ]
rpl000012                     0.05     0.02     0.34       [ pass ]
rpl000013                     0.01     0.02     0.44       [ pass ]
rpl000014                     0.03     0.02     0.41       [ pass ]
rpl000015                     0.02     0.01     0.69       [ pass ]
rpl000016                     0.16     0.03     0.54       [ pass ]
rpl000017                     0.03     0.00     0.10       [ pass ]
rpl000018                     0.01     0.01     0.14       [ pass ]
rpl_alter                     0.02     0.01     0.22       [ pass ]
rpl_empty_master_crash        0.02     0.02     0.18       [ pass ]
rpl_get_lock                  0.55     0.13     6.23       [ pass ]
rpl_mystery22                 0.01     0.02     1.42       [ pass ]
rpl_skip_error                0.03     0.00     0.12       [ pass ]
rpl_sporadic_master           0.02     0.01    12.48       [ pass ]
sel000001                     0.01     0.01     0.05       [ pass ]
sel000002                     0.02     0.01     0.06       [ pass ]
sel000003                     0.01     0.02     0.05       [ pass ]
sel000031                     0.01     0.01     0.10       [ pass ]
```

Listing 4.1 Common results generated by mysql-test-run. (continues)

```
sel000032                     0.02     0.02     0.18       [ pass ]
sel000033                     0.02     0.01     0.18       [ pass ]
sel000100                     0.01     0.00     0.18       [ pass ]
select                        2.90     0.09    96.82       [ pass ]
select_safe                   0.02     0.01     0.26       [ pass ]
show_check                    0.04     0.01     0.95       [ pass ]
status                        0.02     0.01     0.69       [ pass ]
tablelock                     0.02     0.01     0.30       [ pass ]
temp_table                    0.02     0.01     0.61       [ pass ]
truncate                      0.02     0.00     0.14       [ pass ]
type_blob                     0.09     0.00     1.29       [ pass ]
type_date                     0.01     0.01     0.80       [ pass ]
type_datetime                 0.03     0.00     0.35       [ pass ]
type_decimal                  0.07     0.01     1.04       [ pass ]
type_enum                     0.03     0.02     2.76       [ pass ]
type_float                    0.03     0.00     0.31       [ pass ]
type_ranges                   0.04     0.01     0.86       [ pass ]
type_set                      0.01     0.01     0.15       [ pass ]
type_time                     0.02     0.01     0.19       [ pass ]
type_timestamp                0.02     0.00     0.31       [ pass ]
type_uint                     0.02     0.00     0.11       [ pass ]
type_year                     0.02     0.01     0.45       [ pass ]
update                        0.03     0.00     0.34       [ pass ]
user_var                      0.01     0.01     0.18       [ pass ]
varbinary                     0.02     0.01     0.10       [ pass ]
variables                     0.03     0.00     0.28       [ pass ]
warnings                      0.02     0.00     0.15       [ pass ]
-------------------------------------------

Ending Tests
Shutting-down MySQL daemon

Master shutdown finished
Slave shutdown finished
All 136 tests were successful.
```

Listing 4.1 Common results generated by mysql-test-run. (continued)

Note that the test suite starts up its own server instance on a nonstandard port. It does not run against your currently executing server.

When a test passes, you will see the [pass] message on the right side of the test line. Otherwise, the message will say [fail]. If the test ran to completion, but produced incorrect results, you will see the output diff showing the differences between expected and actual results. If the test failed for a different reason (e.g., a server crash), the message will explain at what stage the problem occurred. In any case, you will see a message advising you to report the bug.

When reporting the failure of the test suite on your system, MySQL AB wants you to follow the guidelines in the manual found at www.mysql.com/doc/

en/Reporting_mysqltest_bugs.html. For your convenience, I have quoted them in the following note.

Reporting Test Suite Failures to MySQL AB

"If your MySQL version doesn't pass the test suite you should do the following:

Don't send a bug report before you have found out as much as possible of what when wrong! When you do it, please use the mysqlbug script so that we can get information about your system and MySQL version. See section 1.6.2.3 How to Report Bugs or Problems.
Make sure to include the output of mysql-test-run, as well as contents of all reject files in mysql-test/r directory.
If a test in the test suite fails, check if the test fails also when run by its own:

```
cd mysql-test
mysql-test-run --local test-name
```

If this fails, then you should configure MySQL with —with-debug and run mysql-test-run with the --debug option. If this also fails send the trace file `var/tmp/master.trace' to ftp://support.mysql.com/pub/mysql/secret so that we can examine it. Please remember to also include a full description of your system, the version of the mysqld binary and how you compiled it.

Try also to run mysql-test-run with the --force option to see if there is any other test that fails.

If you have compiled MySQL yourself, check our manual for how to compile MySQL on your platform or, preferable, use one of the binaries we have compiled for you at http://www.mysql.com/downloads/. All our standard binaries should pass the test suite !

If you get an error, like Result length mismatch or Result content mismatch it means that the output of the test didn't match exactly the expected output. This could be a bug in MySQL or that your mysqld version produces slight different results under some circumstances. Failed test results are put in a file with the same base name as the result file with the .reject extension. If your test case is failing, you should do a diff on the two files. If you cannot see how they are different, examine both with od -c and also check their lengths.

If a test fails totally, you should check the logs file in the mysql-test/var/log directory for hints of what went wrong.

If you have compiled MySQL with debugging you can try to debug this by running mysql-test-run with the --gdb and/or --debug options. See section E.1.2 Creating Trace Files. If you have not compiled MySQL for debugging you should probably do that. Just specify the --with-debug options to configure! See section 2.3 Installing a MySQL Source Distribution."

If you do not have enough time to recompile with the —debug option, or for some other reason are unable to follow all of the preceding steps, you should at least use mysqlbug script to gather the system information, include the logs from the mysql-test/var/log directory, report the name of the test that failed, and provide the console output of mysql-test-run script related to the failed test. Failure reports should be sent to bugs@lists.mysql.com. Subscription to the list is not required to post.

The Server Limit Test (crash-me)

The crash-me test name is rather self-explanatory. It explores the limits of the server to see how it will respond and whether it will crash when given certain types of input, such as

- Using particularly long queries
- Handling requests to create a table with a large number of columns
- Creating a key over a large number of fields
- Creating a column of a particular type that is very long

In order to run crashme you first need to make sure you have Perl installed along with DBI/DBD modules for MySQL (for installation instructions, see Chapter 10).

Change to the sql-bench directory (on source and binary installations, it is /usr/local/mysql/sql-bench and on the RPM installation it is /usr/share/sql-bench) and type ./crash-me –config-file=/tmp/mysql.cnf. You will first see output similar to that shown in Listing 4.2.

```
Running ./crash-me 1.54 on 'MySQL 3.23.51 debug log'

I hope you didn't have anything important running on this server....

NOTE: You should be familiar with './crash-me –help' before continuing!

This test should not crash MySQL if it was distributed together with the
running MySQL version. If this is the case you can probably continue without
having to worry about destroying something.

Some of the tests you are about to execute may require a lot of memory. Your
tests WILL adversely affect system performance. It's not uncommon that either
this crash-me test program, or the actual database back-end, will DIE with an
```

Listing 4.2 crash-me disclaimer message. (continues)

```
out-of-memory error. So might any other program on your system if it requests
more memory at the wrong time.

Note also that while crash-me tries to find limits for the database server it
will make a lot of queries that can't be categorized as 'normal'. It's not
unlikely that crash-me finds some limit bug in your server so if you run this
test you have to be prepared that your server may die during it!

We, the creators of this utility, are not responsible in any way if your
database server unexpectedly crashes while this program tries to find the
limitations of your server. By accepting the following question with 'yes',
you agree to the above!

You have been warned!

Start test (yes/no) ?
```

Listing 4.2 crash-me disclaimer message. (continued)

At this, point if you really mean it, type yes and press the Enter key. As the test progresses, you will see output similar to that shown in Listing 4.3.

```
Tables without primary key: yes
SELECT without FROM: yes
Select constants: yes
Select table_name.*: yes
Allows ' and " as string markers: yes
Double '' as ' in strings: yes
Multiple line strings: yes
" as identifier quote (ANSI SQL): error
` as identifier quote: yes
[] as identifier quote: no
Column alias: yes
Table alias: yes
Functions: yes
Group functions: yes
Group functions with distinct: yes
Group by: yes
Group by position: yes
Group by alias: yes
Group on unused column: yes
Order by: yes
Order by position: yes
Order by function: yes
Order by on unused column: yes
```

Listing 4.3 crashme results. (continues)

```
Order by DESC is remembered: no
Compute: no
INSERT with Value lists: yes
INSERT with set syntax: yes
allows end ';': yes
LIMIT number of rows: with LIMIT
SELECT with LIMIT #,#: yes
Alter table add column: yes
Alter table add many columns: yes
Alter table change column: yes
Alter table modify column: yes
Alter table alter column default: yes
Alter table drop column: yes
Alter table rename table: yes
rename table: yes
truncate: yes
Alter table add constraint: yes
Alter table drop constraint: no
Alter table add unique: yes
Alter table drop unique: with drop key
Alter table add primary key: with constraint
Alter table add foreign key: yes
Alter table drop foreign key: with drop foreign key
Alter table drop primary key: drop primary key
Case insensitive compare: yes
Ignore end space in compare: yes
Group on column with null values: yes
Having: yes
Having with group function: yes
Order by alias: yes
Having on alias: yes
binary numbers (0b1001): no
hex numbers (0x41): yes
binary strings (b'0110'): no
hex strings (x'1ace'): no
Value of logical operation (1=1): 1
Simultaneous connections (installation default): 1000
query size: 2097150

Supported sql types
Type character(1 arg): yes
Type char(1 arg): yes
Type char varying(1 arg): yes
Type character varying(1 arg): yes
Type boolean: no
Type varchar(1 arg): yes
Type integer: yes
```

Listing 4.3 crashme results. (continues)

```
Type int: yes
Type smallint: yes
Type numeric(2 arg): yes
Type decimal(2 arg): yes
Type dec(2 arg): yes
Type bit: yes
Type bit(1 arg): yes
Type bit varying(1 arg): no
Type float: yes
Type float(1 arg): yes
Type real: yes
Type double precision: yes
Type date: yes
Type time: yes
Type timestamp: yes
Type interval year: no
Type interval year to month: no
Type interval month: no
Type interval day: no
Type interval day to hour: no
Type interval day to minute: no
Type interval day to second: no
Type interval hour: no
Type interval hour to minute: no
Type interval hour to second: no
Type interval minute: no
Type interval minute to second: no
Type interval second: no
Type national character varying(1 arg): yes
Type national character(1 arg): yes
Type nchar(1 arg): yes
Type national char varying(1 arg): yes
Type nchar varying(1 arg): yes
Type national character varying(1 arg): yes
Type timestamp with time zone: no

Supported odbc types
Type binary(1 arg): yes
Type varbinary(1 arg): yes
Type tinyint: yes
Type bigint: yes
Type datetime: yes

Supported extra types
Type blob: yes
Type byte: no
Type long varbinary: yes
```

Listing 4.3 crashme results. (continues)

```
Type image: no
Type text: yes
Type text(1 arg): no
Type mediumtext: yes
Type long varchar(1 arg): no
Type varchar2(1 arg): no
Type mediumint: yes
Type middleint: yes
Type int unsigned: yes
Type int1: yes
Type int2: yes
Type int3: yes
Type int4: yes
Type int8: yes
Type uint: no
Type money: no
Type smallmoney: no
Type float4: yes
Type float8: yes
Type smallfloat: no
Type float(2 arg): yes
Type double: yes
Type enum(1 arg): yes
Type set(1 arg): yes
Type int(1 arg) zerofill: yes
Type serial: no
Type char(1 arg) binary: yes
Type int not null auto_increment: yes
Type abstime: no
Type year: yes
Type datetime: yes
Type smalldatetime: no
Type timespan: no
Type reltime: no
Type int not null identity: no
Type box: no
Type bool: yes
Type circle: no
Type polygon: no
Type point: no
Type line: no
Type lseg: no
Type path: no
Type interval: no
Type serial: no
Type inet: no
Type cidr: no
```

Listing 4.3 crashme results. (continues)

```
Type macaddr: no
Type varchar2(1 arg): no
Type nvarchar2(1 arg): no
Type number(2 arg): no
Type number(1 arg): no
Type number: no
Type long: no
Type raw(1 arg): no
Type long raw: no
Type rowid: no
Type mlslabel: no
Type clob: no
Type nclob: no
Type bfile: no
Remembers end space in char(): no
Remembers end space in varchar(): no
Supports 0000-00-00 dates: yes
Supports 0001-01-01 dates: yes
Supports 9999-12-31 dates: yes
Supports 'infinity dates: error
Supports YY-MM-DD dates: yes
Supports YY-MM-DD 2000 compilant dates: yes
Storage of float values: round
Type for row id: auto_increment
Automatic row id: _rowid

Supported sql functions
Function +, -, * and /: yes
Function ANSI SQL SUBSTRING: yes
Function BIT_LENGTH: no
Function searched CASE: yes
Function simple CASE: yes
Function CAST: no
Function CHARACTER_LENGTH: yes
Function CHAR_LENGTH: error
Function CHAR_LENGTH(constant): yes
Function COALESCE: yes
Function CURRENT_DATE: yes
Function CURRENT_TIME: yes
Function CURRENT_TIMESTAMP: yes
Function CURRENT_USER: no
Function EXTRACT: yes
Function LOCALTIME: no
Function LOCALTIMESTAMP: no
Function LOWER: yes
Function NULLIF with strings: yes
Function NULLIF with numbers: yes
```

Listing 4.3 crashme results. (continues)

```
Function OCTET_LENGTH: yes
Function POSITION: yes
Function SESSION_USER: no
Function SYSTEM_USER: no
Function TRIM: yes
Function UPPER: yes
Function USER: no
Function concatenation with ||: error

Supported odbc functions
Function ASCII: yes
Function CHAR: yes
Function CONCAT(2 arg): yes
Function DIFFERENCE(): no
Function DIFFERENCE(): no
Function INSERT: yes
Function LEFT: yes
Function LTRIM: yes
Function REAL LENGTH: yes
Function ODBC LENGTH: error
Function ODBC LENGTH: error
Function LOCATE(2 arg): yes
Function LOCATE(3 arg): yes
Function LCASE: yes
Function REPEAT: yes
Function REPLACE: yes
Function RIGHT: yes
Function RTRIM: yes
Function SPACE: yes
Function SOUNDEX: yes
Function ODBC SUBSTRING: yes
Function UCASE: yes
Function ABS: yes
Function ACOS: yes
Function ASIN: yes
Function ATAN: yes
Function ATAN2: yes
Function CEILING: yes
Function COS: yes
Function COT: yes
Function DEGREES: yes
Function EXP: yes
Function FLOOR: yes
Function LOG: yes
Function LOG10: yes
Function MOD: yes
Function PI: yes
```

Listing 4.3 crashme results. (continues)

```
Function POWER: yes
Function RAND: yes
Function RADIANS: yes
Function ROUND(2 arg): yes
Function SIGN: yes
Function SIN: yes
Function SQRT: yes
Function TAN: yes
Function TRUNCATE: yes
Function NOW: yes
Function CURDATE: yes
Function DAYNAME: yes
Function MONTH: yes
Function MONTHNAME: yes
Function DAYOFMONTH: yes
Function DAYOFWEEK: yes
Function DAYOFYEAR: yes
Function QUARTER: yes
Function WEEK: yes
Function YEAR: yes
Function CURTIME: yes
Function HOUR: yes
Function ANSI HOUR: yes
Function MINUTE: yes
Function SECOND: yes
Function TIMESTAMPADD: no
Function TIMESTAMPADD: no
Function TIMESTAMPDIFF: no
Function TIMESTAMPDIFF: no
Function USER(): yes
Function DATABASE: yes
Function IFNULL: yes
Function ODBC syntax LEFT & RIGHT: yes

Supported extra functions
Function & (bitwise and): yes
Function | (bitwise or): yes
Function << and >> (bitwise shifts): yes
Function <> in SELECT: yes
Function =: yes
Function ~* (case insensitive compare): no
Function ADD_MONTHS: no
Function AND and OR in SELECT: yes
Function AND as '&&': yes
Function ASCII_CHAR: no
Function ASCII_CODE: no
Function ATN2: no
```

Listing 4.3 crashme results. (continues)

```
Function BETWEEN in SELECT: yes
Function BIT_COUNT: yes
Function CEIL: no
Function CHARINDEX: no
Function CHR: no
Function CONCAT(list): yes
Function CONVERT: no
Function COSH: no
Function DATEADD: no
Function DATEDIFF: no
Function DATENAME: no
Function DATEPART: no
Function DATE_FORMAT: yes
Function ELT: yes
Function ENCRYPT: yes
Function FIELD: yes
Function FORMAT: yes
Function FROM_DAYS: yes
Function FROM_UNIXTIME: yes
Function GETDATE: no
Function GREATEST: yes
Function IF: yes
Function IN on numbers in SELECT: yes
Function IN on strings in SELECT: yes
Function INITCAP: no
Function INSTR (Oracle syntax): no
Function INSTRB: no
Function INTERVAL: yes
Function LAST_DAY: no
Function LAST_INSERT_ID: yes
Function LEAST: yes
Function LENGTHB: no
Function LIKE ESCAPE in SELECT: yes
Function LIKE in SELECT: yes
Function LN: no
Function LOCATE as INSTR: yes
Function LOG(m,n): no
Function LOGN: no
Function LPAD: yes
Function MDY: no
Function MOD as %: yes
Function MONTHS_BETWEEN: no
Function NOT BETWEEN in SELECT: yes
Function NOT LIKE in SELECT: yes
Function NOT as '!' in SELECT: yes
Function NOT in SELECT: yes
Function ODBC CONVERT: no
```

Listing 4.3 crashme results. (continues)

```
Function OR as '||': yes
Function PASSWORD: yes
Function PASTE: no
Function PATINDEX: no
Function PERIOD_ADD: yes
Function PERIOD_DIFF: yes
Function POW: yes
Function RANGE: no
Function REGEXP in SELECT: yes
Function REPLICATE: no
Function REVERSE: yes
Function ROOT: no
Function ROUND(1 arg): yes
Function RPAD: yes
Function SEC_TO_TIME: yes
Function SINH: no
Function STR: no
Function STRCMP: yes
Function STUFF: no
Function SUBSTRB: no
Function SUBSTRING as MID: yes
Function SUBSTRING_INDEX: yes
Function SYSDATE: yes
Function TAIL: no
Function TANH: no
Function TIME_TO_SEC: yes
Function TO_DAYS: yes
Function TRANSLATE: no
Function TRIM; Many char extension: error
Function TRIM; Substring extension: yes
Function TRUNC: no
Function UID: no
Function UNIX_TIMESTAMP: yes
Function USERENV: no
Function VERSION: yes
Function WEEKDAY: yes
Function automatic num->string convert: yes
Function automatic string->num convert: yes
Function concatenation with +: error

Supported where functions
Function = ALL: no
Function = ANY: no
Function = SOME: no
Function BETWEEN: yes
Function EXISTS: no
Function IN on numbers: yes
```

Listing 4.3 crashme results. (continues)

```
Function LIKE ESCAPE: yes
Function LIKE: yes
Function MATCH UNIQUE: no
Function MATCH: no
Function MATCHES: no
Function NOT BETWEEN: yes
Function NOT EXISTS: no
Function NOT LIKE: yes
Function NOT UNIQUE: no
Function UNIQUE: no

Supported sql group functions
Group function AVG: yes
Group function COUNT (*): yes
Group function COUNT column name: yes
Group function COUNT(DISTINCT expr): yes
Group function MAX on numbers: yes
Group function MAX on strings: yes
Group function MIN on numbers: yes
Group function MIN on strings: yes
Group function SUM: yes
Group function ANY: no
Group function EVERY: no
Group function SOME: no

Supported extra group functions
Group function BIT_AND: yes
Group function BIT_OR: yes
Group function COUNT(DISTINCT expr,expr,...): yes
Group function STD: yes
Group function STDDEV: yes
Group function VARIANCE: no

mixing of integer and float in expression: yes
No need to cast from integer to float: yes
Is 1+NULL = NULL: yes
Is concat('a',NULL) = NULL: yes
LIKE on numbers: yes
column LIKE column: yes
update of column= -column: yes
String functions on date columns: yes
char are space filled: no
Update with many tables: no
DELETE FROM table1,table2...: no
Update with sub select: no
Calculate 1—1: yes
ANSI SQL simple joins: yes
```

Listing 4.3 crashme results. (continues)

```
max text or blob size: 2097119
constant string size in where: 2097115
constant string size in SELECT: 2097141
return string size from function: 2096128
simple expressions: 1507
big expressions: 10
stacked expressions: 1507
OR and AND in WHERE: 84773
tables in join: 31
primary key in create table: yes
unique in create table: yes
unique null in create: yes
default value for column: yes
default value function for column: no
temporary tables: yes
create table from select: yes
index in create table: yes
create index: yes
drop index: with 'ON'
null in index: yes
null in unique index: yes
null combination in unique index: yes
null in unique index: yes
index on column part (extension): yes
different namespace for index: yes
case independent table names: no
drop table if exists: yes
create table if not exists: yes
inner join: yes
left outer join: yes
natural left outer join: yes
left outer join using: yes
left outer join odbc style: yes
right outer join: yes
full outer join: no
cross join (same as from a,b): yes
natural join: yes
union: no
union all: no
intersect: no
intersect all: no
except: no
except all: no
except: no
except all: no
minus: no
natural join (incompatible lists): yes
```

Listing 4.3 crashme results. (continues)

```
union (incompatible lists): no
union all (incompatible lists): no
intersect (incompatible lists): no
intersect all (incompatible lists): no
except (incompatible lists): no
except all (incompatible lists): no
except (incompatible lists): no
except all (incompatible lists): no
minus (incompatible lists): no
subqueries: no
insert INTO ... SELECT ...: yes
transactions: no
atomic updates: no
views: no
foreign key syntax: yes
foreign keys: no
Create SCHEMA: no
Column constraints: no
Table constraints: no
Named constraints: no
NULL constraint (SyBase style): yes
Triggers (ANSI SQL): no
PSM procedures (ANSI SQL): no
PSM modules (ANSI SQL): no
PSM functions (ANSI SQL): no
Domains (ANSI SQL): no
lock table: yes
many tables to drop table: yes
drop table with cascade/restrict: yes
– as comment (ANSI): yes
// as comment (ANSI): no
# as comment: yes
/* */ as comment: yes
insert empty string: yes
Having with alias: yes
table name length: 64
column name length: 64
select alias name length: +512
table alias name length: +512
index name length: 64
max char() size: 255
max varchar() size: 255
max text or blob size: 2097119 (cache)
Columns in table: 3398
unique indexes: 32
index parts: 16
max index part length: 255
```

Listing 4.3 crashme results. (continues)

```
index varchar part length: 255
indexes: 32
index length: 500
max table row length (without blobs): 65534
table row length with nulls (without blobs): 65502
number of columns in order by: +64
number of columns in group by: +64
crash-me safe: yes
reconnected 0 times
```

Listing 4.3 crashme results. (continued)

In some cases, crashme might actually succeed in crashing the server. If that were to happen, it is important to understand the reason and verify that the limitation is not critical for your application. For example, some crashme tests might push the server usage of system resources to the point that the operating system decides to panic, but the tested limit might actually far exceed your needs. In this case, you may lower the appropriate limit in /tmp/mysql.cnf and try the test again.

Each test explores a certain limit of the server. The output of each test is rather self-explanatory. For example, indexes: 32 means that a table can have the maximum of 32 indexes, while index length: 500 means that the longest key can span no more than 500 bytes. If you see a + before a number, this means that the MySQL team has tested the lower limit but the actual value can be higher. For example, number of columns in order by: +64 means that the server can handle a query using 64 columns in an ORDER BY expression but that the team has not tested a higher limit. In actuality, it might be able to do more.

The One-Threaded Standard MySQL Benchmark

The standard MySQL benchmark is a very comprehensive test of the performance of the server. It is good for performance comparisons and can answer many questions about how fast a certain type of query will execute. However, it does have a couple of drawbacks. The results are difficult to read and digest for the typical database programmer or administrator. In addition, the benchmark is one-threaded, which means that it does not answer questions about server scalability.

As with crash-me, in order to run the standard MySQL benchmark, you will also need to have Perl with the DBI/DBD MySQL driver installed. To run the test, you again need to be in the sql-bench directory. The benchmark needs to be able to

create the output directory if it does not exist, and if it does exist, it needs to be able to create files in that directory. If you are not running the benchmark as root, you may run into permission problems. One way to avoid them is to first run mkdir /tmp/output as a non-root user, then in the root shell change to the sql-bench directory and run ln -s /tmp/output output. After taking care of this minor issue, you can type ./run-all-tests as a regular non-root user and you will see something similar to Listing 4.4.

```
Benchmark DBD suite: 2.10
Date of test:        2002-06-10 21:53:45
Running tests on:    Linux 2.4.18-rc1 i686
Arguments:
Comments:
Limits from:
Server version:      MySQL 3.23.43 Max log

ATIS: Total time: 72 wallclock secs (12.22 usr  3.76 sys +  0.00 cusr
  0.00 csys =  0.00 CPU)
alter-table: Total time: 540 wallclock secs ( 0.45 usr  0.09 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
big-tables: Total time: 50 wallclock secs (11.10 usr  7.05 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
connect: Total time: 119 wallclock secs (46.18 usr 18.40 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
create: Total time: 300 wallclock secs (15.56 usr  2.43 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
insert: Total time: 3813 wallclock secs (547.14 usr 133.09 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
select: Total time: 1609 wallclock secs (77.05 usr 10.74 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)
wisconsin: Total time: 84 wallclock secs ( 4.75 usr  1.49 sys +
  0.00 cusr  0.00 csys =  0.00 CPU)

All 8 tests executed successfully

Totals per operation:
Operation                       seconds  usr     sys     cpu   tests
alter_table_add                  300.00   0.25   0.04   0.00   992
alter_table_drop                 228.00   0.14   0.03   0.00   496
connect                           18.00   9.22   3.27   0.00   10000
connect+select_1_row              22.00   9.79   3.88   0.00   10000
connect+select_simple             20.00   9.95   3.43   0.00   10000
count                             53.00   0.07   0.01   0.00   100
count_distinct                   145.00   1.13   0.08   0.00   2000
count_distinct_big               137.00   8.70   2.28   0.00   120
count_distinct_group              92.00   1.49   0.40   0.00   1000
```

Listing 4.4 Sample results generated by the MySQL benchmark test. (continues)

```
count_distinct_group_on_key        72.00   0.72   0.06   0.00   1000
count_distinct_group_on_key_parts  91.00   1.34   0.36   0.00   1000
count_group_on_key_parts           63.00   1.65   0.40   0.00   1000
count_on_key                      608.00  22.91   2.51   0.00   50100
create+drop                        61.00   4.31   0.55   0.00   10000
create_MANY_tables                 67.00   3.22   0.24   0.00   10000
create_index                        6.00   0.00   0.00   0.00   8
create_key+drop                    74.00   5.59   0.79   0.00   10000
create_table                        0.00   0.01   0.02   0.00   31
delete_all                         22.00   0.00   0.00   0.00   12
delete_all_many_keys               80.00   0.03   0.01   0.00   1
delete_big                          1.00   0.00   0.00   0.00   1
delete_big_many_keys               80.00   0.03   0.01   0.00   128
delete_key                          8.00   0.87   0.29   0.00   10000
drop_index                          6.00   0.00   0.00   0.00   8
drop_table                          0.00   0.00   0.00   0.00   28
drop_table_when_MANY_tables        68.00   0.91   0.34   0.00   10000
insert                            928.00  32.82  11.70   0.00   350768
insert_duplicates                  41.00   6.35   2.82   0.00   100000
insert_key                        301.00  13.34   3.50   0.00   100000
insert_many_fields                 20.00   0.72   0.11   0.00   2000
insert_select_1_key                 8.00   0.00   0.00   0.00   1
insert_select_2_keys               10.00   0.00   0.00   0.00   1
min_max                            34.00   0.04   0.03   0.00   60
min_max_on_key                    245.00  34.56   4.22   0.00   85000
multiple_value_insert              11.00   2.28   0.07   0.00   100000
order_by_big                       72.00  27.38  10.69   0.00   10
order_by_big_key                   52.00  24.67  10.29   0.00   10
order_by_big_key2                  51.00  26.29  10.71   0.00   10
order_by_big_key_desc              53.00  25.78  10.33   0.00   10
order_by_big_key_diff              68.00  27.40  11.16   0.00   10
order_by_key                        4.00   1.44   0.38   0.00   500
order_by_key2_diff                  5.00   2.54   0.88   0.00   500
order_by_range                      6.00   1.48   0.55   0.00   500
outer_join                         80.00   0.01   0.00   0.00   10
outer_join_found                   75.00   0.00   0.00   0.00   10
outer_join_not_found               57.00   0.02   0.00   0.00   500
outer_join_on_key                  66.00   0.01   0.00   0.00   10
select_1_row                        3.00   0.79   0.58   0.00   10000
select_2_rows                       4.00   0.78   0.44   0.00   10000
select_big                         96.00  41.57  16.71   0.00   10080
select_column+column                5.00   0.59   0.43   0.00   10000
select_diff_key                   235.00   0.43   0.05   0.00   500
select_distinct                    15.00   2.31   0.76   0.00   800
select_group                       76.00   2.29   0.34   0.00   2911
select_group_when_MANY_tables      28.00   1.53   0.51   0.00   10000
select_join                        21.00   6.02   2.12   0.00   200
```

Listing 4.4 Sample results generated by the MySQL benchmark test. (continues)

```
select_key                    177.00  94.70  11.90   0.00   200000
select_key2                   187.00  95.59  12.16   0.00   200000
select_key_prefix             190.00  89.38  11.97   0.00   200000
select_many_fields             30.00  10.37   6.94   0.00   2000
select_range                  278.00   9.37   0.90   0.00   410
select_range_key2              25.00   8.91   1.44   0.00   25010
select_range_prefix            25.00   8.51   1.23   0.00   25010
select_simple                   3.00   0.76   0.34   0.00   10000
select_simple_join              2.00   0.88   0.15   0.00   500
update_big                     77.00   0.00   0.00   0.00   10
update_of_key                 122.00   4.66   1.65   0.00   50256
update_of_key_big              41.00   0.05   0.02   0.00   501
update_with_key               508.00  23.04   9.38   0.00   300000
wisc_benchmark                  6.00   2.34   0.58   0.00   114
TOTALS                       6663.00 714.33 177.04   0.00  1946237
```

Listing 4.4 Sample results generated by the MySQL benchmark test. (continued)

The first column contains the name of the operation, which in most cases is self-descriptive. The second column gives the total amount of time it took to run *N* iterations of the test. The third column shows the user CPU time on the client, while the fourth displays the system CPU on the client. The fifth column is ideally supposed to indicate the total CPU for the client, but because of a bug in the benchmark code, it gets zeroed out. The sixth column shows the number of iterations.

While the results of this benchmark are difficult to digest at first glance, if you spend some time studying them and if you are familiar with Perl and are willing to dig down into the source of the benchmark code and learn the details, it could prove helpful in anticipating what kind of performance your server will give you under your mix of queries.

I must admit, though, that this is not a task for a novice. It requires a fair amount of experience, insight, and skill. Nevertheless, I would encourage you to give it a try. To get you started, let's take a look at the insert_key test and do the digging. Once you are familiar with the paradigm, you will be able to decipher other tests.

In the sql-bench directory, we find a group of files that start with test-. We need to find which one of them is responsible for the insert_key test. To do so, we use the grep utility:

```
grep "Time for insert_key" test-*
```

which produces the following results:

```
test-insert:print "Time for insert_key ($many_keys_loop_count): " .
```

We now open test-insert file in a text editor and locate the line matched by grep. Prior to that line, you will see the code in Listing 4.5.

```
if (($opt_fast || $opt_fast_insert) && $server->{'limits'}-
>{'insert_multi_value'})
{
  $query_size=$server->{'limits'}->{'query_size'};

  print "Inserting $opt_loop_count multiple-value rows in order\n";
  $res=$query;
  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $tmp= "($i,$i,$i,'ABCDEFGHIJ'),";
    if (length($tmp)+length($res) < $query_size)
    {
      $res.= $tmp;
    }
    else
    {
      $sth = $dbh->do(substr($res,0,length($res)-1)) or die $DBI::errstr;
      $res=$query . $tmp;
    }
  }
  print "Inserting $opt_loop_count multiple-value rows in reverse order\n";
  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $tmp= "(" . ($total_rows-1-$i) . "," .($total_rows-1-$i) .
      "," .($total_rows-1-$i) . ",'BCDEFGHIJK'),";
    if (length($tmp)+length($res) < $query_size)
    {
      $res.= $tmp;
    }
    else
    {
      $sth = $dbh->do(substr($res,0,length($res)-1)) or die $DBI::errstr;
      $res=$query . $tmp;
    }
  }
  print "Inserting $opt_loop_count multiple-value rows in random order\n";
  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $tmp= "(" . $random[$i] . "," . $random[$i] . "," . $random[$i] .
      ",'CDEFGHIJKL')," or die $DBI::errstr;
    if (length($tmp)+length($res) < $query_size)
    {
      $res.= $tmp;
```

Listing 4.5 Test-insert partial listing. (continues)

```
    }
    else
    {
      $sth = $dbh->do(substr($res,0,length($res)-1)) or die $DBI::errstr;
      $res=$query . $tmp;
    }
  }
  $sth = $dbh->do(substr($res,0,length($res)-1)) or die $DBI::errstr;
}
else
{
  print "Inserting $opt_loop_count rows in order\n";
  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $sth = $dbh->do($query . "($i,$i,$i,'ABCDEFGHIJ')") or die $DBI::errstr;
  }

  print "Inserting $opt_loop_count rows in reverse order\n";
  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $sth = $dbh->do($query . "(" . ($total_rows-1-$i) . "," .
                        ($total_rows-1-$i) . "," .
                        ($total_rows-1-$i) . ",'BCDEFGHIJK')")
      or die $DBI::errstr;
  }

  print "Inserting $opt_loop_count rows in random order\n";

  for ($i=0 ; $i < $opt_loop_count ; $i++)
  {
    $sth = $dbh->do($query . "(". $random[$i] . "," . $random[$i] .
                        "," . $random[$i] . ",'CDEFGHIJKL')") or die
$DBI::errstr;
  }
}

$end_time=new Benchmark;
print "Time for insert (" . ($total_rows) . "): " .
  timestr(timediff($end_time, $loop_time),"all") . "\n\n";
```

Listing 4.5 Test-insert partial listing. (continued)

If you are familiar with Perl, you can see from the source code that we have two paths of execution depending on the benchmark configuration options. The variables that start with *opt* are affected only by command-line arguments. Members of the $server->{'limits'} object are set in the server-cfg file. If the program is being run with --fast or --fast-insert options and if server-cfg tells us that the server is capable of multirow insert (e.g., INSERT INTO t1 (n1,n2)

VALUES (1,2),(3,4),(5,6);), we take advantage of multirow insert. Otherwise, we perform regular inserts, first in the key order, then in reverse key order, and then in random order.

Since we did not give any special command-line arguments, our test was run using regular, one-row inserts, which are slower than multirow ones when performance is measured in terms of the number of rows inserted over a fixed period of type as opposed to the number of queries over a fixed period of time. Nevertheless, from the test results, we see that we were able to insert 100,000 rows in 301 seconds, which amounts to 332 inserts per second.

The Basic Multithreaded Benchmark (mysqlsyseval)

While the standard benchmark answers many questions about MySQL's performance on a particular system, because it is one-threaded it does not address the issue of performance under high concurrency. Unfortunately, as of this writing MySQL AB does not have any official multithreaded benchmark suites, although some work is currently in progress. However, the MySQL team has written some in-house tools for taking a quick look at scalability patterns of MySQL. One such tool, mysqlsyseval, can be downloaded from the book Web site (www.wiley.com/compbooks/pachev). You will find binaries for x86 Linux, FreeBSD, and Solaris, as well as the source. To compile the source, first make sure you have MySQL client libraries installed, including the thread-safe client (libmysqlclient_r); then download mysqlsyseval.c and run the following command:

```
cc -o mysqlsyseval  -I/usr/include/mysql mysqlsyseval.c -
  L/usr/lib/mysql -lmysqlclient_r -lz -lpthread
```

To run this test, simply type ./mysqlsyseval. You may need to give it a socket argument with --socket=/path/to/mysql.sock if the compiled-in default value differs from the actual location of the socket. You should see the output similar to this:

```
MySQL BogoMips: 370
Data inserted in 0.731 s, 13679 rows per second
Blob field updated in 3.661 s, 2731 rows per second
Starting multi-threaded test, each test will run a total of 32768
  queries, please be patient...
Clients Throughput(q/sec)       Scale factor(%)
1       2140    100
2       2161    100
4       2168    101
8       2208    103
```

```
16      2187    102
32      2176    101
64      2164    101
```

The first line shows the result of a crude CPU benchmark. The name "bogomips" was borrowed from a Linux kernel CPU calibration test, which runs a loop to roughly guess the number of millions of instructions per second the processor is capable of executing. Since the test is only an approximation, the result would be called "bogus" by a harsh purist. Thus the name "bogomips." In this benchmark, we perform a similar test of the server CPU by making it execute a simple computation in a loop (1+1) using the BENCHMARK() SQL function.

The number of bogomips tends to hover in the area of the number of MHz plus or minus 25 percent or so for Pentium III. For other hardware, the result will depend on how well it can execute the loop that adds two 64-bit integers inside the MySQL code. The result, although only a rough guess, gives you an idea of the CPU capacity.

The first column shows the number of concurrent threads. The second column lists the number of queries per second that MySQL Server was able to perform during the test. The third column displays the percentage throughput relative to a one-client test.

You should pay particular attention to the third column if you are planning to have many concurrent connections. For a single-CPU system, the result should hover around 100 percent for all tests and not drop significantly. For a dual x86 processor, it is reasonable to expect a 150–170 percent scale factor for tests with two or more clients. For a quad x86, you should see something around 250–300 percent.

The test queries by default randomly select count of rows on a key range from a table with 10,000 records. The type of query and the number of rows could be changed with special option arguments to mysqlsyseval.

Note that the client and the server do not have to be on the same machine. You can take advantage of that if you would like to measure the performance of a Windows server, for example, since mysqlsyseval does not yet work on Windows. To make it run against a remote server, you should give it the --host= argument, specifying the name of the host.

In addition to --host, the utility supports a number of configuration options. The full listing can be obtained by running it with the --help option, which produces the following output:

```
Usage: ./mysqlsyseval [options]
Available options:
```

```
-D,—database Database to connect to
-h,—host Host to connect to
-p,—password MySQL password
-P,—port Port to connect to
-S,—socket Unix socket to connect to
-T,—table-type Table type - one of: HEAP,MyISAM,InnoDB,ISAM,BDB
-u,—user MySQL user
-c,—max-clients Increase concurrency up to this number of clients
-m,—min-clients Start tests with this number of clients
-q,—num-queries Total number of queries to perform
-t,—query-type Query type - one of: select-count-range,select-
distinct,update,select-blob,update-blob
-n,—num-rows Number of rows in the test table
-b,—bogoloops-time Low number of seconds limit for the MySQL bogomips
test - the higher the value, the more precise the result.
-B,—blob-size Size of the blob to insert into the test table
-?,—help This message
-v,—version Show version
```

Probably the most interesting—or at least worth additional discussion beyond the help message itself—of those options are num-queries, num-rows, table-type, min-clients, max-clients, blob-size, and query-type.

The option num-queries affects the duration of the test. It is worthwhile increasing this value if you are running a lot of concurrent clients to minimize the effect of the timing inaccuracy caused by the client thread creation and connection overhead. If you are running a test with only a few clients against a slow server, you may want to use a lower value of num-queries. The default is 10,000.

The option num-rows will affect the size of the test table. Increasing the value will make the test run longer, largely because it will take longer to insert all of the rows for the test, but also because the larger the table gets the slower the queries become. The default is 10,000. Increase (or decrease) it as appropriate for your circumstances and goals.

The option table-type allows you to benchmark different table handlers in MySQL. Thus, you can find out, for example, whether MyISAM or InnoDB would be faster for a particular query and table size on your system, and how this will change as you increase the number of clients. The default table type is MyISAM.

The option min-clients represents the number of clients used in the first test. Each subsequence test will double the number of clients until max-clients is reached. If you want to run only one test with, for example, 300 clients, you can execute the following:

```
./mysqlsyseval —max-clients=300 —min-clients=300
```

The option blob-size specifies the size of the blob in the benchmark table in bytes and can be used to control the length of the record. You can also use it to see how efficient MySQL is with large blobs. The default is 8192.

The option query-type, which specifies the type of the query to run during the benchmark, can be one of the following:

- select-count-range
- select-distinct
- update
- select-blob
- update-blob

The default is select-count-range.

For those of you who are somewhat familiar with C, perhaps the most concise way to illustrate what each of these options does is to quote the source (Listing 4.6).

```
 switch(query_type)
      {
      case Q_SELECT_COUNT_RANGE:
      {
        int start_range = rand() % test_rows;
        sprintf(buf, "select count(*) from t1 where id >= %d and id <=
%d",
            start_range,
           start_range + 10 );
      break;
      }
      case Q_SELECT_BLOB:
      {
        int id = rand() % test_rows;
        sprintf(buf, "select id,word,b from t1 where id = %d",
                        id);
              break;
      }
      case Q_UPDATE:
      {
        int id = rand() % test_rows;
        sprintf(buf, "update t1 set word = '%d' where id = %d", rand(),
                    id);
              break;
      }
      case Q_UPDATE_BLOB:
```

Listing 4.6 Explanation of query-type from MySQL source code. (continues)

```
      {
        int id = rand() % test_rows;
        char* buf_end;
        int blob_val = rand();
        sprintf(buf, "update t1 set b = '");
        buf_end = buf + strlen(buf);
        memset(buf_end, 'g', blob_size);
        buf_end += blob_size;
       *buf_end++ = '\'';
        sprintf(buf_end, " where id = %d",
                  id);
       break;
      }
      case Q_SELECT_DISTINCT:
      {
        int start_range = rand() % test_rows;
        sprintf(buf, "select distinct word from t1 where id > %d and
id < %d",
                      start_range, start_range + 100);
        break;
      }
      }
```

Listing 4.6 Explanation of query-type from MySQL source code. (continued)

For those not familiar with C, select-count-range, which is the default, will select the number of records from a range of keys starting at a random location and ending at 10 after. select-blob locates a record based on a random primary key value. It will then pull the entire record out, including the blob. update just updates a string column one random record, selected on a random value of the primary key. update-blob does the same as update, except it updates the blob column. select-distinct selects distinct strings from a random 100-wide range of primary keys.

Your Own Tests

While using pre-existing tests is convenient and saves development time, in some cases it is not sufficient to accurately predict the performance of MySQL for a certain application. Additionally, although MySQL has a reputation for stability and performance, occasionally bugs are introduced during upgrades that may affect your application. Having your own testing procedures will make your upgrades much more trouble-free. You will not need to guess how stable the newly released version would be, as you will have a way to test it for yourself. To create your own test suite, you can expand the existing tests or write your own from scratch.

If you already have an application, you can simply run your MySQL server with the —log option while you try different things with your application as a user (or have a number of your beta testers to try it out), and then examine the log in the data directory to see your query patterns. If the application does not yet exist, you have to use your imagination and think of what kind of queries it will run most frequently.

Once you have an idea of your query patterns, you should write a program in C, Perl, PHP, Java, or whatever other client language you prefer that will execute queries with approximately the same pattern, and then time the operations that you are interested in. You might want to record the results of each query run against a test sample of your database on the first run, and use them as "master results" to compare against on subsequent runs. If the result does not match, you should signal an error. You should also record the performance timing for each query, and flag errors on subsequent runs if the new timing exceeds the "master" timing by a wide margin. These suggestions, of course, assume that your application is functioning properly on the first run of the test.

When writing your own tests, focus on two aspects: functionality and performance. Functionality tests will ensure that all queries that are important in your application produce correct results, while performance tests will ensure that the application is stable under high load and provides acceptable response times.

One common way to create a functionality test for MySQL is to have a test suite for your entire application. As you test various features of your application, certain MySQL queries will be executed. If there are any problems with MySQL that affect your application, they will show up in a thorough test suite. The same approach works for performance. The essence of this approach is that you treat MySQL as a module in your application, which needs to be tested just like any other module.

Many users like to test MySQL performance before they commit to developing their application with MySQL. They like to test 5 to 10 queries that will be common in their application, and see how MySQL will handle them under load. One way to accomplish this task is to modify mysqlsyseval and add the desired type of query. To add a new query type, just add a new type to the QUERY_TYPE enum at the end, add the string for the new query type at the end of the types array in get_query_type(), and add a case statement for the new query type to run_thread() similar to the ones quoted earlier. Be sure to include at the end of the statement the variable buf, specifying the query you want to run. You can also change the table definition by modifying the argument to the sprintf() call, which builds the CREATE TABLE statement in benchmark_threads().

You may also consider extending the standard Perl benchmark, although this is a more difficult task. It would be worthwhile, though, if you want to compare MySQL performance on a particular query with other databases. The Perl benchmark uses the DBI interface, which is highly portable, and requires you to change only the value of $self->{'data_source'} and perhaps a couple of server-specific limit variables to be able to run the test against a different server.

If you would like to test raw functionality, you may want to consider extending the standard regression test suite. The advantage of this kind of test is that you can submit it to MySQL AB, and it will likely be accepted and included in the standard test suite. This guarantees that your test succeeds in all subsequent releases. To submit a test case in this format, send an e-mail to internals@lists.mysql.com.

Let's illustrate the process with an example. Suppose we want to test the following query:

```
SELECT a,b FROM t1 WHERE a > b;

on the table with the following data:

a    b       c
1    2       3
8    3       1
4    4       7
```

We first decide on the name of our test. Suppose we want to call it abctest. Create a file in the mysql-test/t directory called abctest.test. We put the following into the file:

```
drop table if exists t1; # clean up from possible previous crash
create table t1 (a int, b int, c int); # create the table
insert into t1 (a,b,c) values  (1,2,3),(8,3,1),(4,4,7); #populate
  the table
select a,b FROM t1 WHERE a > b; # run the query and check or record
  the result
drop table t1;    # clean up after the work is done
```

and in the mysql-test directory run the following to record the result to compare against in the future:

```
./mysql-test-run —record —local abctest
```

Afterward, we check the result file to make sure it is correct and edit it as necessary. The result file is located in r/abctest.result, and will contain the output of the SELECT query. In version 4.0, in addition to the result of SELECT the queries that have been run during the test will also be recorded.

Now that we have created t/abctest.test manually and r/abctest.result by recording and possible subsequent manual editing, the test suite will contain

the abctest test, and it will be run every time you execute ./mysql-test-run. To submit the test for inclusion, you need to send those two files to the MySQL development team. Please be aware that MySQL mailing lists do not accept attachments. You will need to put up the files at some download location, for example, on your FTP or HTTP server, and send the URL to internals@lists.mysql.com.

As you can see, there are numerous ways to create your own tests. You can choose the most appropriate method depending on your situation. Having a customized test suite for MySQL and your application as it integrates with MySQL will facilitate the development and maintenance processes and increase your system stability.

CHAPTER 5

Access Control and Security

This chapter provides an overview of how to secure your MySQL server and discusses two particular aspects of security: security at the database level, and security at the operating system and network level. Every aspect of security is critical and needs to be scrutinized when you're configuring a server. In a house with open windows, extra-strong door locks will protect you only from a criminal who is not good at climbing in through those windows.

When you're considering system security, you should learn to think like a person who is trying to compromise your system. To be successful in securing a system, you need to develop a benevolently malicious mind—in other words, understand the mind of a bad guy well enough to predict what he is going to try. However, you must also remember this fine line: Legitimate users should not be required to jump through an unreasonable number of hoops every time they need access to a system resource.

Let's discuss how these general security principles apply to a MySQL server installation.

MySQL Access Privilege System

The first important concept of the MySQL access privilege system is that of a *user*. Each user has a given set of privileges. A user is defined not only by a username, but also by the host from which the user is connecting. For example, joe@company1.com could have a completely different set of privileges than joe@company2.com.

It is possible to have a wildcard character (%) in both the username and the connection originating host. Thus, you can give the same set of privileges to a group of users who match a certain wildcard pattern because MySQL will treat them as a single user, and all of them will have the same password. However, MySQL does not support the concept of a group of users beyond what can be accomplished with this wildcard hack.

Each user can have privileges on a database, table, or column level. Additionally, some server administration and system access privileges are not connected with the concept of a database, table, or column. Table 5.1 lists the privileges in MySQL 3.23, which are the same as in version 4.0.1.

Table 5.1 MySQL Privileges

PRIVILEGE NAME	SCOPE	DESCRIPTION
Select	Database,Table,Column	Allows the user to select data.
Insert	Database,Table	Allows the user to insert rows.
Update	Database,Table,Column	Allows the user to update data.
Delete	Database,Table	Allows the user to delete rows.
Create	Database,Table	Allows the user to create tables.
Drop	Database,Table	Allows the user to drop tables.
Reload	Server	Allows the user to perform various refresh operations.
Shutdown	Server	Allows the user to shut down the server.
Process	Server	Allows the user to view information about currently running connections that are owned by other users, and kill those connections.
File	Server	Allows the user to execute commands that read from and write to files residing on the server. Also allows the user to read the binary replication log.
Grant	Database,Table,Column	Allows the user to grant their privileges to other users.
Alter	Database,Table	Allows the user to modify table schema.

Beginning in version 4.0.2, new privilege concepts were added, and some of the old privilege capabilities now require the new privilege type. Here is a summary of the changes:

- **Super:** This privilege takes some of the functionality of the former Process and File privileges. It is now required for KILL, SHOW MASTER STATUS, SHOW SLAVE STATUS, LOAD MASTER DATA, SHOW INNODB STATUS, SHOW BINARY LOGS, and PURGE LOGS TO.
- **Replication Slave:** This privilege permits the user to read the master binary update log, and thus be able to replicate. One needs to grant it to the replication user that will be connecting from the slave. In 3.23, the privilege to read binary logs was covered under the File privilege.
- **Create Temporary Table:** One deficiency of the 3.23 privilege system was that the Create privilege was required in order to create a temporary table. This requirement caused headaches for DBAs who wanted their users to be able to create temporary tables as a workaround for the lack of subselect support, but did not want them to be able to create regular permanent tables. This privilege now allows users to create a temporary table, but does not allow them to go beyond that step and create a table that will exist after they disconnect.
- **Show Databases:** Without this privilege, the SHOW DATABASES command no longer works. In 3.23 and pre-4.0.2 releases of the 4.0 branch, it works for all users unless the server is running with the --safe-show-database or --skip-show-database option.

In addition, 4.0.2 provides two new privilege types reserved for future use: Execute for stored procedures, and Replication Client for fail-safe automatic recovery replication.

Granting Privileges

To grant a user a set of privileges, you need to issue the GRANT command at the prompt in the MySQL command-line client. Let's see how this command works by considering several examples of its use.

The following statement creates a power user superman who is only allowed to connect from localhost. The user can do anything on the server except grant their privileges to another user. Because having access to the server is one of the privileges, this limitation implies that this user cannot create other users. The user's password is mag1cpa3s:

```
GRANT ALL ON *.* TO 'superman'@'localhost' IDENTIFIED BY
'mag1cpa3s';
```

Now let's create a more powerful local user admin who can do anything, including grant their privileges to others. This user's password is secr3tkey:

```
GRANT ALL ON *.*  TO 'admin'@'localhost' IDENTIFIED BY 'secr3tkey'
WITH GRANT OPTION;
```

This time, let's create a user joe who is allowed to select any data as long as it is in the database sales. Because Joe should be connecting only from his workstation, which is called joe.mycompany.com, you will not allow him to connect from anywhere else:

```
GRANT SELECT ON sales.* TO 'joe'@'joe.mycompany.com' IDENTIFIED BY
'sloppyjoe';
```

Joe's boss, Larry, travels a lot and wants to be able to connect from anywhere. He needs Insert, Update, and Delete privileges in addition to Select:

```
GRANT SELECT,INSERT,UPDATE,DELETE ON sales.* TO 'larry'@'%'
IDENTIFIED BY  'cat /dev/mouse';
```

Suppose Joe has earned some favors through his hard work and is now entrusted with being able to insert, update, and delete in addition to his existing Select privilege:

```
GRANT INSERT,UPDATE,DELETE ON sales.* TO 'joe'@'joe.mycompany.com';
```

Suppose you have an employee database with several tables, one of which is named personnel. You want Jim and Jane to have access to everything in the database, Sally to be able to do anything with the personnel table, and Jenny to be able to read fields name and position, but only be able to update position. All users are connecting from their desktop workstations. Also assume that all the users already exist in the privilege system, so you do not need to assign them passwords. Execute the following set of statements:

```
GRANT ALL ON employee.* TO 'jim'@'jim.mycompany.com' ;
GRANT ALL ON employee.* TO 'jane'@'jane.mycompany.com';
GRANT ALL employee.personnel TO 'sally'@'sally.mycompany.com';
GRANT  SELECT (position,name), UPDATE (position) ON
employee.personnel TO 'jenny'@'jenny.mycompany.com';
```

Revoking Privileges

If you need to remove a set of privileges from a user, you should use the REVOKE command. For example, if decide Sally does not need to be able to update the position column, you would execute the following command:

```
REVOKE UPDATE (position) FROM 'sally'@'sally.mycompany.com';
```

Removing Users

To remove a user from the privilege system, delete their entry from the *user* table in the mysql database, and then execute FLUSH PRIVILEGES. For example:

```
DELETE FROM mysql.user WHERE host='jim.mycompany.com' AND
user='jim';
FLUSH PRIVILEGES;
```

System Security

The data in your database is no more secure than the database server itself on the system level. A user who can execute commands on the database server can circumvent a properly configured MySQL access privilege system if they have direct access to the database files. This section focuses on Unix systems. Similar principles apply to Windows, although the specifics of securing the system will be different. The main idea is that in either case, you must do what it takes to prevent a potentially malicious user from executing system commands or arbitrary code on the database server.

On a Unix system, it is important to make sure your directory and file permission are properly configured. The standard installation process usually takes care of this configuration. A user mysql will be created in the group mysql or root; all the database files will have read and write permissions for the user and the group, and no world permission. However, sometimes even experienced system administrators execute arbitrary chmod and chown commands in an effort to get things to work a certain way, and they often set permissions that are too lax.

It is highly recommended that you not only visually examine the permission and the ownership of the database files, but also log in to the shell accounts of your untrusted users and see whether you can read the files in the data directory with their privileges. You can determine the name of the data directory by running this command:

```
mysql -uroot -p -e "show variables like 'datadir'"
```

Check Setuid Binaries

If you have untrusted local users or if you are running a public service on the system (such as a Web server), it is also important to verify that your system is free from local exploitable setuid binaries. Attackers commonly crack systems by first discovering a way to execute arbitrary commands as non-privileged users, using a hole in a publicly accessible application (such as a Web form); then, using their ability to execute local commands, they find a setuid binary that has a security bug and take advantage of it to upgrade their privileges to root. After that, the attacker has free reign on the system until the administrator implements drastic measures, such as disconnecting the machine from the Internet and performing a solid clean-up—which frequently means reinstalling the operating system.

The problem with the setuid binary is that when executed, it will run as the user who owns it (usually root), and not as the user who executed the command. This allows non-root users to perform operations as root, which means

unrestricted privileges. If the setuid utility is bug-free, potential attackers will not be able to use it to upgrade their privileges beyond what they already have. Unfortunately, though, programmers make mistakes, and sometimes a setuid utility has a bug (frequently a buffer overrun) that allows attackers to hijack the execution and take full control of the system. To find all setuid binaries on your system, you can execute the following command:

```
find / -perm +4000
```

As a rule of thumb, you should begin by removing from your system all setuid binaries you do not need. The next step is to secure the ones you do need. For each binary, make sure the version you are running does not have any known security holes.

Run Only Necessary Services

Another important procedure in securing a system is making sure you are running only services that are necessary, and ensuring the services that are running are secure. The first step is to find out which services you are running. You can do so by executing the following command:

```
netstat -a | grep LISTEN | grep tcp
```

This command gives you a listing of TCP sockets in the LISTEN state, along with the name or value of the port for each socket. Each port corresponds to a service that can be accessed remotely. On some systems, netstat supports the -p option when run as root, which will also give you the name and the process ID of the process providing that service. Adding -p to netstat arguments on those systems will make it easier for you to track down the origin of each listening socket.

To account for UDP sockets, do the following:

```
netstat -a | grep udp
```

Once you have accounted for all the services, determine which ones your system has been configured to run by default during installation, and turn off any you do not need. Then apply the same validation procedures to the services you need that you used with the setuid binaries: Check vendor updates and make sure each service does not have any known security holes.

After the initial security hole check, the work is not finished by any means. As new holes are discovered on a daily basis, you should stay up-to-date on the latest security news. You can do so by regularly visiting security sites such as www.securityfocus.com and subscribing to a security mailing list such as Bugtraq. (More information about Bugtraq can be found at www.securityfocus.com/popups/forums/bugtraq/faq.shtml.)

Set Up a Firewall

Another line of defense for your server is a firewall. The firewall should allow only the minimum necessary connections to your network from the outside. It is recommended that you disable port 3306 on your firewall for the connections coming from the outside. Although MySQL is never released with a known security hole, it is not a good idea to tempt the crackers to find one by opening the port unnecessarily. If you need outside access to your MySQL server, limit it to a host-by-host or network-by-network basis, not only on the MySQL server itself, but also on the firewall.

Monitor the System Daily

Even after you have secured your system and protected it with a firewall, it is not sufficient to kick back and assume you are safe. You must take measures to monitor your system on a daily basis for attempted and possibly successful intrusions. You can do so with the help of a freely available utility with a rather humorous name: Snort. For more information about Snort, visit www.snort.org.

Database Application Security

One of the possible ways to subvert system security is to craft a surprise input that exposes a security bug and allows the attacker to take control (at least, to a certain degree). The first line of defense against such attacks is to always validate all user input to ensure it is strictly within its predefined domain. For example, if an input field is numeric by design, you must make sure it contains only digits; if it does not, you should give the user an error message. If the application accepts a person's name, and the input field is only 20 characters wide, then when you get the input, you must make sure the string does not exceed 20 characters; if it does, you should signal an error and refuse to process the input.

The second line of defense is related to the first and can be seen as its extension: Define the range of valid input as defensively as possible. For example, if you ask for the name of a city in the United States, you should only accept alphabetical input plus the - and ‘ characters. If the user wants to try `, ~, !, #, $, ^, &, or binary characters, chances are that their goal is not to provide the name of a city to the system. Even if the user's intent is to play around, or perhaps express frustration about the city they live in (or about your system, because they can't crack it), you should not accept the input and the user should get an error message.

The third line of defense is to check and disarm the potentially dangerous input before performing an operation that puts trust in the input's sanity. The two

potentially dangerous operations are passing the input to the shell in some way—for example, by executing system(), exec(), or Perl backticks (`)—and passing the input to a routine that executes a database query. To filter out the suspicious input from a shell argument, you can use a regular expression that eliminates backticks, semicolons (;), and pipe characters (|). When constructing a query, use the following functions to escape string values:

- **C/C+:** use mysql_real_escape_string() function
- **Perl**: use DBI->quote() method or prepared query with placeholders (? characters)
- **PHP:** use addslashes() function, or mysql_escape_string() starting in PHP 4.0.3
- **Java:** use PreparedStatement object and placeholders (? characters)

Server Configuration Security

One aspect of data security involves making sure the database server is configured with proper safety options and measures. A common-sense rule is to not open any more access privileges than are needed to do the job. Let's discuss some common ways to restrict unnecessary access.

To begin with, check whether the server has any wildcard entries in its privilege system, and remove all that are not absolutely necessary. You can do so with the following query:

```
SELECT * FROM mysql.user WHERE user LIKE '%\%%' OR host LIKE '%\%%';
```

You can then delete the unneeded entries that match the query on a case-by-case basis (or altogether) based on the value of the user and host columns, which form the primary key.

For each user you create, you should clearly define the minimum needed set of privileges; do not give any more than required. For example, if a user executes only INSERT queries, they should only have the Insert privilege. Avoid unnecessary use of GRANT ALL when granting privileges; this command grants all privileges, including the administrative FILE and PROCESS—which could potentially be dangerous if they end up in the wrong hands.

Make sure your MySQL root user has a password. In addition to the obvious issue of having unlimited access to the data, the MySQL root user is also dangerous because it can create files on the file system with arbitrary content. This ability could serve as a step in the process of compromising a system. For example, Apache.org (the site of the Apache server project) was compromised

with the help of a poorly configured MySQL server that did not have a password set for root.

You can run the server with special configuration options that restrict access in several ways and provide an additional line of defense:

- **--safe-user-create:** If enabled, means a user must have Insert privilege for the *mysql.user* database to be able to add new users.
- **--skip-name-resolve:** Disables reverse DNS lookups during authentication, which can protect you against DNS spoofing attacks. If you use this option, you must use numeric IP addresses instead of host names when creating users and granting privileges.
- **--skip-networking:** Prevents the server from listening on a TCP port. It is helpful when you have only local users who connect through Unix sockets, because you know in this case that any TCP/IP connection is unauthorized.
- **--bind-address:** Tells the server to bind only to the interface associated with the given IP address. If you have more than one network interface, but all your TCP/IP database connections come from only one interface, you already know that connections through other interfaces are unauthorized; therefore you can use this option to exclude the possibility.

Data Transfer Security

In 3.23, the MySQL data communication protocol does not encrypt data. With 3.23, the most reasonable option to secure a data transfer is to use a Virtual Private Network (VPN).

VPNs allow you to establish an encrypted tunnel between two trusted networks across an untrusted network as if the two networks were connected directly. This technique lets you overcome the limitation of the MySQL client-server protocol by setting up one interface on the client side and the other interface on the server side, and pointing all the clients on the client side to the local client interface attached to the server through the encrypted tunnel. Many VPN implementations are available, both free and commercial. A good place to begin learning about them is directory.google.com/Top/Computers/Security/Virtual_Private_Networks/.

Beginning in 4.0, SSL connections are supported, although at the time of this writing (4.0.4 is the current version), they are in what I would call an early alpha condition. Nevertheless, MySQL AB has a dedicated developer who is working full steam on making this feature stable. Documentation of SSL, unfortunately, is also in the early alpha condition, but it should soon improve. When it does, it will probably appear at www.mysql.com/doc/en/Secure_basics.html.

Dealing with Especially Sensitive Data

In theory, properly configuring your access privileges, removing the security holes from your operating system, and securing your network should be sufficient to protect MySQL data from unwanted access. But sometimes these steps are not good enough. In some cases, you may need an additional layer of security to protect your data if someone gets access to the data files.

If you are storing data only for the purpose of checking whether the stored pattern matches a given pattern (the most common use is password authentication), it's sufficient to use the MD5() or, in 4.0, SHA1() function, which performs one-way encryption. For two-way encryption (when you need to be able to decrypt what you have originally stored), the options are rather limited in 3.23. You can use the ENCODE()/DECODE() functions, but they do not provide strong encryption, and the encrypted data can be easily cracked. You can also store the data in a BLOB and perform the encryption and decryption operations on the client.

Version 4.0 provides some additional options. AES_ENCRYPT() and AES_DECRYPT() are available in the standard version and provide 128-bit encryption. When compiled with the --with-openssl option (which is the case for 4.0-max binaries), MySQL also supports the DES_ENCRYPT() and DES_DECRYPT() functions, but AES_ENCRYPT() is easier to use and also stronger than DES.

Whichever method you use, encryption does not happen transparently. You must do it in the client code. For example:

```
UPDATE customer  SET credit_card_no = AES_ENCRYPT('123412341234',
'secretkey') WHERE cust_id = 345;
```

Conclusion

We have discussed several aspects of securing MySQL server installation, access control, and data security. MySQL access control system allows you to restrict access on a per user/host combination basis to databases, tables, columns, and server functions. The security of your database server is greatly affected by the general system security. Database applications must be written with security in mind and thoroughly validate user input. MySQL data can be protected through encryption, which can be partially accomplished with built-in MySQL functionality. Certain aspects of data encryption require the use of external tools.

CHAPTER 6

Choosing the Client Language and Client-Server Network Architecture

When you're designing a MySQL application, you need to make a number of important decisions about the client side. In this chapter, we will discusses various issues pertaining to MySQL client setup.

Choosing a Client Language

MySQL has drivers or APIs available for a number of client languages, including C/C+, PHP, Perl, Java, Python, Ruby, and TCL. MySQL can also be accessed through ODBC. You must consider the following factors when choosing a client language:

- Performance requirements
- Development time constraints
- Code integration issues
- Portability requirements
- Developer skills and preferences

Performance

If client code performance is your top priority, there is no question about the language choice: Short of Assembler, which is not a viable option for most developers nowadays, the best thing you can do is write the client in C. Some scripting languages can give you decent performance; however, no scripting

language will ever outperform properly written C code, for this simple theoretical reason: Scripting engines interpret the code and then do the job, whereas the executable written in C/C++ just does the job.

Development Time

In many cases, C/C++ is not the optimal language in terms of development time. For example, every time you make a change in the code, you must recompile it before you can test it. The overhead may be only a few seconds, but it adds up not only in time, but also in mental stress on the developer. Scripting languages such as Perl and PHP do not have this disadvantage.

The syntax of C/C++ requires the developer to always declare and initialize variables before use, whereas most scripting languages are capable of implicit variable declaration and initialization if the programmer chooses to omit it. This C/C++ restriction tends to encourage the programmer to write more structured code, which is helpful in large projects containing modules that interact with each other on the code rather than the system level; however, it can be an unnecessary annoyance in a small project targeted to solve a specific problem, or in a large project consisting of a number of small modules that each solve a specific problem and interact with each other on the system rather than code level.

Additionally, languages such as Perl and PHP come with a number of modules that provide APIs for solving common application problems, such as generating certain HTML or XML components, connecting to another server, extracting information from text, processing information sent by a browser, or generating a PDF document. All those problems can be solved in C/C++, but the process of locating, installing, and learning how to use each relevant module is lengthy.

Development time in C/C++ is decent for number crunching, but it gets worse when you're processing text and constructing database queries. Perl and PHP, on the other hand, naturally lend themselves to text manipulation and make it convenient to construct queries. You can address C/C++'s awkwardness at text processing by creating a set of libraries. In a large project, the time required to develop the libraries will not be significant relative to the entire project, which will reduce the edge Perl and PHP have over C/C++.

Code Integration

The choice of the client language can be largely affected by the need to integrate with existing code. As a rule of thumb, the easiest way to integrate is to write in the same language as the original code.

Portability

If your client has to run on a number of platforms, you can consider several options to address the situation. Contrary to a common misconception, you can write portable code in C/C++—as MySQL, Apache, PHP, and other projects demonstrate—although doing so requires careful effort. However, it's much easier to do in Java, Perl, and PHP.

If your code must support multiple databases, you again have several options. One of them is ODBC. Another possibility is Perl, which can connect to a number of different databases using the DBI interface. You can also consider using C++ with QT library from TrollTech (www.troll.no; the company is based in Norway). QT version 3.0 has support for a portable interface to MySQL, PostgreSQL, Oracle, MS SQL, and Adaptive SQL. QT will also allow you to write portable GUI applications, connect to the network, perform file I/O, manipulate images, and do a number of other tasks for which people usually reach out to Java when they need to write a portable application.

Java is always a choice with MySQL, although MySQL AB only recently began to put some effort into maintaining the MySQL JDBC driver by hiring Mark Matthews, the original developer. (Prior to that, Mark supported it while working independently.) However, the development of MySQL/Connecter J (the driver's new name) is progressing at a fast pace. It now passes the Sun JDBC Compliance Test with the exception of stored procedures, which the MySQL engine does not support.

The Java interface does not go through the standard C API to access MySQL, whereas all other interfaces eventually do. This difference results in a considerable performance drawback. With Java, low-level operations such as constructing client-server communication packets, buffering them, and passing them to the operating system have to go to the Java code and go through the Java virtual machine. With other languages, a call is made to a compact C routine written by Monty (a programmer who sweats profusely over every line of his code to make sure it is top-notch optimal). So, if client performance is a concern and you have not yet begun your project, I recommend that you avoid Java. Fortunately for many Java projects, today's hardware is so cheap and powerful that the Java overhead does not cause many practical issues for most applications.

Developer Skills and Preferences

Developer skills and preferences should also play an important role when you're choosing the client language. If you select a language with which your development staff is already proficient, it will obviously take less time to

implement a new task than if the staff is forced to use a language they do not know and may not be particularly willing to learn (even if the new language is ultimately more suitable to the task). On the other hand, developers may not be familiar with a language; but if they have enough general programming experience, believe the new language is a better tool for the job, and are enthusiastic about learning it, chances are they will be productive and will get the job done in a timely way and with a high degree of quality.

Client Language Performance Comparison

Performance of a database application is determined by the efficiency of both the client and the server. Naturally, client performance is affected by the language in which it is written. Comparing clients written in different languages is always a challenge, due to the myriad of possible configurations. Even if you settle on one configuration, writing equivalent code in any two languages can be tricky.

Despite the difficulties involved, I followed the rationale that some data is better than none, and decided to write a simple benchmark for the languages discussed in this book: C, PHP, Perl, and Java. I ran it on one test system (Pentium III 500 with 256MB RAM running Linux); the results appear in this section. I've provided the source code, so you can try it on the systems you have access to and modify the benchmark code to accommodate your particular needs.

In each language, the benchmark is a command-line application that accepts three arguments from the user:

- **num_rows:** Number of rows in the table
- **num_cols:** Number of string columns in the table in addition to the primary key, which is always an integer
- **num_loops:** Number of select queries to run in the benchmark

The benchmark creates a table with an integer primary key and num_cols string columns of type CHAR(10) NOT NULL. The code populates the table with data using num_rows one-row inserts, timing the insert operation in the process. Then, it performs num_loops selects of one column (physically in the middle of the record) in a random row on the primary key, also timing the operation.

The benchmark code is available on the book's Web site. Each benchmark assumes you have a MySQL server running on the local machine, that you can log in it to it as root with no password, and that you have a database called test. If this is not the case, you can either temporarily modify the server configuration or modify the appropriate lines in the source code of the benchmarks you would like to run. The instructions to run the benchmark for each particular language are provided in the following subsections.

C

The source file is called benchmark.c. To compile it, use the standard C client compilation procedure described in Chapter 8. For example:

```
gcc -o benchmark -I/usr/include/mysql benchmark.c -L/usr/lib/mysql
-lmysqlclient -lz -static
```

To run the benchmark, use the following syntax:

```
./benchmark num_rows num_cols num_loops
```

For example, you can run it with a table of 1000 rows and 10 string columns performing 2000 select queries, as follows:

```
./benchmark 1000 10 2000
```

PHP

The source file is called benchmark.php. To run it, you need a PHP command-line interpreter, which you can produce on a Unix system by downloading the PHP source from www.php.net and running the following:

```
./configure --with-mysql
make
make install
```

On Windows, the binary distribution contains the command-line interpreter PHP.EXE.

To run the benchmark, use the following syntax:

```
php benchmark.php num_rows num_cols num_loops
```

For example, you can run it with a table of 1000 rows and 10 string columns performing 2000 select queries, as follows:

```
php benchmark.php 1000 10 2000
```

Perl

The source file is called benchmark.pl. You need to have Perl installed. The benchmark code uses the Time::HiRes module available from www.cpan.org.

To run the benchmark, use the following syntax:

```
perl benchmark.pl num_rows num_cols num_loops
```

For example, you can run it with a table of 1000 rows and 10 string columns performing 2000 select queries, as follows:

```
perl benchmark.pl 1000 10 2000
```

Java

The source file is called Benchmark.java. To be able to compile and run it, you need to have a JDK (Java Development Kit) installed. You also need the MySQL Connector/J JDBC driver.

To compile the source, use the following command:

```
javac Benchmark.java
```

To run it, use the following syntax:

```
java Benchmark num_rows num_cols num_loops
```

For example, you can run it with a table of 1000 rows and 10 string columns performing 2000 select queries, as follows:

```
Java benchmark 1000 10 2000
```

Test Results

The tests (run on a Pentium III 500 with 512KB CPU cache and 256MB RAM running Linux 2.4.19) yielded the results shown in Table 6.1 for 1000 rows with 5000 loop iterations.

Table 6.1 Client Language Benchmark Results

NUM_COLS/OPERATION	10	50	100	200
C/inserts per second	1367	984	693	445
C/selects per second	1748	1738	1708	1575
PHP/inserts per second	1283	838	607	393
PHP/selects per second	1493	1361	1336	1321
Perl/inserts per second	716	612	402	332
Perl/selects per second	801	759	782	827
Java/inserts per second	659	523	424	262
Java/selects per second	856	820	802	803

The tests were run against MyISAM tables on MySQL 3.23.52. using PHP version 4.0.4pl1, Perl version 5.005_03, and Sun JDK version 1.3.0 with MySQL Connector/J version 2.0.14. I ran the tests several times for each configuration, and the variations were quite significant due to caching. In each case, I ignored the off-the-curve slow results that occurred due to bad caching.

As you would expect, as the record becomes longer, the speed of the insert operation decreases for all languages. The speed of a one-column select tends to decrease somewhat as the record length increases, but the change is not significant; and, in some cases, a longer record yields produces faster performance, contrary to expectations, due to variations introduced by caching.

The C client is fastest. However, PHP is not far behind; it stays within a 10% margin. This result is easy to understand if you consider the PHP client architecture—all calls are basically direct wrappers around the C client library routines.

Perl and Java are nearly tied, with Perl perhaps a tiny bit faster, but both fall behind C and PHP. This result is also easy to understand. The Perl client has a thick DBI layer to penetrate before it gets down to the low-level C API calls. This is the price you pay for the portability of DBI. Java is unique in that it does not call the low-level C API routines. The Connector/J JDBC driver implements the MySQL client/server communication protocol. The speed loss is due to the JDBC overhead, as well as the Java-versus-C overhead in the protocol implementation.

Network Architecture

You have several options for your MySQL application's network architecture. Basically, you can break them into five scenarios:

- Client and server on the same host
- One server host and one remote client host
- One server host and many remote client hosts
- Data distributed across several server hosts and queried by one client host
- Data distributed across several server hosts and queried by several client hosts

The scenario you choose will naturally depend on your hardware budget, your security and performance needs, the complexity of your application, and probably a few other factors. Let's analyze these scenarios one by one.

Client and Server on the Same Host

This setup is a good solution in several cases: If you can afford only one machine, you are shipping an application that includes MySQL server and that needs to run out-of-the-box on the customer's computer, or you are transferring a lot of data between client and server and do not want to deal with the network overhead.

There are a couple of disadvantages to having the client and the server on the same machine. First, if your client is CPU- or memory-intensive, it will reduce the amount of resources available to the server—and that could affect performance. Second, the system is more failure-prone in some ways. Suppose your client has a bug, runs amok, and causes a system crash. When this happens, the server shuts itself down in an inconsistent state; now you have to deal with data-recovery issues while bringing the server back up.

On the other hand, having both client and server on the same machine gives you a fast, reliable, secure client-server data transfer; reduces hardware costs; and simplifies system maintenance. So, considering pros and cons, this option is worth consideration.

One Server Host and One Remote Client Host

This setup is most commonly found in Web applications that do not have enough traffic to justify having more than one database client/Web server host, but that have enough budget to afford the separation of database and Web server.

This approach can work in many instances, but you must address some issues to make it viable. First, you need to ensure that either the network between the client and the server is trusted (in other words, although the data travels unencrypted, unwanted individuals cannot tap into it and sniff the traffic) or the traffic is carried by a secure tunnel. You also need to ensure that the network is fast enough for your data transfer needs. To estimate your data transfer needs, you can, for example, execute SHOW STATUS LIKE 'Bytes_%' before and after a series of queries while there is no other activity on the server; then look at the difference in the values of Bytes_sent and Bytes_received.

It is important to remember that having the client and the server on separate machines will not always lead to performance improvement. If the data transfers are sufficiently large and the network is sufficiently slow, these factors could offset the performance gains that result from removing client-server resource contention.

One Server Host and Many Remote Client Hosts

This configuration is basically used in two common situations. In the first, you have a database and you want remote users connecting from various machines to be able to modify the data directly. For example, you might have a payroll or inventory database on one company server, and employees might be connecting to it using a desktop application from their workstations.

In the second common situation, a Web application is sufficiently CPU- or other resource-intensive to exhaust the capacities of the Web server long before it pushes the database server to the limit. Of course, in that case you should ideally try to optimize the Web application, but sometimes it is cheaper to buy more hardware. As atrocious as this solution may sound to a purist programmer, who is to say it is not valid if the application is only for internal use and this approach helps the business reach its goal at a reasonable cost?

As in the previous case, you need to have a fast and secure network.

Data Distributed Across Several Server Hosts and Queried by One Client Host

This setup is less common, although theoretically possible, which is why we will cover it briefly. You can use it if you have relatively few queries that are resource-intensive, or if you break down a query to run in parallel on several subsets of the data located on different servers.

Data Distributed Across Several Server Hosts and Queried by Several Client Hosts

In a typical situation for this setup, one master server receives updates, several slave servers replicate from the master, and a number of client machines query the data, sending the updates to the master and performing reads on the slaves. We will discuss this setup in more detail in Chapter 16, "Replication."

Estimating Load from Clients

When you're planning the client configuration, it's important to ask whether the server(s) have sufficient resources to accommodate the client load. The issues to consider are as follows:

- How many concurrent connections can come from each client host in the worst case?
- How many queries per second will each client host generate, and how long does each of those queries run if there is no other activity on the server?
- How much network traffic will come from each client host?

You can obtain the answers to those questions by analyzing the client application's configuration and source code, followed by running customized tests that simulate the real-life application. Before you begin to work on the application, it is recommended that you first benchmark a crude prototype you can code

quickly to get a feel for the limits you might have to deal with. It is very likely that during the client load evaluation, you will discover that you need to change certain parameters on the server, as well as optimize some queries. In that case, I refer you to Chapter 14, "Configuring the Server for Optimal Performance."

Client Programming Principles

In my years of working as a MySQL support engineer, I have frequently observed how an improperly coded client puts an undue burden on the server, resulting in reduced performance. By the time the problem is discovered, changing the client code significantly is not an option. However, had the issue been taken care of from the start, the same amount of development time and effort would have resulted in more optimal performance, more robust code, and easier code maintenance.

Based on this experience, I have put together a list of principles to be aware of when coding. These suggestions will help you create an optimal client:

- **Put wrapper functions around the database API calls:** Check for errors inside the wrapper, log failed queries, and have a special troubleshooting mode that logs all queries and times them, failed or successful. You may want to extend the wrapper to also log query result or just the number of rows when a special debugging option is enabled, as well as perform more advanced diagnostics and error handling.
- **Try to funnel complex queries through a common interface:** For example, if you have four different types of queries that differ only slightly, write a function that will generate all of them depending on the value of a certain argument. Doing so will allow you to change several queries at once if you discover you need a special optimization that applies to all of them.
- **Cache the query data:** Do not ask the server the same question in the same client session, but instead cache it in a client variable.
- **Use modularized design and reuse code:** This way, if you have to fix something, you will need to do so in only one place. You will also be able to replace components of your application with a different implementation without breaking it.
- **Cache HTML content in Web applications:** For example, if you have a news page and you update news items no more often than once a minute, you should make the page static and regenerate it every time you update your news items, instead of generating the HTML dynamically on every hit.

- **Store small, frequently used data sets inside the client code instead of the database:** For example, you should put a list of states into a client code module instead of a database.
- **As you write a query, learn the habit of thinking which keys it will use, how many records the database server will have to examine in order to provide the result, and how many records it will return:** Don't wait to consider these issues when the code has been running in production for a couple of months and your dataset has outgrown the limitations of the original design.
- **Avoid fads:** If you choose to use a certain feature of the language or a programming concept, make sure you have a clear idea about how it will make your life better. For example, you should not feel compelled to use the object-oriented model just because everyone around you says it is the way to go. If that is the only reason you are using the object-oriented model, then even if it could provide your application with some benefits, it is likely that without a clear vision you won't be able to take advantage of it—and it will only make things worse.
- **Do not be afraid to push the database to the limit of its functionality:** Suppose you wonder whether the database server has a certain feature, or you wish it had one. Instead of assuming it does not have the feature and coding around the limitation, read the manual; even if you cannot find the feature, try it anyway. You might be rewarded by discovering that it is there and works exactly the way you would expect. This step will save you development time and also give you better performance.

CHAPTER 7

MySQL Client in a Web Environment

Web applications are among the most common uses of MySQL. There are a number of reasons, including MySQL's high performance, rich functionality, low cost, and ease of maintenance. Web developers—from high-school wizards building Web sites for their friends to small businesses to such Web giants as Yahoo and Google—consider MySQL a viable choice for their needs. This chapter gives you a good head start toward building Web applications that connect to MySQL.

Choosing a Web Server

The Web server most commonly used in connection with MySQL is Apache (www.apache.org). Reasons include Apache's popularity, stability, flexibility, community support, and free license; and the availability of PHP. I can summarize all those reasons as follows: Apache is free, and it is popular enough to make it the first thing people try—and when they try it, it usually does the job well enough for their needs.

Apache is a great Web server choice for a MySQL application. During the entire time I've worked with MySQL support, I have seen only a few incidents involving Apache's stability, and I cannot recall one that had to do with Apache's performance.

Apache's top capacity on a 1GHz x86 machine is 2000–3000 static requests per second, and perhaps 500 PHP requests per second. (These figures assume files

of approximately 2–3K and reasonably efficient PHP code.) Some benchmarks show that on a quad system, Microsoft IIS running on Windows will outperform Apache running on Linux under high load on static HTML content. However, this information probably won't influence an experienced MySQL Web system architect, for the following reasons:

- Because much of the content is dynamic, static content performance is not as significant in the decision. As far as dynamic content is concerned, PHP has a reputation of outperforming ASP. Although it is difficult to come up with a fair benchmark for two different languages, many users claim a ten-fold increase in performance after converting their ASP applications to PHP.
- It is more cost-effective to scale Web performance by creating a Web server farm of uniprocessor or dual-processor systems than to buy quad servers.
- The lack of license fees becomes an important cost factor when you're building a Web server farm (a common practice on high-load Web sites).
- PHP can connect to MySQL using a native interface, whereas ASP must go through the ODBC layer.

Of course, you can choose from a number of other Web servers, such as Netscape Enterprise, Roxen, WebSphere, and iPlanet. MySQL applications will run on those Web servers. However, our focus will be on Apache because it is the most commonly chosen Web server.

Server-Application Integration Methods

A Web application can interface with a Web server in two primary ways: through CGI (Common Gateway Interface), or through an internal scripting language or precompiled modules (if the Web server supports those functionalities). In the CGI model, the Web application is a stand-alone executable that reads its input from the standard input stream and writes its output to the standard output stream according to a special protocol understood by the Web server. All the application has to do is follow a simple protocol. It can be written in any language and will work with most Web servers without significant porting effort.

Alternatively, a Web server can have the capability to execute a script internally without loading an external interpreter, or to load a precompiled module. In this case, the script or the module can usually run on a Web server that implements the standard.

Although CGI offers more flexibility, a performance penalty is associated with the fact that a Web server must create a separate process on every hit. The internal execution approach overcomes this problem and tends to produce better results—especially when the execution time is very small and the overhead of process creation is significant. The difference is less significant if the application executes a long time. In practice, the issue of process creation overhead on modern hardware becomes important for applications that handle more than 100 requests per second.

The most common languages for writing CGI Web applications interfacing with MySQL are Perl and C. Perl has the advantage of a faster development cycle, whereas C gives you the upper hand on performance.

If you decide to go the internal interpreter route, the two most common options are mod_perl and PHP. Which one is a better choice is a matter of debate. Generally, people who really like Perl prefer mod_perl, but those who do not know Perl that well or are not attached to it prefer PHP.

Apache provides another option for increasing performance: You can write a server module in C. The development process is more complicated and requires more skill, but you can get top performance because a module blends with the rest of the server and basically becomes its integral part.

If you have a simple targeted Web application that requires absolutely top performance, such as a banner server or request logger, you may want to consider writing your own Web server. Although it is a challenging task, it is doable; it can give you serious hardware savings and, in some cases, can make the difference between being able to handle the load and not being able to do so.

Web Optimization Techniques

The process of optimizing a Web application that connects to MySQL can be broken down into the following areas:

- Avoiding long queries
- Avoiding unnecessary queries
- Avoiding unnecessary dynamic execution
- Using persistent connections

Avoiding Long Queries

Long database queries are probably the top killer of Web performance. A Web developer may write a query, test it on a small dataset, and not notice anything

wrong because the query runs very fast. However, after the server has been in production for a month or two, the response time becomes terrible. The poorly written query is now being executed concurrently from several threads on a large dataset, causing heavy disk I/O and CPU utilization and bringing the server to its knees.

Even very experienced developers can put ugly queries in their code by accident. To avoid unoptimized queries, proactive measures are required. First, you must have a sense of how many rows the query you are writing will have to examine. When in doubt, it is a good idea to try the EXPLAIN command on the query in question.

Additionally, you should run the query with the log-long-format and log-slow-queries options on your development server, which in combination will log all the queries that are not using a key to the slow log. You should then run EXPLAIN on all the queries you find in the slow log as you develop your application and see what keys you need to add. In some cases, you may find it acceptable for a query to scan the entire table, but most of the time this is something you should avoid. It is a good idea to add a /* slow */ comment to your query so you can easily distinguish between the queries that are supposed to be slow and those that are slow by mistake in the slow query log.

Avoiding Unnecessary Queries

Although unnecessary queries are usually not as destructive to the server's sanity as slow queries, they still take the edge off your system's fighting capacity. Of course, if the unnecessary query is also slow, you are dealing with double trouble; but otherwise, the server can usually survive this kind of abuse.

Unnecessary queries usually result from errors and oversights in the application design process. Developers may forget they already have retrieved a certain piece of data and can use the value stored on the client. Or, perhaps they do not think the data will need to be reused in other parts of the code, and either do not store it after retrieving it, or do not retrieve it at all (when it could have been retrieved and stored easily without slowing the original query much).

Experienced developers make errors of this kind (not just novices), and the best time to catch the mistakes is during development. It's helpful to learn to visualize your paycheck being reduced by a certain amount for every query you execute in proportion to the number of rows the query has to examine—let's say one cent per row, and ten cents for initiating a query. (Creative managers might consider making this more than a game; have a set bonus amount that is reduced for inefficiency and also for each day past the deadline.)

In addition to the suggested mental exercise, it is also helpful to enable the *log* option on the development server, which will log all the queries into the usage log along with the connection ID. Then, periodically run the application and examine the log to evaluate the sanity of the query sequence. This process usually makes it easy to spot unnecessarily repeated queries.

Beginning in version 4.0.1, MySQL has a query cache that alleviates the burden of running the same query repeatedly. However, as of this writing the 4.0 branch is still in beta and may take a long time to fully stabilize.

Avoiding Unnecessary Dynamic Execution

In many cases, you can greatly optimize an application by caching the content. A classic example is a news site. News items arrive fairly often, but rarely more often than once a second. On the other hand, the Web site could easily receive several hits per second. A straightforward approach to a news Web site is to generate the news HTML by querying the database on every hit.

Sometimes this approach gives sufficient performance, especially if the news queries are optimized. However, the load may be so high that the developers need to look for an optimization. A more efficient approach with little extra code involves first noticing that a functional dependency exists between the data in the database and the content of the page. In other words, the content of the news page changes only when the database is modified. You can take advantage of this observation by making the news page static and regenerating it every time the database is updated.

The static-when-possible approach has two advantages. First, you avoid unnecessary database queries. Second, even without the database queries, serving a static page requires significantly fewer CPU resources and somewhat less memory. The actual performance improvement from switching to a static pages will, of course, depend on the application. As a semi-educated guess, I would expect on average a three- to tenfold speed increase.

Using Persistent Connections

Some languages, such as PHP, allow you the option of being persistently connected to MySQL, as opposed to connecting at the start of the request before the first database access and disconnecting once the request is over. The Apache Web server process that has just handled the request continues to run while waiting to handle another request; so, in some cases it makes sense to not disconnect from the database at all because the next request will come soon.

The process stays connected for its entire lifetime, which in Apache is controlled by the value of MaxRequestsPerChild.

This process prevents MySQL from having to deal with connection setup overhead, which could be significant on systems that have a problem creating threads under high load (for example, Linux 2.2). Even if thread creation is fast, maintaining a persistent connection still reduces network bandwidth and saves CPU cycles.

Unfortunately, the disadvantage of using persistent connections is the following common scenario: A large number of Web server children are spawned, each connecting to the database. Although as a whole the Web server is busy, each client performs a query only once in a while. This situation creates a large number of connections that are idle most of the time. Although an idle connection consumes memory and CPU resources, they are not significant. The real problem is that the database server reaches its connection limit. To keep this from happening, you must make sure the sum of the maximum number of concurrent clients for each Web server connecting to the database (on Apache, the setting is MaxClients) does not exceed the value of max_connections on the MySQL server. For example, if you have a Web server farm with three servers, each having a MaxClients value of 256, your MySQL server should have max_connections set to at least 3*256 = 768. This rule applies even if you are not using persistent connections, although not using persistent connections is less likely to expose the problem if you break that rule.

In practice, the rule is often not followed, but everything works fine for a while. Then somebody drops a key or writes a very slow query and adds it to a live application. The database server begins to get overloaded because each client stays connected for a long time. This situation drives the number of concurrent connections up past the limit, and eventually new connection attempts result in an error.

You can mitigate the idle connection issues by setting the value of wait_timeout sufficiently low—perhaps 15 seconds. Then, all the connections that have remained idle for more than 15 seconds will be terminated, and the unfortunate client will have to reconnect. This approach accomplishes the desired goal only when most clients perform queries at intervals closer than 15 seconds.

Stress-Testing a Web Application

It is difficult to overestimate the importance of running a stress test. Many problems that are discovered in production could have been detected and corrected with even minimal stress testing. Although no stress test can possibly simulate exactly what will happen when an application goes live, carefully chosen stress tests can help identify ugly application bottlenecks.

Using ApacheBench

A minimum approach to stress testing involves running the ApacheBench tool (ab), which comes packaged with the Apache distribution and is installed by default on most Linux systems on a fixed URL. You can run a total of 5000 requests coming from 10 concurrent clients as follows

```
ab -c 10 -n 5000 http://localhost/
```

The output will be similar to this:

```
This is ApacheBench, Version 1.3c <$Revision: 1.38 $> apache-1.3
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd,
http://www.zeustech.net/
Copyright (c) 1998-1999 The Apache Group, http://www.apache.org/

Server Software:        Apache/1.3.12
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        157 bytes

Concurrency Level:      10
Time taken for tests:   8.275 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      2395958 bytes
HTML transferred:       785314 bytes
Requests per second:    604.23
Transfer rate:          289.54 kb/s received

Connection Times (ms)
              min   avg   max
Connect:        0     3    16
Processing:     4    12   108
Total:          4    15   124
```

One common mistake is to forget the slash (/) at the end of the URL. You will get an error message if you do, at least in the version of ApacheBench I have tried. ApacheBench has a number of other options, which you can view by executing it with the -h option:

```
Usage: ab [options] [http://]hostname[:port]/path
Options are:
    -n requests     Number of requests to perform
    -c concurrency  Number of multiple requests to make
    -t timelimit    Seconds to max. wait for responses
    -p postfile     File containing data to POST
    -T content-type Content-type header for POSTing
    -v verbosity    How much troubleshooting info to print
    -w              Print out results in HTML tables
```

```
    -i              Use HEAD instead of GET
    -x attributes   String to insert as table attributes
    -y attributes   String to insert as tr attributes
    -z attributes   String to insert as td or th attributes
    -C attribute    Add cookie, eg. 'Apache=1234. (repeatable)
    -H attribute    Add Arbitrary header line, eg. 'Accept-Encoding:
zop'
                    Inserted after all normal header lines.
(repeatable)
    -A attribute    Add Basic WWW Authentication, the attributes
                    are a colon separated username and password.
    -p attribute    Add Basic Proxy Authentication, the attributes
                    are a colon separated username and password.
    -V              Print version number and exit
    -k              Use HTTP KeepAlive feature
    -h              Display usage information (this message)
```

Other Approaches

The disadvantage of relying on ApacheBench alone is that it cannot perform dynamic Web requests—it cannot, for example, iterate through a dictionary of possible values for form inputs in order to simulate a more random load. For some applications, this limitation might not make much difference, but for others the lack of variation in the request may seriously misrepresent the applications' capacities.

You can perform more thorough stress testing by doing the following:

1. Find a way to log requests that preserves form variables with their values. For example, your Web code can include a special debugging option that dumps all the variables, or you can do something as sophisticated as modifying the source of the Squid proxy to log POST request bodies. If you are not using POST forms in your code, you can simply use your Web access log.
2. Connect to your application from a browser and manually perform as many frequently run operations as possible as if you were a user.
3. Extract the requests from your logs. Replace the actual values of the form inputs with a special placeholder tag that indicates it is to be replaced with a dictionary value. For each form input, have a separate dictionary reference. For example, replace

```
first_name=Steve&last_name=Jones
```

with

```
first_name=${first_name}&last_name=${last_name}
```

4. Create dictionary files for each form input, populating them a wide range of possible values.
5. Write a program that parses your log files and plays back the requests randomly, replacing dictionary references with the appropriate values from the dictionary. You may want to use ApacheBench as a base; or, if you prefer Perl, you can use the LWP::UserAgent module.
6. Run the load, see what happens, and then fix the bugs in your application.

If a commercial solution exists that will let you do all this with less hassle, I am not aware of it. (Perhaps this is because I have always found it easier and quicker to solve a problem by downloading free tools from the Internet and making up for the missing functionality with my own code than spending time looking for a commercial solution that will do the job for me.)

CHAPTER 8

C/C++ Client Basics

The purpose of this chapter is to give you a head start on writing MySQL client applications in C/C++. In all truth, this chapter is about C. Although there exists a separate C++ API called MySQL++, it is not very stable, and is rather redundant because the C API works perfectly with the C++ code. If you plan to write a client in C++, I recommend that you use the direct C API interface described in this chapter.

Even if you do not plan to develop in C/C++, it is still important to understand the C API because it serves as a basis for all other interfaces (with the exception of Java). All other interfaces are simply wrappers around the C API calls.

This chapter provides instructions for setting up your system, a concise introduction to the API, and then a fully functional sample program. We conclude the chapter with a few useful tips.

Preparing Your System

All you basically need to do to be able to write MySQL applications in C/C++ as far as system administration is concerned is to install the client library file and the headers. If you are using a binary distribution on Unix, the library will be located in /usr/local/mysql/lib/mysql/ and the header files in /usr/local/mysql/include/. The same locations apply if you have installed from source using configure defaults (except unlike the binary distribution, the source distribution by default compiles and installs a shared library instead of a static library).

If you have installed from the RPM, the library is in /usr/lib/mysql, and the headers are in /usr/include/mysql. On the RPM distribution, you need the packages MySQL-devel and optionally MySQL-shared if you plan to link against a shared client library. The static library on Unix is called libmysqlclient.a, and the shared library is called libmysqlclient.so.

When compiling on Unix systems, you should give the compiler the -I argument, indicating the location of the header files. For example:

```
cc  -I/usr/local/mysql/include/mysql -o program.o -c program.c
```

When linking on Unix, you need to pass in libmysqlclient the location and name of the library as well as include the libraries that will resolve external symbols, as indicated in the following example:

```
cc -o program program.o -L/usr/local/mysql/lib/mysql -lmysqlclient
-lz
```

If you are linking against a shared client library that is not in a standard location, you should set LD_LIBRARY_PATH in your environment to point to that location before running the binary. For example, in the Bourne Shell, you type

```
LD_LIBRARY_PATH=$LD_LIRBARY_PATH:/usr/local/mysql/lib/mysql
export LD_LIBRARY_PATH
```

If you're a Windows user, first install the MySQL distribution. Make sure it has the files mysql.h and libmysql.lib. By default, they will install in C:\mysql\include and C:\mysql\lib, respectively. At the top of the source file, check to see that you have the following line:

```
#include <windows.h>
```

Add the path to mysql.h to your include path, and add the path to libmysql.lib to the library path. Then, add libmysql.lib to the list of libraries and build the project.

Structures and Functions of the API

The MySQL client API uses the following structures:

- **MYSQL:** When a connection is made to a MySQL database through C, the connection information is stored in a structure called MYSQL. This structure is used in the majority of the MySQL C functions.
- **MYSQL_RES:** For the query statements that return a result such as SELECT, the MYSQL_RES structure is used to hold and reference the result rows.

- **MYSQL_FIELD:** Within each row is a field, and the MYSQL_FIELD structure contains information about each field, such as its name, width, and type.
- **MYSQL_ROW_OFFSET:** This structure describes the position of a row in the result set.

In addition to the above structures, the API interface uses the type MYSQL_ROW when iterating through the result set, which is established with a typedef to be char** and also MYSQL_FIELD_OFFSET for field movements in the result set, which is established with typedef to be int.

The following list represents the C API functions available in the C client API library. We've presented the functions in alphabetical order. You can find a full description of each function, with examples, at www.mysql.com/documentation/mysql/bychapter/manual_Clients.html#C.

```
my_ulonglong mysql_affected_rows(MYSQL *mysql)
```

Returns a count of the rows affected by an INSERT, UPDATE, or DELETE query.

```
my_bool mysql_change_user(MYSQL *mysql, const char *username, const
char *password, const char *database)
```

Changes the current user to a different one.

```
const char *mysql_character_set_name(MYSQL *mysql)
```

Returns the character set in use by the current connection.

```
void mysql_close(MYSQL *mysql)
```

Closes the connection associated with the argument.

```
void mysql_data_seek(MYSQL_RES *result, my_ulonglong offset)
```

Relocates the internal current row pointer in the result set to the record number specified by the offset. Works only if the result argument was obtained through a call to mysql_store_result() (as opposed to mysql_use_result()).

```
void mysql_debug(const char *debug)
```

Turns on debugging on the client (if the client was compiled with debugging options enabled). You can find more information at www.mysql.com/doc/en/Debugging_client.html.

```
int mysql_dump_debug_info(MYSQL *mysql)
```

Causes the server to write debugging information to the MySQL error log.

```
unsigned int mysql_errno(MYSQL *mysql)
```

Returns the error code for the most recently executed MySQL function. If the previous function succeeded, it returns 0.

```
char *mysql_error(MYSQL *mysql)
```

Returns the error message for the most recently executed MySQL function. If the function succeeded, it returns an empty string.

```
MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)
```

Returns a pointer to MYSQL_FIELD structure for each of the fields in the result set. The first call returns the first field. Subsequent calls return the remaining fields in order. A value of NULL indicates there are no more fields.

```
MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result)
```

Returns an array of all fields in the result set.

```
MYSQL_FIELD *mysql_fetch_field_direct(MYSQL_RES *result, unsigned
int i)
```

Returns a MYSLQ_FIELD structure variable for the *i*th field in the result set.

```
unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

Returns an array of long integers representing the lengths of each field in the current row of the result set. A value of 0 is used for both NULL and empty columns.

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

Returns the current row in the result set. The current row pointer will be moved to the next row. Returns NULL when there are no more rows to return.

```
unsigned int mysql_field_count(MYSQL *mysql)
```

Returns the total number of fields in the result of the most recently executed SELECT query on the connection associated with the argument.

```
MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result,
MYSQL_FIELD_OFFSET offset)
```

Positions the field cursor in the field specified by the offset. Affects the return value of subsequent calls to mysql_fetch_field(). Returns the previous value of the field cursor.

```
MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)
```

Returns the current value of the field cursor.

```
void mysql_free_result(MYSQL_RES *result)
```

Frees all of the memory associated with the result set.

```
char *mysql_get_client_info(void)
```

Returns a string with the current library version it is using.

```
char *mysql_get_host_info(MYSQL *mysql)
```

Returns a string describing the connection between the client and the server.

```
unsigned int mysql_get_proto_info(MYSQL *mysql)
```

Returns the client-server protocol version used in the connection associated with the argument.

```
char *mysql_get_server_info(MYSQL *mysql)
```

Returns a string containing the server version of the connection described by the argument.

```
char *mysql_info(MYSQL *mysql)
```

Returns a string containing some information about the most recently processed query when it is an INSERT, LOAD DATA, ALTER TABLE, or UPDATE. For other queries, NULL is returned.

```
MYSQL *mysql_init(MYSQL *mysql)
```

If the argument is NULL, allocates memory for, initializes, and returns a pointer to the MYSQL structure. Otherwise, the function initializes just the structures at the address specified by the argument. It is always successful if the argument is not NULL; otherwise, the function may return NULL if memory could not be allocated.

```
my_ulonglong mysql_insert_id(MYSQL *mysql)
```

Returns the value of the AUTO_INCREMENT column generated by the previous query.

```
int mysql_kill(MYSQL *mysql, unsigned long pid)
```

Kills the connection thread specified by pid (*pid* is the value reported in the output of SHOW PROCESSLIST or in the result set of mysql_list_processes().

```
MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wildcard)
```

Returns a list of databases on the server to which we are connected through the connection descriptor specified by the first argument. The wildcard value may be NULL for all databases or may include the % and _ wildcards.

```
MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const
char *wildcard)
```

Returns a result set consisting of the fields in the specified table matching the wildcard pattern in the second argument. All fields in the table will be returned if the wildcard argument is NULL.

```
MYSQL_RES *mysql_list_processes(MYSQL *mysql)
```

Returns a result set with a list of all threads currently executing on the database server.

```
MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wildcard)
```

Returns a result set with the tables found in the current database matching the wildcard pattern in the second argument. All fields will be returned if the wildcard argument is NULL.

```
unsigned int mysql_num_fields(MYSQL_RES *result)
```

Returns the number of fields in the result set.

```
my_ulonglong mysql_num_rows(MYSQL_RES *result)
```

Returns the number of rows in the result set.

```
int mysql_options(MYSQL *mysql, enum mysql_option option, const char
*arg)
```

Sets the options for a connection to the database. You should specify the options before making the connection. More detailed documentation is available at www.mysql.com/doc/en/mysql_options.html.

```
int mysql_ping(MYSQL *mysql)
```

Pings a database server. If the server is alive, the function returns 0; otherwise, it returns a non-zero value.

```
int mysql_query(MYSQL *mysql, const char *query)
```

Executes the query specified by the second argument. On success, the function returns 0. This function will not work correctly if the query contains a \0 character that was not meant as a string terminator, in which case you should use mysql_real_query(). On the other hand, you should not have queries that have an unescaped \0 as part of the query to be sent to the server, and should escape such data with mysql_real_escape_string().

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char
*user, const char *passwd, const char *db, unsigned int port, const
char *unix_socket, unsigned int client_flag)
```

Attempts a connection to a database server using the supplied arguments. On success, the function returns the value of the first argument; on failure, it returns NULL.

```
unsigned long mysql_real_escape_string(MYSQL *mysql, char *to, const
char *from, unsigned long length)
```

Escapes the provided *from* string and places it in the *to* string parameter. The last argument (*length*) is the length of the *from* string. The *to* argument must point to an area that has at least 2**length*+1 bytes available.

```
int mysql_real_query(MYSQL *mysql, const char *query, unsigned long
length)
```

Executes the query specified by the query argument. The last argument (*length*) specifies the length of the query.

```
int mysql_reload(MYSQL *mysql)
```

Causes the server to reload the grant tables.

```
MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET
```

```
offset)
```

Moves the current row pointer in the result set to the row specified by the offset. The second argument is a pointer to a structure, not just a number. It must be the return value of mysql_row_tell() or mysql_row_seek(). This function can be used only if the result set was retrieved with mysql_store_result().

```
MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)
```

Returns the current row pointer in the result set.

```
int mysql_select_db(MYSQL *mysql, const char *database)
```

Attempts to change the current database to the value specified by the database argument. Returns 0 on success and a non-zero value on failure.

```
int mysql_shutdown(MYSQL *mysql)
```

Tells the server to shut itself down if the current user has sufficient privileges. Returns 0 on success and a non-zero value on failure.

```
char *mysql_stat(MYSQL *mysql)
```

Returns a string with server statistics such as uptime, number of queries since startup, and number of queries per second. Returns NULL on failure.

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

Should be called immediately after mysql_query() or mysql_real_query(). Retrieves the entire result set from the server and returns a pointer to the result set descriptor. On success, the function returns a non-zero value. On failure—which could mean that there was not enough memory to store the result—or that the last query on the given connection did not receive a result set (e.g., it was an UPDATE)--the function returns NULL. mysql_field_count() can be used to test whether the last query was supposed to return a result set. If it was, mysql_field_count() returns a non-zero value.

```
unsigned long mysql_thread_id(MYSQL *mysql)
```

Returns the thread ID of the connection specified by the argument. The return value can be used as an argument to mysql_kill().

```
MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

Allocates a result descriptor structure and initiates the retrieval of rows of the last query's result. Returns a non-zero value on success and NULL on error. The reasons for the NULL return value are the same as those for mysql_store_result(), except it is much less likely that NULL will be returned due to the lack of memory because only a few bytes are allocated. mysql_fetch_row() retrieves the actual rows. mysql_use_result() should be used when the result set is large enough to cause memory problems on the client.

API Overview

The MySQL client library provides a set of routines to communicate with a MySQL server. In order to use them, the source code must include mysql.h first. Most API routines accept a pointer to the MYSQL structure as the first argument. The MYSQL structure is a connection descriptor that stores internal information needed to communicate with the server. You should never modify members of this structure directly; use API calls instead.

Before you use the MYSQL structure, it has to be initialized with a call to mysql_init(). There are two ways to do this. If you have already allocated memory, you should pass a pointer to the MYSQL structure you have already allocated yourself. In this case, the function always succeeds and returns the value of the pointer you have passed to it as an argument. The other method is for the case when you want mysql_init() to allocate the memory for the MYSQL structure. In this case, you should pass it a NULL pointer. If the allocation is successful, a pointer to an initialized MYSQL structure is returned. Otherwise, a NULL pointer is returned. The caller must check the return value of mysql_init() in this case.

The MySQL application connects to the database using the mysql_real_connect() function, which has eight arguments:

- **MYSQL* mysql:** A pointer to the connection descriptor structure
- **const char* host:** The server host name to connect to
- **const char* user:** The database username
- **const char* password:** The plain-text password for the database user
- **const char* db:** The name of the database on the database server to select
- **unsigned int port:** The port to connect to—if 0, the default will be used
- **const char* sock:** The Unix socket to connect to. If NULL, the default is used
- **unsigned int flag:** Client flags—usually left at 0

mysql_close() is the cleanup function for mysql_real_connect() and mysql_init() at the same time. This function closes the connection and deallocates any resources that have been allocated with mysql_init() or other API functions that are associated with the MYSQL structure internals, although some functions (mysql_store_result() and mysql_use_result()) require a separate resource deallocation.

The MySQL client API uses the concept of having a selected active database. When a database is selected, references to a table without the mention of the database (e.g., tbl_stocks as opposed to db_finance.tbl_stocks) are assumed to refer the table with that name in the currently selected database. If you do not plan to change the active database throughout the lifetime of the connection, you should simply pass the name of the active database as the db argument to mysql_real_connect(). Otherwise, you can use mysql_select_db() to change databases in the middle of the connection.

On success, mysql_real_connect() returns the value of the first argument; on failure, it returns a NULL pointer. You must check the return value of mysql_real_connect() and take measures in case of an error. You can obtain the error code by using mysql_errno() and passing it the pointer to the MYSQL structure; you can obtain the text of the error message in a similar way by calling mysql_error().

To send a query, use mysql_query() or mysql_real_query(). The difference between them is that mysql_real_query() accepts the string length as the third argument. mysql_query() is actually implemented as a wrapper around mysql_real query(), so if you already know the length of the query, you should call mysql_real_query() instead.

mysql_query() and mysql_real_query() may fail for several reasons. On success, they return 0 and they return a non-zero value on failure. In a way similar to mysql_real_connect(), mysql_errno() returns the error code from the failed query, while mysql_error() returns the text of the error message.

Before sending a query, ensure that all unprintable and otherwise ugly characters in the strings are properly escaped because they can cause problems during query parsing and for programs that read query logs, in addition to introducing security risks. You can accomplish this with a call to mysql_real_escape_string(). Refer to the sample program in the next section to learn the details.

For queries that do not produce a result set (e.g., UPDATE), all you need to do to have them run from a C program is call mysql_query() or mysql_real_query() on the appropriate connection and then check for errors. If the query produces the result set, however, the above call is not sufficient. You must retrieve the result set by calling mysql_store_result() or mysql_use_result().

The difference between these two retrieval functions is in the retrieval method. mysql_store_result() will fetch all of the results from the server, while mysql_use_result() will initially retrieve a small chunk, with the remainder of the results being retrieved a chunk at a time as the client iterates through the

result set with mysql_fetch_row() calls. You should call mysql_store_result if it is acceptable or desirable to have the entire result set in memory on the client. Otherwise, you should call mysql_use_result().

Both result retrieval functions return a pointer to a structure of type MYSQL_RES, which can later be used in calls to mysql_fetch_row() to read one row at a time, or in calls to mysql_fetch_fields() to read the field information. It is important to remember that a call to mysql_close() does not free the memory allocated by the mysql_store_result() or mysql_use_result() function. You must free this memory separately by calling mysql_free_result().

A Sample Application

Some people learn things by first studying the theory and then trying to apply it. Others prefer to just watch someone do it, follow the example, and only then begin to wonder about how things actually work. The sample program following tries to meet the needs of both groups. Those who want to study how things work can read the comments, and those who want to see it done can scrutinize the code.

The sample program is a primitive interface to a one-table inventory database. It allows you to create databases, add new records, delete old ones, display and update selected records, and compute the totals for the item count and the price. To keep the program portable and easy to compile, we had to keep the interface to a low level of sophistication and decided to stick with the good old command line.

To compile the program on a Unix system, copy the file sample.c from the Web site and execute

```
cc -o sample -I/usr/include/mysql sample.c -L/usr/lib/mysql
-lmysqlclient -lz
```

You may need to modify the -I and -L arguments according to the specifics of your installation.

On Windows, the instructions will depend on your development environment. If you are using Visual Studio 6.0, create a new project defined as Win32 Console Application. Add sample.c to it, and at the top of that file include the following:

```
#include <windows.h>
```

Choose Settings from the Project menu, click the C/C++ tab, and then select the preprocessor category. Add the path to the MySQL include files (usually C:\MYSQL\INCLUDE) in the section called Additional Include Directories. In the same dialog box, click the Link Tab and then click Input. Add the Object/library modules and then add the text libmysql.lib in the list of libraries.

In the Additional Library Path box, add the full path to the library files of the MySQL installation (usually C:\MYSQL\LIB).

Once the program is compiled, you can try it out by first creating the database:

```
./sample i
```

After you have created the database, add a few records:

```
./sample a
```

To view all of the records in the database:

```
./sample s
```

You can determine other options by executing the command with no arguments and examining the output or by checking the source.

Now, let's get down to business and take a look at the program, shown in Listing 8.1.

```
/* First, include some standard C headers */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
/* Include header for MySQL client API functions */
#include <mysql.h>

/*
   Hard-coded defines for database connection parameters. OK for a
   demo, but in real-life applications those should be read from a
   configuration file.
*/

#define USER "root"
#define PASS ""
#define DB "test"
#define HOST "localhost"

/*
   This define is needed for the size of the query buffer when we
   construct a query. In a perfect program, we would use a dynamic
   string instead, but for our purposes it is sufficient to have a
   static query buffer.
*/

#define MAX_QUERY 16384
char query_buf[MAX_QUERY];
```

Listing 8.1 Source of sample.c. (continues)

```
/*
   Function prototypes. For the purpose of the demo, it is OK to have
   them in the C file. However, a more serious application consisting
   of several files would usually put such prototypes in its own .h
   file.
*/
void die(int flags, const char* fmt, ...);
void safe_connect();
MYSQL_RES* safe_queryf(int flags, const char* query, ...);
void prompt_gets(char* buf, int buf_size, const char* prompt);
int check_money(char* s);
int check_smallint(char* s);

void show_record(const char* rec_id);
void delete_record(const char* rec_id);
void update_record(const char* rec_id, const char* col);
void add_record();
void init_db();
void show_totals();

/*
   We use a global variable and allocate it statically on the heap.
   Alternatively, we could have used a pointer with dynamic
   allocation.
*/
MYSQL mysql;

/*
   Flags to control the behavior of the die() function.
   A large portion of a good program has to deal with errors.
*/
#define DIE_CLEAN 0x1 /* close the connection on fatal exit */
#define DIE_MYSQL_ERROR 0x2 /* print MySQL error message on fatal exit
*/

/* Flags for safe_queryf */
#define QUERYF_ABORT_ON_ERROR 0x1 /* fatal error if query failed */
#define QUERYF_USE_RESULT     0x2 /* call mysql_use_result() instead
of                                                    mysql_store_result()
*/

/* Convenient exit with some cleanup and error message.
   Although it's boring and tedious to write, once we are done with
   it, we can do clean and informative emergency exits with just one
   call when we notice something wrong. The name comes from a similar
   function in Perl and PHP.
 */
```

Listing 8.1 Source of sample.c. (continues)

```
void die(int flags, const char* fmt, ...)
{
  /* Here we do some work to be able to have printf() style arguments */
  va_list args;
  va_start(args,fmt);
  fprintf(stderr,"FATAL ERROR:");
  vfprintf(stderr,fmt,args);
  va_end(args);
  /* As you see, it is OK to call die() before a call to mysql_init()
     as long as we do it with flags set to 0. If one of the flags is
     set, we will be using the MYSQL structure, which needs to be
     initialized first.
  */
  if ((flags & DIE_MYSQL_ERROR))
    fprintf(stderr, " MySQL errno = %d: %s", mysql_errno(&mysql),
                 mysql_error(&mysql));
  /* Note the order - we first print the error, and only then call
     mysql_close(). If we did it in reverse order, we would have been
     referencing invalid memory. Results of any call on the MYSQL
     structure except for mysql_init() are unpredictable after a call
     to mysql_close().
  */
  if ((flags & DIE_CLEAN))
    mysql_close(&mysql);
  fprintf(stderr,"\n");
  exit(1);
}

/*
   Another convenience wrapper. Although in this particular
   application we will be calling this function only once, it is a
   good idea to have this wrapper. It makes the core code more
   readable by removing laborious
   error checks from the view of the reader, and it also makes it more
   extendable were the specifications to change. For example, we might
   need to reconnect, or we might redefine the concept of connection
   as connecting to one server, and if that fails, connect to another.
   In those events, this simple code abstraction will help us maintain
   the code more easily.
*/
void safe_connect()
{
  /* We do not check the return value of mysql_init() because it is a
     static initialization. If we were using a dynamically initialized
     pointer, we would have done:
     if (!(mysql=mysql_init(NULL)))
       die(0,"Out of memory in mysql_init()");
```

Listing 8.1 Source of sample.c. (continues)

```
  */
  mysql_init(&mysql);

  /* Note that on failure, we want the error message from MySQL, and
     we also request a cleanup, although in this case it only serves
     good manners - we have not established a connection, and since
     mysql_init() was called on a static memory area, there is nothing
     to deallocate.
  */
  if (!mysql_real_connect(&mysql,HOST,USER,PASS,DB,0,0,0))
    die(DIE_CLEAN|DIE_MYSQL_ERROR,"Failed connecting");
  /*
    Now the only way we can return is if the connection has actually
    been established.
  */
}

/*
  A pre-processor macro trick to avoid typing the same thing over and
  over again in the function below.
*/
#define CHECK_BUF if (q_p == q_end) \
 die(DIE_CLEAN,"Buffer overflow in safe_queryf()");

/*
  This is a convenience wrapper that has two functions - being able to
  send a query without having to check for errors, and to be able to
  have printf()-style interface with string escaping.
 */
MYSQL_RES* safe_queryf(int flags, const char* query, ...)
{
  const char* p;
  char *q_p, *q_end;
  char c;
  va_list args;
  va_start(args,query);
  /*
    First, we parse out the query resolving the printf()-style
    arguments. This part is a little boring, but it takes a lot of
    tedium out of future coding.
  */
  q_end = query_buf + MAX_QUERY - 1;
  for (p = query,q_p = query_buf; (c = *p); p++)
  {
    if (c == '%')
    {
      switch ((c = *++p))
```

Listing 8.1 Source of sample.c. (continues)

```
{
       /* string */
case 's':
{
       char *s = va_arg(args,char*);
       int len = strlen(s);
       /*
         Note 2*len - in the worst case,
         mysql_real_escape_string() will
         escape every character which will produce the new
         string that is double the size of the original.
       */
       if (q_p + 2*len >= q_end)
       {
         die(DIE_CLEAN,"Buffer overflow in safe_queryf()");
       }
       q_p += mysql_real_escape_string(&mysql, q_p, s, len);
       break;
}
/* money string that needs to be converted to the number of
   cents*/
case 'm':
{
       char* s = va_arg(args,char*);
       char* q_dest;
       int len = strlen(s);
       char c;
       /*
         Note that unlike the case above, we do not escape the
         string. We assume that it conforms to the money format
         (without $), and simply convert the value to cents
         before incorporating it into the query.
       */
       if (q_p + len >= q_end)
       {
         die(DIE_CLEAN,"Buffer overflow in safe_queryf()");
       }
       for (;(c=*s);s++)
       {
         if (isdigit(c))
           *q_p++ = c;
         else
         {
           s++;
           break;
         }
       }
```

Listing 8.1 Source of sample.c. (continues)

```
            *q_p = q_p[1] = '0';
            q_dest = q_p + 2;
            for (;(c=*s);s++)
              *q_p++ = c;
            q_p = q_dest;
            break;
      }
      /*
            As '%' now has become a special character, we need to
            provide a way for % to be included in the query. We do it
            by establishing the convention that %% in the format
            string translates into % in the actual query in a way
            similar to how it is done in printf().
      */
      case '%':
            *q_p++ = c;
            CHECK_BUF;
            break;
      default:
            die(DIE_CLEAN,"Invalid format character in safe_queryf()");
            break;
      }
    }
    else
    {
      *q_p++ = c;
      CHECK_BUF;
    }
  }
  *q_p = 0;
  va_end(args);
  /* Now the fun part. We actually get to send a query to the server */
  if (mysql_real_query(&mysql, query_buf, q_p - query_buf))
  {
    if ((flags & QUERYF_ABORT_ON_ERROR))
      die(DIE_CLEAN|DIE_MYSQL_ERROR,"failed executing '%s'", query_buf);
    else
      return 0;
  }
  /*
    As you may remember from the API overview, mysql_store_result() is
    for result sets that are sufficiently small to fit into memory,
    while mysql_use_result() should be called when memory is scarce or
    the result set is very big. In most cases, mysql_store_result() is
    preferred, since nowadays systems usually have plenty of RAM.
  */
  return (flags & QUERYF_USE_RESULT) ? mysql_use_result(&mysql) :
```

Listing 8.1 Source of sample.c. (continues)

```
    mysql_store_result(&mysql);
}

/*
  We do not need this define any more, so we undef it just in case
  somebody else might want it.
*/
#undef CHECK_BUF

/* This one should be pretty obvious - print usage info and exit */
void usage()
{
  printf("Usage: sample command [rec_id]\n\
Possible commands are: i (initialize database), a (add record),\n\
d (delete record), s (show record), q (update quantity), p (update price),\n\
q (update quantity), n (update name), t (show totals)\n");
  exit(1);
}

/*
  This function performs sanity checks on the money string. As this is
  just a demo, I have not comprehensively covered everything that can
  go wrong when a user tries to type in monetary amount. It is here
  more for the educational purpose to show the need for user input
  validation.
*/
int check_money(char* s)
{
  char c,c1;
  for (;(c=*s);s++)
  {
    if (!isdigit(c))
    {
      if (c != '.')
            return 0;
      else
      {
            s++;
            break;
      }
    }
  }
  if (!c)
    return 1;
  if (!isdigit(c=*s))
    return 0;
  if (!(c1=*++s) || (isdigit(c1) && !*++s))
```

Listing 8.1 Source of sample.c. (continues)

```
    return 1;
  return 0;
}

/*
  Again, just like the function above,
  this serves a didactic purpose to emphasize the importance of
  input validation.
*/
int check_smallint(char* s)
{
  int val = atoi(s);
  char c;
  if (strlen(s) > 5 || val >= 65536 || !val)
    return 0;
  for (;(c=*s);s++)
    if (!isdigit(c))
      return 0;
  return 1;
}

/*
  Convenience function to prompt the user to enter data, and
  then read it into a buffer. The caller must provide the buffer,
  and we promise we will not write more than buf_size characters of
  input here, truncating if needed.
*/
void prompt_gets(char* buf, int buf_size, const char* prompt)
{
  char *p;
  printf("%s: ",prompt);
  fflush(stdout);
  fgets(buf, buf_size, stdin);
  p = buf + strlen(buf);
  /* If fgets() appended '\n', replace it with '\0' */
  if (*--p == '\n')
    *p = 0;
}

/*
  As the name suggests, displays a record from the database, If rec_id
  is not NULL, show the match; otherwise, show all records.

  This function serves as an example of sequential record retrieval
  from a SELECT or any other query that returns a result set.
*/
void show_record(const char* rec_id)
```

Listing 8.1 Source of sample.c. (continues)

```
{
  MYSQL_RES* res; /* stores the result set descriptor */
  MYSQL_ROW row;  /* pointer to a row returned by mysql_fetch_row() */
  /*
    Note that when we retrieve all records, we pass QUERYF_USE_RESULT
    flag to safe_queryf(), which is our own convenience wrapper around
    mysql_real_query(). We do this because we do not know how many
    records we are reading, and it might be more than we have RAM for.

    Also note how our work invested into implementing safe_queryf() is
    now beginning to pay off. Think about what it would take to
    construct the query from user input otherwise.
   */
  if (rec_id)
    res = safe_queryf(QUERYF_ABORT_ON_ERROR,
                      "SELECT id,name,quantity,price/100 FROM inventory \
WHERE id = '%s'", rec_id);
  else
    res = safe_queryf(QUERYF_ABORT_ON_ERROR|QUERYF_USE_RESULT,
                      "SELECT id,name,quantity,price/100 FROM
inventory");
  if (!res)
    die(DIE_CLEAN|DIE_MYSQL_ERROR, "Error retrieving result");
  printf("Id\tName\tQuantity\tPrice\n");
  /*
    A call to mysql_fetch_row() on the result set returns a row of
    type MYSQL_RES, which is actually just an alias for char**. So row
    is just an array of pointers to character arrays containing the
    values of the corresponding columns. After we have read the last
    row, a call to mysql_fetch_row() will return 0.

    If we wanted to iterate through the result set one more time, this
    would be possible if the result set was obtained with
    mysql_store_result() (as opposed to mysql_use_result()). To rewind
    or to place the internal iterator on a particular row we would use
    mysql_data_seek().

    Note that we do not check row[n] for NULL. This is acceptable
    because all of the columns have been declared as NOT NULL. If the
    column value is NULL for a particular row, the corresponding value
    of row[n] will be NULL, not "NULL" or "". So if there is a
    possibility of getting NULL in a column from a query, row[n] would
    have to be checked for NULL.
   */
  while ((row = mysql_fetch_row(res)))
  {
    printf("%s\t%s\t%s\t%s\n", row[0], row[1], row[2], row[3]);
```

Listing 8.1 Source of sample.c. (continues)

```
  }
  /*
    Note that mysql_close() will not free res. We have to do it
    separately with mysql_free_result().
  */
  mysql_free_result(res);
}

/* Here the name of the function tells us exactly what it does */
void delete_record(const char* rec_id)
{
  /*
    Again, having safe_queryf() pays off. The job of deleting a record
    is reduced to one statement.
  */
  safe_queryf(QUERYF_ABORT_ON_ERROR, "DELETE FROM inventory WHERE id = '%s'",
                  rec_id);
  printf("Record deleted\n");
}

/* Again, function name is self-explanatory. */
void update_record(const char* rec_id, const char* col)
{
  char buf[30];
  char prompt[64];
  sprintf(prompt, "New value of %s", col);
  prompt_gets(buf,sizeof(buf),prompt);
  safe_queryf(QUERYF_ABORT_ON_ERROR,
                  "UPDATE inventory SET %s = '%s' WHERE id = '%s'",
                  col, buf, rec_id);
  printf("Record updated\n");
}

/* Again, self-explanatory name. */
void add_record()
{
  char name[30], quantity[7], price[10];
  prompt_gets(name,sizeof(name),"Name");
  /* No validation for the name in this case */
  prompt_gets(quantity,sizeof(quantity),"Quantity");
  /*
    Quantity has to be a positive number within the range of
    smallint unsigned, which is from 0 to 65535.
  */
  if (!check_smallint(quantity))
    die(DIE_CLEAN,"Invalid quantity");
```

Listing 8.1 Source of sample.c. (continues)

```
  prompt_gets(price,sizeof(price),"Price");
  /* We also validate the price */
  if (!check_money(price))
    die(DIE_CLEAN,"Invalid price");

  /* Now we are ready for the actual insert operation */
  safe_queryf(QUERYF_ABORT_ON_ERROR,"INSERT INTO inventory \
(name,quantity,price) VALUES ('%s',%s,%m)", name, quantity, price);
}

/* Initialize the database, deleting the old data if present */
void init_db()
{
  /* first, clean up */
  safe_queryf(QUERYF_ABORT_ON_ERROR, "DROP TABLE IF EXISTS
 inventory");
  /* now create the table. */
  safe_queryf(QUERYF_ABORT_ON_ERROR, "CREATE TABLE inventory (\
id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,\
name VARCHAR(30) NOT NULL, \
price MEDIUMINT UNSIGNED NOT NULL, \
quantity SMALLINT UNSIGNED NOT NULL )");
}

/*Self-explanatory name. */
void show_totals()
{
  /*
    Any time we are retrieving a result set from a query, we need the
    two variables below.
  */
  MYSQL_RES* res;
  MYSQL_ROW row;
  if (!(res = safe_queryf(QUERYF_ABORT_ON_ERROR,"SELECT SUM(quantity),
\
SUM(quantity*price)/100 FROM inventory")))
    die(DIE_CLEAN|DIE_MYSQL_ERROR,"Error retrieving result");
  if (!(row = mysql_fetch_row(res)))
    die(DIE_CLEAN,"SUM query returned no rows - something is wrong
with \
the server");
  if (!row[0] || !row[1])
    printf("Database contains no records\n");
  else
    printf("The inventory contains %s items worth $%s\n", row[0],
row[1]);
}
```

Listing 8.1 Source of sample.c. (continues)

```
/* Another convenience macro */
#define CHECK_ARG2 if (argc != 3) usage();

int main(int argc, char** argv)
{
  if (argc < 2)
    usage();
  /*
    safe_connect() makes our life easier - we do not have to check for
    error, as safe_connect() will.
  */
  safe_connect();
  switch (*(argv[1]))
  {
  case 'i':
    init_db();
    break;
  case 'a':
    add_record();
    break;
  case 'd':
    CHECK_ARG2;
    delete_record(argv[2]);
    break;
  case 's':
  {
    char* rec_id = 0;
    if (argc == 3)
      rec_id = argv[2];
    show_record(rec_id);
    break;
  }
  case 'q':
    CHECK_ARG2;
    update_record(argv[2],"quantity");
    break;
  case 'n':
    CHECK_ARG2;
    update_record(argv[2],"name");
    break;
  case 'p':
    CHECK_ARG2;
    update_record(argv[2],"price");
    break;
  case 't':
    show_totals();
```

Listing 8.1 Source of sample.c. (continues)

```
    break;
  default:
    /*
      since we will be exiting through usage(), we want to first close
      the connection.
    */
    mysql_close(&mysql);
    usage();
  }
  /* cleanup */
  mysql_close(&mysql);
}

#undef CHECK_ARG2
```

Listing 8.1 Source of sample.c. (continued)

My goal in writing the sample program was to write the code as clearly as possible and document it thoroughly, so that you would not need to flip the pages of this book back-and-forth between code sample and explanation. However, some highlights are worth mentioning:

- We use wrappers around MySQL API calls that provide error handling and other conveniences (safe_connect(), safe_queryf()).
- We validate user input with our own validation routines (check_smallint() and check_money()).
- Strings that could have contained potentially problematic characters (e.g., a single quote) are "auto-magically" escaped in safe_queryf().
- safe_queryf()is also capable of converting money strings into cents on the fly to support our storage optimization of storing monetary value in an integer instead of decimal or float.
- We have a "magic" function die() that we call every time we hit a fatal error, which prints the message and optionally performs the cleanup. This allows us to have a more convenient error handling. Having a convenient way to handle errors is very important. Psychologically, if error handling is burdensome, even the most thorough programmer will be inclined to say at some point that a certain error does not need to be checked because it will never happen. This phenomenon eventually results in not handling errors when you should and, consequently, the program suffers a reduction in stability.

Tips and Tricks

In this section we list some small issues and techniques that are easy to miss when you are learning MySQL C API, but that will help you save a lot of time if you know them.

- Do not forget to call mysql_free_result() for each invocation of mysql_store_result() or mysql_use_result().
- Remember to check the value of the NULL column if possible. Remember that although the column itself might be declared as NOT NULL, some functions on it may still return NULL.
- When using mysql_real_connect(), if you want to connect via TCP/IP to a local server, pass "127.0.0.1" for host, not "localhost". If you pass "localhost" you will be connecting through a Unix socket, even if your port argument is not 0.
- Server administration queries, such as SHOW STATUS, SHOW PROCESSLIST, and CHECK TABLE, from the developer's point of view are not different from a regular SELECT. To execute them, just call mysql_real_query() or mysql_query() followed by mysql_store_result(), mysql_fetch_row(), and mysql_free_result(), just as you would if you wanted to read some data from a table.
- If you called mysql_store_result(), you do not have to iterate through all the rows before freeing it, but in the case of mysql_use_result(), you do. It is possible to call mysql_store_result() and issue queries retrieving their results before you are done reading all the rows. It is also possible to keep the result set around for the entire lifetime of the program, even after you have disconnected from the database. This is not possible with mysql_use_result().
- If you think you may have lost the connection, call mysql_ping(). If the connection has been lost, an attempt to reestablish it will be made.
- When calling mysql_real_escape_string(), make sure the buffer is at least double the size of the string to provide for the worst case when every character has to be escaped.
- If you want compression in the client-server protocol, pass the CLIENT_COMPRESS flag in the last argument of mysql_real_connect().

CHAPTER 9

PHP Client Basics

PHP is one of the most popular Web development languages. Many programmers appreciate it for its ease of learning, convenience of use, rich functionality, excellent performance–and free license on top of everything.

PHP was created by Rasmus Lerdorf in 1995, initially just to solve a particular problem. Since then, however, it has quickly developed into a mature language noted for its richness of features and intuitive syntax. An experienced C or Perl programmer with no prior PHP exposure should be productive in PHP after looking at a couple of code examples. PHP is supported by all major Web servers; its install base is estimated at about 10 million users at the time of this writing and keeps growing daily. You can find more information about PHP at the PHP Web site, www.php.net. (Note the *.net* in the URL; the www.php.*com* site is not affiliated with the PHP language project.)

MySQL and PHP developers have a lasting relationship and work in close cooperation. As a result, PHP has excellent support for MySQL connectivity in addition to all the other advantages of PHP. This makes PHP an excellent language for developing a Web site with a MySQL backend.

In this chapter, we provide instructions for setting up your system to be able to use MySQL with PHP, briefly discuss the MySQL API, and illustrate our discussion with a sample application. In addition, we include a few tips you'll find helpful.

Preparing Your System

There are three ways to run a PHP application: as a Web server-side script, as a CGI, or as a stand-alone non-Web executable interpreted by the PHP command-line interpreter. The first two are suitable for the Web environment, while the third can be used for various tasks, such as data load cron jobs and other command-line utilities.

If you are using Linux, chances are you already have Apache installed with PHP support. To see if this is the case, type the following:

```
lynx -head -dump http://localhost/ | grep PHP
```

If the output contains something like PHP/4.2.1, the PHP module is installed and you are set. Otherwise, if the output is empty, PHP is not installed, which means you need to do some work. If you prefer to stick with the packages from your distribution, you should make sure that both the Apache and PHP packages have been installed.

Another simple method to test whether PHP has already been installed is to place a file called test.php somewhere under the HTTP documented root on your Web server with the following content:

```
<?
 phpinfo();
?>
```

Then visit the URL that will correspond with the newly created file. If PHP has been installed, you will see a page containing the information about your PHP configuration.

If you are running a different Unix operating system, or you simply prefer to install software yourself, you can download PHP from www.php.net and compile PHP. The distribution contains good installation instructions, so we refer you to the INSTALL file rather than repeat its contents in the book. Just remember to add --with-mysql to the configure arguments if you want MySQL support.

If you plan to use PHP with a Web server other than Apache and PHP does not have an option to build a module for that server, or if you want to build a command-line PHP interpreter, your configure command should be

```
./configure --with-mysql
```

If at all possible, you should use the configuration in which the PHP interpreter resides inside the server, because this will give you better performance. If you are using a non-Apache Web server, you should check the vendor documentation, as well as www.php.net/manual/en/install.otherhttpd.php.

On Windows, PHP will not come preinstalled; you need to download it and install it. Detailed instructions for different configurations are available at www.php.net/manual/en/install.windows.php. Follow the instructions that apply to your particular configuration.

API Functions

In this section, we provide a brief listing and description of the PHP MySQL module functions in alphabetical order. The list corresponds to what is available in PHP version 4.2.1, but excludes the functions that are listed as depreciated. If you have an older PHP version, some functions or some of their features may not be available. For those functions, we provide backward-compatibility information. More detailed documentation is available at www.php.net/manual/en/ref.mysql.php.

The PHP online manual also provides a convenient shortcut for function entry lookups. Function entry URLs follow the pattern of www.php.net/manual/en/function.*func-name*.php. For example, if you want to look up mysql_connect(), the URL would be www.php.net/manual/en/function.mysql-connect.php. Do not forget to replace any underscore (_) characters in the function name with hyphens (-).

Most functions make use of the optional connection link argument. All of them, when the link is not specified, will use the default link, which is the last link opened by mysql_connect() or mysql_pconnect().

```
int mysql_affected_rows([resource link])
```

Returns the total number of rows changed in the last operation on the connection specified by the link identifier.

```
bool mysql_close([resource link]
```

Attempts to close a MySQL connection specified by the link identifier. Returns TRUE on success and FALSE on error.

```
resource mysql_connect()[string server [, string username [, string
password [, bool new_link [, int client_flags]]]]])
```

Attempts to open a new connection to the specified MySQL server using the specified authentication credentials. Returns a connection link reference, which can be used as an argument to other API functions.

```
bool mysql_data_seek(resource result_identifier, int row_number)
```

Moves the current row pointer to the row specified by row_number in the result set specified by result_identifier. Returns TRUE on success and FALSE on error.

```
string mysql_db_name(resource result, int row)
```

Returns the name of the database at the position row in the result set returned by mysql_list_dbs().

```
int mysql_errno([resource link])
```

Returns the error code of the last MySQL operation on the connection specified by the link argument.

```
string mysql_error([resource link])
```

Returns the error message string of the last MySQL operation on the connection specified by the link argument.

```
string mysql_escape_string(string column_value)
```

Returns a string with properly escaped potentially problematic characters. Available starting in PHP version 4.0.3.

```
array mysql_fetch_array(resource result_set[,int result_type])
```

Returns an array corresponding to the row fetched from the result set, or FALSE if there are no more rows left to read. The type of the array is determined by the second argument, which can be one of the following: MYSQL_NUM (array with only numeric indexes), MYSQL_ASSOC (associative array indexed by field name), or MYSQL_BOTH (array with both numeric and field name indexes). The second argument was added in PHP version 3.0.7, and defaults to MYSQL_BOTH.

```
array mysql_fetch_assoc(resource result_set)
```

Returns an associative array corresponding to the currently fetched row. Available since PHP version 4.0.3.

```
object mysql_fetch_field(resource result_set [,int field_offset])
```

Returns an object describing the properties of the field in the specified result set at the field_offset position. The object consists of the following members:

- **name:** Column name
- **table:** Name of the table the column belongs to
- **max_length:** Maximum length of the column
- **not_null:** 1 if the column cannot be NULL
- **primary_key:** 1 if the column is a primary key
- **unique_key:** 1 if the column is a unique key
- **multiple_key:** 1 if the column is a non-unique key
- **numeric:** 1 if the column is numeric
- **blob:** 1 if the column is a BLOB
- **type:** The column type

- **unsigned:** 1 if the column is unsigned
- **zerofill:** 1 if the column is zero-filled

```
array mysql_fetch_lengths(resource result_set)
```

Returns an array containing the lengths of each column in the specified result set.

```
object mysql_fetch_object(resource result_set)
```

Returns an object corresponding to the fetched row in the specified result set. Members of the object correspond to the names of the fields.

```
array mysql_fetch_row(resource result_set)
```

Returns an enumerated array for the fetched row in the specified result set.

```
string mysql_field_flags(resource, int field_offset)
```

Returns the flags associated with the specified field in the specified result set. The flags are reported as a single word per flag separated by a single space and can be split into an array using explode(). The possible flag values are

- not_null
- primary_key
- unique_key
- multiple_key
- blob
- unsigned
- zerofill
- binary
- enum
- auto_increment
- timestamp

```
int mysql_field_len(resource result_set, int field_offset)
```

Returns the length of the specified field in the specified result set.

```
string mysql_field_name(resource result_set, int field_offset)
```

Returns the name of the specified field in the specified result set.

```
int mysql_field_seek(resource result_set, int field_offset)
```

Positions the current field cursor to the given field offset in the given result set. Affects the return value of mysql_fetch_field() invoked without the second argument. Returns the value of field_offset.

```
string mysql_field_table(resource result_set, int field_offset)
```

Returns the name of the table containing the specified field in the specified result set.

```
string mysql_field_type(resource result_set, int field_offset)
```

Returns the type of the specified field in the specified result set. The type value corresponds to the definition of the table in the CREATE TABLE statement, for example, int, real, char, or blob.

```
bool mysql_free_result(resource result_set)
```

Frees the memory associated with the specified result set. On success, returns TRUE; returns FALSE on error.

```
string mysql_get_client_info()
```

Returns a string describing the client library version. Available since PHP version 4.0.5.

```
string mysql_get_host_info([resource link])
```

Returns a string describing the type of server connection (e.g., whether done through TCP/IP or through a local socket). Available since PHP version 4.0.5.

```
int mysql_get_proto_info([ resource link])
```

Returns the protocol version number for the specified connection. Available since PHP version 4.0.5.

```
string mysql_get_server_info([resource link])
```

Returns a string containing version information for the server we are connected to through the specified link. Available since PHP version 4.0.5.

```
int mysql_insert_id([resource link])
```

Returns the AUTO_INCREMENT value generated by or associated with the last query on the connection specified by the argument.

```
resource mysql_list_dbs([resource link])
```

Returns a result set containing a list of databases available on the MySQL server we are connected to through the link specified by the argument.

```
resource mysql_list_fields(string database, string table [, resource
link])
```

Returns a result set containing a list of the fields available in the table residing in the specified database on the server to which we are connected through the specified link.

```
resource mysql_list_tables(string database [, resource link])
```

Returns a result set containing a list of tables in the specified database on the MySQL server we are connected to through the link specified by the argument.

```
int mysql_num_fields(resource result_set)
```

Returns the number of fields in the specified result set.

```
int mysql_num_rows(resource result_set)
```

Returns the number of rows in the specified result set.

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

Attempts to open a new connection to the specified MySQL server using the specified credentials if the matching connection is not found in the persistent connection pool. If it is found, the matching connection will be used instead of opening the new one. Returns a connection link reference, which can be used as an argument to other API functions.

```
resource mysql_query(string query [, resource link,[ int
result_mode]])
```

Sends the specified query on the specified connection link to the server, and retrieves the result in the manner specified by result_mode, which can be either MYSQL_STORE_RESULT (read all rows at once and store them in memory) or MYSQL_USE_RESULT (read one row at a time once per fetch operation). The default is MYSQL_STORE_RESULT.

```
mixed mysql_result(resource result_set, int row [,mixed field])
```

Returns the value of the specified field in the specified row of the result set. The row is specified by number (offset). The field can be specified by number (offset) or by name.

```
bool mysql_select_db(string database [,resource link])
```

Changes the currently active database on the connection specified by the link argument to the one specified by the database argument. Returns TRUE on success and FALSE otherwise.

```
string mysql_tablename(resource result_set, int i)
```

Returns the *i*th table in the result set returned by mysql_list_tables().

```
resource mysql_unbuffered_query(string query [,resource link[,int
result_mode]])
```

Sends the specified query on the specified connection link to the server, and retrieves the result in the manner specified by result_mode, which can be either MYSQL_STORE_RESULT (read all rows at once and store them in memory) or MYSQL_USE_RESULT (read one row at a time once per fetch operation). The default is MYSQL_USE_RESULT. Available since PHP version 4.0.6.

API Overview

The MySQL API in PHP is conceptually very similar to the C API. This should not come as a surprise, given the fact that it is essentially a wrapper around the C API calls. Nevertheless, this is a very handy wrapper. PHP, being a scripting language with loose typing (no variable declaration required prior to its use), introduces some shortcuts. Additionally, in the true spirit of PHP, the API has a lot of convenience features and is quite successful at guessing programmers' intentions when they fail to state them explicitly. When the implicit behavior happens to be out of harmony with the programmer's intentions, there is always an option, of course, to override it and get the code to do what is desired.

Because of the great degree of similarity to the C API, and also to encourage you to understand what happens under the hood of PHP in terms of C API calls, we frequently compare the PHP API with its C equivalent in this overview.

To connect to MySQL Server, you call either mysql_connect() or mysql_pconnect(). While the former establishes a connection that will be terminated when the work is finished or when mysql_close() is called, the latter will establish a persistent connection.

PHP/Apache persistent database connections work in the following manner. Apache at any given time runs a number of child processes. Each PHP request is handled by a child process, and each child process has a pool of persistent connections. When the persistent connection function is called (mysql_pconnect() in the case of MySQL), first the lookup is done in the pool for a matching connection that may have already been established. If such connection is found, it will be used and no new connection will be opened. Otherwise, a new connection is made and entered into the pool. All of the pool connections will be maintained for the lifetime of the child.

The advantage of persistent connections is that the connection overhead on most requests will be eliminated. The disadvantage is that connections are often being maintained idly, unnecessarily tying up the database server resources. Because the connection overhead with MySQL is relatively small on most systems, the performance overhead of using nonpersistent connections is not significant. On the other hand, inexperienced users frequently run into problems when the limits on Apache and on MySQL have not been set properly.

The functions mysql_connect() and mysql_pconnect() return a connection handle on success or FALSE on error. Unlike with other MySQL functions, if an error occurs, the error message has to be read from $php_errormsg, not mysql_error(), because mysql_error() requires a connection handle that we do

not get. Persistent connections do not need to be closed. Nonpersistent connections can optionally be closed with mysql_close(), but this is not required. Unlike C, PHP is nice enough to clean up after us if we don't. Perhaps this "free maid service" feature is one of the reasons it is so popular.

On the C level, PHP's mysql_connect() amounts to a combination of mysql_init() and mysql_real_connect(), and mysql_pconnect() may not result in any calls to the C API if the connection match is found in the pool. If it is not found, we do essentially what mysql_connect() does. PHP's mysql_close() translates into C's mysql_close() for nonpersistent connections and does nothing for persistent ones.

If your application is going to connect to just one MySQL server and maintain just one connection the entire time, you may want to take advantage of the default connection feature that allows you to omit the connection link argument in most API calls.

Unlike mysql_real_connect() in the C API, mysql_connect() and mysql_pconnect() do not allow the user to automatically select a database. Therefore, unless you plan to reference all of your tables using the db_name.tbl_name syntax, you should also call mysql_select_db() immediately after the connection has been established and before you start executing queries.

The query is executed with mysql_query(), which wraps around two C API calls: mysql_real_query() and mysql_store_result()/mysql_use_result(), depending on the value of the optional third argument (result_mode). The possible values are MYSQL_STORE_RESULT and MYSQL_USE_RESULT. The default is MYSQL_STORE_RESULT. On success for queries that return a result set the result set handle is returned. If the query was not supposed to return a result set (e.g,. UPDATE or DELETE), TRUE is returned; on failure, FALSE is returned.

Just like in the case of mysql_close(), it is possible to clean up early with a call to mysql_free_result(). However, if you forget, PHP cleans up after you at the end of the request.

Just as with the C API, the result set can iterate through sequentially. However, unlike the bland C API, which provides only mysql_fetch_row(), PHP offers a rich menu of mysql_fetch_*() routines to satisfy a gourmet taste: mysql_fetch_row(), mysql_fetch_array(), mysql_fetch_assoc(), and mysql_fetch_object(). All of these functions are essentially fancy or not-so-fancy wrappers around the C API call mysql_fetch_row(). mysql_fetch_row() is the fastest, while mysql_fetch_object() is perhaps most convenient, although not by far, and some might disagree depending on their personal view and typing habits.

All strings passed to queries as column values must be properly escaped. To accomplish this, use mysql_escape_string(). The good news is that unlike with

the C API, you do not have to worry about buffer size. PHP again comes to the rescue, and, depending on your viewpoint, you can say that it either frees programmers from thinking about annoying low-level implementation details and allows them to focus on the core of the problem, or, as some would say, allows lazy and sloppy programmers to produce working code without correcting their evil ways—and thus reinforcing the bad habits.

Sample Code

For the sample application, I have chosen to write the Web version of our command-line inventory program from Chapter 8. Due to the Web nature of the code, a good portion of it is dedicated to HTML generation and user form input processing. Although this may distract you from the database-manipulation techniques, which is our primary goal, I believe that since most uses of PHP are Web related—and database programming is tightly coupled with the Web interface—it is only natural to study HTML integration along with the database programming.

To see the application in action, copy sample.php from the Web site into a Web-accessible directory on your Web server, make sure MySQL is installed and running on the same system, and then type

```
http://webserver-hostname/path/to/sample.php
```

replacing the placeholder tokens with the actual values; for example, http://localhost/learn-mysql/sample.php. You will see a menu of options. The first thing you need to do is select Initialize Database. Afterward, you may add, view, delete, and update records, as well as view the totals.

Listing 9.1 contains the sample code with my comments. It is my hope that the comments inside the code will help you understand the code, but see the section that follows for a further discussion.

```
<?
// auxiliary variable for proper HTML generation
$html_started = 0;

// The line below tells PHP not to automatically escape user input
values with quotes, since we will be escaping them ourselves in the
// code.
ini_set("magic_quotes_gpc","0");
// Enable error tracking. We need this to have MySQL error in
// $php_errormsg in case mysql_connect() fails.
ini_set("track_errors","1");
```

Listing 9.1 Source code of sample.php. (continues)

```
// database connection parameters
$mysql_user = "root";
$mysql_host = "localhost";
$mysql_pass = "";
$mysql_db = "test";

// This class is used for the drop-down list during HTML form
// generation. No rocket science involved, but very convenient.
class form_option
{
  // Class members
  var $text, $val;

  // A very boring constructor
  function form_option($val_arg,$text_arg)
    {
      $this->text = $text_arg;
      $this->val = $val_arg;
    }

  // The name of this method should be self-explanatory
  function print_html($selected_val)
    {
      // Note that this method allows us to preselect items of our
      // choice by default.
      if ($this->val == $selected_val)
            $selected = "selected";
      echo("<option value=\"$this->val\" $selected>$this->text</option>");
    }
}

// Helper function to generate the HTML for a submit button
function submit_button($name, $text)
{
  echo("<input type=submit name=$name value=\"$text\">");
}

// Helper function to generate HTML for a drop-down list
// Note that we assume that $opts is an array of form_option objects.
function select_input($name, $prompt,$opts)
{
  global $action;
  echo("<table><tr><td>$prompt</td><td><select name=$name>");
  $len = count($opts);
  for ($i = 0; $i < $len; $i++)
  {
```

Listing 9.1 Source code of sample.php. (continues)

```
    $opts[$i]->print_html($action);
  }
  echo("</select></td><td>");
  submit_button("${name}_submit","Go!");
  echo("</td></tr></table>");
}

// Function to generate a form with a drop-down menu prompting the
// user to select an action
function menu_form()
{
 echo("<form method=post>");
 // After we have written some rather tedious auxiliary HTML code,
 // now we can generate a drop-down HTML input with appropriate
 // default preselection of items quite easily. The dirty work is
 // finally paying off.
 select_input("action","Select Action:",
                  array(new form_option("init","Initialize Database"),
                                  new form_option("add", "Add Record"),
                                  new form_option("delete","Delete
Record"),
                                  new form_option("select","Select
Record"),
                                  new form_option("update", "Update
Record"),
                                  new form_option("show_totals","Show
Totals")
                                  ));
 echo("</form>");
}

// Helper function to generate customized HTML at the top.
function html_start($title)
{
  // Do not generate it if we have already done it earlier.
  if ($html_started) return;

  echo("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01
 Transitional//EN\">");
  echo("<html><head><title>Sample Inventory Application -
 $title</title>
</head>");
  echo("<body><h1>$title</h1>");
  // Ensure that this function will be called only once by setting a
  // special flag. As you can see, we check this flag in the first
  // statement.
  $html_started = 1;
```

Listing 9.1 Source code of sample.php. (continues)

```
}

// Helper function to generate the standard bottom HTML
function html_end()
{
  echo("</body></html>");
}

// Function for gracious abort with a properly written message.
function error_exit($msg)
{
  if (!$html_started)
    html_start("Error");
  echo("Fatal error: $msg");
  exit();
}

// Function to connect to the database, and select the default one for
// this application, checking for errors and if any discovered, abort
// with an error message. We assume that the caller is always pleased
// with our error handling and does not want us to return if we are
// not successful.
function safe_connect()
{
  global $mysql,$mysql_host,$mysql_user,$mysql_pass,$mysql_db,
    $php_errormsg;
  // Note the @ in front of MySQL API calls. This prevents PHP
  // from automatically spitting out the error message in the HTML
  // output if the call fails.
  $mysql = @mysql_connect($mysql_host,$mysql_user,$mysql_pass);
  if (!$mysql)
    error_exit("Could not connect to MySQL: $php_errormsg");
  if (!@mysql_select_db($mysql_db,$mysql))
    error_exit("Could not select database $mysql_db:".mysql_error($mysql))  ;
}

// Safety wrapper function for mysql_query(). We take care of the
// error handling here. The assumption is that the caller expects this
// function to succeed and is satisfied with just aborting on error.
function safe_query($query)
{
  global $mysql;
  // Again, note @ before mysql_query() to prevent an error message
  // "spill" to the user. We want to control how the message is
  // printed in case of error.
  if (!($res=@mysql_query($query,$mysql)))
    error_exit("Failed running query '$query': ".mysql_error($mysql));
```

Listing 9.1 Source code of sample.php. (continues)

```
  return $res;
}

//Another convenience wrapper function for error messages. As the name
//suggests, signals to the user that there was an error in the entered
// data.
function input_error($msg)
{
  echo("<font color=red>Input error:</font>$msg<br>");
}

//Helper function to prompt the user to select a record by entering
//the record id.
function pick_record_form()
{
  global $action;
  // If the record is being picked for update or delete, we do not
  // want to give the user the options of picking all records.
  // However, this is an option if the user just wants to view the
  // record(s).
  if ($action == "select")
    $empty_for_all = "(empty for all)";
  // Note the use of hidden inputs to pass messages about the current
  // state of the application.
  echo("<form method=post><input type=hidden name=record_selected value=1>
<input type=hidden name=action value=\"$action\">
<table><tr><td>Record id $empty_for_all:</td>
<td><input name=record_id type=text size=5></td>
<td><input type=submit name=submit value=\"Submit\"></td>
</table></form>");
}

// Helper function for generating form HTML. Generates a table row
// consisting of a prompt with the input name, followed by the text
// input itself.
function table_input($prompt, $name,$size)
{
  echo("<tr><td>$prompt</td><td><input type=text name=$name size=$size
value=\"".$GLOBALS[$name]."\"></td></tr>");
}

// Another HTML generating helper. Writes out the HTML for the entire
// record form, pre-populating the fields if the record has been
// retrieved from the database already, or if the user has partially
// entered the data but made some errors or omissions. Again, note the
// use of hidden fields for state tracking. Also note the use of
// table_input() and submit_button() helpers, which make the code much
```

Listing 9.1 Source code of sample.php. (continues)

```
// less cumbersome, easier to read, and easier to modify.
function record_form()
{
  global $record_id,$action;
  echo("<form method=post><input type=hidden name=action
value=\"$action\"> <input type=hidden name=record_id
value=\"$record_id\">
 <input type=hidden name=record_entered value=1>
 <table>");
  table_input("Name","name", 15);
  table_input("Quantity","quantity", 5);
  table_input("Price", "price", 6);
  echo("<tr><td>");
  submit_button("submit","Submit");
  echo("</td></tr>");
  echo("</table></form>");
}

// Helper function to generate HTML for buttons in the record view
// provided by "Select Record" to prompt the user to update or delete
// the displayed record.
function td_form($record_id,$text,$action)
{
  echo("<td><form method=POST><input type=hidden name=record_id
value=$record_id><input type=hidden name=action value=\"$action\">
 <input type=submit name=submit value=\"$text\">
 </form></td>");
}

// Now the validation functions. Each of them returns TRUE if the
// respective input element was valid. Otherwise, it prints the
// appropriate error message about the input error and returns FALSE.
//
// Note the use of trim() to remove the heading and trailing space,
// and the intended side effect of setting the corresponding global
// value in all three of the functions, which we need to do manually,
// since we do not want to assume that register_globals is set to ON.

function validate_name()
{
  global $name;
  if (!strlen(($name=trim($_POST["name"]))))
  {
    input_error("Empty name");
    return FALSE;
  }
  return TRUE;
```

Listing 9.1 Source code of sample.php. (continues)

```
}

function validate_price()
{
  global $price;
  $price = trim($_POST["price"]);
  if (!strlen($price))
  {
    input_error("Empty price");
    return FALSE;
  }
  if (!preg_match("/^\d+(\.\d{1,2}){0,1}/",$price))
  {
    input_error("Invalid price");
    return FALSE;
  }
  return TRUE;
}

function validate_quantity()
{
  global $quantity;
  $quantity = trim($_POST["quantity"]);
  if (!strlen($quantity))
  {
    input_error("Empty quantity");
    return FALSE;
  }
  if (!is_numeric($quantity))
  {
    input_error("Invalid quantity");
    return FALSE;
  }
  return TRUE;
}

// Convenience wrapper to combine all three functions into one call
function validate_input()
{
  return validate_name() && validate_quantity() && validate_price();
}

// Helper function to fetch the record specified by global setting of
// $record_id from the database and set the corresponding global
// variables in accordance with the values of the fields. Note that we
// assume that the caller has already set the record_id to a numeric
// value, which is accomplished by get_record_id(). If the record is
```

Listing 9.1 Source code of sample.php. (continues)

```
// found in the database, we return TRUE, otherwise FASLE, expecting
// the caller to take measures.
function fetch_record()
{
  global $record_id,$name,$quantity,$price;
  $res = safe_query("SELECT name,quantity,price/100 as dollar_price
FROM inventory WHERE id = $record_id");
  if (!mysql_num_rows($res))
    return FALSE;
  $r=mysql_fetch_object($res);
  $name = $r->name;
  $quantity = $r->quantity;
  $price = $r->dollar_price;
  return TRUE;
}

// Helper function to ensure that the record id is set to a numeric
// value. After it returns, global $record_id will be set to a numeric
// value either entered by the user or passed through a hidden input,
// and TRUE will be returned. Otherwise, the user is prompted with a
// record id entry form, and FALSE is returned.
function get_record_id()
{
  global $record_id;
  $record_id = $_POST["record_id"];
  if (!isset($record_id))
  {
    pick_record_form();
    return FALSE;
  }
  if (!is_numeric($record_id))
  {
    input_error("Record id must be a number");
    pick_record_form();
    return FALSE;
  }
  return TRUE;
}

// Now we get to the meat of the application. The next functions are
// actually calling the shots. They are action handlers. Each of them
// is called from main after executing some common preamble.

// Drop the old table if it exists and re-create an empty one.
// Note that we use mediumint for price because we store the money in
// cents to save space. This function is called every time the user
// selects "Initialize database". There is no other user input to
```

Listing 9.1 Source code of sample.php. (continues)

```
// process, so we get straight down to business.
function init_db()
{
  safe_query("DROP TABLE IF EXISTS inventory");
  safe_query("CREATE TABLE inventory (
                id smallint unsigned NOT NULL auto_increment,
                name varchar(30) NOT NULL,
                price mediumint unsigned NOT NULL,
                quantity smallint unsigned NOT NULL,
                PRIMARY KEY  (id)
    )");
  // Having finished the database work, give the menu and report
  // success.
  menu_form();
  echo("Database initialized<br>");
}

// Handler for "Add Record" action. We collect the data, validate user
// input, and then process it. Note how the function is able to handle
// being called from different states, and how it determines the state
// by checking the record_entered variable.
function add_record()
{
  global $name,$quantity,$price;
  if (!isset($_POST["record_entered"]) || !validate_input())
  {
    record_form();
    return;
  }
  else
  {
    // Note the use of mysql_escape_string() to escape potentially
    // problematic characters. Also note the dollars->cents conversion
    // to accommodate for the use of mediumint for price storage
    safe_query("INSERT INTO inventory (name,quantity,price) VALUES
('".
                  mysql_escape_string($name)."',$quantity,$price*100)");
    // After we've done the database work, make users happy by giving
    // them a menu and telling them the work was successful.
    menu_form();
    echo("Record added<br>");
  }
}

// Handler for "Show Totals" action. No additional user input is
// required. Note how we handle NULL values in the fields by using
// isset() on the field member in the row object. Also note how we do
```

Listing 9.1 Source code of sample.php. (continues)

```
// not use the while loop, but instead call mysql_fetch_object() only
// once because the query will always return only one row.
function show_totals()
{
  // Since the price is stored in cents, and we want the output in
  // dollars, we divide the price by 100.
  $res = safe_query("SELECT SUM(quantity) as total_quantity,
 SUM(price*quantity)/100 as total_price FROM inventory");
  $r = mysql_fetch_object($res);
  // Give menu at the top before we print the results of the
  // computation.
  menu_form();
  // If we have no records, the query will return NULL in both
  // columns, not 0.
  if (!isset($r->total_quantity))
  {
    echo("The database has no records<br>");
  }
  else
  {
    echo("<table border=1><tr><td>Total Items</td><td>Total Value</td></tr>
 <tr><td>$r->total_quantity</td><td>$r->total_price</td></tr></table>");
  }
  // Although PHP cleans up for us at the end, we will be good and
  // clean up here to make the code cleaner and read easier.
  mysql_free_result($res);
}

// Handler for "Delete Record" action. We first need to get $record_id
// from the user.
function delete_record()
{
  global $record_id,$mysql;
  if (!get_record_id())
    return;
  safe_query("DELETE FROM inventory WHERE id = $record_id");
  // Again, make users happy with a menu and a report of the results.
  // Note that in some cases the record may not exist.
  menu_form();
  if (mysql_affected_rows($mysql))
    echo("Record deleted<br>");
  else
    echo("No such record, nothing deleted<br>");
}

// Handler for "Update Record". We require $record_id, as well as the
// entire record form filled out. If something is missing we ask for
```

Listing 9.1 Source code of sample.php. (continues)

```
// it.
function update_record()
{
  global $record_id,$name,$quantity,$price;
  if (!get_record_id())
    return;
  if (!($record_entered=$_POST["record_entered"]) ||
!validate_input())
  {
    if (!$record_entered)
    {
      // deal with a record id that is not in the database
      if (!fetch_record())
      {
            input_error("No such record");
            pick_record_form();
            return;
      }
    }
    record_form();
    return;
  }
  // Again, note the dollars->cents conversion and name escaping.
  safe_query("UPDATE inventory SET name =
'".mysql_escape_string($name).
                "', quantity=$quantity, price = $price * 100
 WHERE id = $record_id");
  // Report success.
  menu_form();
  echo("Record updated<br>");
}

// Handler for "Select Record". We deal with three cases--users have
// just come here from the menu, they have seen the record id prompt
// but entered no record id to see all of them, or they selected a
// record id. Once we understand the user's intentions, we proceed to
// do the work.
function select_record()
{
  // This condition would be true if the user comes here fresh from
  // the menu. Prompt the user for the record id.
  if (!isset($_POST["record_selected"]))
    pick_record_form();
  else
  {
    // Otherwise, we can proceed to do the work, although we still may
    // need to check for errors. But first print the menu since we
```

Listing 9.1 Source code of sample.php. (continues)

```
    // need to present it in any case.
    menu_form();
    // Start forming the query.
    $query = "SELECT id,name,price/100 as dollar_price,
quantity FROM inventory";
    // Did the user enter something in the record id field?
    if (strlen($record_id=trim($_POST["record_id"])))
    {
      // If yes, was it a number?
      if (!is_numeric($record_id))
      {
            // If not a number, signal an error and have the user try
            // again.
            input_error("Record id must be a number");
            pick_record_form();
            return;
      }
      // If the id was valid, incorporate the constraint into the
      // query.
      $query .= " WHERE id = $record_id";
    }
    // If there was no record id, we simply omit the WHERE clause and
    // select all the records.
    $res = safe_query($query);
    // Avoid the ugly table with just column names and no values
    // underneath. It looks less confusing if we just say that the
    // table is empty instead.
    if (!mysql_num_rows($res))
    {
      echo("No matching records found<br>");
      return;
    }
    // If that table is not empty, display the HTML for it.
    echo("<table border=1><tr><td>id</td><td>name</td><td>price</td>
<td>quantity</td><td></td><td></td></tr>");
    while (($r=mysql_fetch_object($res)))
    {
      echo("<tr><td>$r->id</td><td>$r->name</td><td>$r->dollar_price</td>
<td>$r->quantity</td>");
      // The calls below generate the HTML for Update and Delete
      // buttons for each displayed record.
      td_form($r->id,"Update","update");
      td_form($r->id,"Delete","delete");
      echo("</tr>");
    }
    echo("</tr></table>");
    // Again, the "good kid" rule. Even if Mom (or PHP) will clean up
```

Listing 9.1 Source code of sample.php. (continues)

```
    // after us, we want to do it ourselves to show that we are neat
    // and responsible.
    mysql_free_result($res);
  }

}

// main function. We do some common preamble initializing the $func to
// the handler we are going to execute as we go based on the setting
// of $action. Then if $func is not 0, call it, and then execute the //
common epilogue.

// Extract action from the _POST array. We do not depend on
// register_globals.
$action = $_POST["action"];
if (!isset($action))
{
  // If this is the first hit, we just say welcome and do nothing
  // else.
  $title = "Welcome to the sample inventory application";
  $func = 0;
}
else
{
  // Long and boring switch/case statement to initialize $title and
  // $func. Note that default is impossible unless we have a bug, the
  // browser is broken, or somebody has faked a strange value of
  // action in a cooked-up POST request, which is most likely an
  // indication that the user is probing for a security hole.
  switch ($action)
  {
  case "init":
    $title = "Initialize Database";
    $func = "init_db";
    break;
  case "add":
    $title = "Add Record";
    $func = "add_record";
    break;
  case "update":
    $title = "Update Record";
    $func = "update_record";
    break;
  case "select":
    $title = "Select Record";
    $func = "select_record";
    break;
```

Listing 9.1 Source code of sample.php. (continues)

```
  case "delete":
    $title = "Delete Record";
    $func = "delete_record";
    break;
  case "show_totals":
    $title = "Show Totals";
    $func = "show_totals";
    break;
  default:
    error_exit("Attempted break-in, script error, or browser bug
 detected");
  }
}
// HTML preamble
html_start($title);
// Is there something serious to do?
if ($func)
{
  // If yes, connect to the database, do it, then disconnect. Note
  // that PHP will disconnect for us, but we will do it ourselves to
  // keep the code clean and maintain a sense of responsibility for
  // cleanup.
  safe_connect();
  $func();
  mysql_close($mysql);
}
else
 menu_form();

html_end();
?>
```

Listing 9.1 Source code of sample.php. (continued)

Now let's summarize the main points of the code:

- The code has a lot of helper functions for HTML generation and user input processing.
- We use wrapper functions for database interface (safe_connect(),safe_query()) that do their job, checking and handling errors as they go. In addition to convenience, this gives us some flexibility. For example, if we later decided to be connected to several servers at a time and run all of our queries in a distributed manner, all we would have to do is reimplement safe_connect() and safe_query(). We could also very easily add a query logger that could log the thread id, the query itself, the result, and the execution time. We'd simply add a few lines to safe_query() and wouldn't have to touch the rest of the code.

- To iterate through the result, we use mysql_fetch_object() called in a loop. In cases where we know for sure that we can get only one row, we just call mysql_fetch_object() once. The returned row object has members with names corresponding to the names of the fields. In the case where the natural name of the field is ugly, such as SUM(quantity), we can alter it by using the AS alias_name syntax in the SQL query. After the last row has been returned, mysql_fetch_object() will return FALSE.
- To test a field value for NULL, we use the isset() PHP function.
- To escape problematic characters in the user input, we filter all the strings to be sent as part of a query through mysql_escape_string().
- Although PHP performs the cleanup by freeing all the result sets and closing all nonpersistent connections at the end of a request, we clean up after ourselves anyway with mysql_free_result() and mysql_close() to maintain the habit of neatness.
- The code assumes that the magic_quotes_gpc setting is off, or in other words, user input comes to us unmodified without the "automagic" quote escaping.
- We do not depend on the setting of register_globals, and do not assume that the POST variables have been preimported into the global variable namespace.
- We use the @ symbol in front of MySQL API calls that can generate error messages and print them into the output HTML.
- Each menu option is handled by a separate function, which is always called when that menu option is selected.
- We use hidden HTML form inputs to track the state of execution.
- We assume that the user is either not knowledgeable or malicious, and we validate all user input.

Tips and Tricks

In this section, we offer a list of techniques and suggestions that will help you avoid pitfalls and improve your application.

- PHP by default tries to make life easier for a novice programmer and handles many problematic issues automatically. However, this is usually not suitable for a serious database application. If you are developing one, you want to disable the default error handling behavior as a rule. The things to take care of are to disable magic_quotes_gpc, enable track_errors, and use @ in front of MySQL API calls, while checking for errors in your code.

- To get the error if mysql_connect() or mysql_pconnect() fails, you need to have the track_errors setting enabled and read it from $php_errormsg. Unlike other MySQL API functions, where you would use mysql_error(), the connection handle will not be initialized if the connection is not successful, and mysql_error() will not be able to read the error message. PHP interpreter settings are set in the php.ini configuration file or can be altered with calls to ini_set(). To learn more about how to work with PHP configuration settings, visit www.php.net/manual/en/configuration.php.
- To check a value retrieved from the database for SQL NULL, use the isset() function.
- Use mysql_escape_string() to escape potentially troublesome characters in the query.
- Use safety wrappers around MySQL API functions.
- Validate user input.
- Since your sysadmin may inadvertently change php.ini or otherwise reconfigure the system to make PHP run with different defaults, if your code depends on any of the php.ini settings, set them explicitly in your code with ini_set().
- Check for errors diligently. While C programmers are pretty good about it, Perl programmers are sometimes good, Java programmers have to do it or otherwise their code does not compile, and PHP programmers are the worst in this regard.

CHAPTER 10

Perl API Basics

Perl is a popular scripting language created by Larry Wall in 1986. In the early history of Web applications, Perl was probably the most popular tool, at least in the open source community, for delivering dynamic content. With the creation of PHP, Perl began to yield ground in the Web area, but it is still the language of choice in many cases for developing command-line data-manipulation scripts and utilities.

One advantage of Perl is that most Unix systems have the Perl interpreter preinstalled. It has become almost as widespread as the Bourne Shell. Unlike earlier scripting languages, such as Unix shells, AWK, and SED, Perl provides extensive capability to invoke system calls. Perl's functionality can be extended through modules, which has enabled the user community to provide a large number of packages that permit programmers to do just about anything they might need to in a very short amount of development time. And all of the above comes free of charge—the only thing required of the developer is to download the packages and learn how to use them.

In the area of databases, the strength of Perl is in the portability of the interface. Database connectivity is accomplished through the DBI and DBD modules, and the interface is designed in such a way that, provided the SQL queries were written with portability in mind, porting the code to another database can be as simple as changing the value of the data source variable.

For the hardcore enthusiasts, Perl is more than just a programming language. It is a way to express themselves. To them, a Perl program is not just a piece of code designed to do a certain job. It is also a poem or a piece of art. Therefore,

some Perl programmers are sensitive to the issues of style and language construct usage. I must admit that although I have done quite a bit of Perl programming—and have been quite fascinated with the profound expressive beauty of statements like *$_ =~ s/\-//g;*—I spend more time writing in C and PHP. This shows in the way I write Perl code. (Some of you notice that I speak Perl with a strong C accent.)

In this chapter, I provide some basic instructions to prepare your system to run Perl clients connecting to MySQL, give an overview of the Perl API, provide some sample code, and offer a few helpful tips.

System Preparation

The preparation process consists of first making sure that Perl is installed and then installing two Perl modules, DBI and DBD::mysql (if they have not been installed earlier during the system installation or some other process). If you plan to use transactions, you will need version 1.22.16 or later of DBD::mysql.

Since most Unix systems come with at least a partial installation of components needed to connect from Perl, instead of providing installation instructions from scratch, for the Unix part we tell you where to find out what you need to install first, and then how to do it.

To test if your system is ready to go, execute the following Perl one-liner:

```
perl -e 'use DBI; DBI->connect("dbi:mysql:test:host=localhost",
"root", "")->disconnect'
```

If the command produces no output, Perl itself and the DBI/DBD modules have been installed, and you may skip to the next section (unless you need to configure another system or are simply curious about how to proceed on a bare-bones system). If you see an error message, there could be several reasons.

If the message says “Access denied,” “Unknown database,” or “Can’t connect,” there is a reason to be happy. It means that Perl and the DBI and DBD modules are already installed but that the command-line test simply did not specify the proper access credentials (e.g., you have a non-empty password for root), it did not find the *test* database because it did not exist, or your server was simply not running on *localhost*. In other words, you do not need to do anything from the aspect of Perl-MySQL interaction, but you need to tweak your server installation or the sample code itself to make it run on your system.

Seeing the message “Command not found” on Unix or “Bad command or file name” on Windows indicates that you have not installed Perl. To install Perl, visit www.cpan.org, find your operating system, and then follow the instructions.

You may also see the message "Can't locate DBI.pm." If that is the case, you do not have the DBI driver installed yet. Visit www.cpan.org and download the DBI module.

If you have the DBI module installed but have not yet installed the MySQL DBD module, you will get a message "Can't locate DBD/mysql.pm." Go to www.mysql.com/api-dbi.html and download the file called Msql-mysql-modules-version.tar.gz, unpack the archive, and then proceed with the standard Perl module installation mantra:

```
perl Makefile.PL
make
make install
```

Note that it is important to install DBI before you install the MySQL module, and you should also have the MySQL client library and headers installed first.

The same process applies if you are planning to upgrade the module. If your DBD::mysql module version is older than 1.22.16, an upgrade is necessary to run the sample program we present later on in this chapter (because of transaction use). The sample program will check the version and tell you if it is too old.

On a Windows system, you need to first install ActivePerl and then get the DBI/DBD modules. The process is described in detail in the MySQL online manual at www.mysql.com/doc/en/Active_state_perl.html.

DBI Methods and Attributes

This section lists the most commonly used methods and attributes in the DBI module. You can find more detailed information by visiting http://search.cpan.org/author/TIMB/DBI-1.30/DBI.pm. To learn more about the DBD::mysql module, which works underneath the DBI module to provide support for MySQL connectivity, visit http://search.cpan.org/author/JWIED/DBD-mysql-2.1018/lib/DBD/mysql.pod.

The methods and attributes listed as follows can be divided into three categories: methods and attributes global to the DBI module, database handle methods and attributes, and statement handle methods and attributes.

Global methods and attributes

available_drivers: Returns an array of available database drivers. Usage example: @drivers = DBI->available_drivers;

connect($datasource, $username, $password): Creates a connection to a database using the datasource information and supplied username/password and returns a database handle reference. The datasource argument will reference a DBD source—for example, DBI:mysql:$database:$hostname:$port.

Usage example: $dbh = DBI->connect("DBI:mysql:products:localhost", "spider", "spider");

err: Attribute containing the database error code from the last operation. Usage example: die "Error code: $DBI::err\n" if ($DBI::err);

errstr: Attribute containing the database error message from the last operation. Usage example: die "Error: $DBI::errstr\n" if ($DBI::errstr);

Database handle methods and attributes

AutoCommit: Attribute controlling whether each query that modifies data should be automatically committed. Usage example: $dbh->{AutoCommit} = 1;

RaiseError: Attribute controlling whether errors should be checked for after each database call. If enabled, an error would result in aborting the program with the appropriate error message. Usage example: $dbh->{RaiseError} = 1;

disconnect: Closes the connection to the database server associated with the database handle. Usage example: $result_code = $dbh->disconnect;

prepare($query): Prepares the query for execution and returns a statement handle. In MySQL, the call is more of a protocol formality because MySQL does not yet support prepared queries (as of this writing). Usage example: $sth = $dbh->prepare($query);

quote($col): Quotes the value (by surrounding it with quotes), escaping potentially problematic characters that may confuse the query parser. Usage example, $quoted_col = $dbh->quote($col);

Statement handle methods and attributes

bind_columns(column variable): Binds all of the columns in a result row to a specific variable. Subsequent calls to fetch() will set the variables to the query values fetched from the result set. Usage example: $sth->bind_columns($var1, $var2, undef, $var3);

do($query): Prepares and executes the query in the same fashion as the prepare/execute combination. Usage example: $rows = $dbh->do($query);

execute: Executes a prepared statement returned by the prepare() function and returns the result value. The result is usually a row count. For example, $rows = $sth->execute;

fetch: Retrieves a row from the statement handle and places it in the variables that have previously been bound to result set columns with a bind_columns() call.

fetchall_arrayref: Fetches all data from the statement handle and returns a reference to an array of references. Each element of the array represents a row. Usage example: $all_rows_ref->fetchall_arrayref;

fetchrow_array: Fetches the next row from the statement handle and returns it as an array. Usage example: @rowData = $sth->fetchrow_array;

fetchrow_arrayref: Fetches the next row from the statement handle and returns it as an array reference. For example, $row_ref = $sth->fetchrow_arrayref;

fetchrow_hashref: Fetches the next row from the statement handle and returns a hash reference. Usage example: $row_hash_ref = $sth->fetchrow_hashref;

finish: Destroys the statement handle after the work has finished freeing the system resources allocated by it. Usage example: $sth->finish;

rows: Returns the number of rows affected by the last query. Meaningful only on a query that modifies the table. Usage example: $affected_rows = $sth->rows;

NUM_OF_FIELDS: Returns the number of fields in the query result. Usage example: $fieldCount = $sth->(NUM_OF_FIELDS);

NULLABLE: Returns a reference to an array of values that indicate which columns can contain NULLs. The values in the array have the following meaning: 0 or empty string—no, 1—yes, 2—unknown.

API Overview

The Perl API is a DBI wrapper around MySQL C API calls. While a detailed discussion of DBI is beyond the scope of this book, we provide a quick primer to get you started and be able to get the job done. (If you're interested, dbi.perl.org is a good start for an in-depth DBI discussion.)

DBI is a database-independent Perl interface to SQL databases. It allows the programmer to write portable code that will run with a large number of databases as a backend while requiring only a very minimal code change: the name of the driver in the connect call. DBI accomplishes this by abstracting the database communication concepts of connecting, preparing a query, executing it, processing the result, and disconnecting. Each concept has a corresponding abstract method, which in turn goes down to a lower-level interface driver that is able to talk to the given database using the specific protocol that it understands.

A typical program utilizing the DBI module will have a use DBI; statement at the beginning. The first call is DBI->connect(), which attempts to establish a connection to the database loading the appropriate DBD driver. Following the motto of Perl ("There is more than one way to do it"), there are many ways to pass arguments to DBI->connect(). Rather than discuss all of them, we mention just one that works well for MySQL in most cases: The first argument is a DSN in the form of dbi:mysql:$db_name:host=$host;port=$port;socket=$socket. The second argument is the MySQL username, and the third is the password of that user. The fourth argument is a hash of options, in which you may want to set PrintError to 0 when you are writing a serious application.

DBI->connect() will return a connection handle object on success; otherwise, it returns undef. If you did not disable PrintError, an error message also will be printed to stderr. You will use the returned handle for all the subsequent operations. For simplicity, we refer to the handle returned from DBI->connect() as $dbh in the discussion that follows.

If a query does not return any results, you should use $dbh->do(). On success, the return value is the number of affected rows. On error, undef is returned. In order to allow statements like $dbh->do($query) || die "Query failed"; where we directly evaluate the return value for truth, to in case of success that nevertheless affected no rows 0E0 is returned, which evaluates to a true value in the boolean sense.

If the query returns a result set, it is performed in two stages. First, you call $dbh->prepare(), which returns a statement handle ($sth in the future context) and then call $sth->execute().The dual-stage prepare/execute mechanism is designed for databases that support prepared statements. MySQL does not, but the low-level DBD driver has to comply with the DBI interface requirements in this regard. $sth->execute() completes the execution of the query and returns a true value on success and undef on error.

Once the statement has been executed, the result can be retrieved one row at a time with $sth->fetchrow_array(), $sth->fetchrow_arrayref(), $sth->fetch(),or sth->fetchrow_hashref(). To retrieve all rows, call one of those methods in a loop. When all rows have been retrieved, undef will be returned. Upon finishing the work with $sth, you must call $sth->finish(). When you are done with the connection handle, you must call $dbh->disconnect().

Sample Code

In Listing 10.1, we use a script that updates bank accounts and prints reports for our sample code. For convenience, we also add a special option on the command line to initialize the database.

To run the script, download sample.pl from the book Web site, and first type

```
perl sample.pl i
```

If all is well, you will see a message telling you that the database has been initialized. After that, you can run perl sample.pl several times in a row and watch how those sample customers get richer and richer with every run. As an exercise, you can add another customer to the database, either through the command-line interface or by adding another call to add_customer_with_deposit() in init_db().

The comments in the code should explain quite well what is happening and provide some instruction on how to interface with MySQL using Perl. In addition to the comments, we provide a review following the listing.

```
#! /usr/bin/perl

# Ask Perl to make it more difficult for us to write sloppy code
use strict;

# Need the DBI driver
use DBI;

# These variables are going to be global - tell 'strict' about it
# Explanation of the cryptic names: dbh - Database Handle, DSN - Data Source
# Name, and drh - Driver Handle
use vars qw($dbh $user $host $db $pass $dsn $drh);

#Now declare our global variables
my $dbh;
my ($user,$host,$db,$pass,$dsn,$drh);

# configuration settings for database connectivity
$host = "localhost";
$user = "root";
$db = "test";
$pass = "";
$dsn = "dbi:mysql:$db:host=$host";

# Error handling routines are as important for a good program as a goalie for
# a good soccer team. That is why before we do anything else we protect
ourselves
# by writing this generic emergency exit function.
sub dbi_die
  {
```

Listing 10.1 Perl MySQL sample code. (continues)

```
    my $msg = shift;
    #We need to save the error message since the cleanup will destroy it.
    #Alternatively, we could have printed the message, done the cleanup, and
    #then exited.
    my $dbi_msg = $DBI::errstr;
    do_cleanup();
    die("FATAL ERROR: $msg: DBI message: $dbi_msg}\n");
  }

# Roll back the transaction and disconnect from the database
sub do_cleanup
  {
    if (defined($dbh))
      {
           $dbh->rollback();
           $dbh->disconnect();
      }
  }

# Wrapper connection function that allows us to not worry about error
handling.
# It also provides flexibility because, for example, it can be modified to
connect
# to an alternative server if the primary one is not available, and we would
# not have to change the rest of the code.
sub safe_connect
  {
    ($dbh = DBI->connect($dsn,$user,$pass,{PrintError => 0,AutoCommit => 0}))
      || dbi_die("Could not connect to MySQL");
  }

# Wrapper query function. Again, the same idea as with safe_connect() - ease
of
# use and flexibility.
sub safe_do
  {
    if (!$dbh->do(@_))
    {
      my $query = shift;
      dbi_die("Error running '$query'");
    }
  }

# Another wrapper query function. This one combines $dbh->do() and
# $sth->execute() in one in addition to error checking.
sub safe_query
  {
```

Listing 10.1 Perl MySQL sample code. (continues)

```
    my $sth;
    my $query = shift;
    $sth = $dbh->prepare($query);
    $sth->execute(@_) || dbi_die("Could not execute '$query'");
    return $sth;
  }

# Helper function to insert a customer record and create one account
# for the customer. Overall, a rather boring function, other than the
# fact that we count money in cents to be able to stick to integer
# arithmetic, which is where money conceptually belongs. How would you
# like it if you came to the store and they told you that the can of
#orange juice you want to buy will cost you the square root of 2
# dollars, and with tax it is going to be PI/2? You round it to $1.58,
# and they file off a piece of a penny to give you for change.
#
# Another noteworthy thing in this function is storing the interest
# rate as an integer. We assume that our interest rate precision is
# %0.01, and therefore multiply the percentage value by 100. Do not do
# such tricks if you plan to have users connect to the database
# directly and look at tables - they will be thoroughly confused. But
# if all access to the data goes through your application, this is
# invisible to the user and helps reduce storage requirements.
sub add_customer_with_deposit
  {
    my $query;
    my ($fname,$lname,$ssn,$street,$city,$state,$zip,$amount) = @_;
    $query = "INSERT INTO customer (fname,lname,ssn,street,city,state,zip)".
                "  VALUES (?,?,?,?,?,?,?)";
    # Note the arguments of safe_do(). It will replace ? with
    # respective values and quote them with $dbh->quote(). Also note
    # that the second argument is an attribute hash and is usually set
    # to undef.
    safe_do($query,undef,$fname,$lname,$ssn,$street,$city,$state,$zip);
    my $interest_rate = 375; # 3.75%
    $query = "INSERT INTO account (customer_id,interest_rate,amount)
        VALUES (LAST_INSERT_ID(),$interest_rate,$amount * 100)";
    safe_do($query);
   }

# Creates the tables initially when the program is run with the i
# argument and populates it with dummy data. Should be run only once.
#
# We have two tables - customer and account. A customer is allowed to
# have more than one account. Each customer and account are identified
# with a surrogate unique id. To demonstrate the use of transactions
# and to remind you that MySQL does support transactions
```

Listing 10.1 Perl MySQL sample code. (continues)

```
# nowadays, we make both tables transactional - TYPE=INNODB.
#
# Note the partial key on only the first 5 letters of
# the first name and only the first 5 letters of the last name. This
# is a heuristical decision - the reasoning that leads us to this
# decision is that if two last names have the same first 5 letters
# there is a high likelihood they are either identical or belong to a
# class of similar names of small variety (e.g., {Anderson,Andersen}),
# and the same is true for the first names. Therefore, this way of
# partial column indexing will help us reduce the size of the key
# without significantly reducing the uniqueness of key values.
#
# Note also that we store money as the number of cents and percentage
# rate as the actual percentage rate multiplied by 100. With the
# assumption that we only need %0.01 precision we can store it as an
# integer. We could optimize even further. Since the interest rate is
# usually attached to the account type, we could have account_type_id
# tinyint unsigned not null, and have in a separate small table a full
# description of the account, including the rate.
#
# For simplicity, we omit a lot of columns you will actually need if
# this was a real bank application.
sub db_init
  {
    # First, get the old tables out of the way
    safe_do("DROP TABLE IF EXISTS customer,account");
    # Now create tables
    my $query = q{CREATE TABLE customer
                        (
                           id int unsigned not null auto_increment primary
                           key,
                           fname varchar(30) not null,
                           lname varchar(30) not null,
                           ssn char(9) not null,
                           street varchar(30) not null,
                           city varchar(30) not null,
                           state char(2) not null,
                           zip char(5) not null,
                           key(lname(5),fname(5)),
                           key(ssn)
                        ) TYPE=INNODB
                        };
    safe_do($query);
    $query = q{CREATE TABLE account
                   (
                       id int unsigned not null auto_increment primary key,
                       customer_id int unsigned not null,
```

Listing 10.1 Perl MySQL sample code. (continues)

```
                    interest_rate smallint unsigned not null,
                    amount  int unsigned not null,
                    key(customer_id),
                    key(amount)
                    ) TYPE=INNODB
                };
    safe_do($query);
    # Populate the created tables. For the curious, I have made sure
    # that the street names and the zips are real with the help of
    # www.mapquest.com. Social Security numbers and names are, of
    # course, made up.
    add_customer_with_deposit("Tim", "Jones", "111234567", "1400
Maple",
                                    "Los Angeles", "CA", "90015",
1000.00);
    add_customer_with_deposit("Chris", "Barker", "111234567", "1145
State",
                                    "Springville", "UT", "84664",
2000.00);
    add_customer_with_deposit("Jim", "Marble", "678456788", "230 N
Pioneer",
                                    "Mesa", "AZ", "85203", 1500.00);
    # Commit the transaction
    $dbh->commit();
    print("Database initialized\n");
  }

# Adds annual interest to all accounts. Note the computation of the
# increase - we divide the interest rate by 10000 instead of 100
# because we store it multiplied by 100 to make it an integer.
sub update_account
  {
    my $query = q{UPDATE account SET amount = amount +
                  amount * (interest_rate/10000)};
    safe_do($query);
    $dbh->commit();
  }

# Prints the summary reports. There are a few subtle details to take
# care of. Look in the comments for details.
sub print_report
  {
    # Report header
    print "Account Summary Report\n";
    print "————————————————————-\n";

    # Get the sum and the maximum
```

Listing 10.1 Perl MySQL sample code. (continues)

```
    my $query = "SELECT SUM(amount)/100,MAX(amount) FROM account";
    my $sth = safe_query($query);
    my ($total, $max_amount,$id,$fname,$lname);

    # bind_columns() + fetch() is the most efficient way to read
    # the rows from a statement handle, according to the DBI
    # documentation. After the following invocation of bind_columns(),
    # the first column from the query will be put into $total, while
    # the second will be in $max_amount.
    $sth->bind_columns(\$total,\$max_amount);

    # There is only one row in the result, so we call fetch() only
    # once.
    $sth->fetch();

    # Need to test for NULL. If the value is SQL NULL we get undef in
    # the corresponding variable from DBI
    if (!defined($total))
      {
           print "There are no accounts\n";
           # Note the cleanup with $sth->finish() before return.
           $sth->finish();
           return;
      }
    # Now we print the total, but hold off on the maximum amount. We
    # will print later when we retrieve the name(s) of the customer(s)
    # with that amount.
    print "Accounts Total is \$$total\n";

    # clean-up of the old statement handle
    $sth->finish();

    # Now let us retrieve the names of the customers who had the
    # maximum amount.
    $query = "SELECT customer.id,account.id,fname,lname,amount/100
FROM customer,account WHERE
customer.id = account.customer_id AND amount = $max_amount";
    $sth = safe_query($query);

    # This trick allows us to keep our English grammar straight.
    # Instead of the ugly Account(s), we print Account if there was
    # only one and Accounts otherwise. The number of rows is in
    # $sth->rows
    my $plural = "" ;
    $plural = "s" if ($sth->rows > 1);
    print "Top Account$plural:\n";
```

Listing 10.1 Perl MySQL sample code. (continues)

```
    # Now we proceed with the names of the customers having the top
    # deposit amounts. We begin with a column heading.
    print "Customer ID\tAccount ID\tFirst Name\tLast Name\tAmount\n";
    my ($customer_id,$account_id,$fname,$lname,$amount);

    # Repeat the bind_columns() manuever
    $sth->bind_columns(\$customer_id,\$account_id,\$fname,\$lname,
                           \$amount);

    # This time we fetch() in a loop since there can be more than one
    # customer with the maximum amount.
    while ($sth->fetch())
      {
           # Print each row, tab-delimited
           print "$customer_id\t$account_id\t$fname\t$lname\t$amount\n";
      }
    # Cleanup
    $sth->finish();
  }

# Now the code for main

# First make sure that the MySQL DBD driver is sufficiently recent
# This is needed because we are using transactions.
#
# Note that there is a discrepancy between the version number in the
# variable# and the official version of the driver, but luckily, they
# correlate and we can do version checks.
($drh = DBI->install_driver("mysql")) ||
  dbi_die("Failed to load MySQL driver");
if ($drh->{"Version"} < 2.0416)
{
  dbi_die("Please install MySQL driver version 1.22.16 or newer\n");
}

# Connect to the database. The earlier work is rewarded - we do not
# need to check for errors and everything is done in one neat line.
# All ugliness has already been put into the implementation of
# safe_connect().
safe_connect();

# Check to see if the first argument is i. If yes, initialize the
# database; otherwise, update the accounts with the interest earned
# and print the report.
#
# This is a rather ugly interface hack. In a real-world application,
# we would have a separate script that would create the tables, plus
```

Listing 10.1 Perl MySQL sample code. (continues)

```
# an interface for updating them. However, to make everything work
# from one file in order to simplify the setup process of this demo,
# we do it this way.
if ($ARGV[0] eq "i")
  {
    db_init();
  }
else
  {
    update_account();
    print_report();
  }
# Disconnect before exit.
$dbh->disconnect();
```

Listing 10.1 Perl MySQL sample code. (continued)

Now let's examine a few significant points in the code:

- We use wrappers around databases that perform error checking and could be extended to do other things, such as error logging, performance measurement, or load-balancing.
- Because we use transactions, we need to make sure that DBD::mysql is the sufficiently recent version.
- Because we have chosen to perform optimizations in our schema to convert the attributes (which are usually given the types float or decimal) to integer (amount and interest_rate), we perform multiplications in the queries to adjust for the difference between internal representation and input/display values. As stated in the comments, this is not recommended if you allow your users direct access to the database.
- Because MySQL does not support prepared queries, we are satisfied with multiple calls to prepare() + execute() combination. The performance gain from one call to prepare() and subsequent calls to execute() with different parameters would bring only a marginal improvement, resulting from a slightly more efficient internal structure management in Perl.
- bind_columns() + fetch() is the fastest way to iterate through the data.

Unlike PHP, Perl does no automatic cleanup for us—we clean up ourselves.

We check for SQL NULL values from the rows for undef.

Tips and Tricks

This section provides a few additional techniques and suggestions for using Perl to work with MySQL.

- Keep in mind that in Perl there is always more than one way to do anything. Choose the way that is most familiar to you and that solves your problem. Do not worry too much about "political correctness."
- If your program is of significant size and complexity, Perl's use strict helps keep you out of trouble. However, if this is a 100-liner cron job to import the data, use strict may create more trouble than it will save you from.
- Write code in a way that makes it easy for you to handle errors. Handle all errors.
- Have wrappers around database calls. This allows room for easy customization in the future.
- Validate user input and properly escape it. This can be done either with $dbh->quote() or by using the binding technique with ? placeholders in the query and passing the value parameters to $dbh->do() and $dbh->execute(). Do not forget that the second argument of dbh->do() is an attribute array, not the first placeholder substitution value.
- Clean up after yourself by always calling $sth->finish() and $dbh->disconnect() when you are done. Perl will not do it for you automatically like PHP will.
- If a column in a row has a NULL value, you do not get a "NULL" string like you might expect. The value is Perl undef.

CHAPTER 11

Java Client Basics

The first public version of Java was released in 1995. The language came about as a result of a group of Sun developers who were not quite satisfied with C++. Since then, it has been gaining in popularity in the corporate world and has become a major player in the enterprise.

Nevertheless, MySQL developers have looked at Java with an eye of skepticism for a long time, and only recently became serious about using the language. For a while, Mark Matthews maintained his MySQL JDBC (Java Database Connectivity) driver on his own time until he was hired in the summer of 2002 to work full-time for MySQL AB.

Although not the most efficient language in terms of CPU and memory resources, Java does have a set of advantages that make it popular in the enterprise. Among them are a very rich set of APIs, a logical and consistent API object hierarchy, excellent documentation, solid internal error-handling mechanisms, garbage collection, and free availability. As the language developed, the initial performance drawbacks were somewhat diminished through technologies such as JIT (the just-in-time compiler) and Sun's HotSpot optimization engine. Nowadays, Java performance, although still behind C/C++, is at least not by an order of magnitude behind if the programmer is able to avoid performance-killer temptations such as instantiating objects in a loop and frequently copying large arrays.

In this chapter, we briefly cover the issues related to communicating with MySQL from Java. To a certain extent, it is written as a Java/MySQL survival guide for C/C++ developers. The "pure" Java programmers will notice a strong C accent in my Java code.

System Configuration

First of all, you must install the Java compiler and virtual machine. If you do not have it installed already, visit java.sun.com and follow the download and installation instructions. The next step is to install MySQL Connector/J, so go to www.mysql.com/downloads/api-jdbc-stable.html and download the distribution. Unpack the archive in the directory where you would like to have it installed, and set your CLASSPATH variable to point at that directory. For example:

```
mkdir /usr/lib/java
cd /usr/lib/java
gtar zxvf /tmp/mysql-connector-java-2.0.14.tar.gz
ln -s mysql-connector-java-2.0.14 mysql-connector
CLASSPATH=$CLASSPATH:/usr/lib/java/mysql-connector
export CLASSPATH
```

Now you should be able to use MySQL Connector/J in your applications.

It is also possible to install the development version of JDBC. To get the development version, visit www.mysql.com/downloads/api-jdbc-dev.html. The development version offers some extra features and speed enhancements, and might be worth a try. It does pass all of the Sun JDBC Compliance test suite with the exception of the stored procedures, which are not supported by the MySQL server.

JDBC Classes and Methods

Java applications connect to MySQL through the JDBC interface. Native MySQL API does not exist—everything happens on the JDBC level. Our discussion of the MySQL API for Java, therefore, is essentially reduced to the discussion of JDBC with the exception of MySQL-specific driver parameters. Let's begin with a listing of the most commonly used JDBC classes and their methods. Strictly speaking, Java has three entities that correspond to the C++ notion of a class: classes, interfaces, and exceptions. However, all of them from the point of view of a programmer can be instantiated, copied, referenced, and have methods to invoke, and for all practical purposes act like what a C++ programmer would call a "class." Therefore, for simplicity's sake, we do not distinguish among them and refer to all three as "classes."

For each class, we state its name, briefly describe it, list its most frequently used methods, and provide a URL containing more detailed information. The issue of which methods are most frequently used is, of course, rather debatable; different programmers have different styles and will therefore use a different

combination of method calls to solve the same problem. The selected listing is based on what in my opinion is the most appropriate for MySQL. I encourage you to visit the detailed-information URLs to look at the entire collection of methods for each class to decide for yourself which methods are the most appropriate for your style and your application.

For more in-depth information on JDBC, see java.sun.com/products/jdbc/.

Class: Blob

Java representation or mapping of the SQL BLOB (binary-large object).

Frequently used methods

```
byte[] getBytes(long pos, int length)
```

Returns the BLOB or its portion as an array of bytes.

```
long length()
```

Returns the length of the BLOB.

Detailed documentation URL

java.sun.com/j2se/1.4/docs/api/java/sql/Blob.html

Class: Connection

Connection descriptor. Stores the data associated with a connection to a database server, and provides methods to perform operations on the given connection.

Frequently used methods

```
void close()
```

Closes the connection and releases the resources associated with it.

```
void commit()
```

If the database supports transactions, commits the current transaction.

```
Statement createStatement()
```

Creates and returns a Statement object used for sending queries to the database.

```
boolean getAutoCommit()
```

Returns TRUE if the connection is in auto-commit mode, and FALSE otherwise.

```
PreparedStatement prepareStatement(String query)
```

Prepares the query for execution. Returns a prepared Statement object.

```
void rollback()
```

If the database supports transactions, rolls back the current transaction.

```
void setAutoCommit(boolean autoCommitMode)
```

Sets the auto-commit mode for the given connection.

Detailed documentation URL

java.sun.com/j2se/1.4/docs/api/java/sql/Connection.html

Class: DatabaseMetaData

Contains and provides information about driver and database capabilities, which facilitates writing portable applications.

Frequently used methods

```
String getDatabaseProductVersion()
```

Returns the version string of the database.

```
String getDriverVersion()
```

Returns the version string of the JDBC driver.

```
int getMaxRowSize()
```

Returns the maximum length of a table record.

```
int getMaxIndexLength()
```

Returns the maximum length of a key.

```
int getMaxStatementLength()
```

Returns the maximum length of a query.

```
int getMaxTableNameLength()
```

Returns the maximum length of a table name.

```
int getMaxTablesInSelect()
```

Returns the maximum number of tables you can use in a SELECT query.

```
boolean supportsOuterJoins()
```

Returns TRUE if the database supports some form of outer join, FALSE otherwise.

```
boolean supportsTransactions()
```

Returns TRUE if the database supports transactions, FALSE otherwise.

```
boolean supportsUnion()
```

Returns TRUE if the database supports UNION queries, FALSE otherwise.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/DatabaseMetaData.html

Class: Driver

This class is loaded at the beginning of each JDBC program, and serves as an entry point for a specific JDBC driver. Each JDBC driver will have its own Driver class.

Frequently used methods

```
boolean acceptsURL(String url)
```

Validates the URL syntax. Returns TRUE if the driver thinks it can open a connection to this URL, and FALSE otherwise.

```
Connection connect(String url, Properties[] info)
```

Attempts to connect to the specified URL with the properties specified in the list. Returns a Connection object associated with the database connection.

```
int getMajorVersion()
```

Returns the major version number of the driver.

```
int getMinorVersion()
```

Returns the minor version number of the driver.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/Driver.html

Class: DriverManager

As the name of the class suggests, manages JDBC drivers. All of its methods are static.

Frequently used methods

```
Connection getConnection(String url)
```

Attempts to establish a connection to the given URL. Returns a Connection object.

```
Driver getDriver(String url)
```

Attempts to locate a driver that understands the argument URL. Returns a Driver object.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/DriverManager.html

Class: PreparedStatement

Defines an object associated with a precompiled SQL statement. Because MySQL does not support prepared statements, there will be no performance gains from using one on the database end, although there might be some on the driver, depending on the query. The main advantage in using one is the ability to use placeholders (?) in the query, along with automatic escaping of quotes and other troublesome characters.

Frequently used methods

```
ResultSet executeQuery()
```

Executes a query, usually SELECT, and returns a ResultSet object.

```
int executeUpdate()
```

Executes a query that produces no result set (e.g., INSERT, UPDATE, DELETE). Returns the number of affected rows.

```
ResultSetMetaData getMetaData()
```

Returns the ResultSetMetaData object, which contains information about the result set (such as the number of columns) when this statement is executed.

```
void setBlob(int i, Blob x)
```

The ith occurrence of the placeholder (?) in the query will be replaced with the properly escaped and quoted SQL representation of x.

```
void setDate(int i, Date x)
```

The ith occurrence of the placeholder (?) in the query will be replaced with the SQL representation of x.

```
void setFloat(int i, float x)
```

The ith occurrence of the placeholder (?) in the query will be replaced with the value of x.

```
void setInt(int i, int x)
```

The ith occurrence of the placeholder (?) in the query will be replaced with the value of x.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/PreparedStatement.html

Class: ResultSet

Defines an object associated with the result set of a query. Returned by the executeQuery() method of Statement and PreparedStatement.

Frequently used methods

```
void close()
```

Releases all of the resources associated with the given ResultSet object. If it is not called explicitly, the resources will be released during garbage collection.

```
void first()
```

Moves the cursor to the first row in the result set. Returns TRUE if the cursor is on a valid row, and FALSE is there are no rows in the result set.

```
Blob getBlob(int columnIndex)
```

Returns a BLOB object with the value of the column specified by the argument in the current row.

```
int getFloat(int columnIndex)
```

Returns the integer value of the column specified by the argument in the current row.

```
int getInt(int columnIndex)
```

Returns the integer value of the column specified by the argument in the current row.

```
String getString(int columnIndex)
```

Returns a String object with the value of the column specified by the argument in the current row.

```
int getRow()
```

Returns the current row number.

```
boolean isFirst()
```

Returns TRUE if the cursor is on the first row, and FALSE otherwise.

```
boolean isLast()
```

Returns TRUE if the cursor is on the last row, and FALSE otherwise.

```
boolean last()
```

Moves the cursor to the last row in the result set. Returns TRUE if the cursor is on a valid row, and FALSE if there are no rows in the result set.

```
boolean next()
```

Moves the cursor to the next row. Returns TRUE if the current row is valid, and FALSE if there are no more rows.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/ResultSet.html

Class: ResultSetMetaData

Defines an object describing properties of a result set, such as the number and names of columns.

Frequently used methods

```
int getColumnCount()
```

Returns the number of columns in the result set.

```
String getColumnName(int i)
```

Returns the name of the *i*th column in the result set.

```
int getColumnType(int i)
```

Returns the type of the *i*th column in the result set.

```
String getColumnTypeName(int i)
```

Returns a string representation of the type of the *i*th column.

```
String getTableName(int i)
```

Returns the name of the table that the *i*th column in the result set belongs to.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/ResultSetMetaData.html

Class: SQLException

Exception object thrown on database errors. Extends Exception.

Frequently used methods

```
int getErrorCode()
```

Returns the database error code number associated with the exception.

```
String getMessage()
```

Returns the error message string.

```
void printStackTrace(PrintWriter s)
```

Prints the stack trace into the stream s.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/SQLException.html

Class: SQLWarning

Exception object thrown on database warnings. Extends SQLException.

Frequently used methods

```
int getErrorCode()
```

Returns the database error code number associated with the exception.

```
String getMessage()
```

Returns the error message string.

```
void printStackTrace(PrintWriter s)
```

Prints the stack trace into the stream s.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/SQLWarning.html

Class: Statement

Defines an object associated with a static (as opposed to prepared) SQL statement.

Most frequently used methods

```
ResultSet executeQuery()
```

Executes a query, usually SELECT, and returns a ResultSet object.

```
int executeUpdate()
```

Executes a query that produces no result set (e.g., INSERT, UPDATE, DELETE). Returns the number of affected rows.

```
ResultSetMetaData getMetaData()
```

Returns the ResultSetMetaData object, which contains information about the result set (such as the number of columns) when this statement is executed.

Detailed information URL

java.sun.com/j2se/1.4/docs/api/java/sql/Statement.html

API Overview

Your application must first load the driver. This is done with Class.forName ("com.mysql.jdbc.Driver").newInstance(). Note that forName() throws ClassNotFoundException , and newInstance() throws InstantiationException

and IllegalAccessException. The exceptions must be handled or declared by the method that loads the driver. A simple way to deal with all of them can be

```
try
{
    Class.forName("com.mysql.jdbc.Driver").newInstance();
}
catch (Exception e)
{
  System.err.println("Error loading driver: " + e.getMessage());
  System.exit(1);
}
```

Once the driver is loaded, you can establish the connection as follows:

```
java.sql.Connection con =
java.sql.DriverManager.getConnection("jdbc:mysql://" + host +
"/" + db + "?username=" + user + "&password=" + password);
```

The argument to getConnection() is a JDBC URL (Uniform Resource Locator) string. The concept of a URL is used not only with JDBC, but in many other client-server protocols. When used with JDBC, the format of the URL is

```
<protocol>:<subprotocol>:<location>
```

For JDBC, the protocol is jdbc, and subprotocol is the name of the driver. In the case of MySQL Connector/J, it is mysql. location defines the name or IP address of the server, port, initial database, authentication credentials, and a few other connection parameters, and should follow this syntax:

```
//host[:port][/db][?driver_arg=driver_arg_val][&driver_arg=driver_arg_
val]*]
```

host is the name or IP address of the database server. The optional *port* argument defines the TCP/IP port to connect to. When this is omitted, the default is 3306. There is no need to specify it explicitly unless MySQL is running on a nonstandard port, but it does not hurt if you do. The db argument specifies the name of the initial database. The URL can optionally specify additional arguments after the ? delimiter. Each argument setting is separated by the & delimiter. For example:

```
jdbc:mysql://localhost/products?user=test&password=test
```

The above URL defines a JDBC connection to a MySQL server running on *localhost* on the default TCP/IP port to the database *products* with the username set to *test* and the password set to *test*.

The most common URL arguments for MySQL Connector/J are

- **user:** the username for authentication.

- **password:** the password to use for authentication.
- **autoReconnect:** if set to true, reconnect if the connection dies. By default, it is set to false.
- **maxReconnects:** if autoReconnect is enabled, specifies the maximum number of reconnection attempts.
- **initialTimeout:** if autoReconnect is enabled, specifies the time to wait between reconnection attempts.

The URL arguments for MySQL Connector/J are fully documented in the README file in its distribution archive. The file also discusses special features, known issues, and tips for this driver. All users of MySQL Connector/J should check out this file.

Note that getConnection() throws SQLException, which you will need to deal with. If you do not want to type *java.sql* all the time in front of java.sql classnames, you can put *import java.sql.*;* at the top of your code. We assume in the future examples that you have done this.

To be able to send a query, you first must instantiate a Statement object:

```
Statement st = con.createStatement();
```

Once a statement is created, you can use it to run two types of queries: the ones that return a result set, and the ones that do not. To run a query that returns a result set, use executeQuery(), and for the query that does not return a result set, use executeUpdate().

executeQuery() returns an object of type ResultSet. You can iterate through the result set using the method next() and then access individual fields of the current row with the getXXX() method—for example, getString(), getFloat(), getInt(). When the result set has been fully traversed, next() returns FALSE.

Another way to execute a query is to use the PreparedStatement object, which is a subclass of Statement. It can be obtained in the following way:

```
PreparedStatement st = con.prepareStatement(query);
```

In the query passed to prepareStatement(), you can use placeholders (?) in place of column values. The placeholder should be set to the actual value through a call to the appropriate setXXX() method of the PreparedStatement object. Note that the placeholder indexes start with 1, not with 0. After the placeholder values have been set, you can call executeQuery() or executeUpdate(), just as in the case of Statement. The advantage of using prepared statements is that the strings will be properly quoted and escaped by the driver.

If the information, such as the number of columns in the result set or their names, is not known in advance, you can obtain it by retrieving the ResultSetMetaData object associated with the ResultSet. To retrieve the object, use the getMetaData() method of the ResultSet. You can then use getColumnCount(), getColumnName(), and other methods to access a wide variety of information.

Note that all of the database access methods we have discussed can throw SQLException.

Sample Code

Listing 11.1 is a simple command-line benchmark that illustrates the basic elements of interacting with MySQL in Java, and as a bonus gives you some idea of performance of the MySQL server in combination with the Java client. We create a table with one integer column, which is also going to be the primary key, plus the specified number of string columns of type CHAR(10). We populate it with dummy data. Then we perform the specified number of SELECT queries that will select one record based on the random value of the primary key. To reduce input/output, we select only one column from the entire record, which is physically located right in the middle of it.

You can download the sample code from the book's Web site. The source is in Benchmark.java. The class files are also provided: Benchmark.class and MySQLClient.class. To compile, execute

```
javac Benchmark.java
```

To run:

```
java Benchmark num_rows num_cols num_queries
```

For example:

```
java Benchmark 1000 20 2000
```

creates a table with 1000 rows and 20 columns in addition to the primary key, and runs 2000 queries. Now let's take a look the sample code.

```
/* Import directives for convenience */
import java.sql.*;
import java.util.Random;

/* Convenience client wrapper class */
class MySQLClient
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
{
  /*
     We encapsulate Connection, Statement, PreparedStatement, and
     ResultSet under one hood. ResultSet and PreparedStatement are
     made public as we will want to give the caller a lot of
     flexibility in operating on it.
   */
  private Connection con;
  private Statement st;
  public  PreparedStatement prepSt = null;
  public ResultSet rs;

  /* Constructor. Accepts the JDBC URL string */
  public MySQLClient(String url)
    {
      try
      {
           /* first, connect to the database */
           con = DriverManager.getConnection(url);
           /* Initialize the Statement object */
        st = con.createStatement();
      }
      catch(SQLException e)
      {
           /* On error in this application we always abort with
              a context-specific message.
           */
           Benchmark.die("Error connecting: " + e.getMessage());
      }
    }

  /*
     We have two sister methods - safeReadQuery(), and
     safeWriteQuery(). They both execute a query and handle errors by
     aborting with a message. The former executes a read query that
     produces a result set, and stores the result set in the rs class
     member. The latter executes a write
     query, which produces no result set.
  */
  public void safeReadQuery(String query)
    {
      try
      {
        rs = st.executeQuery(query);
      }
      catch (SQLException e)
      {
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
            Benchmark.die("Error running query '" + query + "': " +
                          e.getMessage());
        }
      }

    public void safeWriteQuery(String query)
      {
        try
        {
          st.executeUpdate(query);
        }
        catch (SQLException e)
        {
            Benchmark.die("Error running query '" + query + "': " +
                          e.getMessage());
        }
      }

    /*
      JDBC allows the programmer to use prepared queries. At this point,
      MySQL does not support prepared queries, but using the semantics
      is still beneficial since we get automatic quoting and escaping of
      strings.

      We create two wrappers, safePrepareQuery() and safeRunPrepared(),
      which are to be used in conjunction - first safePrepareQuery()
      with the query, and then after setting the column values with one
      of the setXXX() methods of PreparedStatement, a call to
      safeRunPrepared(). In our convention, the prefix "safe" in the
      method name indicates that we catch the exception ourselves and
      deal with it. The caller does not need to worry about
      handling any exceptions.
     */
    public void safePrepareQuery(String query)
      {
        try
        {
            prepSt = con.prepareStatement(query);
        }
        catch (SQLException e)
        {
            Benchmark.die("Error preparing query '" + query + "': " +
                          e.getMessage());
        }
      }

    public void safeRunPrepared()
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
    {
      try
      {
           prepSt.execute();
      }
      catch (SQLException e)
      {
           Benchmark.die("Error running prepared query : " +
                         e.getMessage());
      }

    }

  /* Get number of columns in the result set from the last query */
  public int getNumCols() throws SQLException
    {
      return rs.getMetaData().getColumnCount();
    }

  /*
     Get the name of the column in the result set at the given
     sequential order index
  */
  public String getColName(int colInd) throws SQLException
    {
      return rs.getMetaData().getColumnName(colInd);
    }
}

/* Main class */
class Benchmark
{
  /*
     Hardcoded values for the connectivity arguments. In a real-life
     application those would be read from a configuration file or from
     the user.
  */
  private static String user = "root", password = "", host = "localhost",
    db = "test";

  /* Convenience emergency exit method */
  public static void die(String msg)
    {
      System.err.println("Fatal error: " + msg);
      System.exit(1);
    }
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
/*
   The name of the method should be self-explanatory. It loads the
   MySQL JDBC driver.
*/
public static void loadDriver()
  {
    try
    {
     Class.forName("com.mysql.jdbc.Driver").newInstance();
    }
    catch(Exception e)
    {
         die("Error loading driver: " + e.getMessage());
    }
  }

/*
   Create a table named t1 with the first column as an integer and
   a primary key, and the rest of the columns strings of type
   CHAR(10). The number of additional columns is determined by the
   numCols argument. Populate the table with numRows rows of
   generated data. When finished, run SHOW TABLE STATUS LIKE 't1'
   and print the results.
*/
public static void makeTable(MySQLClient c, int numRows, int numCols)
  throws SQLException
  {
    /*
          First, clear the path. Remove a stale table t1 if
          it exists.
    */
    c.safeWriteQuery("DROP TABLE IF EXISTS t1");

    /* Initializations to prepare for constructing the queries */
    String query = "CREATE TABLE t1(id INT NOT NULL PRIMARY KEY";
    String endInsert = "", startInsert = "INSERT INTO t1 VALUES(?";
    int i;

    /*
          Construct the CREATE query, and the common end of all
          insert queries while we are going through the loop.
    */
    for (i = 0; i < numCols; i++)
    {
         query += "," + "s" + String.valueOf(i) + " CHAR(10) NOT NULL";
         endInsert += ",?";
    }
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
        query += ")";
        endInsert += ")";

        /* This executes the CREATE */
        c.safeWriteQuery(query);

        /* Start the timer */
        long start = System.currentTimeMillis();
        /* Prepare the query before the insert loop */
        c.safePrepareQuery(startInsert + endInsert);

        /* Set the constant string values */
        for (i = 0; i < numCols; i++)
        {
             /*
               note that setXXX() functions count the columns starting
               from 1, not from 0.
             */
             c.prepSt.setString(i+2,"abcdefghij");
        }

        /* Execute numRows INSERT queries in a loop to populate the
           table
        */
        for (i = 0; i < numRows; i++)
        {
             c.prepSt.setInt(1,i);
             c.safeRunPrepared();
        }

        /* Stop the timer */
        long runTime = System.currentTimeMillis() - start;

        /* Compute and print out performance data */
        System.out.println(String.valueOf(numRows) +
                                    " rows inserted one at a time in " +
                                    String.valueOf(runTime/1000) + "." +
                                    String.valueOf(runTime%1000) + " s, " +
                                    (runTime > 0 ? String.valueOf((num-
Rows*1000)

/runTime) +
                                     " rows per second" : "") + "\n");

        /*
              Now we examine the table with SHOW TABLE STATUS. This
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
      serves several purposes. It allows us to check if the rows
      we have inserted are really there. We can see how many
      bytes the table and each row are taking. And we can
      provide an example of how to read the results of a query.
*/
c.safeReadQuery("SHOW TABLE STATUS LIKE 't1'");

/*
      Get the number of columns in the result set. See
      MySQLClient.getNumCols() for low-level details.
*/
int numStatusCols = c.getNumCols();
String line = "TABLE STATUS:\n";

/*
      Read column names. See MySQLClient.getColName() for the
      low-level details.
*/
for (i = 1; i <= numStatusCols; i++)
{
     line += c.getColName(i) + "\t";
}

/* Print the tab-delimited line with column names */
System.out.println(line);

/*
      Iterate through the result set.
      MySQLClient.safeReadQuery() stores the result set in the
      rs member. There is actually only one result row. But for
      the sake of example, we do a loop anyway.
*/
while (c.rs.next())
{
     line = "";

     /* Iterate through the columns of the result set, and
        concatenate the string values with \t as the delimiter.
      */
     for (i = 1; i <= numStatusCols; i++)
     {
       line += c.rs.getString(i) + "\t";
     }
     line += "\n";

     /* Output the tab-delimited result line */
     System.out.println(line);
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
      }
    }

  /*
     Run numQueries randomized selects of the type SELECT sN FROM t1
     WHERE id = M, with N being the number of columns divided in half
     to hit the middle column, and M a random number between 0 and
     numRows-1. Time the process and compute performance data.
  */
  public static void runSelects(MySQLClient c, int numRows, int
numCols,
                                  int numQueries) throws
SQLException
    {
      int i;

      /* Initialize the common query prefix */
      String queryStart = "SELECT s" + String.valueOf(numCols/2) +
           " FROM t1 WHERE id = ";

      /* Instantiate the random number generator object. */
      Random r = new Random();

      /* Start the timer*/
      long start = System.currentTimeMillis();

      /* Now run generated queries in a loop randomizing the key
         value.*/
      for (i = 0; i < numQueries; i++)
      {
           c.safeReadQuery(queryStart + String.valueOf(Math.abs(r.nextInt())
                                                                  % num-
Rows));
           while (c.rs.next())
           {
             /* Empty - just retrieve and dispose of the result */
           }
      }

      /* Stop the timer */
      long runTime = System.currentTimeMillis() - start;

      /* Compute and print performance data */
      System.out.println(String.valueOf(numQueries) +
                                " selects in " +
                                String.valueOf(runTime/1000) + "." +
                                String.valueOf(runTime%1000) + " s,
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
                                    " + (runTime > 0 ?
                                     String.valueOf((numQueries*1000)/runTime)
+
                                     " queries per second" : "") +
"\n");

    }

  /* The name speaks for itself. This is the main method in the main
     class.*/
  public static void main(String[] args)
    {
      /* Again, the name speaks for itself. We load the JDBC MySQL
         driver. */
      loadDriver();

      /* Construct the JDBC URL. */
      String url = "jdbc:mysql://" + host + "/" + db + "?user=" + user
 + "&password=" + password;

      /* Parse the command-line arguments */
      int numRows = 0, numCols = 0, numQueries = 0;
      if (args.length != 3)
           die("Usage: java Benchmark num_rows num_cols num_queries");
      try
      {
           numRows = Integer.parseInt(args[0]);
           numCols = Integer.parseInt(args[1]);
           numQueries = Integer.parseInt(args[2]);
      }
      catch(NumberFormatException e)
      {
           die("Arguments must be numeric");
      }

      /*
            Instantiate the Client object. The constructor establishes
            the connection.
      */
      MySQLClient c = new MySQLClient(url);

      /*
            Now we actually do the job - create the table and
            run selects on it, as the names of the methods suggest.
      */
      try
      {
```

Listing 11.1 Source code of Benchmark.java. (continues)

```
            makeTable(c,numRows,numCols);
               runSelects(c,numRows,numCols,numQueries);
            }
            catch (Exception e)
            {
               die("Uncategorized exception: " + e.getMessage());
            }
         }
      }
```

Listing 11.1 Source code of Benchmark.java. (continued)

A run of the sample application with 1000 rows, 10 columns, and 1000 selects (Java Benchmark 1000 10 1000) with Sun JDK 1.3.1 produces the following output on my desktop (a Pentium III 500 with 256MB of RAM, running Linux 2.4.19):

```
1000 rows inserted one at a time in 1.696 s, 589 rows per second
*
*
*
1000 selects in 1.917 s, 521 queries per second
```

CHAPTER 12

Writing the Client for Optimal Performance

In this chapter, we discuss how to write MySQL client code efficiently. We have examined some related topics in Chapters 6 and 7; in this chapter, we look at caching techniques, replication-aware code, ways to improve write-dominant applications, methods of reducing network I/O, and query optimization. In the query optimization section, we learn how the MySQL optimizer works.

Query Caching

A frequent cause of inefficiency in database applications is that the same SELECT query is being run multiple times, and each time it produces the same result because the tables have not changed. One easy way to address this problem is to use server query caching. The server will cache the results of all SELECT queries. The next time the same query is run, if the tables involved in the query have not been modified, the result is read from the cache and not from the tables. When at least one of the tables involved in the query is modified, the cached result is invalidated. Thus, performance is improved, and the optimization is transparent to the user. This feature is available in the 4.0 MySQL branch starting in version 4.0.3.

To enable the query cache, you need to set query_cache_size to a non-zero value (--set-variable query_cache_size=16MB, for example). This does mean, however, that you would have to use a relatively new feature along with quite a bit of new code that is in the 4.0 branch. The code in the query cache has proven

itself reasonably stable, though, so you may consider this an option as long as you test your application thoroughly to make sure it does not hit any unknown bugs (something you should do anyway, even with a stable release of MySQL). By the time you read this book, it is very likely that the 4.0 branch will have reached stable status.

However, having to depend on the server query cache is not the best option for improving performance itself, although it may save you some development time, especially if you have already written the application. The problem is that you still have to contact the server and have it do the work that it really does not have to do, such as reading network packets, parsing the query, checking if it is stored in the query cache, checking if the tables have been modified, and sending the stored result back to the client. A better alternative is to perform caching on the client inside your application logic. There are several caching techniques you can use in your code:

- When retrieving query results, ask yourself if you will need them again in another routine. If you will, save the result for later access. An elegant way to do it in C is to not call mysql_free_result() on the pointer returned from mysql_store_result() immediately, but to save it in a global variable for later access; then when it is needed again, call mysql_data_seek() to rewind to the first row, and iterate through it another time. A similar method will work in PHP. In Perl, you can use $dbh->fetchrow_arrayref(), although it is not as elegant as with C or PHP because it will make an unneeded copy of the result.
- If you are generating an HTML page dynamically, you can have a static page instead that is always regenerated when tables that it depends on are updated. The advantage of this approach is that even if the database server goes down, you will still have something to show, and it will be current, too!
- For non-Web clients, you can identify logical blocks of information that depend on a set of tables, and cache/regenerate those blocks only if the table has been updated. To check the timestamp on a table, you can use SHOW TABLE STATUS LIKE 'tbl_name' if the updating agent is not capable of notifying clients that the update has happened otherwise.

If the application performs intensive frequent reads and relatively infrequent writes, proper query caching can significantly improve performance and help greatly reduce hardware costs.

Replication-Aware Code

An application that performs a high volume of reads but relatively few writes can benefit in terms of performance from a replicated setup. All updates will go to the

master, while non-time-critical selects can go to slaves. This method allows you to scale your application with a cluster of inexpensive servers to the kind of performance that might be difficult to achieve even with an expensive mainframe.

However, to take advantage of this setup in MySQL 3.23, the client has to be aware of the set of replication servers. The connect function must be able to establish two connections: one to the master and the other to one of the slaves, based on some evenly distributed algorithm (random pick, hash on process id, etc.). All queries will go through three query functions: safe_query_write(), safe_query_read(), and safe_query_critical_read(). You call safe_query_write() for any queries that modify the data, safe_query_read() for any query that can read data that is slightly behind (a slave may be a few queries behind the master), and safe_critical_read() for reads that need the most current data.

MySQL AB plans to add a proxy in 4.0 or 4.1 that will look at the query and intelligently decide if it should go to the master or to the slave. This allows old code to take advantage of the replicated setup. For more information on replication, see Chapter 16.

Improving Write-Dominant Applications

One of the challenges in scaling a MySQL application is that while it is relatively easy to scale read performance through caching and replication, those techniques will not improve performance when writes are dominant. Let's discuss a couple of ways to speed up write-intensive applications:

- Combine several inserts into one. For example, instead of INSERT INTO tbl VALUES (1); INSERT INTO tbl VALUES (2); INSERT INTO tbl VALUES (3), you can do INSERT INTO tbl VALUES (1),(2),(3);.
- If the number of records you insert at once is significant (more than 100 or so), it is advisable to save the data in a file (e.g., tab-delimited columns) and then use LOAD DATA INFILE (if the file can be placed directly on the server) or LOAD LOCAL DATA INFILE otherwise to load the data. LOAD DATA INFILE is the fastest possible way to get a large chunk of data into a table, and is worth the overhead of creating a temporary file.
- Do all you can to reduce the length of a record by choosing efficient data types for each field. If you are using sizeable BLOBs (over 1K), consider whether it would be possible to store them separately on a regular file system and store references to them instead in the database. MySQL does not perform very well with BLOBs; it does not have a separate BLOB space for storage, and the network and storage code in the server is written with the assumption that the inserted record is short and that making one extra copy of a field does not hurt.

- Eliminate unnecessary keys, and reduce the size of the ones you are going to keep if possible (e.g., by indexing only a prefix of a string instead of the whole string).
- If possible, write your application with the assumption that the data can be logically divided into groups. For example, if you are logging some information based on an IP address, you can hash on the last 4 bits of the address and create 16 different groups. Then you can put each group of records on its own server. The statistical and other queries that need to examine the entire data set will poll multiple servers. This will add some development overhead for your application because you will need to merge the data coming back. On the other hand, it will give you the advantage of being able to run the query in parallel on several servers.
- For applications that collect randomly arriving data one piece at a time (e.g., if you are logging IP packets, ISP dial-up information, or cell-phone calls), you can greatly improve performance by pooling the data into batches and then using LOAD DATA INFILE. This can be done by having the clients connect to a simple proxy server instead of MySQL directly, which will read the data from the client, add it to the queue, and periodically flush the queue, sending it to the MySQL server in a batch.

Reducing Network I/O

Unnecessary network I/O overhead is obviously detrimental to application performance. Not only does it tie up network resources and cause delays on the client, it also increases the amount of time locks are being held on the server. In addition, it ties up the server CPU since it pipes the data to the client it should not have to pipe. Two basic concepts are behind achieving efficient network I/O:

- Read only what you need (in other words, if you read it, use it; if you do not use it, do not read it).
- Do not retrieve data into the client just to process it. Let the server do the work as much as possible.

Excessive network I/O is often caused by the client requesting too many rows or columns of data back from a query. Let's look at an example where the client is reading from a table with 100 columns, each an average of 10 bytes in length. Once it receives the data, it accesses only 5 of those 100 columns. Yet the client performs SELECT * FROM tbl instead of SELECT c1,c2,c3,c4,c5 FROM tbl. Or, on the other hand, the client executes SELECT * FROM tbl WHERE c1 > 'start' AND c2 < 'end' and reads the first record of the result, discarding the rest of the records. A more efficient approach would be to use SELECT * FROM tbl

WHERE c1 > 'start' AND c2 < 'end' LIMIT 1, which will give the client the record it needs.

Other times, the client actually uses what it reads, but it is still not efficient because the server is capable of doing the work to get the end result the client is actually after. Let's consider a simple example in PHP (see Listing 12.1).

```
($res=mysql_query("SELECT c1,c2,c3,c4,c5 FROM tbl WHERE $constr")) ||
die("Error running query");
echo("<table border=1><tr><td>c1</td><td>total</td></tr>");
while ($r=mysql_fetch_object($res))
{
   echo("<tr><td>$r->c1</td><td>".($r->c2+$r->c3+$r->c4+$r-
>c5)."</td></tr>");
}
echo("</table");
```

Listing 12.1 Inefficient code can lead to higher network I/O.

The code in Listing 12.1 can be improved by moving the addition to the server, as shown in Listing 12.2. We now read only two columns from the server instead of five, which reduces network traffic.

```
$res=mysql_query("SELECT c1,(c2+c3+c4+c5) as total FROM tbl WHERE
$constr")) || die("Error running query");
echo("<table border=1><tr><td>c1</td><td>total</td></tr>");
while ($r=mysql_fetch_object($res))
{
   echo("<tr><td>$r->c1</td><td>$r->total)</td></tr>");
}
echo("</table");
```

Listing 12.2 Example of reducing network I/O by letting the server do the work.

Understanding the Optimizer

To master the art of writing efficient queries, you must learn how the MySQL optimizer thinks. In a way, you must become one with it, just as experienced cyclists become one with their bike or experienced musicians become one with their instrument. Getting there takes a lot of time, practice, research, and more practice. In the next few pages, we provide an introduction that we hope will get you started on the right track.

The MySQL optimizer was written mostly by one person: Monty Widenius. It is an expression of Monty's programming genius, and as such is quite efficient and capable. At the same time, however, it inherited some of Monty's quirks and prejudices, and at times exhibits rather odd behavior that some programmers may find annoying. Despite these quirks, the optimizer gets the job done very well if you take the time to understand how it works.

The goal of any optimizer is to improve application performance by examining the minimum possible amount of data necessary to return accurate results for any query. The most common way to accomplish this is to perform key reads as opposed to table scans. When a key read is being performed, we want to use the one that will require the least amount of I/O to retrieve the data we need.

The MySQL optimizer can use only one key per table instance in a join (even if you are selecting from just one table, the optimizer still considers this a join, albeit a simplified one). It distinguishes among four methods of key lookup: const, ref, range, and fulltext. A const lookup involves reading one record based on a known value of a unique key. It is the most efficient lookup method. ref involves two situations: a lookup on a key when only some prefix parts are known but not the entire value of the key, and lookups on a non-unique key. In a const lookup, we always read only one record, while a ref lookup may need to read additional ones. A range lookup is used when we know the maximum and minimum possible values of a key (the range of values) or a set of such ranges. The efficiency of a ref lookup greatly depends on key distribution. full-text lookups are used when you have a full-text key on a column and are using the MATCH AGAINST() operator.

The optimizer is capable of examining different possibilities for keys and will choose the method that requires, in its judgment, the least number of records. In some cases, it can make a mistake in estimating how many rows a certain key would require it to examine, and for that reason it can choose the wrong key. If that happens, you can tell it to use a different key by including the *USE KEY(key_name)* syntax in the query.

If the key interval estimated number of records exceeds 30 percent of the table, a full scan will be preferred for MyISAM tables. The reason for this decision is that it takes longer to read one key than one record from the data file. This is not the case for InnoDB tables, however, because a full scan is really just a traversal of the primary key (if no primary key or unique key is specified when the table is created, a surrogate key will be created). It would definitely not be faster than partially traversing itself, and is not likely to be faster than traversing a portion of a secondary key.

MyISAM and InnoDB tables use the B-tree index. HEAP (in-memory) tables use hash index. Therefore, HEAP tables are very efficient at looking up records

based on the value of the entire key, especially if the key is unique. However, they are not able to read the next key in sequence, and this makes it impossible for them to use the range optimization or to be able to retrieve the records based on a key prefix.

The optimizer is capable of figuring out from the WHERE clause that it can use a key if the columns comprising it are compared directly with a constant expression or with the value from another table in a join. For example, if a query has WHERE a = 4+9, the optimizer will be able to use a key on a, but when the same condition is written as WHERE a–4 = 9, the key would not be used. It is, therefore, important to have a habit of writing comparison expressions with a simple left side. The optimizer looks at =, >, >=, <=, LIKE, AND, OR, IN, and MATCH AGAINST operators in the WHERE clause when deciding which keys could possibly be used to resolve the query.

In some cases, MySQL will "cheat" by noticing something special in the WHERE clause. For examine, MyISAM and HEAP tables keep track of the total number of records. Therefore, SELECT count(*) FROM tbl can be answered by simply looking up the record count in the table descriptor structure. A query like SELECT * FROM tbl WHERE a = 4 AND a = 5 can be answered immediately without ever looking at the table data: a cannot be 4 and 5 at the same time; therefore, the result is always an empty set. Suppose we have a key on a in a MyISAM or InnoDB table. During initialization, the optimizer will look up the highest and the lowest values of the key. If it notices that the key value requested is out of range, it will immediately return an empty set.

Remembering that only one key can be used per table instance is important. A common mistake that can significantly reflect on performance is repeatedly executing something like SELECT * FROM tbl WHERE key1 = 'val1' OR key2 = 'val1', assuming that key1 and key2 are keyed columns in the table tbl. In this situation, MySQL will not be able to use either one of the keys. It is much better to split the query into two in this case: SELECT * FROM tbl WHERE key1 = 'val1', followed by SELECT * FROM tbl WHERE key2 = 'val2'.

The optimizer does a decent job with joins, but it does require you to think carefully when writing them. You have to have the proper keys in place. No temporary keys will be created to optimize a join—only the existing ones will be used. MySQL does not perform a hash-join. It simply examines each table to be joined to see how many records it would have to retrieve from each, sequences them in smaller-first order, and then begins to read records according to that order. The first record is read for the first table, then the first for the second, up until the last record. All records are read from the last table that matches the conditions. Then the second record is read in the next-to-last table, and all the matching records are read from the last table again.

Each time the optimizer finishes the iteration through a table, it backtracks to the previous one in the join order, moves on to the next record, and repeats the iterative process. These steps are repeated until we have iterated through the first table. Thus, we end up examining the product of the number of candidate rows for each table.

As you can see, this process will be very fast if a key can always be used in all tables to look up a record and there are not very many possibilities to examine in each table. However, it could be a performance disaster if all the tables are large and have to be fully scanned.

To see what the optimizer is doing, you can look at the output of EXPLAIN; you can also run a query on the server (when it has no other activity) and use the output from SHOW STATUS to track the differences in the optimizer statistic variables. Let's examine a few examples that will illustrate how this can be done to understand how efficient a certain query might be. For more in-depth information on EXPLAIN and SHOW STATUS, please refer to Chapter 15, where we document the output of both in detail.

To facilitate our optimizer strategy research, I have written a special command-line utility called query-wizard. Both the source and the executables are available on this book's Web site (www.wiley.com/compbooks/pachev). When performing your own studies, you could either use query-wizard as is without modifications, extend or customize it to fit your needs, or perhaps use it as an inspiration to write your own tool. The essence of the research is to run EXPLAIN on your query to have the optimizer tell you what it is going to do; then run the query and see how the optimizer statistics have changed to see what it has actually done.

Here is a listing of the program's options (obtained by running ./
query-wizard -?):

```
Usage: query-wizard [options] file
Options:
-x -- EXPLAIN
-s -- report Handler_ variable changes
-r -- run the query N times and time it, requires an argument
-a -- equivalent of -x -s -r 1
-u -- MySQL user
-h -- MySQL host
-S -- MySQL socket
-d -- MySQL database
-p -- MySQL password
-f -- continue on error
-o -- output file name
-q -- maximum query length, default 65535

-? -- show this message and exit
```

As evident from the above help message, query-wizard is capable of reading an input file of queries and running them against a specified MySQL server. Depending on the supplied options, it can EXPLAIN a query, run it one or more times and time it, and collect optimizer statistics under the assumption that no other queries were running at the same time.

The input file will be standard input since no file is specified as the file argument. Either you can just type up a series of semicolon-terminated queries you would like to explore, or you can feed it the contents of the slow log. SELECT queries are processed directly. CREATE TABLE ... SELECT or INSERT INTO ... SELECT will be processed by keeping only the SELECT part. A DELETE or an UPDATE query will be converted to an equivalent SELECT so that EXPLAIN could be used on it (EXPLAIN does not work on non-SELECT queries). USE queries will result in changing the active database. Other queries will be simply ignored.

The output, depending on the command-line options, will include the results of EXPLAIN, the number of rows the optimizer believes it will have to examine, the number of returned records, query timing information, and the change in the optimizer statistic counters over the runtime of the query.

Let's begin by creating a table and populating it with some dummy data. We execute the set of SQL statements shown in Listing 12.3. In this example, we assume that we have a dictionary file in /usr/dict/words, which will be the case on many Unix systems. If you do not have a dictionary file on your system, there are many places on the Internet where you can download one—for example, www.puzzlers.org/secure/wordlists/dictinfo.html.

```
DROP TABLE IF EXISTS employee;
CREATE TABLE employee
 (
   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   ssn CHAR(9) NOT NULL,
   fname VARCHAR(20) NOT NULL,
   lname VARCHAR(20) NOT NULL,
   gender ENUM('Male','Female') NOT NULL,
   occupation ENUM('Engineer','Sales Rep','Accountant','Manager'),
   UNIQUE KEY(ssn),
   KEY(lname(5),fname(5)),
   KEY(gender,occupation),
   KEY(occupation,gender)
 );
DROP TABLE IF EXISTS dict;
```

Listing 12.3 SQL script to populate the sample table. (continues)

```
CREATE TABLE dict
 (
   id INT NOT NULL AUTO_INCREMENT  PRIMARY KEY,
   word VARCHAR(20) NOT NULL ,
   KEY(word)
 );
LOAD DATA INFILE '/usr/dict/words'  INTO TABLE dict (word);
SELECT (@max_id:=MAX(id)) FROM dict;
INSERT INTO employee (ssn,fname,lname,gender,occupation)
 SELECT 999999999-a.id,a.word,b.word,
 IF(SUBSTRING(a.word,LENGTH(a.word),1) = 'a'
  OR SUBSTRING(a.word,LENGTH(a.word),1) = 'e'
  OR SUBSTRING(a.word,LENGTH(a.word),1) = 'y'
  OR SUBSTRING(a.word,LENGTH(a.word)-2,3) = 'ing'
 ,
'Female','Male'),
 ELT((4*RAND()) % 4 + 1 , 'Engineer','Sales Rep',
'Accountant','Manager')
FROM dict a, dict b WHERE a.id = @max_id - b.id + 1;
DROP TABLE dict;
```

Listing 12.3 SQL script to populate the sample table. (continued)

Although the sample sequence of SQL statements in Listing 12.3 may look rather complex, the idea is very simple: We load a dictionary into a temporary table; then join it with itself going in the reverse order of ids. The join gives us a set of senseless (but on occasion rather humorous) combinations of first and last name. As we read the results of the join, we randomly pick the occupation field value for the field, set SSN to 999999999-id, and "guess" the gender based on the word ending. This gives us a reasonably sized table of employee records that appears realistic enough to be somewhat entertaining if you look at individual records and try to imagine what kind of person could possibly match it.

Now let's run some queries. We will put them in a file called queries.txt (see Listing 12.4) and run them through query-wizard.

```
USE test;
SELECT COUNT(*),occupation FROM employee GROUP BY occupation;
SELECT COUNT(*),occupation FROM employee GROUP BY CONCAT(occupation);
SELECT fname,lname FROM employee WHERE ssn = '999999333';
SELECT fname,lname FROM employee WHERE fname = 'Hope';
SELECT fname,lname FROM employee WHERE lname = 'Hope';
SELECT fname,lname FROM employee WHERE ssn = '999999333' OR lname = 'Hope';
```

Listing 12.4 Sample queries. (continues)

```
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555';
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555' ORDER BY ssn;
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555' ORDER BY CONCAT(ssn);
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555' ORDER BY lname DESC;

SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P';
SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P'
  ORDER BY lname,fname;
SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P'
  ORDER BY fname,lname;
```

Listing 12.4 Sample queries. (continued)

Now let's run query-wizard -a queries.txt and look at the pieces of output for each query. Here we have just changed to the test database where we have our employee table:

```
# Query, 8 bytes
USE test
```

Listing 12.5 contains a GROUP BY query on a key (occupation). The query was resolved by reading the very first key entry (see Handler_read_first) and then sequentially traversing the key (see Handler_read_next). MySQL breezed through 45,407 key entries in 0.235 seconds on my Pentium 500 (256MB of RAM) Linux desktop. Note that Using index in Extra in the EXPLAIN output tells us that the data file was not accessed.

```
# Query, 60 bytes
SELECT COUNT(*),occupation FROM employee GROUP BY occupation
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     index     NULL     occupation     3     NULL     45407
# Using index
# Estimated rows to examine: 45407
# Timing: run 1 times in 0.235 s, 0.235 s per query
# Returned 4 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_first: 1
# Handler_read_next: 45407
# Select_scan: 1
```

Listing 12.5 Optimization of GROUP BY on a key.

```
# Query, 68 bytes
SELECT COUNT(*),occupation FROM employee GROUP BY CONCAT(occupation)
# Results of EXPLAIN:
# table                  type                 possible_keys     key
# key_len     ref     rows     Extra
# employee     index                NULL                gender     3
# NULL     45407     Using index; Using temporary
# Estimated rows to examine: 45407
# Timing: run 1 times in 0.430 s, 0.430 s per query
# Returned 4 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_first: 1
# Handler_read_key: 45407
# Handler_read_next: 45407
# Handler_read_rnd: 4
# Handler_read_rnd_next: 5
# Handler_update: 45403
# Handler_write: 4
# Select_scan: 1
# Sort_rows: 4
# Sort_scan: 1
```

Listing 12.6 Unoptimized GROUP BY.

In Listing 12.6, we make it a bit more difficult for MySQL. We confuse it on purpose so it will not be able to use the occupation key by making it group by CONCAT(occupation) instead of occupation. Of course, CONCAT(occupation) is always equal to occupation, but the optimizer does not know that. The query is resolved by first creating a temporary table for the GROUP BY containing the columns that will be present in the result with a unique key on the GROUP BY expression (we can tell by Using temporary in Extra).

Then, we iterate through the original table; for each record, we check if a record already exists with the corresponding key in the temporary table. If there is one, we update it; otherwise, we insert a new one (see Handler_update, Handler_write, and Handler_read_key). Then we read the records in the order of the GROUP BY expression. To be able to do that, we have to use the filesort algorithm--we cannot just iterate through the key. The table is of type HEAP and supports only hash keys. A hash key cannot be traversed in the key order; the only thing you can do is look up a given value. We know that the sorting has taken place by looking at the values of Handler_read_rnd and Sort_rows.

Despite the extra complexity, the query still takes only 0.43 seconds. Having only a few distinct values of the GROUP BY expression helps quite a bit because it keeps the temporary table very small. Most of the overhead actually occurs from having to jump through the hoops of a formal key read, record

update, or record insert, as opposed to just executing the record expression evaluation routine in the previous query. If we had significantly more distinct values for the GROUP BY expression, the overhead of sorting the temporary table would have been noticeable, which would have added significantly to the execution time.

The kind of query shown in Listing 12.7 is every DBA's dream. We just read one record on a unique key; it takes less than 0.001 of a second to execute. Running it in a loop (by giving -r 2000 to query-wizard) indicated that the execution time was 0.0005s on my system, or in other words, it could do 2000 of those per second on just one CPU.

```
# Query, 56 bytes
SELECT fname,lname FROM employee WHERE ssn = '999999333'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     const     ssn     ssn     9     const     1
# Estimated rows to examine: 1
# Timing: run 1 times in 0.000 s, 0.000 s per query
# Returned 1 row
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
```

Listing 12.7 Lookup on a primary key.

Listing 12.8 shows what happens when we cannot use a key . fname is a part of a key (lname(5),fname(5)), but it does not constitute a prefix. Therefore, the key cannot be used, and the optimizer has to do a full scan. Instead of 0.0005 seconds, the query now takes 0.187 seconds, about 300 times longer. The larger the table, the bigger the gap between a key lookup and a scan.

```
# Query, 53 bytes
SELECT fname,lname FROM employee WHERE fname = 'Hope'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     ALL     NULL     NULL     NULL     NULL     45407
# where used
# Estimated rows to examine: 45407
# Timing: run 1 times in 0.187 s, 0.187 s per query
# Returned 1 row
# Optimizer statistics for the query (assuming no other running):
# Handler_read_rnd_next: 45408
# Select_scan: 1
```

Listing 12.8 Unoptimized record lookup.

At first glance, Listing 12.9 contains a query similar to the one in Listing 12.8, but with one very important difference: We are using lname instead of fname, which allows us to use the key prefix. The query now has to perform only two key reads (for two independent reasons—the key is not unique, and we do not have the full value of the key). It now takes less than 0.001 seconds (a more precise test shows the value is 0.0005). We are, however, reminded that it is a good idea to explicitly state that a key is unique if we do not expect any duplicate values in it, not only for documentation purposes but also for performance.

```
# Query, 53 bytes
SELECT fname,lname FROM employee WHERE lname = 'Hope'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     ref     lname     lname     5     const     1     where
# used
# Estimated rows to examine: 1
# Timing: run 1 times in 0.000 s, 0.000 s per query
# Returned 1 row
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 1
```

Listing 12.9 Record lookup on the key prefix.

```
# Query, 74 bytes
SELECT fname,lname FROM employee WHERE ssn = '999999333' OR lname =
'Hope'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     ALL     ssn,lname     NULL     NULL     NULL     45407
# where used
# Estimated rows to examine: 45407
# Timing: run 1 times in 0.209 s, 0.209 s per query
# Returned 2 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_rnd_next: 45408
# Select_scan: 1
```

Listing 12.10 Record lookup with OR over two different keys.

The record lookup in Listing 12.10 is very interesting. Each individual expression ORed in the WHERE clause is using a key reference. In fact, we have already run a query for each of those expressions with a great degree of success—both completed in about 0.0005 seconds. However, doing an OR causes

terrible results. We now have to scan the entire table and end up with a whopping 0.209-second CPU hog.

The problem is that MySQL cannot use more than one key to read records from a table. The workaround is to execute two queries. If you absolutely need to read the combined result, create a temporary table, select into it twice, select from the temporary table, and then drop it. Another alternative is to use the UNION operator available in MySQL 4.0.

In Listing 12.11, we try a range query on a key. As you would expect, the key is read from the start of the range; then we iterate from that point up to the end of the range. This results in a total of 6221 records in 0.412 seconds.

```
# Query, 83 bytes
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     ssn     ssn     9     NULL     5294     where
# used
# Estimated rows to examine: 5294
# Timing: run 1 times in 0.412 s, 0.412 s per query
# Returned 6221 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 6221
# Select_range: 1
```

Listing 12.11 Lookup on key range.

Listing 12.12 contains a query similar to the one in Listing 12.11, but we run through in ORDER BY. Fortunately, ORDER BY uses the same key that is used for the range lookup, so there is really nothing different compared to the previous query. ORDER BY does not change anything because the optimizer notices that by simply using the range key the records will be read in the correct order. And the execution time is about the same.

```
# Query, 96 bytes
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555' ORDER BY ssn
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
```

Listing 12.12 Lookup on key range with optimized ORDER BY. (continues)

```
# employee     range     ssn     ssn     9     NULL     5294     where
# used
# Estimated rows to examine: 5294
# Timing: run 1 times in 0.417 s, 0.417 s per query
# Returned 6221 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 6221
# Select_range: 1
```

Listing 12.12 Lookup on key range with optimized ORDER BY. (continued)

```
# Query, 104 bytes
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
 ssn < '999999555' ORDER BY CONCAT(ssn)
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     ssn     ssn     9     NULL     5294     where
# used; Using filesort
# Estimated rows to examine: 5294
# Timing: run 1 times in 0.430 s, 0.430 s per query
# Returned 6221 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 6221
# Handler_read_rnd: 6221
# Select_range: 1
# Sort_range: 1
# Sort_rows: 6221
```

Listing 12.13 Lookup on key range with unoptimized ORDER BY.

In Listing 12.13, we examine what looks like a really nasty oddball we've thrown at the optimizer. CONCAT() obscures the query, preventing the optimizer from realizing it can use the key. This adds the overhead of filesort. However, since the result of the sort fits into the sort buffer (we can tell this by the absence of Sort_merge_passes in the output of query-wizard, which means that there were none), it is sorted with a very efficient radix sort algorithm. So the sort overhead is barely noticeable in this case. We would have seen some performance degradation, though, if we had overflowed the sort buffer and forced the use of merge sort, with some intermediate data being stored in temporary files.

```
# Query, 103 bytes
SELECT fname,lname,ssn FROM employee WHERE ssn > '999993333' AND
ssn < '999999555' ORDER BY lname DESC
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     ssn     ssn     9     NULL     5294     where
# used; Using filesort
# Estimated rows to examine: 5294
# Timing: run 1 times in 0.419 s, 0.419 s per query
# Returned 6221 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 6221
# Handler_read_rnd: 6221
# Select_range: 1
# Sort_range: 1
# Sort_rows: 6221
```

Listing 12.14 Unoptimized key range lookup due to ORDER BY DESC.

Again, in Listing 12.14 we make it impossible for the optimizer to use a key (lname is not in the range key used for retrieving the records). To test the efficiency of the sorting algorithm, we also force the sort to be in the order opposite of the ssn ordering, which would make the sort work on a list sorted in reverse order. The reverse order of the original list does not seem to affect performance at all—radix sort works just as efficiently.

```
# Query, 70 bytes
SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P'
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     lname     lname     5     NULL     5186
# where used
# Estimated rows to examine: 5186
# Timing: run 1 times in 0.272 s, 0.272 s per query
# Returned 5602 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 5602
# Select_range: 1
```

Listing 12.15 Lookup on key range to establish timing comparison base.

For a base line timing, let's run the range key lookup query shown in Listing 12.15. Later we will modify it to demonstrate how additional query elements affect performance.

```
# Query, 93 bytes
SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P'
ORDER BY lname,fname
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     lname     lname     5     NULL     5186
# where used; Using filesort
# Estimated rows to examine: 5186
# Timing: run 1 times in 0.420 s, 0.420 s per query
# Returned 5602 rows
# Optimizer statistics for the query (assuming no other running):
# Handler_read_key: 1
# Handler_read_next: 5602
# Handler_read_rnd: 5602
# Select_range: 1
# Sort_range: 1
# Sort_rows: 5602
```

Listing 12.16 ORDER BY when the key uses column parts.

Listing 12.16 is the same as the query in Listing 12.15, except this time we force an ORDER BY on fields in the range key. There is one catch, though: because the key used prefixes (lname(5),fname(5)) instead of full-length (lname,fname), ORDER BY lname,fname cannot be done by traversing the key and we have to resort to post-sorting the result. In this case, we notice a more significant, although not terribly large sorting overhead.

```
# Query, 93 bytes
SELECT fname,lname,ssn FROM employee WHERE lname > 'L' AND lname < 'P'
ORDER BY fname,lname
# Results of EXPLAIN:
# table     type     possible_keys     key     key_len     ref
# rows     Extra
# employee     range     lname     lname     5     NULL     5186
# where used; Using filesort
# Estimated rows to examine: 5186
# Timing: run 1 times in 0.381 s, 0.381 s per query
# Returned 5602 rows
# Optimizer statistics for the query (assuming no other running):
```

Listing 12.17 ORDER BY with key parts in reverse order. (continues)

```
# Handler_read_key: 1
# Handler_read_next: 5602
# Handler_read_rnd: 5602
# Select_range: 1
# Sort_range: 1
# Sort_rows: 5602
```

Listing 12.17 ORDER BY with key parts in reverse order. (continued)

Now, in Listing 12.17 let's make our ORDER BY work in what will turn out to be a reverse order due to the way we have generated the first name and last name values. Performance has actually slightly improved, most likely due to the caching of the previous run.

In the previous examples, we have seen how to perform a quick performance analysis of your queries. Using this technique you could research and understand the basic functionality as well as the deep dark corners of MySQL optimizer, as well as get an idea of what the optimizer will actually do with a particular query and how fast it will run.

CHAPTER 13

Table Design

Proper table design is a critical factor in the cost of maintenance and in the performance of your application. This is true for all relational database servers, not just for MySQL. Although many principles we discuss in this chapter apply not just to MySQL but to all database servers, we examine some issues that pertain to MySQL only.

Column Types and Disk Space Requirements

When you're designing a table, it is important to have a sense of how much disk space will be required to store each row—and how to reduce it. Reducing disk space is important not just for the sake of being able to fit all of your data on the disk. Smaller data sets fit better into disk cache and into the CPU cache, which results in better performance. Therefore, let's examine the process of creating a table and learn the science of estimating row and table size as we go along.

Suppose we need to design a database of employees. We store each employee's first name, last name, social security number, position in the company, hire date, number of annual vacation days, and annual salary. We solve this problem with the following SQL statement:

```
CREATE TABLE employee (
  ssn CHAR(9) NOT NULL PRIMARY KEY,
  fname CHAR(20) NOT NULL,
  lname CHAR(20) NOT NULL,
  hire_date DATE NOT NULL,
  vac_days TINYINT  UNSIGNED NOT NULL,
```

```
  position ENUM ('CEO','CTO','Engineer', 'Manager','Accountant' )
NOT NULL,
  salary  MEDIUMINT UNSIGNED NOT NULL
);
```

The number of bytes required for each column depends on its type and definition. CHAR columns take one byte per character. DATE columns take 3 bytes. TINYINT and ENUM each take 1 byte. MEDIUMINT takes 3 bytes. Table 13.1 lists all column types and storage requirements (taken from the MySQL online manual at www.mysql.com/doc/S/t/Storage_requirements.html).

Table 13.1 Storage Requirements for All Column Types (continues)

COLUMN TYPE	STORAGE REQUIRED
TINYINT	1 byte
SMALLINT	2 bytes
MEDIUMINT	3 bytes
INT	4 bytes
INTEGER	4 bytes
BIGINT	8 bytes
FLOAT(X)	4 if X <= 24 or 8 if 25 <= X <= 53
FLOAT	4 bytes
DOUBLE	8 bytes
DOUBLE PRECISION	8 bytes
REAL	8 bytes
DECIMAL(M,D)	M+2 bytes if D > 0, M+1 bytes if D = 0 (D+2, if M < D)
NUMERIC(M,D)	M+2 bytes if D > 0, M+1 bytes if D = 0 (D+2, if M < D)
DATE	3 bytes
DATETIME	8 bytes
TIMESTAMP	4 bytes
TIME	3 bytes
YEAR	1 byte
CHAR(M)	M bytes, 1 <= M <= 255
VARCHAR(M)	L+1 bytes, where L <= M and 1 <= M <= 255
TINYBLOB	L+1 bytes, where L < 2^8

Table 13.1 Storage Requirements for All Column Types (continued)

COLUMN TYPE	STORAGE REQUIRED
TINYTEXT	L+1 bytes, where L < 2^8
BLOB	L+2 bytes, where L < 2^16
TEXT	L+2 bytes, where L < 2^16
MEDIUMBLOB	L+3 bytes, where L < 2^24
MEDIUMTEXT	L+3 bytes, where L < 2^24
LONGBLOB	L+4 bytes, where L < 2^32
LONGTEXT	L+4 bytes, where L < 2^32
ENUM('*value1*','*value2*',...)	1 or 2 bytes, depending on the number of enumeration values (65535 values maximum)
SET('*value1*','*value2*',...)	1, 2, 3, 4, or 8 bytes, depending on the number of set members (64 members maximum)

Thus, the ssn column takes up 9 bytes; fname and lname each take up 20 bytes; hire_date takes up 3 bytes; vac_days takes 1 byte, position takes 1 byte; and salary takes 3 bytes. This adds up to 57 bytes. In this case, MySQL also reserves 1 byte for internal bookkeeping purposes, which brings the total record length to 58. If you declare all of your columns as NOT NULL, there will always be just 1 byte reserved for bookkeeping per record. Otherwise, for each column that could possibly be NULL, MySQL will need a bit to indicate whether it is NULL or not. If you have more than 7 NULL-capable columns, MySQL will use up another byte, and then for each additional 8 columns there will be another overhead byte reserved.

Note that we have chosen vac_days to be TINYINT UNSIGNED. This assumes that no employee gets more than 255 days of vacation per year. If this assumption is not correct, the type would have to be SMALLINT UNSIGNED, which allows up to 65,535 vacation days.

Also note that we store salary in a MEDIUMINT UNSIGNED column. A more conventional approach is to put money into DECIMAL(10,2). While the conventional approach will make life slightly easier for the application programmer, it wastes disk space—DECIMAL(10,2) would require 12 bytes, while MEDIUMINT UNSIGNED takes only 3. The drawback is that we now have to store the salary value in cents, which means that the application programmer has to multiply the dollar amount by 100 before inserting it and divide the amount by 100 when retrieving it. This tradeoff is worthwhile if the space requirements are critical.

If you are trying to optimize your design for space, MySQL has a built-in data type selection adviser called PROCEDURE ANALYSE(). The usage syntax can be inferred from the following example. Suppose you want to improve the storage types of the employee table. You execute the following query:

```
SELECT * FROM employee PROCEDURE ANALYSE();
```

Or if you want advice only on position and salary columns, you type the following:

```
SELECT position,salary FROM employee PROCEDURE ANALYSE();
```

The last column of the output contains the recommended type for the column. This method needs a table with a lot of rows in order to be effective. On small tables, it tells you that all columns should be ENUM due to the low cardinality of the existing values. It is recommended that you put a large chunk of real-life data in the table before you run this procedure.

Variable-Length versus Fixed-Length Records

In the previous example, we have not used one optimization that could save us a lot of disk space—variable-length records. In MySQL, there are two types of records: fixed length and variable length. On average, a string of characters in a particular record will tend to be quite a bit shorter than the maximum limit allocated for its length. While a fixed-length record will simply pad the bytes not used for storage with spaces, in a variable-length record, all character strings are stored with 1 byte specifying its length, followed by the actual data.

Variable-length records typically give significant space savings, although at the cost of higher CPU overhead. This cost, though, will be compensated for in many instances by performance gains resulting from reduced I/O. As a rule of thumb, if your entire table fits into RAM, you should prefer fixed-length records; if it does not fit into RAM, use variable-length records.

Whether the record will be fixed length or variable length is determined by the combination of column types that comprise the record. All data types are divided into two groups: fixed-length types and variable-length types. TEXT, TYNYTEXT, SMALLTEXT, MEDIUMTEXT, LARGETEXT, BLOB, TYNYBLOB, SMALLBLOB, MEDIUMBLOB, LARGEBLOB, and VARCHAR are variable length. The rest are fixed length. If the table definition contains at least one variable-length column, the entire record becomes variable length. Fixed-length record format will be used only when all columns are fixed length.

To make a table variable length, it is sufficient to simply change one of the columns that is at least four characters long from the CHAR type to VARCHAR, although it is good for documentation purposes to use the conceptually correct

type. For example, you would use CHAR(9) for SSN because all SSN values are exactly nine characters long. However, for the first and the last name, you would use VARCHAR(20) because each can be shorter than 20 characters.

If you need to store columns that are longer than 255 bytes (the limit for CHAR/VARCHAR), you would have to use either the BLOB or TEXT type, which will make your record variable length. If those long fields are queried infrequently, it might be a good idea to split the table into two—one with short fixed-length columns, and the other containing the rest of the columns. Both tables will have the same short primary key (e.g., an id number). Queries that need access to BLOB or TEXT columns will have to join the two tables, which will give you a small performance drawback, but on the other hand you can keep the queries that do not need BLOB or TEXT columns from being slowed down by their presence in the table. This will give you an advantage, assuming that most of your queries do not use the BLOB or TEXT columns.

Normalization

As you may remember from your database textbooks, normalization refers to organizing the data in a format that complies with a certain set of rules. Database theory defines several sets of rules, each known under the title of some normal form (e.g., first normal form, second normal form, third normal form). In practice, when we talk about normalization, we are usually referring to the third normal form. The third normal form means that each column will contain only one value (a requirement of the first normal form), each non-key column will be functionally dependent on the primary key (a requirement of the second normal form), and there will be no functional dependencies except on the primary key (the third normal form requirement).

Let's illustrate the concept of normalization by taking a denormalized table and bring it step by step into the first normal form. Consider the following table:

RUNNER	CITY	STATE	ZIP	RACE
Bob	Provo	UT	84606	Salt Lake Classic, Freedom Run, Deseret News Marathon
Bill	Denver	CO	80202	Bolder Boulder, Salt Lake Classic
Jane	Mesa	AZ	85201	St. George Marathon, Freedom Run
Jill	Provo	UT	84606	Freedom Run

We are not even in the first normal form since we have several races separated by a comma delimiter in the Races column. Let's bring it into the first normal form:

REGID	RUNNER	CITY	STATE	ZIP	RACE
1	Bob	Provo	UT	84606	Salt Lake Classic
2	Bob	Provo	UT	84606	Freedom Run
3	Bob	Provo	UT	84606	Deseret News Marathon
4	Bill	Denver	CO	80202	Bolder Boulder
5	Bill	Denver	CO	80202	Salt Lake Classic
6	Jane	Mesa	AZ	85201	St. George Marathon
7	Jane	Mesa	AZ	85201	Freedom Run
8	Jill	Provo	UT	84606	Freedom Run

We still are not in the second normal form, though; City and State depend on Runner, while RegId is our primary key. Now let's progress to the second normal form:

RUNNER	CITY	STATE	ZIP
Bob	Provo	UT	84606
Bill	Denver	CO	80202
Jane	Mesa	AZ	85201
Jill	Provo	UT	84606

REGID	RUNNER	RACE
1	Bob	Salt Lake Classic
2	Bob	Freedom Run
3	Bob	Deseret News Marathon
4	Bill	Salt Lake Classic
5	Bill	Bolder Boulder
6	Jane	St. George Marathon
7	Jane	Freedom Run
8	Jill	Freedom Run

Now we are still not meeting the requirement of the third normal form because City and State are functionally dependent on Zip. Let's fix it by adding another table:

ZIP	CITY	STATE
84606	Provo	UT
80202	Denver	CO
85201	Mesa	AZ

RUNNER	ZIP
Bob	84606
Bill	80202
Jane	85201
Jill	84606

REGID	RUNNER	RACE
1	Bob	Salt Lake Classic
2	Bob	Freedom Run
3	Bob	Deseret News Marathon
4	Bill	Salt Lake Classic
5	Bill	Bolder Boulder
6	Jane	St. George Marathon
7	Jane	Freedom Run
8	Jill	Freedom Run

Now we are in the third normal form. Normalization has both advantages and disadvantages. The advantages are as follows:

- In many cases, normalization avoids unnecessary repetition of data and saves disk space needed to store it.
- As the system requirements evolve, the chances of having the application break down while you're trying to make it meet the requirements are smaller.
- In many cases, normalized data results in better query performance.

The disadvantages are as follows:

- The queries are somewhat harder to write because more tables are involved.
- With the increased number of tables, you must address the issue of referential integrity more frequently in the application.

- In some cases, normalization increases the amount of disk space required to store the data.
- In some cases, normalization degrades performance due to the join overhead.

Knowing when to normalize and when to avoid it—at least with MySQL—is more of an art than a science. The knowledge comes with experience. However, in general it's important to have a measure of common sense, remember your priorities, and strive for balance. For example, if you have a table with only 5000 records, you are confident it is not going to grow, and it is critical to develop the application fast, normalization probably would be a waste of time. However, if the table will eventually grow to 1,000,000 records, you should normalize it at least to a certain degree to make sure that you are not wasting too much disk space and that your queries can properly use keys.

Some people are afraid to normalize because of a concern that MySQL will not be able to execute complex joins efficiently. This concern is not valid. Practical experience and benchmarks show that as long as you have proper keys, you do not need to worry about MySQL being slow on a join.

The important principle to remember when deciding when to normalize and when not to normalize as far as performance is concerned is that the speed of a particular query primarily depends on the amount of data it would have to examine in order to produce the answer. Normalized design, therefore, is usually worthwhile if it significantly reduces the amount of data stored or if it improves the use of keys. Decisions should be made on a case-by-case basis, and an understanding of the basics of how the MySQL optimizer works will be very helpful. (We discuss the optimizer in detail in Chapter 15.)

My personal approach to normalization is very pragmatic. I ask myself: Is there a way to save some disk space with a normalized table design? Will some important queries run faster if I normalize? In the future, am I likely to use queries that will benefit from normalization? Will my application be easier to maintain if I normalize? If I can answer yes to these questions, I do it.

The Need for Proper Keys

First, let's briefly review what a key is and look at a simple example. Suppose you are trying to locate your friend's phone number in a phone book. If you know his last name, the task is trivial—based on the first letter of his last name, you take a guess at which section of the phone book his entry will be located in, open it there, and then navigate skipping back and forth, depending on where you are, and reducing the length of your skip until the desired entry comes into your view.

Now let's consider another situation. All you know about your friend is his address. If your phone book is like mine, you would potentially have to scan through the whole book before you find the target entry. Instead of the couple of minutes you have spent locating the entry by last name, you are now going to spend hours and hours. What is it that makes the difference?

The book has an index or a key on the last name field, but does not have one on the address. Database keys work pretty much like the phone book in this regard. Data is organized into a special structure that greatly decreases the amount of work needed to find a particular entry. The lookup that would take days without a key will now take a couple of minutes, the work that takes minutes will be done in seconds with a key, and what takes seconds will be done in a fraction of a second.

An index or a key can be created on one or more columns. The order of columns is significant. Some queries will not be able to use a particular key if you put the columns in the wrong order.

As you can guess, the "no free lunch" law suggests that the great advantages of creating a key do come at a cost. A key takes up a certain amount of disk space, and it must be updated every time the value of the key for a particular record is modified or a record is either inserted or deleted.

In a table with *N* columns, you could create 2^*N*-1 different keys. That would be 1023 possibilities for a table with only 10 columns, and the number of possibilities increases exponentially as the number of columns grows. Out of all those possibilities, which ones do we select?

Again, the phone book analogy is helpful. The question you should ask is very simple: What exactly do I want to be able to look up by? Consider the employee table we designed previously. We want to be able to look up employees by their last name. Sometimes we know their first name, too, but we never want to look up just by their first name. To do this kind of lookup, the best key is (lname,fname)—note the order. We sometimes want to ask, "How many managers make more than $80,000 per year?" A good key for this one would be (position,salary). And one last question type for now: "How many people were hired between January and May of 2001?" To optimize queries of this kind, we use a key on just one column: hire_date. To summarize everything into one SQL statement:

```
CREATE TABLE employee (
  ssn CHAR(9) NOT NULL PRIMARY KEY,
  fname CHAR(20) NOT NULL,
  lname CHAR(20) NOT NULL,
  hire_date DATE NOT NULL,
  vac_days TINYINT  UNSIGNED NOT NULL,
  position ENUM ('CEO','CTO','Engineer', 'Manager','Accountant' )
```

```
    NOT NULL,
    salary  MEDIUMINT UNSIGNED NOT NULL,
    key lname_fname(lname,fname),
    key position_salary(position,salary),
    key hire_date(hire_date)
  );
```

As a rule of thumb, when you're creating a key, you should try to accomplish more than one goal. You do not want to create a key that will optimize just one query that is run infrequently. You want to try to pick a key in such a way that it will also help other queries. In our previous example, instead of creating a (lname) key, we used (lname,fname). We assumed that we would do a lookup on the combination of last name and first name much more frequently than just the last name. If the majority of the lookups used only the last name, the (lname) key would be more preferable because it would require less space to store and would also be used more efficiently by the optimizer.

The same concept applies to the (position,salary) key. We chose the (hire_date) key because we assumed that when we want to do a lookup by hire date, there is rarely any other information available to use that we can include in the query to narrow down the range of records. If this were not the case, we would have considered adding other columns.

Note the order of columns in the key. It is very important. The first (prefix) column or columns in a key must be the ones for which we always know the value or the range for the queries that we would like to optimize. The subsequent (suffix) columns are the ones for which we may or may not have the value or the range. For example, when we decided to create the key on (position,salary), we did so assuming that we always know the position and that sometimes we know the salary. If we reversed the order of the fields in the key, we would be able to use that key only for queries where the salary value or range is known.

One good approach is to begin with a few self-evident keys and then add more as needed. An effective technique in determining the needed keys is to enable log-long-format and log-slow-queries options on the server, and periodically perform test runs of your application as you develop it, followed by the examination of the slow log. Not all queries that end up in the slow log with this configuration are critical to optimize. For example, if a query scans a table with only 50 records, adding a key will not make that much of a difference. However, if you know that the table with 50 records at some point will have several thousand, you should add the proper keys before things get out of control.

It is impossible to overemphasize the importance of creating proper keys. Although I do not have the exact statistics, I would roughly estimate from my experience that about 70 percent of performance problems with MySQL can be corrected by creating proper keys in the table.

Data Wisdom

A lot of times, a thorough understanding of your application's requirements and the nature of the data can help you design a more efficient schema. Let's illustrate this concept with an example.

Suppose we are storing e-mail addresses. A straightforward approach would be as follows:

```
CREATE TABLE user ( /* some other fields*/, email VARCHAR(50) NOT
NULL, KEY(email))
```

This is a perfect approach if our table fits entirely into RAM and we want to optimize for random full-address retrieval, as opposed to grouping by domain or pulling out users with an e-mail address at a certain domain, such as aol.com. Now suppose we have additional requirements. We need to be able to group by domain so that we can generate domain statistics and optimize our special mailing list delivery program. We try this:

```
CREATE TABLE user ( /* some other fields*/, email_user VARCHAR(50)
NOT NULL, email_domain VARCHAR(50) NOT NULL,
KEY(email_user),KEY(email_domain))
```

Although the storage and retrieval of full addresses is now slightly more complex (we have to use CONCAT(email_user,'@',email_domain) to get them), our queries that involve the domain can use keys. Additionally, we have save 1 byte per e-mail by not storing the @ character.

Now suppose that as the table grows, we cannot fit it into RAM anymore, so it becomes critical to reduce the data size to keep performance at a satisfactory level. We notice that 90 percent of our addresses come from only 10 domains. We can, therefore, replace the domain string with an integer id reference to another table, and thus save a little bit of space for all domains that are longer than 4 characters (an integer takes 4 bytes). We optimize our schema the following way:

```
CREATE TABLE domain (id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
domain VARCHAR(50) NOT NULL, KEY(domain));
CREATE TABLE user (/*some other fields*/, email_user VARCHAR(50) NOT
NULL, email_domain_id INT NOT NULL,
KEY(email_user),KEY(email_domain_id));
```

Now most of our queries dealing with the e-mail address will involve a join, but it will be on a key, which will reduce the join overhead penalty to a minimum. On the other hand, our records are now smaller, so the queries that do not need the e-mail address will be faster.

Similar application-specific observations can help you achieve great improvements in performance if you take advantage of them.

The Proper Table Type

MySQL gives you the choice of selecting a table type. To specify the table type, you can either use the option default-table-type in your configuration file (my.cnf)—which will make all the newly created tables the specified type—or you can provide it as part of the CREATE TABLE statement—for example:

```
CREATE TABLE employee (fname varchar(30) not null, lname varchar(30)
not null, ssn char(9) not null, key(lname(5),fname(5)), primary
key(ssn)) TYPE=INNODB;
```

MySQL supports the following table types:

- **ISAM** is supported only to facilitate migration from MySQL 3.22.
- **MyISAM** is the recommended nontransactional table type in 3.23.
- **HEAP** creates an in-memory table with hash indexes (as opposed to B-tree for disk-based tables) that will lose all of its data (but not the schema) when the server shuts down.
- **MERGE** allows you to view several MyISAM tables as one.
- **InnoDB** is the recommended transactional handler.
- **BDB** is an old transactional handler, which is around for historical reasons but is no longer in active development.

The choice for a regular permanent table type is usually between MyISAM and InnoDB. Both have advantages and disadvantages. MyISAM does not have any transactional overhead. This makes selects overall slightly faster, and offers major improvement with queries that modify a lot of rows at once. However, InnoDB tables do much better under a mixed load of reads and writes due to row-level locking as opposed to table locking with MyISAM. Additionally, because of its transactional nature, InnoDB will preserve data better in case of a crash, and has a very fast crash recovery even for large tables. Large MyISAM tables take a long time for check and repair after a crash.

The MyISAM handler is ideal for transient data collection tables that are kept relatively small by archiving the old data or for read-only data warehousing tables. InnoDB is ideal for large tables that are updated frequently (with only one or a few rows being modified at a time) but that are not purged very often (by deleting many records at once).

The table handler behavior is essentially the same in 3.23 and in 4.0, and our earlier comparison applies to both versions.

If you ever make the wrong choice for the table handler, you can change it with the ALTER TABLE command. For example:

```
ALTER TABLE student TYPE=INNODB
```

Note that in 3.23 you will need the max binary to be able to use InnoDB tables. In the later versions for 4.0, the InnoDB handler should be present in the non-max version.

CHAPTER 14

Configuring the Server for Optimal Performance

Several types of tasks are involved in tuning your server, and the most important is optimizing your schema. After that come the variables, and only then hardware and operating system tweaks. In this chapter, we cover all of those aspects, including a listing of server variables along with an explanation of how each affects performance.

Optimizing the Schema

In my experience working with MySQL support, I have noticed that it is typical for a good system administrator to take a hardware/OS approach to performance improvement. When MySQL does not perform as it should, the questions frequently asked are, "What should I upgrade?" and "What operating system setting should I tweak?" A DBA approach is frequently, "What server configuration setting should I change to make the database work better?" While there are cases when one of those two approaches will help, more often than not the key to enhancing performance is improving the table design.

We have already discussed the principles of table design in Chapter 13. Here is a brief review of the relevant principles from that chapter:

- **Normalization:** A healthy degree of normalization reduces storage requirements, and reduces the amount of data the average query would cause the database server to examine.

- **Keys:** Properly created keys speed up many queries significantly. The speedup is frequently a thousand-fold or higher. In many cases, the speedup achieved by creating proper keys can never be accomplished by even the most expensive hardware upgrade.
- **Optimal column type:** Choose storage type with the storage requirements in mind. This helps reduce the size of the record and speeds up a lot of queries.
- **Data wisdom:** By being aware of the nature of your data and the requirements of your application, you can design your tables to take a minimum amount of space and give you better performance.

To determine whether the table schema on your server is optimal, you can do the following:

- Make sure that the server is running with --log-slow-queries and --long-log-format options. You may specify the path to the slow query log as an argument to --log-slow-queries. Otherwise, the file will be in your data directory under the name of `hostname`-slow.log.
- Run your application under load.
- Examine your slow log. You may find the mysqldumpslow script (written by Tim Bunce, the author of DBI and a great master of Perl) quite helpful in viewing the contents of the slow log in a more easily readable form. You may also use the mysql_explain_log script (written by the programmers at mobile.de). myslqdumpslow and mysql_explain_log are included into MySQL distribution.
- Account for every query in the slow log and see if you can add some keys or change the schema in other ways to get rid of them.

Optimizing Server Variables

DBAs with a strong Oracle background are accustomed to tweaking buffer sizes extensively to gain performance improvements. With MySQL, however, changing a buffer size from the default value more often than not either makes no difference or makes things worse. Yet there are situations when it improves performance.

Let's first familiarize ourselves with the server variables by examining the configuration of a running server. If you execute the SHOW VARIABLES command from the command-line client, your output will resemble that shown in Listing 14.1.

```
Variable_name      Value
back_log      50
basedir      /usr/
bdb_cache_size      8388600
bdb_log_buffer_size      256000
bdb_home      /var/lib/mysql/
bdb_max_lock      10000
bdb_logdir
bdb_shared_data      OFF
bdb_tmpdir      /tmp/
bdb_version      Sleepycat Software: Berkeley DB 3.2.9a: (June 3,
2002)
binlog_cache_size      32768
character_set      latin1
character_sets      latin1 big5 czech euc_kr gb2312 gbk sjis tis620
ujis dec8 dos german1 hp8 koi8_ru latin2 swe7 usa7 cp1251 danish
hebrew win1251 estonia hungarian koi8_ukr win1251ukr greek win1250
croat cp1257 latin5
concurrent_insert      ON
connect_timeout      2
datadir      /var/lib/mysql/
delay_key_write      ON
delayed_insert_limit      100
delayed_insert_timeout      300
delayed_queue_size      1000
flush      OFF
flush_time      0
have_bdb      YES
have_gemini      NO
have_innodb      YES
have_isam      YES
have_raid      NO
have_openssl      NO
init_file
innodb_additional_mem_pool_size      1048576
innodb_buffer_pool_size      8388608
innodb_data_file_path      ibdata1
innodb_data_home_dir
innodb_file_io_threads      4
innodb_force_recovery      0
innodb_thread_concurrency      8
innodb_flush_log_at_trx_commit      OFF
innodb_fast_shutdown      ON
innodb_flush_method
innodb_lock_wait_timeout      50
innodb_log_arch_dir
innodb_log_archive      OFF
```

Listing 14.1 Output of SHOW VARIABLES. (continues)

```
innodb_log_buffer_size      1048576
innodb_log_file_size      5242880
innodb_log_files_in_group      2
innodb_log_group_home_dir      ./
innodb_mirrored_log_groups      1
interactive_timeout      28800
join_buffer_size      131072
key_buffer_size      16773120
language      /usr/share/mysql/english/
large_files_support      ON
locked_in_memory      OFF
log      OFF
log_update      OFF
log_bin      ON
log_slave_updates      OFF
log_long_queries      OFF
long_query_time      1
low_priority_updates      OFF
lower_case_table_names      0
max_allowed_packet      2096128
max_binlog_cache_size      4294967295
max_binlog_size      1073741824
max_connections      2000
max_connect_errors      10
max_delayed_threads      20
max_heap_table_size      16777216
max_join_size      4294967295
max_sort_length      1024
max_user_connections      0
max_tmp_tables      32
max_write_lock_count      4294967295
myisam_max_extra_sort_file_size      256
myisam_max_sort_file_size      2047
myisam_recover_options      1
myisam_sort_buffer_size      8388608
net_buffer_length      16384
net_read_timeout      30
net_retry_count      10
net_write_timeout      60
open_files_limit      0
pid_file      /var/lib/mysql/mysql.pid
port      3306
protocol_version      10
record_buffer      131072
record_rnd_buffer      131072
query_buffer_size      0
safe_show_database      OFF
```

Listing 14.1 Output of SHOW VARIABLES. (continues)

```
server_id      1
slave_net_timeout      3600
skip_locking      ON
skip_networking      OFF
skip_show_database      OFF
slow_launch_time      2
socket      /var/lib/mysql/mysql.sock
sort_buffer      2097144
sql_mode      0
table_cache      500
table_type      MYISAM
thread_cache_size      200
thread_stack      131072
transaction_isolation      READ-COMMITTED
timezone      MDT
tmp_table_size      33554432
tmpdir      /tmp/
version      3.23.51-Max-log
wait_timeout      28800
```

Listing 14.1 Output of SHOW VARIABLES. (continued)

If you want to see only one of those variables, just type SHOW VARIABLES LIKE 'variable_like_pattern'. This does produce a rather humorous syntax where you type the entire variable name in the LIKE pattern. For example, you type

```
SHOW VARIABLES LIKE 'record_buffer'
```

when all you really want is record_buffer and nothing else.

Not all of the variables we've discussed are actually variables that you can modify. Some reflect the build of the server. Most of the true variables can be set with the --set-variable option to mysqld or the set-variable line in the my.cnf configuration file. The syntax is --set-variable=*varname=varval*. The option parser understands the K and M multipliers for kilobyte and megabyte. The my.cnf file is parsed by the same code that parses the command line. Therefore, any command-line option is also supported in the [mysqld] section of my.cnf. The only difference is that each option has to be on a new line and there is no -- prefix. You just type the actual name of the option. For example, in my.cnf you can include a line like this:

```
set-variable = key_buffer_size=256M
```

You can accomplish the same on the command line with this:

```
safe_mysqld --set-variable=key_buffer_size=256M &
```

Note that spaces are allowed between set-variable and = and after the first = in my.cnf, but are not allowed on the command line. Most users prefer setting variables in my.cnf because doing so facilitates configuration maintenance. Command-line options are useful for a testing environment, or when you have a script that starts the MySQL server in a customized environment.

For an example of the my.cnf file syntax, take a look at the my-small.cnf file in the distribution. On RPM distributions, you can find the file in /usr/share/mysql, and on binary Unix distributions you can find it in the support-files directory.

In MySQL 4.0, you can also use the --variable=varval syntax. The 3.23 --set-variable syntax is officially considered depreciated, but it is still supported in 4.0. Since there is a chance you may have to go back to 3.23, I recommend sticking with the --set-variable syntax up until about 4.0.15 or so when it comes out (the current version is 4.0.5 as of this writing). That is when I expect 4.0 to be in a condition where the probability of having to downgrade for any reason would be extremely low.

Version 4.0 also adds the capability to change the server configuration variables at runtime using the SET GLOBAL syntax in a query from a command-line MySQL client or your own application. For example:

```
SET GLOBAL key_buffer_size=65536
```

Note that SET GLOBAL does not understand the K and M multiplication prefixes, so you have to perform the multiplication yourself and give it the result in bytes.

Variable Descriptions

As you can see, it is possible to change many variables. Let's examine each one, explaining what it affects and the role it plays in the performance of the server. We explain every line you see in the output of SHOW VARIABLES. Some of these lines are not changeable or do not affect performance, but we discuss them for the sake of completeness.

First, the 3.23 variables, which are also present in 4.0:

back_log: Determines the size of the server socket listen queue. The default is 50, which should be enough even for a highly active server. However, if you start experiencing strange connection errors, you may want to increase it and see if that helps. To set this value, use the set-variable syntax.

basedir: A file-system path relative to which some other paths are calculated if not explicitly set. The variable does not explicitly affect server performance. To set this value, use the basedir option.

bdb_cache_size: The memory cache allocated by the Berkeley DB table handler to cache data rows and keys. This setting is relevant only if the Berkeley DB driver is built into the binary. The default is 8MB. If you are not using BDB tables but the BDB driver is built in, you should run mysqld with the --skip-bdb option to avoid unnecessary memory allocation. This advice applies for all bdb_ options that allocate memory. To set the variable, use the set-variable option.

bdb_log_buffer_size: The size of the BDB transaction log buffer. The default is 256K. This setting is relevant only for the BDB table handler, which is no longer in active development. It is recommended that you use the InnoDB handler instead for the transactions.

bdb_home: Specifies the home directory for BDB bookkeeping. The default is datadir. To set the variable, use the bdb-home option.

bdb_max_lock: Specifies the maximum number of active locks on a BDB table.

bdb_logdir: Specifies the directory for BDB logs. To set the variable, use the bdb-logdir option.

bdb_shared_data: Controls whether the BDB handler will use shared data. It is set to OFF by default; change it to ON with the bdb-shared-data option.

bdb_tmpdir: Specifies the directory for temporary files used by the BDB handler. The default is tmpdir. To set the variable, use the bdb-tmpdir option.

bdb_version: Specifies the version string of the BDB handler. You cannot change this variable with an option.

binlog_cache_size: MySQL keeps a logical binary update log used for replication and backup. To preserve consistency in the log when running with transactions enabled, all updates are written into a temporary cache log until the transaction commits. When the transaction commits, the contents of the temporary cache log are flushed into the main binary log. The temporary binary log uses cached I/O. This variable controls the size of the I/O buffer for this temporary log. You may benefit from increasing the value if you have either long queries or a lot of them in your transactions. By default, the value is 32K. To set the variable, use the --set-variable option.

character_set: Specifies the default character set used for sorting. The default is latin1. To set the variable, use the --default-character-set option.

character_sets: Shows which character sets have been compiled into the server. You cannot change this variable with an option.

concurrent_insert: MyISAM tables have originally required exclusive table-level locks to be updated. However, in the early days of MySQL 3.23 we received a patch from Joshua Chamas that allowed SELECTs on a MyISAM table while an INSERT was in progress for tables with no holes from deleted records. After

some rework and adaptation, the patch was included into the server. This option controls whether this feature is enabled. You should keep it on unless you suspect a bug in the concurrent insert code. The option is set to ON by default. To turn it off, use the --skip-concurrent-insert option.

connect_timeout: When the MySQL client connects to the server, there is a place in the server code where the server holds a global mutex while it is waiting for the client to respond to the server during the handshake stage. If the server had no time limit on this wait, this could prevent all other clients from connecting in the meantime. Thus, a malicious client could stage a denial-of-service attack by not responding to the greeting on purpose, or a client connecting over a troublesome network could cause service delays. To avoid this type of problem, the server has a timeout that specifies how long a client can wait to respond during the handshake before it gets the "Bad handshake" message. The default value is 2 seconds; you can change it with the --set-variable option.

datadir: Specifies the data directory of the server. All databases are implemented as subdirectories of the data directory. MyISAM tables are stored in three files in the directory corresponding to the database: tablename.frm (the table definition file), tablename.MYI (the table index file), and tablename.MYD (the table data file). InnoDB tables have a table definition file just like MyISAM tables, but the data and the index are stored in a separate table space. It is possible that some time in the future the InnoDB tables will be stored in the table space file or partition entirely.

delay_key_write: Controls whether the key buffer pages will be flushed immediately after an update that made them dirty, or if the write-out of the dirty pages will be delayed until the key cache decides to displace it or until the table is flushed. Each table has a delay_key_write attribute that is set to ON by default and that is specified during table creation. When set to OFF, this option disallows delayed key writes even on individual tables that have the per-table delay_key_write enabled. You should not change this setting unless you suspect a bug in the delay_key_write code. By default, delay_key_write is enabled, and tables are created with the corresponding attribute set to ON. To turn it off, use the --skip-delay-key-write option.

delayed_insert_limit: All delayed_ options relate to the delayed insert feature in MySQL. If a table is locked when a client requests to insert a row, the query will normally not return until the lock becomes available and the insert is able to proceed to completion. This can potentially take a long time, and it may be more important to the client to return fast than to make sure that the row actually gets inserted into the table. To satisfy this requirement, Monty Widenius at some point implemented INSERT DELAYED. When run with the DELAYED attribute, INSERT will never tie up the client while waiting for a lock. If the lock

can be obtained, the insert proceeds as if there were no DELAYED attribute. However, if the lock is not available, the insert is queued for later processing, and the client is notified of success. The insert then is being processed in the background by the delayed insert thread assigned to the given table. The delayed insert thread in a loop inserts delayed_insert_limit rows, then checks to see if there are SELECT queries asking for a lock on the table and yields the lock if require it at a later time. The default value is 100; you can change it with the --set-variable option.

delayed_insert_timeout: When a delayed insert thread has de-queued and inserted all of the delayed rows, instead of exiting immediately it will hang around for delayed_insert_timeout seconds in anticipation that another query will perform a delayed insert affecting the lock and will be forced to put the rows in the delayed insert queue. When the timeout expires, the thread exits. The purpose of this wait period is to reduce a possibly unnecessary thread creation/destruction overhead. The default value is 300 seconds; you can change it with the --set-variable option.

delayed_queue_size: When a delayed insert process cannot be performed immediately, the inserted rows are put in the delayed queue that holds up to delayed_queue_size rows. If the queue is full, all subsequent delayed inserts that encounter the lock conflict and are forced to be processed as delayed will wait for the queue to clear out and make room for the new row. The default setting is 1000; you can change it with the --set-variable option.

flush: If enabled, FLUSH TABLES will force the operating system to write out the disk cache associated with MyISAM data and key files. This option is useful when you have reason to believe the system might go down uncleanly during operation and would like to minimize the loss of data in that case. The default setting is OFF. To enable it, use the --flush option.

flush_time: If this variable is set to a non-zero value, a special manager thread will be created that will periodically FLUSH TABLES (or in other words, close and reopen them, which writes out the dirty buffers), and will perform log cleanup for the BDB handler. If BDB is enabled, the thread is always started. However, if flush_time is 0 (the default), FLUSH TABLES will not be performed, and instead, the log cleanup will be invoked whenever the BDB handler reports the need to do so. You can change this setting with the --set-variable option.

have_bdb: Specifies whether Berkeley DB tables can be created by the running server. The variable is set to YES if BDB support has been compiled and no --skip-bdb option has been used, and is set to NO otherwise.

have_gemini: Specifies whether Gemini tables can be created by the running server. The variable is set to YES if Gemini support has been compiled and no --skip-gemini option has been used, and is set to NO otherwise.

have_innodb: Specifies whether InnoDB tables can be created by the running server. The variable is set to YES if InnoDB support has been compiled and no --skip-innodb option has been used, and is set to NO otherwise.

have_isam: Specifies whether the compatibility ISAM table support (the 3.22 table format) has been compiled into the server. Currently this variable is always set to YES.

have_raid: Specifies whether rapid application development (RAID) table support has been compiled into the server. A data file of a RAID table is fragmented into several files and in theory can be spread across the disks, thus creating kludgy application-level RAID support. However, this feature was implemented to overcome the 2GB file size limitation on the file systems of the old days, which reduced the maximum size of a MyISAM table. Most modern file systems do not have that limitation, so RAID table support is no longer developed or maintained.

have_openssl: Specifies whether the server was compiled with support for Secure Socket Layer (SSL) encryption in the client-server protocol. As of this writing, this feature is in a very early alpha stage in both the 4.0 and 3.23 branches, and unfortunately, it is not being developed, either. I hope in the future this will change.

init_file: When the MySQL server starts up, if init_file is set it will execute SQL statements from it. This feature is helpful, for example, if you are using auxiliary HEAP (or in-memory) tables that need to be populated on server startup because whatever you had in them is lost when the server goes down. The variable is empty by default; you can set it with the --init-file option.

innodb_additional_mem_pool_size: All variables starting with innodb_ apply to the configuration of the InnoDB table handler, which is the recommended handler for transactions. You can find more up-to-date information on InnoDB handlers at www.innodb.com/ibman.html. The InnoDB handler has two buffer pools: one for caching the data and the keys (the main pool), and the other for meta information (the additional pool). The innodb_additional_mem_pool_size variable determines the size of the additional buffer pool. If the additional pool is not sufficient, InnoDB starts to allocate directly from the operating system, and writes messages to the MySQL error log. The default size is 1MB, and you can set it with the --set-variable option.

innodb_buffer_pool_size: Determines the size of the main InnoDB buffer pool, which caches keys and data rows. Note that unlike MyISAM, which caches only keys, InnoDB caches both keys and rows. The recommended value of innodb_buffer_pool_size is about 80 percent of the total physical memory if the machine is entirely dedicated to MySQL. The default is 8MB, and you can set it with the --set-variable option.

innodb_data_file_path: Specifies the path to the InnoDB table space file or partition device relative to innodb_data_home_dir. There is no default value. If you do not set it in MySQL 3.23, the InnoDB handler will be deactivated during startup and a message will be printed to the error log. Starting after 4.0.2, there is a default to create a ibdata1 file in the data directory. You can set the variable with --innodb_data_file_path. The syntax requires that you specify the size of the data file (separated with a colon from the path), and allows you to have more than one data file on a line separated with a semicolon. At the end, a special autoextend option is allowed. Example: innodb_data_file=ibdata1:40MB;ibdata2:20MB:autoextend

innodb_data_home_dir: Specifies the directory relative to which the paths to all InnoDB files are constructed. The setting defaults to datadir. It is also possible to set it to an empty string and use absolute paths for all InnoDB files. You can change this variable with --innodb_data_home-dir.

innodb_file_io_threads: Specifies the number of I/O threads. This is a legacy option, so you do not have to worry about it. Maybe by the time this book is published, this option will be removed from SHOW VARIABLES. Check www.innodb.com/ibman.html for the most current information.

innodb_force_recovery: If the server crashes, InnoDB will attempt a recovery when the server comes up. Unfortunately, in some extreme cases the corruption is so bad that a consistent recovery is not possible. By default, the server refuses to start if the data cannot be consistently recovered. However, whatever inconsistent data is left may be of value to the user. innodb_force_recovery sets a corruption tolerance level of the recovery process and will allow the server to come up even if the data is inconsistent. The possible values are from 0 (no tolerance for inconsistency at all) to 6 (the highest possible inconsistency tolerance that will allow the server to run). The default is 0, and you change it with the --set-variable option.

innodb_thread_concurrency: An advisory parameter passed to the thread implementation by the InnoDB handler to help optimize thread performance. The recommended value is the number of processors plus the number of disks. The setting defaults to 8, and you can change it with the --set-variable option.

innodb_flush_log_at_trx_commit: Controls the transaction log flushing behavior on commit. By default, it is set to OFF. This gives better performance, but you run a small risk of losing a transaction if the system crashes with the transaction log data still in the disk cache. To change it, use the innodb_flush_log_at_trx_commit option. Note that it accepts a boolean numeric argument (1 or 0). Example: innodb_flush_log_at_trx_commit=0

innodb_fast_shutdown: If this setting is enabled, the shutdown occurs without some extra safety measures that could take a long time. It is enabled by

default. You can disable it with innodb_fast_shutdown=0 . However, be warned in that case that shutdown may take several minutes or, in some situations, even hours.

innodb_flush_method: This option applies only to Unix and was added for troubleshooting purposes. It can be set to either fsync or O_DSYNC, and determines which system call and parameters will be used to flush transaction logs. We recommend you not mess with it unless you are curious or are instructed by a member of the support team. You change the setting by using the --innodb_flush_method option.

innodb_lock_wait_timeout: If a transaction has been waiting for a lock for longer than innodb_lock_wait_timeout seconds, it will be rolled back and an error will be reported to the client. You can set this variable with the --innodb_lock_wait_timeout option.

innodb_log_arch_dir: Specifies the directory where logs are moved after rotation. The setting defaults to the MySQL data directory. You can change it with --innodb_log_arch_dir.

innodb_log_archive: Specifies whether you want to archive old logs. By default, it is set to OFF. To change it, use innodb_log_archive=1.

innodb_log_buffer_size: Specifies the size of the transaction log buffer. According to the current InnoDB documentation, sensible values range from 1MB to 8MB. The default is 1MB. If you have large transactions, the value of 8MB is recommended. You can change the setting with the --set-variable option.

innodb_log_file_size: Specifies the size of an individual transaction log file. The default is 5MB. InnoDB writes to a group of log files in a circular manner. innodb_log_file_size * innodb_log_files_in_group should not exceed 4GB. The value can be set with the --set-variable option.

innodb_log_files_in_group: Specifies the number of logs in the log group. The default is 2; you can change it with the --set-variable option.

innodb_log_group_home_dir: Specifies the home directory for the logs. The setting defaults to the MySQL data directory; you can set it with the --innodb_log_group_home_dir option.

innodb_mirrored_log_groups: Specifies the number of identical log groups. It is set to 1 by default. Keep it that way.

interactive_timeout: Specifies the number of seconds an interactive client is allowed to stay connected to the server without sending commands. A client lets the server know that it is interactive with a special option flag passed as an argument to mysql_real_connect(). The default is 28800 seconds (8 hours). To change it, use --set-variable.

join_buffer_size: Specifies the buffer used to store records while performing a join without keys. If increasing the size of this buffer improves performance, that's a good sign of improper schema design or poorly written queries. When trying to speed up a join, you should first consider finding a way to do it with keys. The default is 128K; you can change the setting with the --set-variable option.

key_buffer_size: Specifies the size of the MyISAM key cache; it affects only MyISAM tables. If you are using only InnoDB tables, set it to 0. If you are using MyISAM tables, the recommended value on a dedicated MySQL server is 25 percent of physical RAM. Do not increase it without a good reason. To see if you need to increase it, execute SHOW STATUS LIKE 'Key_read%' and compute the ratio of Key_read_requests/Key_reads. If it is significantly over 1 percent, you should consider increasing the key cache. The default is 8MB; set it with the --set-variable option.

language: Specifies the directory containing the language-specific error message file errmsg.sys. The default is $basedir/mysql/share/English. You can change it with the --language option.

large_files_support: Shows whether the server was compiled with large files support. No option is available to change this setting.

locked_in_memory: This variable is always set to OFF on systems not supporting mlockall(). For Unix systems that support mlockall(), this setting specifies whether the call has been made after memory initialization. The effect of the call is to instruct the operating system not to swap the process memory. The main reason for enabling this option is to protect a large key buffer from being swapped out. The server needs root privileges to be able to use this option. The default is OFF; to enable it, use the --memlock option.

log: Shows whether general logging has been enabled. The general log logs every command, and is useful for application debugging, server troubleshooting, and security audits. There is a slight performance penalty, and on a heavily loaded system the log may grow very fast and fill up the log partition. We suggest you have this setting enabled on development systems, and on production systems use discretion depending on your priorities. If you do use it in production, make sure your logs are frequently rotated and moved to the archive. The setting is disabled by default. To enable it, use the --log option. This option accepts an argument, which is the name of the log. If no argument is given, logs will be stored in the data directory under the name of `hostname`.001, `hostname`.002, `hostname`.003, and so forth. Note that the general log does not autorotate, and you need to make it happen manually by periodically running FLUSH LOGS. If you want to be able to rotate the log, do not give the extension in the log argument. For example: log=/var/log/mysql/querylog (not log=/var/log/mysql/querylog.txt).

log_update: Shows whether the textual logical update log has been enabled. This option is a carryover from the pre-binary log days. It is set to OFF by default. The argument syntax and rotation caveats are the same as with the log option. You can change this setting with the --log-update option.

log_bin: Shows whether the logical update log (binary format) has been enabled. The log is essential in replication and can be used for incremental backup. The variable is set to OFF by default. Enable it if you want to replicate or do incremental backups. You change this setting with the --log-bin option. The option can have a path argument.

log_slave_updates: Shows if the replication slave has been instructed to log replicated updates into its own binary update log. This option was originally implemented to enable a safe two-way co-master replication, in which each server can receive updates from clients, and as a replication slave of the peer. Not logging the updates prevented update loops. However, with the introduction of server_id two-way replication can occur safely, even when log_slave_updates is enabled. You need to have this enabled if you are performing relay replication—this slave serves as a master to another slave. The variable is set to OFF by default; you can change it with the --log-slave-updates option. For example: log-slave-updates=1

log_long_queries: Shows whether the slow query log has been enabled. You enable the slow log with --log-slow-queries, which supports an optional path argument. The variable is set to OFF by default. This option is very helpful in finding performance problems, so I suggest you keep it enabled.

long_query_time: Specifies the minimum amount of time in seconds a query needs to take to be considered slow. Note that if you are using the --long-log-format option, all queries that do not use keys will be considered slow regardless of how long they actually take. You can this setting with the --set-variable option.

low_priority_updates: MyISAM tables use table-level locks. When a locking decision has to be made as to which query gets the contested lock, preference is given to the query that has requested the write lock. If this option is enabled, the preference will be given to the query requesting the read lock. The default is OFF. To enable the setting, use low-priority-updates=1.

lower_case_table_names: If this setting is enabled, table names are forced to be in lower case, thus making them case-insensitive even on case-sensitive file systems. The default is 1 on Windows and 0 otherwise. To change it, use the --set-variable option.

max_allowed_packet: The client-server protocol has the concept of its own packet. max_allowed_packet determines the maximum size of a packet in

bytes in the client-server transmissions. The same concept applies in the replication protocol. Because a query is being sent in one packet, it can be no longer than max_allowed_packet bytes. The setting defaults to 1MB. In MySQL 3.23, the maximum value is 16MB, and in 4.0 the maximum is 2GB. To change the setting, use the --set-variable option.

max_binlog_cache_size: Specifies the size of the binlog cache, which is used for writing transactions to the binary log (see binlog_cache_size for a more detailed explanation). The default is 4GB - 1. To change the setting, use the --set-variable option.

max_binlog_size: If the binary update log exceeds max_binlog_size bytes, it will be automatically rotated. The default is 1GB, and it cannot be set any higher. To change it, use the --set-variable option.

max_connections: Specifies the number of concurrent connections the server is allowed to accept. There is always one more connection reserved for the root user for emergency purposes. The default is 100; it can be changed with the --set-variable option.

max_connect_errors: As we've already discussed under connect_timeout, when a client delays a response during the handshake, the connection will be terminated with an error message. To prevent abuse, if a client from a certain IP has been slow in performing a handshake max_connect_errors times, it gets blacklisted and is not allowed to connect until the DBA issues FLUSH HOSTS on the server. The default is 10; to change the setting, use the --set-variable option.

max_delayed_threads: Delayed inserts are handled by assigning a thread to processed queued rows for each table. max_delayed_threads sets a limit on how many tables can have delayed inserts in progress at the same time. If a delayed insert is requested after the max_delayed_threads limit has been reached, the insert will be processed in the non-delayed manner. The default is 20; to change the setting, use the --set-variable option.

max_heap_table_size: MySQL supports a special in-memory table type called HEAP. You can create it explicitly by adding TYPE=HEAP to the end of the CREATE TABLE statement, or it can be created when a temporary table is needed to resolve a complex query. Naturally, because the table is in memory, there is a risk that accidentally a table will become so large that it runs the system out of memory. max_heap_table_size sets a limit on how much memory a HEAP table can take. If this limit is exceeded during a regular insert into a HEAP table, an error is given to the user. If the limit is exceeded during query processing when a temporary table was required, the table is converted to MyISAM (disk-based) and the operation progresses. The default value is 16MB; use the --set-variable option to change the value.

max_join_size: Serves as a safety limit to block potentially long joins. If the optimizer guesses that in order to resolve the query it would have to examine more than max_join_size rows, it will issue an error. Note that the optimizer could be wrong as to how exactly how many rows it will need to examine, but it usually guesses within a very low margin of error. Also note that even if you are selecting from just one table, the process is still technically considered a join as far as the optimizer is concerned. It is a good idea to set a low max_join_size on a development system (to catch bugs early) or on systems where ad hoc queries are being run. The default is 4GB-1, or virtually infinite. To change the variable, use the --set-variable option.

max_sort_length: When you're performing an ORDER BY or GROUP BY on a BLOB or a TEXT field, sorting will be done based on the first max_sort_length bytes, which defaults to 1024, rather than the entire field. This restriction allows a much better memory utilization while rarely sorting in the wrong order. If you want higher precision, you can increase this value with the --set-variable option.

max_user_connections: Specifies the maximum number of connections an individual user is allowed to have opened simultaneously. If it is set to 0, there is no limit. The purpose of this feature is to limit possible abuse on a host where a large number of users are allowed to connect. The default value is 0; to change the value, use the --set-variable option.

max_tmp_tables: Reserved for future use. When implemented, this option will limit the maximum number of temporary tables a client is allowed to have opened at the same time. The default is 32, and you will be able to change the value with the --set-variable option.

max_write_lock_count: If the number of write table locks reaches max_write_lock_count, the next lock grant will favor read lock over write lock. The default is 4GB-1, or for practical purposes, virtual infinity. You can change the setting with the --set-variable option.

myisam_max_extra_sort_file_size: Some table operations such as ALTER, REPAIR, OPTIMIZE, and LOAD DATA INFILE require building or rebuilding of the keys. There are two methods of building an index: by creating a temporary file or by using a key cache. Using the temporary file is significantly faster, but may require a lot of disk space. This parameter is used to help determine which method to use based on the degree of variation in key-value lengths across the key. The default is 256, and you can set it with the --set-variable option. Note that the value is in megabytes.

myisam_max_sort_file_size: When rebuilding a key, you can either use a temporary file or the key cache. This value puts a limit on how large a temporary

file for index rebuild can get. Note that the value is in megabytes. The default is 2047; to change the setting, use the --set-variable option.

myisam_recover_options: The MySQL server supports a special option, --myisam-recover, which tells it to automatically repair corrupted MyISAM tables. The behavior of the recovery process depends on the setting of several flags: DEFAULT (1), BACKUP(2), FORCE(4), and QUICK(8). Each flag corresponds to a bit in the myisam_recover_options bitmask. The numbers in parentheses show the corresponding bit in the myisam_recover_options value. DEFAULT means the server will try to check the table and repair it if the server discovers that the table was not closed properly. BACKUP means that corrupted table will be backed up before repair is attempted, FORCE means that repair will proceed even if we are going to lose some data, and QUICK means we only check the key file and do not compare it with the data file. You can control the setting of myisam_recover_options with arguments to --myisam-recover. The following example illustrates the usage syntax: --myisam-recover=BACKUP,FORCE

myisam_sort_buffer_size: When rebuilding a key, the MySQL server allocates a buffer for sorting key values, the size of which is controlled by myisam_sort_buffer_size. The default is 8MB. It is recommended that you increase this value to about one-fourth of your total physical RAM. Large values of this buffer greatly improve ALTER TABLE performance. The same applies to REPAIR, OPTIMIZE, and LOAD DATA INFILE. To change the setting, use the --set-variable option.

net_buffer_length: MySQL allocates a buffer for network I/O. The initial size is net_buffer_length, but it will be expanded to max_allowed_packet if necessary. You may want to increase this value if the majority of your queries are very long. The default is 16KB; to increase it, use the --set-variable option.

net_read_timeout: To avoid denial-of-service attacks as well as potential resource problems on unreliable networks, MySQL puts a time limit on how long a network read can take. net_read_timeout is the value in seconds that a read of one MySQL client-server protocol packet must complete within before timing out. The default is 30; you want to increase it on slow networks. To change the setting, use the --set-variable option.

net_retry_count: If a network I/O operation fails "innocently"—in other words, the error code returned gives us a reason to hope that a simple retry may succeed—we retry instead of returning an error. However, in case our judgment of this error was wrong, we put a limit on how many times we are going to retry by using net_retry_count. The default is 10; to change the setting, use the --set-variable option.

net_write_timeout: Everything said under net_read_timeout applies here, except read should be replaced with write.

open_files_limit: If this variable is not set to 0, the server will ask the operating system to allow it to open open_files_limit files. Otherwise, the value requested is the larger of max_connections*5 and max_connections+table_cache*2. The default is 0; you can change it with the --set-variable option.

pid_file: Specifies the location of the file containing the pid value of the server process. The default is $datadir/`hostname`.pid. You can change it with the --pid-file option.

port: Specifies the TCP/IP port that the server will listen on. The default is 3306. You can change it with the --port option.

protocol_version: Displays the client-server protocol version. In MySQL 3.23, the version is 10. Newer versions have always been backward-compatible with the old ones, and the plan is to keep it this way.

record_buffer: In MySQL 4.0, this option has been renamed to read_buffer. When reading a record in a thread, the server allocates a read-ahead buffer and places other records in it. The recommended value is the size of a typical record times the number of examined records in a typical select. Do not set this value too large, or you may run out of memory if you have a lot of connections. The default is 128K; to change the setting, use the --set-variable option.

record_rnd_buffer: Specifies the buffer used for read-ahead caching when reading presorted records based on position. An increased value can speed up ORDER BY. Do not set this value too high without good reason because it is allocated per thread. The default is 128K; to change the setting, use the --set-variable option.

query_buffer_size: Not used. This option was originally added for the query cache code, but in MySQL 4.0 the query cache was implemented differently and does not use this setting.

safe_show_database: Specifies whether --safe-show-database has been enabled. It is set to OFF by default. If the setting has been enabled, issuing the SHOW DATABASES command shows only the databases that the current user has privileges on.

server_id: Contains a unique server id value used in replication. The concept is similar to that of an IP address. To set this value, use the --server-id option.

slave_net_timeout: Specifies how long a network I/O operation can take before timing out on a replication slave. It must be set on the slave; it has no effect on the master. The default is 3600, and to change the value, use the --set-variable option.

skip_locking: This option has confused some users in the past, and perhaps it should have been called disable_filesystem_locks from the very start. When opening a MyISAM table, the server can optionally acquire the file system locks on the table files. If this kind of locking is enabled, several server instances can share the data directory. If you are using only one server instance, which is the case on most installations, you should have skip-locking enabled by adding skip-locking to your my.cnf file.

skip_networking: Shows whether the server has been instructed with the --skip-networking option to not listen on a TCP/IP port. If you are not using remote connections, we recommend enabling this option for improved security.

skip_show_database: Shows whether the server is running with the --skip-show-database option. This option tells the server to give an error when the SHOW DATABASES command is issued. In some cases, this option can be used to improve security.

slow_launch_time: On some systems, thread creation is slow. To keep track of this, the status parameter Slow_launch_threads. slow_launch_time defines the number of seconds that a thread can take to be created before it is counted as slow. The default is 2; to change the setting, use the --set-variable option.

socket: Specifies the Unix socket that the MySQL server is listening on. The default depends on the distribution. To change the setting, use the --socket option.

sort_buffer: Specifies the buffer used for sorting record positions when performing GROUP BY and ORDER BY queries. An increased value improves performance of large ORDER BY and GROUP BY queries. Do not set the value too high, because it is allocated per thread. The default is 2MB; you can change the value with --set-variable.

sql_mode: MySQL can be forced to use an ANSI-compliant behavior with the --ansi option, or you can optionally alter some of its non-ANSI-compliant behavior into compliance. The sql_mode variable shows the bitmask of ANSI-compliance on a feature-by-feature basis. The features are identified as: REAL_AS_FLOAT (bit value 1; makes the REAL type equivalent to FLOAT instead of DOUBLE), PIPES_AS_CONCAT (bit value 2; makes || a string concatenation operator instead of logical OR), ANSI_QUOTES (bit value 4; makes " quote identifiers instead of literal strings; equivalent of `), INGORE_SPACE (bit value 8; allows a space between a function name), SERIALIZABLE (bit value 16; sets the default transaction isolation level to serializable), and ONLY_FULL_GROUP_BY (bit value 32; will not permit selecting a field in a GROUP BY query that is not also used in GROUP BY). You can change this option with the --sql-mode option. The following example illustrates the usage syntax: --sql-mode=ANSI_QUOTES,SERIALIZABLE

table_cache: The first time a table needs to be accessed, the server opens it by creating an in-memory table descriptor. After the table is not needed any more, instead of closing the table the server caches the table descriptor in the table cache, which speeds up subsequent queries using the table. The table_cache setting determines the maximum number of table descriptors that can be present in the table cache at one time. The default is 64. It is a good idea to set this value as high as possible. The only concern is that you may run out of file descriptors because each open MyISAM table requires two file descriptors—one for the data file, and one for the key file. To change this variable, use the --set-variable option.

table_type: Contains the default table type as set by the --default-table-type option. The setting defaults to MyISAM in the standard binaries.

thread_cache_size: On systems with slow thread creation, the server can avoid the thread creation overhead by caching threads after they have finished processing a connection for later reuse instead of allowing the thread to exit. No caching is done if thread_cache_size is set to 0. The default is 0; to change the setting, use the --set-variable option.

thread_stack: Limits the thread stack size. Each thread has to have its own stack space. The default is 128K. Sometimes, this default is not enough, but sometimes it is too much. On Linux, the setting of this variable has little effect—reducing the stack size below the LinuxThreads precompiled value will make MySQL crash when the thread_stack limit is reached as opposed to when the precompiled stack spacing default is reached. However, you cannot increase the stack size with the thread_stack setting. If you need to increase it on Linux, you have to recompile LinuxThreads with a different value of STACK_SIZE.

transaction_isolation: Shows the default transaction isolation level. The default is READ COMMITTED. Other possible values are READ UNCOMMITTED, REPEATABLE READ, and SERIALIZABLE. To set this value, use the --transaction-isolation option.

timezone: Shows the current time zone. The time zone setting is used in date and time functions and can be controlled with the --timezone option as well as the time zone-related environment variables used in libc.

tmp_table_size: In some cases, the server needs to create a temporary table to resolve a query. If the table has no BLOBs, the table will be in-memory (type HEAP). However, if the table reaches tmp_table_size, it will be converted to disk-based MyISAM. This may occasionally happen with large GROUP BY queries. If you have a lot of memory, you may want to increase the value of

tmp_table_size. The default is 32MB.

tmpdir: Specifies the directory used for disk-based temporary tables and other temporary storage. The default is /tmp. You can specify this setting with the --tmpdir option, with the environment variable TMPDIR on all platforms, and with the environment variables TMP and TEMP on Windows and OS/2.

version: Displays the server version.

wait_timeout: Specifies the number of seconds that a non-interactive client (any client that does not pass the INTERACTIVE_CLIENT option to mysql_real_connect()) will be allowed to stay connected without sending any commands. The default is 28800 (8 hours). You can lower this value if you have a lot of idle clients that make you run out of connections. You can change the value with the --set-variable option.

Additional Variables for MySQL 4.0

bulk_insert_buffer_size: Specifies the size of the buffer used to optimize MyISAM bulk inserts. The default is 8MB. Setting it to 0 disables the optimization of bulk inserts. Increasing the value will speed up large multirow inserts, INSERT ... SELECT, and LOAD DATA INFILE for MyISAM tables.

ft_min_word_len: Specifies the minimum length of a word to be indexed in a full-text key. Shorter words will not be indexed. Changing the value requires rebuilding all the full-text keys with the REPAIR TABLE command. The default is 4.

ft_max_word_len: Specifies the maximum length of a word to be indexed in a full-text key. Longer words will not be indexed. Changing the value requires rebuilding all the full-text keys with the REPAIR TABLE command. The default is 254.

ft_max_word_len_for_sort: When you are rebuilding a full-text index (e.g., during ALTER TABLE or REPAIR TABLE), an optimization can be applied for short words. This option allows you to define the cutoff for "short". If the cutoff is too small, the optimization does not get used when it should be. If the cutoff is too big, it backfires on long words actually slowing down the process. The default is 20.

ft_boolean_search_syntax: Contains a list of full-text search boolean operators. More information is available at www.mysql.com/doc/en/Fulltext_Search.html.

query_cache_limit: Specifies the maximum size of a result set to be cached in the query cache. The default is 1MB. The purpose of the limit is to prevent the query cache from being overloaded with large junk results that take up space

without improving server performance. For example, if the queries that your clients run repeatedly hardly ever return a result set that exceeds 30K, you should set query_cache_limit to 30K.

query_cache_size: Specifies the size of the query cache. By default, the value is 0 (query cache disabled). The larger the value, the more result sets can be cached. Set it to a non-zero value (e.g., 16MB) if your clients keep running the same queries over and over, and the tables involved in those queries do not change very frequently.

query_cache_type: Specifies the type of the query cache. Settings are as follows: 0 (OFF)--no cache; 1 (ON)--cache everything except queries with SELECT SQL_NO_CACHE; 2 (DEMAND)--cache only queries with SELECT SQL_CACHE. The default is 1 (ON).

Sometimes new variables are added even after the version has become stable. You can find the most current documentation of variables at www.mysql.com/doc/en/SHOW_VARIABLES.html.

Verifying Support for Variables

Another way to learn which options your current binary supports is to run mysqld --help and study the output. You should see something like Listing 14.2.

```
./mysqld  Ver 3.23.52 for pc-linux-gnu on i686
Copyright (C) 2000 MySQL AB, by Monty and others
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL license
Starts the MySQL server

Usage: ./mysqld [OPTIONS]

  --ansi            Use ANSI SQL syntax instead of MySQL syntax
  -b, --basedir=path     Path to installation directory. All paths
                         are usually resolved relative to this
  --big-tables           Allow big result sets by saving all
                         temporary sets on file (Solves most 'table
                         full' errors)
  --bind-address=IP      Ip address to bind to
  --bootstrap            Used by mysql installation scripts
  --character-sets-dir=...
                         Directory where character sets are
  --chroot=path          Chroot mysqld daemon during startup
  --core-file            Write core on errors
  -h, --datadir=path     Path to the database root
```

Listing 14.2 Output of mysqld --help. (continues)

```
--default-character-set=charset
                Set the default character set
--default-table-type=type
                Set the default table type for tables
--delay-key-write-for-all-tables
                Don't flush key buffers between writes for any
                MyISAM table
--enable-locking       Enable system locking
-T, --exit-info        Used for debugging; Use at your own risk!
--flush                Flush tables to disk between SQL commands
-?, --help             Display this help and exit
--init-file=file       Read SQL commands from this file at startup
-L, --language=...     Client error messages in given language. May
                       be given as a full path
--local-infile=[1|0]   Enable/disable LOAD DATA LOCAL INFILE
-l, --log[=file]       Log connections and queries to file
--log-bin[=file]       Log queries in new binary format (for
                       replication)
--log-bin-index=file   File that holds the names for last binary log
                       files
--log-update[=file]    Log updates to file.# where # is a unique
                       number if not given.
--log-isam[=file]      Log all MyISAM changes to file
--log-long-format      Log some extra information to update log
--low-priority-updates INSERT/DELETE/UPDATE has lower priority than
                       selects
--log-slow-queries=[file]
                       Log slow queries to this log file.  Defaults
                       logging to hostname-slow.log
--pid-file=path        Pid file used by safe_mysqld
--myisam-recover[=option[,option...]] where options is one of
                       DEAULT, BACKUP or FORCE.
--memlock              Lock mysqld in memory
-n, --new              Use very new possible 'unsafe' functions
-o, --old-protocol     Use the old (3.20) protocol
-P, --port=...         Port number to use for connection

-O, --set-variable var=option
                       Give a variable a value. --help lists
                       variables
-Sg, --skip-grant-tables
                       Start without grant tables. This gives all
                       users FULL ACCESS to all tables!
--safe-mode            Skip some optimize stages (for testing)
--safe-show-database   Don't show databases for which the user has
                       no privileges
--safe-user-create
```

Listing 14.2 Output of mysqld --help. (continues)

```
                      Don't new users cretaion without privileges
                      to the mysql.user table
--skip-concurrent-insert
                      Don't use concurrent insert with MyISAM
--skip-delay-key-write
                      Ignore the delay_key_write option for all
                      tables
--skip-host-cache     Don't cache host names
--skip-locking        Don't use system locking. To use isamchk one
                      has to shut down the server.
--skip-name-resolve   Don't resolve hostnames.
                      All hostnames are IP's or 'localhost'
--skip-networking     Don't allow connection with TCP/IP.
--skip-new            Don't use new, possible wrong routines.

--skip-stack-trace    Don't print a stack trace on failure
--skip-show-database  Don't allow 'SHOW DATABASE' commands
--skip-thread-priority
                      Don't give threads different priorities.
--socket=...          Socket file to use for connection
-t, --tmpdir=path     Path for temporary files
--sql-mode=option[,option[,option...]] where option can be one of:
                     REAL_AS_FLOAT, PIPES_AS_CONCAT, ANSI_QUOTES,
                     IGNORE_SPACE, SERIALIZE, ONLY_FULL_GROUP_BY.
--transaction-isolation
                      Default transaction isolation level
--temp-pool           Use a pool of temporary files
-u, --user=user_name  Run mysqld daemon as user
-V, --version         output version information and exit
-W, --warnings        Log some not critical warnings to the log
                      file
--bdb-home=  directory  Berkeley home directory
--bdb-lock-detect=#     Berkeley lock detect (DEFAULT, OLDEST,
                        RANDOM or YOUNGEST, # sec)
--bdb-logdir=directory  Berkeley DB log file directory
--bdb-no-sync           Don't synchronously flush logs
--bdb-no-recover        Don't try to recover Berkeley DB tables on
                        start
--bdb-shared-data       Start Berkeley DB in multi-process mode
--bdb-tmpdir=directory  Berkeley DB tempfile name
--skip-bdb              Don't use Berkeley db (will save memory)
--innodb_data_home_dir=dir   The common part for Innodb table spaces
--innodb_data_file_path=dir  Path to individual files and their
                             sizes
--innodb_flush_method=#  With which method to flush data
--innodb_flush_log_at_trx_commit[=#]
```

Listing 14.2 Output of mysqld --help. (continues)

```
 Set to 0 if you don't want to flush logs
  --innodb_log_arch_dir=dir    Where full logs should be archived
  --innodb_log_archive[=#]     Set to 1 if you want to have logs
                               archived
  --innodb_log_group_home_dir=dir  Path to innodb log files.
  --skip-innodb                Don't use Innodb (will save memory)
Default options are read from the following files in the given order:
/etc/my.cnf /usr/local/mysql/var/my.cnf ~/.my.cnf

The following groups are read: mysqld server
The following options may be given as the first argument:
--print-defaults      Print the program argument list and exit
--no-defaults         Don't read default options from any options file
--defaults-file=#      Only read default options from the given file #
--defaults-extra-file=# Read this file after the global files are read

To see what values a running MySQL server is using, type
'mysqladmin variables' instead of 'mysqld --help'.
The default values (after parsing the command line arguments) are:

basedir:     /usr/
datadir:     /var/lib/mysql/
tmpdir:      /tmp/
language:    /usr/share/mysql/english/
pid file:    /var/lib/mysql/mysql.pid
binary log:
binary log index:
TCP port:    3306
Unix socket: /var/lib/mysql/mysql.sock

system locking is not in use

Possible variables for option --set-variable (-O) are:
back_log              current value: 50
bdb_cache_size        current value: 8388600
bdb_log_buffer_size   current value: 0
bdb_max_lock          current value: 10000
bdb_lock_max          current value: 10000
binlog_cache_size     current value: 32768
connect_timeout       current value: 2
delayed_insert_timeout  current value: 300
delayed_insert_limit  current value: 100
delayed_queue_size    current value: 1000
flush_time            current value: 0
innodb_mirrored_log_groups  current value: 1
innodb_log_files_in_group  current value: 2
innodb_log_file_size  current value: 5242880
```

Listing 14.2 Output of mysqld --help. (continues)

```
innodb_log_buffer_size  current value: 1048576
innodb_buffer_pool_size  current value: 8388608
innodb_additional_mem_pool_size  current value: 1048576
innodb_file_io_threads  current value: 4
innodb_lock_wait_timeout  current value: 50
innodb_thread_concurrency  current value: 8
innodb_force_recovery  current value: 0
interactive_timeout   current value: 28800
join_buffer_size      current value: 131072
key_buffer_size       current value: 16773120
long_query_time       current value: 1
lower_case_table_names  current value: 0
max_allowed_packet    current value: 2096128
max_binlog_cache_size  current value: 4294967295
max_binlog_size       current value: 1073741824
max_connections       current value: 2000
max_connect_errors    current value: 10
max_delayed_threads   current value: 20
max_heap_table_size   current value: 16777216
max_join_size         current value: 4294967295
max_sort_length       current value: 1024
max_tmp_tables        current value: 32
max_user_connections  current value: 0
max_write_lock_count  current value: 4294967295
myisam_max_extra_sort_file_size  current value: 256
myisam_max_sort_file_size  current value: 2047
myisam_sort_buffer_size  current value: 8388608
net_buffer_length     current value: 16384
net_retry_count       current value: 10
net_read_timeout      current value: 30
net_write_timeout     current value: 60
open_files_limit      current value: 0
query_buffer_size     current value: 0
record_buffer         current value: 131072
record_rnd_buffer     current value: 0
slave_net_timeout     current value: 3600
slow_launch_time      current value: 2
sort_buffer           current value: 2097144
table_cache           current value: 500
thread_concurrency    current value: 10
thread_cache_size     current value: 200
tmp_table_size        current value: 33554432
thread_stack          current value: 131072
wait_timeout          current value: 28800
```

Listing 14.2 Output of mysqld --help. (continued)

If you find an option that interests you but unfortunately has not yet been documented, or perhaps has not been documented as clearly as you would like it to be, you can take advantage of the open source nature of MySQL and study the source. You should begin by opening the file sql/mysqld.cc in the source distribution and searching for the name of the option you are interested in.

Operating System Adjustments

The most important thing on the operating system level for MyISAM tables is to make sure that disk caching is configured fairly aggressively. The MyISAM handler is purposefully conservative in the buffer allocations with the idea that the operating system cache will do the remainder of the necessary caching. On the other hand, the InnoDB handler does not have much faith in the operating system and caches everything that it can in its own buffers.

If you are using MyISAM tables, the file system that the data directory is on can make a bit of a difference. If you are not satisfied with MyISAM's performance on your system, you may want to try different file systems to see which one is better suited for your application.

Some operating systems support mounting with the noatime option, or setting a special attribute on certain files on a directory that tells the kernel not to update the access time on this file when the data is read. This can be helpful on a system that performs frequent small reads from a file. You could try running chattr +A /var/lib/mysql (or wherever your data directory happens to be) to see if that gives you any improvement in performance.

With InnoDB tables, you can try putting the table space on a raw device. Don't expect dramatic gains in performance in this case, but if the tables are significantly larger than the size of RAM you could see a difference.

If you are putting the data directory on an NFS volume, I strongly recommend using the InnoDB handler because of the caching issues. Since MyISAM depends on the operating system cache, it falls prey to the NFS surprises when in certain situations the data cannot be cached locally and has to be written immediately across the network.

It is important to ensure that MySQL gets enough operating system resources to do the job. The resources you should pay particular attention to are the number of open files and the number of child processes. The latter applies only to the operating systems that count threads as true child processes (for example, Linux). If you are running MySQL on a dedicated server, you may want to increase the priority of the MySQL process to prevent the operating system

from throttling it down. You must also make sure there are no unreasonable restrictions on how much RAM it is allowed to allocate.

Hardware Upgrades

If you find that after all the application, server configuration, and operating system tuning the server still does not give you satisfactory performance, you may consider updating your hardware. It is very important to understand what your bottleneck is—and what causes it—before you do an upgrade.

Perhaps the most common bottleneck that makes the DBA start thinking about a hardware upgrade is heavy disk I/O. This problem should be first addressed by getting more RAM, and only then by improving the disks. Again, I will refer to our trip from New York to Los Angeles analogy: If you want to improve your travel time, you will get better results by increasing the portion of the way that you travel by plane and decreasing the portion that you travel on foot, as opposed to increasing your foot speed.

If disk I/O is not your bottleneck, and you would like to improve performance with a hardware upgrade, you may consider buying a CPU with a larger cache. At this point, I do not have enough data to know when and if getting another CPU will give you a larger performance increase than simply getting a faster CPU. Give both options given equal consideration, and let the nature of the application and the type of the platform it runs on influence your decision.

Here's a common mistake a number of people have made: They migrated from x86 to a reduced instruction set computing (RISC) architecture when MySQL could not handle the load on x86. The migration usually resulted in reduced performance because x86 has a much higher "bang per buck" value than RISC. In fact, I must admit that I have not logged in to a RISC system that could outperform a single-processor AMD 1.3GHz running Linux. This is not because I do not get to use many RISC systems—the ones that would outperform it are simply so expensive and therefore so rare among MySQL AB customers that MySQL AB staff does not get to play with them.

CHAPTER 15

Analyzing and Improving Server Performance

In this chapter, we describe how you can take the pulse of your server—and we suggest some treatments in case you find its heartbeat is faltering. We also provide some suggestions for keeping your server healthy and avoiding fatalities, even when something goes terribly wrong in the middle of activity on your production system.

Staying Out of Trouble

The best defense is a strong offense; you will be much more successful in keeping your server working optimally if you proactively prevent problems rather than reactively patch holes. First, let's review a few key principles for giving our server and application a solid foundation:

- Invest in a solid test/development environment. Build a test system that you can populate with production data and run production-type loads on.
- Maintain a test suite for your application--both functionality and performance.
- Do not wait for your data volume to reach a critical point and reduce performance. Detect potentially slow queries during development; you can do this by enabling --log-slow-queries and --long-log-format during development and policing the slow log. You should account for every query that hits the slow log during development.
- Make sure your tables have proper keys, and are otherwise properly designed. See Chapter 13 an in-depth discussion of table design.

- Choose the proper table type. For example, for read-only tables, use MyISAM. For high-concurrency read-write tables, use InnoDB.
- Anticipate growth. Test your application under the maximum possible production load with the largest possible dataset, and address any exposed issues that arise.
- Treat MySQL Server as your own system component as opposed to a perfect third-party tool that cannot possibly fail. When you upgrade MySQL, rerun your application test suite even if nothing has changed in your application.
- Do not be in a hurry to use new MySQL versions or new features on your production system—but try them on the test system first. Once you've ensured that everything functions correctly, integrate the new features into your production system, although you should be prepared to revert to the old setup in case anything goes wrong.
- Log all query failures and investigate each one. You may also want to introduce heuristics to detect abnormal behavior for certain queries—for example, too few or too many rows are returned for the given context. For example, you can include in your application a safe_query() function that will accept the maximum amount of time for the query to run before MySQL issues an alert.
- Perform frequent backups. See Chapter 17 for details on how to set up backups.
- Perform various checks on your backups. Check table integrity with the CHECK TABLE command as a start. You may also want to check data-integrity issues specific to your application. If a check fails, investigate the matter thoroughly.
- Have a cron job that monitors your server, not only the actual connectivity, but also the output of SHOW STATUS and SHOW PROCESSLIST commands for anything abnormal. Investigate all abnormality reports.
- Monitor the error log. Investigate all errors.
- Take good care of the operating system. Make sure you know which components MySQL depends on, and that each of those components is reliable. Those components always include the kernel and, for dynamically linked binaries, system libraries that you can list by running ldd on mysqld on a Unix system.
- If you are doing your own builds, ensure that your build system is configured properly and that you are using proper compiler flags for your platform.
- Make sure your hardware is solid. Before you put the system into production, run available hardware checks on your hardware, especially on the

RAM and hard disk. Follow some common sense rules for hardware maintenance, such as connecting your system to a robust uninterruptible power supply (UPS) and installing software on the server that can gracefully deal with power outages.

- Build a reasonable measure of redundancy into the system. If circumstances allow, maintain a hot spare server that replicates from the production system and is ready to take over in case the primary server fails.

These guidelines do not guarantee that you will never have trouble, but they will reduce the chance of problems. When problems do happen, you will very likely detect and fix them early before they cause any real damage.

Dealing with Slow Performance

When server performance begins to decline, you can take several measures to identify the cause. The first thing is to run SHOW FULL PROCESSLIST and identify some potentially problematic queries that are currently running. Here's a sample output:

```
Id  User   Host       db    Command  Time   State   Info
1   sasha  localhost  test  Sleep    21367  NULL
8   sasha  localhost  test  Query    2436   Locked  lock tables
                                                   employee read
9   root   localhost  test  Sleep    2438   NULL
32  root   localhost  NULL  Query    0      NULL    SHOW FULL
                                                   PROCESSLIST
```

The first column shows the connection id. If your application is careful to log the connection id, you can use this id to track down the problem. You can accomplish this in C and in newer versions of PHP by using mysql_thread_id(). In Perl you can use $dbh->{"thread_id"}. In older versions of PHP and in other languages, you can get the id by using SELECT CONNECTION_ID(). The connection id is also printed in the general log (which you enable with the --log option). You can also use this value to kill a thread by using the KILL command to manually terminate a long-running query.

The User and Host fields display the name of the thread owner and the host from which the connection came. If you catch the thread in the process of authenticating the user, you will see "unauthenticated user" in the User field. This is normal and should not give you a reason for concern if the subsequent run of SHOW FULL PROCESSLIST gives you the name of the user for that thread. However, if the authentication process is taking longer than a second, this indicates a problem--usually, it means that your clients are running very slowly or that the network connection between the client and the server is poor.

The db field shows the default database for the given thread: It may be empty. The Command field displays the current database command the thread is executing. A regular query thread will usually have this set to either Sleep, which means it is waiting to receive a query to execute, or Query, which means it is in the middle of executing a query.

The Time field shows the number of seconds the command has been in progress. A sleeping thread may have a very high value—for example, 600 seconds or higher. This is normal for systems with persistent connections. However, if you are not aware of any persistent connections on your server and you see a thread that has been sleeping for several seconds, this should be a cause for concern. Most likely you have a "secret" daemon process that forgot to disconnect from MySQL after finishing its work. If you want to force automatic disconnects for such processes, lower the value of wait_timeout. Another possibility is that you have accidentally enabled persistent connections in a client. If a thread shows that it is running a Query command and the Time value is very high, this is also an obvious cause for alarm.

The State field is mostly useful for someone familiar with the MySQL source, such as a MySQL support engineer, community MySQL guru, or your local MySQL expert who likes to dig through the source code. The State field corresponds to the value of the proc_info member of the THD structure defined in sql/sql_class.h. Instances of THD are called thd in most places in the code, so if you are interested in becoming an expert on different values of the State field, you should look through the source code in the sql directory for thd->proc_info and then study the surrounding code. The value of State shows how far the execution of a query has progressed.

Finally, the Info field contains the name of the currently running query.

Because the output of SHOW FULL PROCESSLIST may be difficult to read on a loaded server with many connections, have a troubleshooting script handy that will help you quickly identify the culprit. Listing 15.1 contains a sample script in Perl that will show you the top 10 threads with the longest running queries. (You can also download the script from the book Web site--www.wiley.com/compbooks/pachev. It is under the name slow.pl.)

```
#! /usr/bin/perl

use strict;
use DBI;
use vars qw($dbh $user $host $db $pass $dsn);
```

Listing 15.1 The slow.pl script. (continues)

```
my $dbh;
my ($user,$host,$db,$pass,$dsn);

# configuration settings for database connectivity
$host = "localhost";
$user = "root";
$db = "test";
$pass = "";
$dsn = "dbi:mysql:$db:host=$host";

sub dbi_die
  {
    my $msg = shift;
    #We need to save the error message as the cleanup will destroy it.
    #Alternatively, we could have printed the message, done the cleanup, and
    #then exited.
    my $dbi_msg = $DBI::errstr;
    do_cleanup();
    die("FATAL ERROR: $msg: DBI message: $dbi_msg}\n");
  }

sub do_cleanup
  {
    if (defined($dbh))
      {
     $dbh->disconnect();
      }
  }

sub safe_connect
  {
    ($dbh = DBI->connect($dsn,$user,$pass,{PrintError => 0}))
      || dbi_die("Could not connect to MySQL");
  }

sub safe_query
  {
    my $sth;
    my $query = shift;
    $sth = $dbh->prepare($query);
    $sth->execute(@_) || dbi_die("Could not execute '$query'");
    return $sth;
  }

#sort comparison function
sub by_time
  {
```

Listing 15.1 The slow.pl script. (continues)

```
    $b->{"Time"} <=> $a->{"Time"};
  }

safe_connect();
my ($sth,$row,$res);
$sth = safe_query("SHOW FULL PROCESSLIST");

$res = $sth->fetchall_arrayref({});

@$res = sort by_time @$res;

$sth->finish();
$dbh->disconnect();

print "Id\tTime\tCommand\tState\tQuery\n";

my($i);

for ($i = 0; $i < 10; $i++)
  {
    last unless (($row = shift @$res));
    next if ($row->{"Command"} ne "Query");
    print $row->{"Id"}."\t".$row->{"Time"}."\t".$row->{"Command"}."\t".
      $row->{"State"}."\t".$row->{"Info"}."\n";
  }
```

Listing 15.1 The slow.pl script. (continued)

Using the EXPLAIN Command

Once you have identified some long-running queries, you may want to run the EXPLAIN command on each to see how the optimizer is processing it. The output will disclose what keys are being used, if any; which way they are going to be used; what other keys can be used; and how many rows the optimizer thinks it will have to examine for each table, among other helpful pieces of information.

Let's illustrate the practical use of the EXPLAIN command with an example. Suppose we have a table, web_downloads, with the following structure (as shown by SHOW CREATE TABLE):

```
CREATE TABLE `web_downloads` (
 `host` char(128) default NULL,
 `ts` datetime NOT NULL default '0000-00-00 00:00:00',
 `ver` tinyint(4) default NULL,
 `branch` tinyint(4) default NULL,
 `release` tinyint(4) default NULL,
```

```
  `desc_tag`
enum('server','shareware','shared','devel','client','other') default
'other',
  `arch` enum('x86','sparc','hp','sgi','alpha','powerpc','ia64','other')
default 'other',
  `os` enum('linux','solaris','hpux','irix','win','freebsd',
'macosx','sco','other') default NULL,
  `src_or_bin` enum('src','bin','unknown') default 'unknown'
) TYPE=MyISAM
```

And suppose we have the following query as the unlucky "winner" in the output of our slow query monitor script:

```
SELECT count(*) FROM web_downloads WHERE arch = 'ia64'
```

We immediately check what is wrong with it:

```
EXPLAIN SELECT count(*) FROM web_downloads WHERE arch = 'ia64'
```

which gives us the following information:

```
table          type  possible_keys  key   key_len  ref   rows    Extra
web_downloads  ALL   NULL           NULL  NULL     NULL  600000 where used
```

The table column shows the name of the table. In our query, this value seems rather redundant because we have only one table. It becomes relevant, though, when we are joining two or more tables.

The type column identifies the method used to retrieve the relevant records from the table. The optimizer uses the method that it thinks will pull out the minimum superset of the table needed to match the conditions of the query—which in the ideal case consists of only the records that the user requested. However, it is not always possible to pull only the records that the user wants immediately—we retrieve the superset of records that will contain the query result, and then filter out those that do not satisfy the query conditions.

The possible values of type are

- **system:** This value is possible only for a table with one or no records, a special case of const.
- **const:** The table has one matching row, which will be read in the initial stage of query processing.
- **eq_ref:** Several records will be retrieved on a unique or primary key using all of its parts, one at a time as the adjacent—in terms of join—table records are being processed.
- **ref:** Uses the same logic as eq_ref, except that the key is either not unique or you do not have the values of all the parts in the lookup, so you retrieve more than one row for each match.

- **range:** You retrieve all of the records for the known key range.
- **fulltext:** A full-text key will be used.
- **index:** You scan all the key values in the table, but you do not touch the data.
- **ALL:** You scan the entire table.

On the scale of 1 to 10, with 10 being the highest score, give your query a 10 if you see const, a 9 for eq_ref, from 4 to 8 for a ref depending on the number of matches, from 3 to 7 for range, from 1 to 3 for index, and a definite 1 for ALL. If you see system, you should ask yourself why you have a table with only one record, or with no records at all, in your database. This should perhaps be counted as -1 for the score. Although rather informal and quite unscientific, this scale should give you a rough feel for the efficiency of your query. It is, of course, important to realize that some queries have no way of being optimized; they will always have the low-score types in the EXPLAIN command output. Such queries should be avoided on heavily loaded systems and instead directed to a replication slave dedicated to running slow queries.

In our case, the type is ALL because we have no keys at all. We make a note of that so we can think about how we can correct the problem.

The possible_keys value shows the keys that the optimizer thinks can be used. In our case, there are no possible keys. In some cases, however, you can have more than one key that the optimizer could use to retrieve the records for the table.

The key column shows which key the optimizer will actually use, which in our case is NULL, meaning that no key will be used. If you see that the optimizer is using a key and the performance is still unsatisfactory, you can give the optimizer a hint to use a different key with the USE KEY directive. For example:

```
SELECT  employee.fname, employee.lname, employee_paycheck.pay_date,
employee_paycheck.amount FROM employee USE KEY(ssn),
employee_paycheck USE KEY(id) WHERE employee.id =
employee_paycheck.emp_id AND employee.ssn = '111222333';
```

Although the optimizer tries its best to predict which key will require the least amount of work, and is usually right, it occasionally makes a bad decision. USE KEY corrects it, but you must give the name of one of the possible keys for each table. Although our example provided explicit key names for both tables in the join, it is not necessary to do so—you can use it on only some of the join tables.

The USE KEY syntax is specific to MySQL; it has the potential to create problems if you run your application on other databases. However, if you are sure that your application is always going to run on MySQL, it is a good idea to employ USE KEY on all tables and in all queries, or document in the comments otherwise that the query will use no keys. This will force you to think about

what keys your query is going to use, or whether it is going to use them at all, and thus will greatly reduce the chance of writing suboptimal queries. The additional advantages are that the optimizer will never be tempted to use the wrong key and that the query will go slightly faster because the optimizer will not spend the extra time wondering which key to use.

The key_len column shows the length of the part of the key (or the whole key) that is going to be used to look up the records. In an indirect way, it shows what part of the key will be used.

The ref column displays the names of the reference keys for each of the key parts of the key that is used to resolve the query. When tables are joined, the join happens in the order decided by the optimizer. You can force a specific order with the STRAIGHT JOIN operator instead of the comma that separates the join tables, but this is usually not necessary. The records in the first table are looked up based on some constants provided in the WHERE clause, or perhaps in the unfortunate case when no keys can be used, the optimizer does a full scan and uses no reference values at all. However, each subsequent table in the join order (depending on the kind of query and key used) may utilize the values of the records in the columns of the preceding tables for the lookup. The ref column shows which of those columns were utilized.

In some cases, a constant value is used for key lookup. If that is the case, the ref value will be const. If the key consists of several parts, the reference columns will be displayed for each part separately. ref is applicable only to const, eq_ref, and ref types, and in the case of const it is always const. Other record lookup types that do not have the concept of lookup reference (system, range, index, and ALL) will display NULL. In our case, it is NULL because we do not have any keys and are using an ALL lookup.

The rows column contains the number of records the optimizer believes it will have to read from the table. Note that this is only an estimate, so it will be wrong in some cases. If the estimate is considerably off, the optimizer may choose the wrong key. An inaccurate rows estimate can also affect the "hog" trigger set off by the value of max_join_size, making it either too aggressive or too lenient. Usually the margin of error is within 10 percent, which means that in most cases the optimizer will behave as if the estimate were perfect. To reduce the chances of an estimate error, you should periodically run ANALYZE TABLE, which updates the key distribution statistics serving as a base for the estimate.

You should do everything possible to ensure that the number of rows examined for each table is as small as possible, especially if you are joining several tables. The total number of row combinations the optimizer would have to examine is the product of the numbers of rows for each table, and it can get ugly very fast.

In our case, even with one table the number of rows is already 600,000, which is not good.

The Extra column provides room for the optimizer to pass along some non-numeric information, sometimes by naming a key or by picking one of the predefined types. It writes a little essay telling you—perhaps with a bit of a Swedish accent—what is happening as the query is being processed. Here are some common expressions it can use:

- **where used:** This means that during a join the WHERE clause was used to eliminate some rows from the possible result subset. If you don't see this, it means that either the optimizer "cheated" by doing a const or eq_ref lookup or possibly some other clever trick, or it scanned the entire table and could not eliminate any rows by evaluating the expression in the WHERE clause.
- **range checked for each record (index map: #):** This means that MySQL did not find a good key to look up rows in this table; however, MySQL specifies that, depending on the values of the preceding table records in the join order, it might sometimes be possible to use a key in this table. This is a rather uncommon condition, but if you see this, you should ask yourself why it happened and how to make things better so that MySQL can find a key to use for this table.
- **Using index:** This means that the data for the table was retrieved completely by using the index without accessing the data records. This approach is usually a plus, although in some rare cases it can be a detriment if the optimizer erroneously decides that the index scan would be faster than reading the data. The advantage of scanning the index as opposed to accessing the data records is that the index scan will require less I/O. However, the disadvantage is that it requires more CPU resources.
- **Not exists:** This means that the "not exists" optimization technique was applied in processing a LEFT JOIN command. This means that the data in the table did not have to be examined at all because the optimizer deduced from the WHERE clause that no data matching the condition would be found in that table. Therefore, only the row of NULL values will be included in the join result from the table. This is not a very common optimization, and if you see it in the EXPLAIN command output, although it is possible that everything is fine, you should verify that your LEFT JOIN is written correctly and actually does what you intended it to do.
- **Using temporary:** This means that a temporary table had to be created, which is a warning for the thoughtful DBA. You should try to figure out a way to make the query work without a temporary table, although this is not always possible. If the use of a temporary table cannot be avoided, you should consider ways to reduce the frequency with which this query is run.

- **Using filesort:** The term *filesort* refers to the optimizer technique of retrieving records in the requested order (needed for resolving ORDER BY). First, the matching records are found and their positions are recorded. Then the positions are sorted according to the value of ORDER BY in the records. Then the records are read based on position in the sorted order of positions. This is not a very efficient way to do ORDER BY—it is much better if we can just go through the index and retrieve the records in the correct order. If you see this message in your EXPLAIN output, you should consider adding a key that will help you do ORDER BY by iterating through the keys as opposed to doing filesort. Note that this must be the same key that is being used to satisfy the WHERE conditions.
- **Distinct:** This means that the optimizer noticed that because we were doing SELECT DISTINCT, the table can be eliminated from the join completely because no records from it were included in the result. If you see this, you probably have made a mistake in writing the query and included an unnecessary table.

In our case, the optimizer writes no essays, because it simply is doing a full scan.

Note that EXPLAIN works only for SELECT queries. UPDATE and DELETE queries should be transformed into SELECT queries with the same WHERE clause first.

Now that we have some understanding of what the output of EXPLAIN actually means, we know what the problem is with our query. In this case it is very simple: The query is using no keys. Since it is doing a lookup on arch , it would be a good idea to add a key on this field. Let's try the following command:

```
ALTER TABLE web_downloads ADD KEY (arch);
```

Now let's look at the results of EXPLAIN:

```
table type    possible_keys  key     key_len ref     rows    Extra
web_downloads       ref     arch    arch    2       const   433     where
used; Using
index
```

Much better. Now we can use our newly created key, and we have to examine only 433 rows instead of 600,000. To make things better, we never have to touch the data records—we stay only inside the index.

Using the mysqldumpslow Script

While picking the trouble queries from the SHOW FULL PROCESSLIST output is helpful in times of emergency, a more methodical approach would be to run the server with –long-log-format and --log-slow-queries enabled and police the

slow query log for signs of trouble. You may try using the mysqldumpslow script contributed by Tim Bunce, who is the author of DBI and a very active MySQL user. mysqldumpslow is distributed as part of MySQL. Let's demonstrate a few examples of using mysqldumpslow. First, the most straightforward example; no special options:

```
mysqldumpslow
```

We get the following output:

```
Reading mysql slow query log from /var/lib/mysql/mysql-slow.log
Count: 1  Time=10.00s (10s)  Lock=0.00s (0s)  Rows=1.0 (1),
root[root]@localhost
  SELECT count(*) FROM web_downloads WHERE os IN ('S','S')

Count: 2  Time=8.50s (17s)  Lock=0.00s (0s)  Rows=1.0 (2),
root[root]@localhost
  SELECT count(*) FROM web_downloads WHERE os = 'S'
```

By default, mysqldumpslow groups the queries and abstracts their parameters. This is quite reasonable because most applications construct queries on the fly. This kind of grouping and categorization will make it easier for the database programmer to identify the part of the code that is generating slow queries. If you want to see individual queries, you can run

```
mysqldumpslow -a
```

and get the following:

```
Reading mysql slow query log from /var/lib/mysql/mysql-slow.log
Count: 1  Time=10.00s (10s)  Lock=0.00s (0s)  Rows=1.0 (1),
root[root]@localhost
  SELECT count(*) FROM web_downloads WHERE os IN ('Windows','Linux')

Count: 1  Time=9.00s (9s)  Lock=0.00s (0s)  Rows=1.0 (1),
root[root]@localhost
  SELECT count(*) FROM web_downloads WHERE os = 'Linux'

Count: 1  Time=8.00s (8s)  Lock=0.00s (0s)  Rows=1.0 (1),
root[root]@localhost
  SELECT count(*) FROM web_downloads WHERE os = 'Windows'
```

Other helpful options to mysqldumpslow are

- **-t <*number*>:** Shows the top <*number*> of queries.
- **-s <*sorttype*>:** Sorts queries by <*sorttype*>, which can be *t* (total time), *at* (average time), *l* (total lock waiting time), *al* (average lock waiting time), *r* (total records returned), or *ar* (average records returned).
- **-g <*substring*>:** Includes only the queries containing <*substring*>.
- **-r:** Indicates a reverse sort order.

- **-n <*num_digits*>:** Abstracts names containing at least <*num_digits*> digits.
- **-l:** Does not subtract the lock waiting time from the total time.

Another useful tool that can be used for analyzing the slow log is query-wizard, which is available on the book Web site (www.wiley.com/compbooks/pachev). We discuss query-wizard in Chapter 12.

The most important part to remember about optimizing a query is the bottom line—it must run fast, and the speed of execution is primarily affected by the number of records or record combinations the optimizer has to examine. Therefore, when optimizing a server, begin with queries that examine the highest number of records; then move down to the query with the next highest degree of inefficiency, and so on. For example, you may want to start with queries that examine more than 10,000 records. Then move on to the ones that examine between 5,000 and 10,000 records, then into the 1,000-5,000 range, and so on. If a query examines all records in a table (the most inefficient way to do it from the algorithmic point of view) but the table is small (fewer than 500 records), it is not as critical to optimize it. Algorithmic efficiency (e.g., using the right key), however, is important during the design stage for queries on tables that you expect to become very large.

Monitoring Server Activity Patterns with SHOW STATUS

In the previous section, we discussed methods of direct attack on potentially or already problematic queries. Next, we approach the issue of server performance from a different angle. Instead of targeting particular queries, we look at how to feel the pulse of the server—to get a general idea of what it is doing, what kind of load it is under, what kind of optimizations it is performing (and not performing), how much data is coming in and going out, how often the clients connect, and so forth. All of this information can be obtained from the output of the SHOW STATUS command, as shown in Listing 15.2.

```
Variable_name     Value
Aborted_clients     0
Aborted_connects     0
Bytes_received     67
Bytes_sent     2590
Com_admin_commands     0
Com_alter_table     0
```

Listing 15.2 SHOW STATUS output. (continues)

```
Com_analyze     0
Com_backup_table     0
Com_begin     0
Com_change_db     0
Com_change_master     0
Com_check     0
Com_commit     0
Com_create_db     0
Com_create_function     0
Com_create_index     0
Com_create_table     0
Com_delete     0
Com_drop_db     0
Com_drop_function     0
Com_drop_index     0
Com_drop_table     0
Com_flush     0
Com_grant     0
Com_insert     0
Com_insert_select     0
Com_kill     0
Com_load     0
Com_load_master_table     0
Com_lock_tables     0
Com_optimize     0
Com_purge     0
Com_rename_table     0
Com_repair     0
Com_replace     0
Com_replace_select     0
Com_reset     0
Com_restore_table     0
Com_revoke     0
Com_rollback     0
Com_select     0
Com_set_option     0
Com_show_binlogs     0
Com_show_create     0
Com_show_databases     0
Com_show_fields     0
Com_show_grants     0
Com_show_keys     0
Com_show_logs     0
Com_show_master_status     0
Com_show_open_tables     0
Com_show_processlist     0
Com_show_slave_status     0
```

Listing 15.2 SHOW STATUS output. (continues)

```
Com_show_status     2
Com_show_tables     0
Com_show_variables     0
Com_slave_start     0
Com_slave_stop     0
Com_truncate     0
Com_unlock_tables     0
Com_update     0
Connections     3
Created_tmp_disk_tables     0
Created_tmp_tables     0
Created_tmp_files     0
Delayed_insert_threads     0
Delayed_writes     0
Delayed_errors     0
Flush_commands     1
Handler_delete     0
Handler_read_first     1
Handler_read_key     0
Handler_read_next     1
Handler_read_prev     0
Handler_read_rnd     0
Handler_read_rnd_next     66
Handler_update     0
Handler_write     0
Key_blocks_used     1
Key_read_requests     2
Key_reads     1
Key_write_requests     0
Key_writes     0
Max_used_connections     0
Not_flushed_key_blocks     0
Not_flushed_delayed_rows     0
Open_tables     0
Open_files     3
Open_streams     0
Opened_tables     6
Questions     3
Select_full_join     0
Select_full_range_join     0
Select_range     0
Select_range_check     0
Select_scan     0
Slave_running     OFF
Slave_open_temp_tables     0
Slow_launch_threads     0
Slow_queries     0
```

Listing 15.2 SHOW STATUS output. (continues)

```
Sort_merge_passes     0
Sort_range     0
Sort_rows     0
Sort_scan     0
Table_locks_immediate     5
Table_locks_waited     0
Threads_cached     0
Threads_created     1
Threads_connected     1
Threads_running     1
Uptime     40000
```

Listing 15.2 SHOW STATUS output. (continued)

Now let's discuss each variable in detail. We use the terms query and command interchangeably, although we prefer *query* when we are talking about a traditional SQL query and *command* when we are telling the database to perform some administrative function through the SQL parser interface. As far as MySQL is concerned, there is no difference in how the commands or queries are processed internally, and from the point of view of the application programmer, all SQL-parser-based administrative commands can be sent as queries.

Aborted_clients: When a client loses the connection to the server because of a network problem, or when it has been idle for longer than wait_timeout seconds or interactive_timeout for interactive clients, the Aborted_clients counter is incremented. Two common reasons for the value to be high are clients that keep up idle connections and clients that disconnect without calling mysql_close(). If this value is high (relative to Connections), double-check your client code to make sure that it does not crash and that it does call mysql_close() (or $dbh->disconnect() in Perl) at the end in case of normal termination (and especially when it terminates because it encountered an error).

Aborted_connects: This counter is incremented when the client is taking longer than connect_timeout seconds to respond during the authentication handshake process. If this is high (relative to Connections), it could be because of network problems, a severely overloaded client, or an attempted denial-of-service attack.

Bytes_received: Specifies the total number of bytes received from all clients. Once 4GB is reached, the counter will wrap around.

Bytes_sent: Specifies the total number of bytes sent to clients. Once 4GB is reached, the counter will wrap around.

Com_admin_commands: Specifies the total number of miscellaneous commands not covered by other command categories. Currently includes

replication commands issued from the slave, ping, debug, and shutdown. Will very possibly be split into separate categories in the future.

Com_alter_table: Specifies the total number of ALTER TABLE commands.

Com_analyze: Specifies the total number of ANALYZE TABLE commands.

Com_backup_table: Specifies the total number of BACKUP TABLE commands.

Com_begin: Specifies the total number of BEGIN commands. BEGIN explicitly starts a transaction.

Com_change_db: Specifies the total number of commands to change the database (USE db_name).

Com_change_master: Specifies the total number of CHANGE MASTER TO commands (used on replication slaves).

Com_check: Specifies the total number of CHECK TABLE commands.

Com_commit: Specifies the total number of COMMIT commands. COMMIT makes permanent the changes performed by the transaction so far since its start.

Com_create_db: Specifies the total number of CREATE DATABASE commands.

Com_create_function: Specifies the total number of CREATE FUNCTION commands (used when loading a user-defined function [UDF]).

Com_create_index: Specifies the total number of CREATE INDEX commands.

Com_create_table: Specifies the total number of CREATE TABLE commands.

Com_delete: Specifies the total number of DELETE queries.

Com_drop_db: Specifies the total number of DROP DATABASE commands.

Com_drop_function: Specifies the total number of DROP FUNCTION commands (used when unloading a UDF).

Com_drop_index: Specifies the total number of DROP INDEX commands.

Com_drop_table: Specifies the total number of DROP TABLE commands.

Com_flush: Specifies the total number of FLUSH commands. This includes FLUSH TABLES, FLUSH LOGS, FLUSH HOSTS, FLUSH PRIVILEGES, FLUSH MASTER, and FLUSH SLAVE.

Com_grant: Specifies the total number of GRANT commands (used for creating users and granting them access privileges).

Com_insert: Specifies the total number of INSERT queries.

Com_insert_select: Specifies the total number of INSERT INTO tbl_name SELECT queries.

Com_kill: Specifies the total number of KILL commands. The KILL command is issued by a DBA to kill rogue threads (connections).

Com_load: Specifies the total number of LOAD DATA INFILE commands. LOAD DATA INFILE allows you to load the data from a text file directly into a table with the minimum possible overhead.

Com_load_master_table: Specifies the total number of LOAD TABLE FROM MASTER commands. This command is executed on the replication slave and was originally added to the parser to facilitate replication testing.

Com_lock_tables: Specifies the total number of times the LOCK TABLES command has been executed. The command is used to lock the tables for the duration of more than one query, and is frequently used in applications to ensure referential integrity in MyISAM tables.

Com_optimize: Specifies the total number of times the OPTIMIZE TABLE command has been executed.

Com_purge: Specifies the total number of the PURGE MASTER LOGS command has been run. This command is used to purge the old replication binary logs in a consistent manner.

Com_rename_table: Specifies the total number of times the RENAME TABLE command has been executed.

Com_repair: Specifies the total number of times REPAIR TABLE has been executed.

Com_replace: Specifies the total number of times the REPLACE query has been executed. REPLACE works like an INSERT if there is no primary or unique key conflict, and otherwise deletes the conflicting record and replaces it with the new one.

Com_replace_select: Specifies the total number of times the REPLACE INTO tbl_name SELECT query has been executed.

Com_reset: Specifies the total number of times the RESET MASTER and RESET SLAVE commands have been executed. RESET MASTER is a synonym for FLUSH MASTER, and RESET SLAVE is a synonym for FLUSH SLAVE. Although identical in functionality, FLUSH and RESET are accounted for differently. Both commands restore the master and the slave respective to their "virgin" states with regard to replication log tracking. The data does not change.

Com_restore_table: Specifies the total number of times the RESTORE TABLE command has been executed. The table would have been previously backed up with BACKUP TABLE.

Com_revoke: Specifies the total number of times the REVOKE command has been executed. REVOKE is used to take away privileges from users (the opposite of GRANT).

Com_rollback: Specifies the total number of times the ROLLBACK command has been executed.

Com_select: Specifies the total number of SELECT queries.

Com_set_option: Specifies the total number of SET commands. SET commands are used to configure various per-connection options on the server.

Com_show_binlogs: Specifies the total number of times the SHOW MASTER LOGS command has been executed. The command lists all the binary replication logs on the master, and can be used prior to PURGE MASTER LOGS.

Com_show_create: Specifies the total number of times the SHOW CREATE TABLE command has been executed. SHOW CREATE TABLE displays a CREATE TABLE statement that will create a table with the exact same structure as the given one.

Com_show_databases: Specifies the total number of times the SHOW DATABASES command has been executed.

Com_show_fields: Specifies the total number of times the SHOW FIELDS command has been executed.

Com_show_grants: Specifies the total number of times the SHOW GRANTS command has been executed.

Com_show_keys: Specifies the total number of times the SHOW KEYS command has been executed.

Com_show_logs: Specifies the total number of times the SHOW LOGS command has been executed. SHOW LOGS displays the current BDB transaction logs, and is of interest only if you are using BDB tables.

Com_show_master_status: Specifies the total number of SHOW MASTER STATUS commands. The command shows the current binary log coordinates of the master, as well as some master replication configuration parameters.

Com_show_open_tables: Specifies the total number of times SHOW OPEN TABLES has been executed. SHOW OPEN TABLES displays the tables for the current database that are currently in the table cache.

Com_show_processlist: Specifies the total number of SHOW PROCESSLIST commands.

Com_show_slave_status: Specifies the total number of SHOW SLAVE STATUS commands. The command shows the progress of replication on the slave, as well as some slave configuration parameters.

Com_show_status: Specifies the total number of SHOW STATUS commands.

Com_show_tables: Specifies the total number of SHOW TABLES commands.

Com_show_variables: Specifies the total number of SHOW VARIABLES commands.

Com_slave_start: Specifies the total number of SLAVE START commands. SLAVE START starts the slave thread on the replication slave.

Com_slave_stop: Specifies the total number of SLAVE STOP commands. The command stops the slave thread on the replication slave.

Com_truncate: Specifies the total number of TRUNCATE TABLE commands. TRUNCATE TABLE removes all the records from the table but does not drop it.

Com_unlock_tables: Specifies the total number of UNLOCK TABLES commands. The command undoes the effect of LOCK TABLES.

Com_update: Specifies the total number of UPDATE queries.

Connections: Specifies the total number of attempted connections from clients.

Created_tmp_disk_tables: Specifies the total number of disk-based (MyISAM) temporary tables created "behind the scenes" to process a query. This number does not include the user-created temporary tables, which is done with CREATE TEMPORARY TABLE. If a temporary table has to be created at all to answer a query, the server first tries to use an in-memory (HEAP) table. If the table is too big, or if it has BLOB or Text columns, a disk-based table will be used. If this number is high relative to the value of Questions, you should be concerned and investigate. Check all of your GROUP BY and ORDER BY queries to make sure they are properly optimized.

Created_tmp_tables: Specifies the total number of memory-based (HEAP) temporary tables created "behind the scenes" to process a query. If this value is high relative to Questions, it is not as bad as in the case of Created_tmp_disk_tables, but should nevertheless be a cause for concern. Again, you should check the GROUP BY and ORDER BY queries.

Created_tmp_files: Specifies the total number of temporary files created by the server. If this is high relative to Questions, check your ORDER BY and GROUP BY queries.

Delayed_insert_threads: Each table receiving records from a delayed insert gets assigned a delayed insert thread. This number shows the number of those threads currently running.

Delayed_writes: Specifies the total number of records written with INSERT DELAYED.

Delayed_errors: A delayed insert does not have a way of informing the user that an error has occurred. For example, we can hit a duplicate key error on a unique key, and the client will think everything is fine because the error will happen in the delayed insert thread, not in the thread that is handling the connection. When an error occurs, the delayed insert thread simply increments the Delayed_errors counter and continues working.

Flush_commands: Same as Com_flush.

Handler_delete: Specifies the total number of deleted records.

Handler_read_first: Specifies the total number of times the first entry was read from an index. The first entry is read from an index when the optimizer performs a full-index scan, which is usually not very efficient and is done because the optimizer could not devise a better way to answer the query. If this value is high relative to Questions, you should be worried. Look for queries with index in the type column of the EXPLAIN output.

Handler_read_key: Specifies the total number of requests to read a record based on the value of a key. Ideally, you want this number to be as high as possible. Reading a record based on the exact value of a key is the most efficient way to use a key.

Handler_read_next: Specifies the total number of requests to read the next key value in an index. Requests of this type occur during key scans, range lookups, and lookups on non-unique keys. Having this number high can be good or bad. When you are analyzing server performance, it is important to understand why this value is high or low, and whether this is acceptable. The answer will depend on the application.

Handler_read_prev: Specifies the total number of requests to read the previous key value in an index. This type of request occurs when processing ORDER BY DESC and is also used in some cases when optimizing MAX() with GROUP BY. Not very many queries will issue this type of request in 3.23. In 4.0, the optimization of ORDER BY DESC is better, and you should see more of such requests for ORDER BY DESC queries.

Handler_read_rnd: Specifies the total number of requests to read a record based on its position. It is used in processing GROUP BY and ORDER BY queries when no optimization is found that allows you to read all the needed records in one pass on some key. This counter is also incremented while running a REPLACE query that overwrites a record having hit a unique key match.

Handler_read_rnd_next: Specifies the total number of requests to read the next record in the physical layout sequence. This counter is incremented for each scanned record. If you see that this number is 100-1000 or more times the value of Questions, you should be concerned, and you should determine whether you can improve some of your queries by adding keys.

Handler_update: Specifies the total number of requests to update a record in a table. Note that this counter may increase during a regular SELECT query if a temporary table is being used.

Handler_write: Specifies the total number of requests to insert a record into a table. This counter may increase during a SELECT if a temporary table is being used.

Key_blocks_used: Shows how many blocks are being used in the MyISAM key cache. Each block is 1KB. If the setting is Key_blocks_used * 1 KB < key_buffer_size, you may consider reducing your key cache size, because it is probably just wasting memory. Note that the InnoDB handler uses its own buffer pool and is not affected by the MyISAM key cache.

Key_read_requests: Specifies the total number of requests to read a key block out of the MyISAM key cache.

Key_reads: Specifies the total number of unsatisfied MyISAM key cache requests that forced the cache manager to perform a read from disk. You can compute the cache hit rate as (Key_read_requests-Key_reads)/Key_read_requests. If the cache hit rate is less than 99 percent, you may consider increasing your key buffer.

Key_write_requests: Specifies the total number of key block write requests to the MyISAM key cache.

Key_writes: Specifies the total number of key block writes to disk in the MyISAM key cache. If this is less than Key_write_requests, this means that some of the DELAY_KEY_WRITE tables have been updated, but the dirty blocks have not yet been flushed out.

Max_used_connections: Specifies the maximum number of concurrent connections the server has experienced since it was started.

Not_flushed_key_blocks: Specifies the total number of dirty key blocks in the MyISAM key cache.

Not_flushed_delayed_rows: Specifies the total number of records waiting to be written in the DELAYED INSERT queues. There is a separate queue for each table.

Open_tables: Specifies the total number of open tables in the table cache. When a table is used for the first time, the table descriptor structure is initialized. MySQL caches the information about each table instance it opens if there is root in the table cache to avoid the overhead of unnecessary reinitialization. If this number is equal to the size of the table cache, your table cache is probably too small.

Open_files: Specifies the total number of files currently in use by the server. This variable is useful in tracking down file descriptor shortage problems. File descriptor shortage problems can be solved by either reducing the number of file descriptors MySQL needs (lower max_connections and/or table_cache) or by making more file descriptors available to the server process.

Open_streams: An obsolete variable. In earlier versions, it counted the number of FILE streams opened with the libc call fopen().The streams were being used for logging. Eventually, it was discovered that fopen() did not work on Solaris if the highest file descriptor number available was greater than 255, and all use of FILE streams was replaced with the internal file I/O cache implementation (see mysys/mf_iocache.c in the source), which does not depend on FILE streams. This variable will probably be removed in future versions.

Opened_tables: Specifies the total number of tables opened. If this value is much higher than the total number of tables frequently used in queries, you should consider increasing the size of the table cache.

Questions: Specifies the total number of queries and special server commands (such as ping, quit, or a request for the dump of the replication binary log by a slave). This is a good indicator of the server load and a good reference value to estimate if the other statistics are high or low.

Select_full_join: Specifies the total number of table joins when no records are being eliminated from a subsequent (not first in the join order) table through a key lookup optimization and the entire table has to be scanned. A full join is often a serious problem. The number of records the server has to examine to resolve a join is a product of the number of records that have to be examined from each table participating in the join. If the records in the subsequent table in the join order can be retrieved with a key, it will usually significantly reduce the number of records in the table that will need to be examined. Not having proper keys and constraints in a join will lead to very serious scalability problems. If you find that the value of this counter is not 0, you should make sure you know which queries are responsible and verify that they all are using only small tables. See Chapter 14 for a more in-depth discussion of this topic.

Select_full_range_join: Specifies the total number of table joins when some records are being eliminated from a subsequent (not first in the join order)

table through a range optimization—the optimizer uses a key to include the records only from a certain value range or a group of ranges. A full-range join is not as bad as a full join, but is still problematic if the range covers a lot of values. If the value of this counter is not 0, you should audit all of your queries, find out which ones are responsible for incrementing this counter, and make sure that all of them cover only a small range. See Chapter 14 for a more in-depth discussion.

Select_range: Specifies the total number of table reads that involved the range optimization (reading on a key range or a set of key ranges from the index). Note that this counter will be incremented only for the first non-constant table in the join order. (A non-constant table is a table that the optimizer believes to have more than one row it will need to examine.) If a range optimization is used on the subsequent tables, Select_full_range_join or Select_range_check will be incremented. The range optimization is effective when the range is small, but performance can drop as the range increases. If you see a high value (relative to Questions) in this counter, you should examine your range queries to see how big the ranges are. One indicator of the average size of a range is the value of Handler_read_next. However, you need to take into account that Hander_read_next also is incremented during full-key scans and ref optimizations (when we look up all records knowing only the value of a the first *N* parts of a multipart key as opposed to the whole key).

Select_range_check: Specifies the total number of table joins that involved a partial range optimization for a subsequent table in the join order. When the values of the reference columns in the previous tables would give a sufficiently narrow key range, the range optimization was used. If the range was too wide in the judgment of the optimizer, the table would be scanned. The decision was being made on a record-by-record basis. Seeing a non-zero value for this counter is a cause for concern. You should find the queries that are responsible for incrementing it, and either optimize them to do a more efficient join or make sure that the tables will always stay sufficiently small not to cause a problem.

Select_scan: Specifies the total number of times the first non-constant table in the join order was scanned. If this value is high relative to Questions and is coupled with a high value of Hander_read_rnd_next, you should track down the scanning queries and see if you can optimize them to use a key. Note, however, that if the table is guaranteed to stay relatively small and you are not performing a join, scans are not that terrible. The MyISAM handler scans in the order of magnitude of 200,000 records per second (100–200 bytes long) on a Pentium 500 when the data file fits into RAM and is cached, whereas the InnoDB handler can do 30,000 records per second on the same machine again with the caching assumption. Scans, however, can cause severe performance problems if a join is involved or if a table keeps growing. A classical case that

periodically appears on newsgroups, mailing lists, and the MySQL support ticket system is a table that is being accidentally scanned for days and weeks in the application, while it keeps growing in the meantime. Up to a certain size, the performance is satisfactory. But when a critical size is reached, the table does not fit into RAM any longer, scans require heavy disk I/O, the operating system begins to swap pages in and out frantically, and the server can barely function. Adding a key or changing the query to use one fixes the problem.

Slave_running: Shows if the slave is running.

Slave_open_temp_tables: Specifies the total number of temporary tables currently in use by the slave thread. This variable is important when you're shutting down a slave server. Shutting down the slave server (not just stopping the slave thread) when this value is not 0 may result in improper replication of some data when replicated queries rely on the data in a temporary table. You have to stop the slave thread first and then check if this value is 0. If it isn't, start the slave thread, let it run a bit, and then stop it and check again. If the master creates temporary tables but never updates other nontemporary tables using the data from the temporary tables, it is okay to shut down the slave server even when this value is not 0.

Slow_launch_threads: Some operating systems have a problem with slow thread creation under load. This counter keeps track of how many threads took longer than slow_launch_time seconds to get started. If this value is high, it indicates that your system has experienced periods of extreme load or that the operating system has some performance bug in thread creation.

Slow_queries: Specifies the total number of queries that took longer than long_query_time seconds. If the server is run with the long-log-format option, the counter will also include queries that executed faster than the long_query_time cutoff but were not, in the opinion of the optimizer, properly optimized.

Sort_merge_passes: When executing ORDER BY queries that cannot be optimized by just iterating through a key, MySQL will use the filesort algorithm. This algorithm uses the sort buffer and temporary files if needed to sort the record positions and then retrieves the records in the correct order based on position. A merge pass takes place when the contents of the sort buffer are being merged with the contents of an existing temporary file that already has positions sorted in the correct order. This counter is incremented on every merge pass. If the entire sort can occur inside the sort buffer, no temporary files will be used, and there will be no merge passes. If this value is greater than 0, it indicates that you are running ORDER BY queries that sort a large number of records. You can speed up those queries by reducing or eliminating merge passes through the increase of the sort buffer size.

Sort_range: This counter is incremented every time a sort takes place on records read from a key range as opposed to a full scan. Note that as the optimizer is able to pass through the key range in the correct sort order, this counter is not incremented. This method of sorting, while usually better than full scan, is still quite inefficient. If this value is high relative to Questions, you should track down the queries that are causing it to increase and see if there is a way to optimize them. Usually you will have to add a better key, or perhaps even add a surrogate column to your table and make it a key.

Sort_rows: Specifies the total number of records that participated in the filesort algorithm sorts. Usually the same as Handler_read_rnd. A high value (relative to Questions) suggests that perhaps you could benefit from optimizing your GROUP BY and ORDER BY queries to use keys to read records in sorted order.

Sort_scan: This counter is incremented every time the filesort algorithm is used while the entire table is being scanned (as opposed to scanning a key range). This is, as a rule, the most inefficient sorting method. The best way to eliminate scanned sorts is to add a key that will allow MySQL to read the needed records in the required order. If that is not possible, you might be able to improve it to use a range scan instead of a full scan while the records are being chosen for the sort.

Table_locks_immediate: Specifies the total number of table lock requests that have been granted by the lock manager immediately without waiting for other threads to yield the lock. Note that InnoDB tables use row-level locking and pass through the lock manager, formally receiving a table lock under all conditions. MyISAM and HEAP tables, though, require a real table lock, and the lock manager may block the thread that requested a lock until the lock becomes available.

Table_locks_waited: Specifies the total number of table lock requests that have required a wait for another thread to yield the lock. For MyISAM and HEAP tables, the ratio of Table_locks_waited/(Table_locks_immediate+ Table_locks_waited) shows the degree of lock contention. If the value is high (over 10–20 percent), you may be able to improve performance of your application by converting to InnoDB tables.

Threads_cached: Specifies the total number of threads currently waiting in the thread cache to handle the next request. The thread cache is helpful on systems that have a lot of quickly connecting and disconnecting clients, or when thread creation is slow.

Threads_created: Specifies the total number of created threads. A thread is created each time a new client connects. If you see that your value of this counter is significantly higher than Max_used_connections, you may benefit from using the thread cache or increasing its size.

Threads_connected: Specifies the total number of currently connected clients plus various administrative threads.

Threads_running: Specifies the total number of threads that are currently processing a query.

Uptime: Specifies the amount of time (in seconds) the server has been running.

A detailed analysis of the SHOW STATUS output frequently leads to pinpointing and correcting the query inefficiency and configuration issues that caused the unsatisfactory performance. It also helps track down potential performance problems that have not yet surfaced but that will at some point as the application scales. We recommend you perform periodic audits of SHOW STATUS output on your production server, and always conduct an audit on the development server after a test run of the application.

Another aspect of monitoring the health of a MySQL server is observing the operating system activity. Most Unix systems have a utility called top. Other utilities, such as vmstat, procinfo, monitor, and systat, are available on various Unix variants. Using a system-monitoring tool, you can watch the CPU utilization, disk I/O, amount of available RAM, context switches, and other parameters.

The ideal server will run at about 70 percent user CPU and 30 percent system when fully loaded, which means that 70 percent of the time is spent inside the server code itself and 30 percent is spent in the system calls performed on behalf on of the server process. If the user CPU share is significantly higher than 70 percent, consider it a warning that a performance bug exists either in the server itself or in the system libraries. It is also, of course, possible that the server is running some very CPU-intensive queries that are being properly processed. In any case, high user CPU always justifies a thorough investigation into the cause.

Low user CPU on a fully loaded server usually indicates that the disk I/O is very high. The most likely cause for this is unoptimized queries that are using large tables or joining a lot of small ones without a key.

High-context switches are an indication of performance bugs in the way the operating system handles threads, or in the thread library that MySQL depends on. (Unfortunately, it is difficult to come up with a generic standard of what constitutes high because it is very much system dependent.) When troubleshooting a scalability problem, you should monitor how context switches change as the number of clients increases. If the increase in the context switches is very dramatic and performance numbers also drop significantly, it is a good indication that MySQL is using the wrong kind of mutex. If you do not find anything in the operating system notes for your system in the MySQL

online manual, you should report the problem to the MySQL development team by sending a mail using the mysqlbug script to bugs@lists.mysql.com.

On Linux, you should be aware that the memory reported for each thread includes global segments shared by all threads. Therefore, if you see that 500 instances of mysqld are using 500MB each, this is not something to worry about. Since most of the memory is global, the total amount of memory consumed by MySQL is only 500*small_thread_overhead higher than the reported 500 MB, not 500*500MB.

CHAPTER 16

Replication

The term *replication* refers to the capability of a system installed in one location to duplicate its data to another system at a remote location. Replicated systems can be used for backups, scaling system performance, redundancy, cross-system data sharing, and other purposes.

MySQL has internal replication capabilities that you might find useful for your application. In this chapter, we will discuss how MySQL replication works, what it can do, how to set it, and how to maintain it. For more information on the subject, visit www.mysql.com/doc/R/e/Replication.html; this site provides more detailed documentation of replication options and updated information about the features currently available. I particularly recommend that you visit the site if this book is no longer new by the time you read this chapter.

Replication Overview

MySQL replication is conceptually simple. Each server instance can act as a master and/or a slave. To be a master, all a server must do is keep a logical update log. To be a slave, it needs to know how to connect to the master, and it must have appropriate privileges on the master to be able to read the replication log. The roles of master and slave are not mutually exclusive. A server can be a master and have slaves connected to it, while in the meantime it can have its own master and connect to it as a slave.

The master keeps track of the updates in a logical update log. It logs the queries that have modified the data or that could potentially affect the way the data will

be modified by a subsequent query. When old logs are rotated, their names are recorded; these log names are listed in their correct order in a special log index.

The slave connects to the master and requests a feed of the replication log. During the first connection, the slave starts with the beginning of the first log in the master log index. Secondary connections request a feed from the appropriate log at the appropriate position. The slave keeps track of its replication coordinates (log name and position), updating the information with each successfully processed master update or when it reaches the end of one log and moves to the next. This way, if the slave disconnects for any reason (network problems, master going down, slave thread or the entire server brought down, and so on), it can reconnect later and continue from the same spot.

In 3.23, the updates are applied immediately. In 4.0 (beginning in 4.0.2), the updates are first read into an intermediate relay log on the slave by the network I/O thread. The relay log is processed by another thread (called the SQL thread) that runs in parallel with the I/O thread. This modification was made to improve reliability for setups in which the slave gets far behind the master and the master has a high risk of disappearing forever (for example, through a fatal hardware failure or an operating system crash that damages the data beyond repair).

In order for the replication between master and slave to function properly, the slave must have an identical data snapshot of the master that corresponds to the state of the master from the time it started the first log, unless the initial replication coordinates are stated explicitly on the slave during the replication setup process. When coordinates (log name and position) are stated on the slave explicitly, they must correspond to the respective values on the master at the time of the snapshot.

You can tell the slave to ignore certain tables and databases during replication. You can also tell the master not to log updates to certain databases. Because replication happens on the logical level, you can even have tables with a different initial dataset on the slave or tables with a different structure under some circumstances. The rule is that the query that succeeded on the master must always succeed on the slave; as long as this rule is followed, replication will progress. Although doing so is generally not recommended, you can perform a shoot-in-the-wild replication by telling the slave to ignore all errors.

Uses of Replication

Several problems can be solved with the help of replication, including making backups, scaling read performance, sharing modifiable data among servers, running lengthy statistical queries, and performing hot failovers.

Backing Up Data

If your server is constantly under heavy load and stores large amounts of data, then performing regular backups can be a challenge—you may not be able to afford the extra CPU cycles and I/O bandwidth to perform the operation. In this case, you can set up a replication slave that stays in sync with the master and is backed up periodically. The backup server can also double as a hot spare; if the master suffers a fatal failure, you can replace it immediately with the slave and experience minimum downtime.

Scaling Read Performance

Replication is also helpful for scaling read performance. You first set up a master and a number of slaves, and then direct all the writes to the master. Non-critical reads (in other words, those that can happen on data that is a few seconds old) are directed to one of the slaves. Critical reads (those that must read the most current data) are directed to the master, because it is the only server that is guaranteed to have an up-to-date copy of the data. If the majority of the queries are non-critical reads, you can accomplish a great degree of scalability for little expense by adding low-cost machines to the slave pool.

Sharing Modifiable Data in Real Time

Suppose several offices in a company are connected over a slow network. All the updates are done by the central office, but the local offices need read access to the data. Because the network connection is slow, it would not be desirable to have local office clients connected to the central server all the time. To reduce the traffic between offices, you can set up a slave in each local office and put the master at the central location. The local clients now connect to their own server and do not have to go to the central server across the slow connection. This setup will work even if the network connection is not persistent, as long as all the updates can be transferred to the slaves during the connectivity window.

Running Lengthy Statistical Queries

Another common scenario aided by replication is a setup in which an intensive data-gathering process is taking place, but you want to be able to run lengthy statistical queries on the data just gathered. You make the data collection server a master and the statistical query server a slave. Now the statistical queries will not lock out the time-critical data collection writes and can proceed at their own leisure.

Carrying Out Hot Failovers

Although MySQL replication does not yet support hot failover, some users have discovered a version that works for them. You can set up two servers and make them co-masters: Each replicates the other. This way, if one server fails, the other is already up to date and can take over.

The same setup can be used when two remote locations perform infrequent updates that need to be seen on both servers, but the majority of the queries are local reads. This co-master bidirectional replication works only if the updates are independent of each other and do not use AUTO_INCREMENT fields.

Setting Up Replication

Now, let's get our hands dirty and look at the specifics of setting up replication. We'll examine how to configure the master first and then the slave.

Configuring the Master

Setting up the master involves the following steps:

1. Enable the log-bin option.
2. Assign the master a server ID.
3. Set up inclusion and exclusion rules.
4. Take a snapshot of the master data.
5. Determine the snapshot's replication coordinates.
6. Create a replication user for each slave on the master.

The following sections walk through each of these steps.

Enabling log-bin

You begin setting up the master server by making sure it is running with the log-bin option enabled. Like any option, you can use it either on the command line or in my.cnf (most experienced MySQL users prefer to use my.cnf). The log-bin option tells the server to keep a binary logical update log. This log is used in replication, but it can also serve for incremental backup purposes in conjunction with the mysqlbinlog tool, which can translate binary logs into a sequence of plain SQL commands.

If you are not satisfied with having the binary logs in the data directory, you can give log-bin an argument—for example, log-bin=/var/log/mysql/binlog. In this case, the binary logs will have the paths /var/log/mysql/binlog.001, /var/log/mysql/binlog.002, /var/log/mysql/binlog.003, and so on.

Assigning the Master a Server ID

Each replication peer (master or slave) is identified by a unique ID. This concept is similar to that of an IP address—each host on the network must have a unique address. Just as IP communication does not work if you give two machines the same IP address, replication peers will not work correctly if they have been assigned the same ID. So, be sure you choose a unique number for the master's ID. Some users set the ID to 1, and others use the last number in the IP address. The important point is that you will not give any of the slaves the same ID you give to the master, and the master will not connect as a slave to any super-masters that have the same ID. To set the server ID, enter server-id=#*num* (where #*num* is the actual value of the ID) in my.cnf.

Setting Up Inclusion and Exclusion Rules

If there are any databases to which you do not want to log updates, you can set up inclusion or exclusion rules with the binlog-do-db and binlog-ignore-db directives. To say "do not log anything except the good stuff I'll name explicitly," use binlog-do-db:

```
binlog-do-db=good_db1,good_db2
```

Updates to all databases that are not included in binlog-do-db directive will not be logged.

To say "log everything except the bad stuff I'll name explicitly," use binlog-ignore-db:

```
binlog-ignore-db=bad_db1,bad_db2
```

Do not use binlog-do-db and binlog-ignore-db in the same configuration file—the results will be unpredictable. One important caveat is that the rules are applied based on the active database, not on the database being updated. For example, suppose you have configured the master with binlog-do-db=personnel and you execute the following:

```
USE sales;
UPDATE personnel.employee SET salary = salary*1.1;
```

Although a table in the personnel database is being updated, the update will be ignored because the active database is sales.

Taking a Snapshot of the Master Data

There are several ways to take a snapshot of the master data. Ideally, if you can afford for the master to stay down during the snapshot, you can make a copy of the entire data directory (on Unix, you use tar; on Windows, you use Winzip). If

you have InnoDB tables, and you have put the logs and data files outside the data directory, you will have to copy them separately.

If you must take the snapshot on a running server, begin by issuing FLUSH TABLES WITH READ LOCK. This command will make sure all MyISAM tables are written out to disk and will prevent all clients from modifying the data. If you are using only MyISAM tables, you can take a snapshot of the data directory. If you are using InnoDB tables, you can either use mysqldump (if you are short on cash) or the native InnoDB hot backup tool (commercially available at www.innodb.com).

When you are finished with the snapshot, execute SHOW MASTER STATUS and record the current log name and position. You will need this information later for configuring the slave. At the end of the process, execute UNLOCK TABLES.

Note that you can use the same snapshot to set up several slaves as long as the master logs are not reset or purged manually.

Determining the Snapshot's Replication Coordinates

The next step is determining the replication coordinates (log name and offset) to which the snapshot corresponds. If you take the snapshot on a running server, no additional brain-wracking is required because the coordinates appear in the output of SHOW MASTER STATUS.

In the case of an offline snapshot, the situation is more complicated. You examine the contents of the binary log index file (the default is `hostname`-bin.index in the data directory) before you restart the master server, and find the last log in the index. Based on that name, you calculate the name of the next log. That name will be the log you will tell the slave to start from. For example:

```
$ hostname
mysql
$ tail /var/lib/mysql/mysql-bin.index
./mysql-bin.083
./mysql-bin.084
./mysql-bin.085
./mysql-bin.086
./mysql-bin.087
./mysql-bin.088
./mysql-bin.089
./mysql-bin.090
./mysql-bin.091
./mysql-bin.092
```

The last log name is mysql-bin.092. The next log, therefore, will be mysql-bin.093, and that is where the slave will begin.

If the master has previously been running without the log-bin option enabled, the slave must begin with the first log. In this case, you record an empty string for the log name.

If the name of the last log is mysql-bin.999, the name of the next log will be mysql-bin.1000. Up to 1000, the numbers are zero-padded at the beginning to make them three digits. Beyond 1000, no padding is necessary.

The position will always be 4 in the case of an off-site snapshot. For a hot snapshot, the position is whatever value appears in the Master_pos column of SHOW MASTER STATUS.

If the master was down during the snapshot, you can start it as soon as you have finished copying the data and have looked at the end of the master log index file. Updates can be allowed on the master only after you have the snapshot and know what replication coordinates it corresponds to.

Creating a Replication User for Each Slave

Now you need to create a replication user for each slave on the master. To be able to replicate in 3.23, the slave user needs to have the FILE privilege. You create the replication with the GRANT command in a way similar to this example:

```
GRANT FILE ON *.* TO 'repl'@'slave1.mycompany.com' IDENTIFIED BY
'secret';
```

In 4.0.3, a new REPL_SLAVE privilege has been introduced, and you should use it instead of FILE:

```
GRANT REPL_SLAVE ON *.* TO 'repl'@'slave1.mycompany.com' IDENTIFIED
BY 'secret';
```

Configuring the Slave

Now it is time to configure the slave. Doing so involves these steps:

- Assign the slave a unique server ID.
- Issue configuration directives.
- Add log-bin to the slave configuration.
- Start the slave server.
- Launch the slave thread.

Assigning the Slave a Server ID

You begin by making sure the slave has a unique server ID. The ID must be different from that of the master and also from that of other slaves connecting to

the same master. If subslaves will connect to this slave, the ID must also be different from theirs. Enter server-id=#*num* in the my.cnf file on the slave.

Issuing Configuration Directives

You may want to tell the slave to exclude some tables from replication or to include only certain tables. The configuration directives are replicate-do-table, replicate-ignore-table, replicate-wild-do-table, and replicate-wild-ignore-table. The following examples and explanations illustrate how these directives work.

The following directives tell the slave to replicate only the updates that modify the sales.contacts and personnel.employee tables:

```
replicate-do-table=sales.contacts
replicate-do-table=personnel.employee
```

In this example, the slave will replicate only the updates that modify tables in either the sales or personnel database:

```
replicate-wild-do-table=sales.%
replicate-wild-do-table=personnel.%
```

These directives tell the slave to replicate all updates except those that modify the sales.contacts and personnel.employee tables:

```
replicate-ignore-table=sales.contacts
replicate-ignore-table=personnel.employee
```

In this case, the slave will replicate all updates except those that modify tables in the personnel and sales databases:

```
replicate-wild-ignore-table=personnel.%
replicate-wild-ignore-table=sales.%
```

Adding log-bin to the Slave Configuration

It is recommended that you add log-bin to the slave configuration, even though this option is not absolutely required for the slave to function. Having an update log on the slave will help you track down bugs in the clients that update the slave erroneously, as well as other problems. The option is also required if you plan for this slave to function as a master to another server. If the slave will be a master to another server, you should also enable log-slave-updates.

Starting the Slave Server

Now that you are done editing my.cnf, you can start the slave server. Depending on how you took the snapshot on the master, you may need to unpack it on the slave while the slave is down, or you may need to start the slave first.

After you have restored the snapshot, start the slave server (if it is not running). Then, connect to it and execute the following:

```
CHANGE MASTER TO MASTER_HOST='masterhostname',
MASTER_USER='slave_user_name_on_master',
MASTER_PASSWORD='slave_user_password_on_master',
MASTER_LOG_FILE='master_log_name_for_snapshot',
MASTER_LOG_POS='master_pos_for_snapshot';
```

You need to replace the values in quotes with the parameters appropriate for the given configuration; for example:

```
CHANGE MASTER TO MASTER_HOST='master.mycompany.com',
MASTER_USER='repl', MASTER_PASSWORD='secret',MASTER_LOG_FILE='master-
092.bin',
MASTER_LOG_POS=4;
```

If the master is running on a non-standard port, you must add a MASTER_PORT=#*num* statement to the command. The previous command tells the slave how to connect to the master and where to start replication.

Launching the Slave Thread

Once oriented, you launch the slave thread:

```
SLAVE START;
```

If you did everything right, the slave thread should be running. To see if it is, execute SHOW SLAVE STATUS; it produces output similar to this:

```
Master_Host   Master_User   Master_Port   Connect_retry   Log_File
Pos   Slave_Running   Replicate_do_db   Replicate_ignore_db
Last_errno   Last_error   Skip_counter
master.mycompany.com   repl   3306   60   master-bin.009      2978
Yes   0 0
```

The value in the Slave_Running column should be Yes. If that is the case, you connect to the master, try an update (such as creating a dummy table and inserting a couple of rows), and then check the slave to see whether the update has been propagated. If it has, replication is working as it should. If not, or if the value in the Slave_Running column is No, it is time to troubleshoot.

Troubleshooting

Most of the time, the answer to a problem can be found in the slave's error log. Following is a list of things that most commonly go wrong during replication setup and how you can address them:

- **Access denied error:** The access rights for the slave have not been set up properly. Check the user name and password. Make sure CHANGE

MASTER TO points to the correct master. Try connecting from the slave to the master using the command-line client with the same authentication credentials. If everything fails, rerun the GRANT command on the master and the CHANGE MASTER TO command on the slave.

- **Slave connects but disconnects immediately:** Make sure you do not have a server ID conflict. If you do, correct it and then rerun the CHANGE MASTER TO command on the slave.
- **Duplicate key error:** Either there is a problem with the snapshot, or the position does not correspond to the snapshot you have taken. Analyze the snapshot data to see what happened. If the problem is still not clear, try taking another snapshot, this time taking particular care that no updates are made during the snapshot and that you record the master log position correctly.
- **Slave thread complains about a bogus log position or non-existent log:** Either the position or the log name was recorded incorrectly during the snapshot, or someone executed RESET MASTER on the master since the snapshot was taken. Take another snapshot on the master, and repeat the slave setup.

Replication Maintenance

Maintaining a replicated setup involves two basic tasks: making sure the master binary logs do not overflow the disk, and monitoring the slaves to make sure they keep up with the master.

The master server automatically rotates logs when an update makes the log size exceed the value of max_binlog_size (which is set to 1GB by default). It is also capped at that limit—even if you set the value higher, it will be trimmed to 1GB. If you want the logs to be rotated more frequently, you can lower the log size with the command set-variable max_binlog_size=*val*. Logs can also be rotated manually at any time with the FLUSH LOGS command.

You must purge the old logs with care. Don't just delete them manually—the binary log index must also be updated. You can do so with one command: PURGE MASTER LOGS TO. Before running PURGE MASTER LOGS TO, list all the logs with the SHOW MASTER LOGS command. Then, make sure all the slaves have processed the logs you are about to purge. Pick the highest log that all slaves have already processed, and give the next log in sequence after that one as an argument to PURGE MASTER LOGS TO. The process is illustrated in the following example.

You first run SHOW MASTER LOGS on the master, and it gives the following listing:

```
master-bin.001
master-bin.002
master-bin.003
master-bin.004
master-bin.005
master-bin.006
```

You have three slaves. You connect to each of them and check how far they have progressed with SHOW SLAVE STATUS. You find that the first slave is processing master-bin.003, and the other two slaves are on master-bin.004. This means you can purge up to and including master-bin.002 because it is the highest log in sequence that all slaves have already processed. The next log after that is master-bin.003, so it will be the argument to PURGE MASTER LOGS TO. You accomplish the purge with the following command:

```
PURGE MASTER LOGS TO 'master-bin.003';
```

Measuring How Far the Slave Is Behind the Master

To measure how far a slave is behind the master, execute SHOW SLAVE STATUS on the slave and SHOW MASTER STATUS on the master, and then compare the replication coordinates on both. If the master has relatively little update activity, the coordinates will frequently be identical if the replication is working properly. On a highly loaded master, if everything is well the slave will be a bit behind, but the gap will not be significant and will not increase with time.

If you notice that the slave is progressing but keeps gradually falling behind the master, there could be several reasons. The most common is that the slave cannot handle the combined load of the master updates and the read queries from clients. You can address this problem by optimizing the read queries, upgrading hardware, or setting up another slave. A less-common problem, although still a possibility, is that the network connection between master and slave is too slow for the update load on the master.

A slave that is not progressing while the master position keeps advancing indicates that an error may have occurred in one of the slave thread queries, or perhaps that there are connectivity problems between the master and the slave. Query errors will appear in the Last_error column of the SHOW SLAVE STATUS output; connectivity error messages will be in the slave error log.

Replication Errors

One of the most common errors during replication is the duplicate key error. It usually happens because the slave snapshot was initially different from that of the master. It may also happen if the slave is erroneously updated outside the slave thread. If you want the replication to continue past the duplicate key error, you can add skip-slave-error=1062 to my.cnf on the slave. Other error codes can also be specified in this configuration directive, delimited by a comma if desired. However, this practice is generally discouraged because it undermines the integrity of the replicated data. If errors do happen, usually a serious problem needs to be understood and addressed.

Another common problem is the low setting of max_allowed_packet. The default value of 1MB is usually sufficient, but if the master performs a large multirow insert, this value may not be enough. The error log will contain a message saying the packet is too large. If that happens, you should set max_allowed_packet to 16MB.

Stopping Replication

If you need to stop the replication process, execute the SLAVE STOP command. You can resume replication later with SLAVE START.

This technique is useful in several situations. For example, there is no need for the slave thread to run while you make a nightly backup. Other times, you may want to run a long statistical query on the slave, you do not necessarily need the most current data, and you want the query to go as quickly as possible. Shutting down the slave thread will give you extra CPU cycles and remove the lock contention.

Sometimes you may want to perform maintenance immediately after the slave server starts up, and you do not want the replication process to interfere. To do so, add the skip-slave-start option to my.cnf. You can then perform the maintenance and then start the slave thread with SLAVE START when you are finished. If you plan to have the slave thread start automatically the next time you restart the server, you should remove or comment out this option.

Replication Caveats

Unfortunately, replication in MySQL is not perfect. It has several known limitations, and it is important for you to understand them so you can adjust your expectations and plan accordingly.

Improperly Replicated Queries

Queries that utilize user variables in updates are not properly replicated because the binary log does not record the setting of a user variable to a value. If you are using the RAND() function in a query, the data will not be identically replicated. In some cases, that is acceptable—if you want a random number in a column anyway, it may not be important that the slave has the same random number as the master. But in other cases, this situation could be important.

Replication of Temporary Table Activity

Temporary table activity is replicated in order to properly replicate a sequence of queries of this kind:

```
CREATE TEMPORARY TABLE t1 SELECT * FROM t2;
INSERT INTO t3 SELECT * FROM t1;
DROP TABLE t1;
```

This process may cause unnecessary logging, data transfer, and slave CPU overhead. If you do not want the creation of a temporary table to be replicated, turn off binary logging for the query from the client code in the following manner:

```
SET SQL_LOG_BIN=0;
CREATE TEMPORARY TABLE t1 SELECT * FROM t2;
....
DROP TABLE t1;
SET SQL_LOG_BIN=1;
```

The client must have the PROCESS privilege to be able to turn off logging. After you grant the client the privilege, test it to be sure it works. If the client does not have the privilege, the attempt to disable logging will be silently ignored.

You can also use this technique to turn off logging in other situations, but you must be careful to be consistent. There is a danger in not logging a query, but logging the query that depends on the first (unlogged) query. Doing so will cause problems on the slave.

Replication of LOAD DATA INFILE

In 3.23, LOAD DATA INFILE is replicated, but only if the file resides on the server by the time the slave processes the query. Otherwise, an error occurs. LOAD LOCAL DATA INFILE is ignored.

Version 4.0 replicates LOAD DATA INFILE correctly, regardless of whether the file is still intact on the master when the slave begins processing the query and

can also replicate LOAD LOCAL DATA INFILE. However, the replication code in version 4.0 as of this writing (version 4.0.5) has not yet stabilized enough to be recommended for general production use. Still, some specialized uses of 4.0 replication have been tried in production, and the users have reported success (for example, the Yahoo! site remember.yahoo.com, which was set up to commemorate the victims of the September 11, 2001 attack on the World Trade Center).

Bidirectional Replication

If you are using bidirectional replication between two servers, you must be aware of several caveats. All of them stem from the fact that due to the lack of any kind of distributed transaction commit protocol, two update queries can take place in one order on one server and in reverse order on the other. Therefore, a query updating the data must be written in such a way that any two queries will leave the database in the same state if applied in reverse order.

Many applications will not be able to comply with this requirement. However, some do—for example, if-only inserts without the use of AUTO_INCREMENT and updates on a unique key that is functionally dependent on the server where the update originates.

You can work around the problem by including a surrogate primary key in all tables that identifies the server. For example, instead of an AUTO_INCREMENT field, you can use a simple integer coupled with your own sequence generation. You create a separate sequence table that is not replicated, and use the following maneuver to insert a new record:

```
SET SQL_LOG_BIN=0;
UPDATE sequence SET seq_num = LAST_INSERT_ID(seq_num+2);
SET SQL_LOG_BIN=1;
INSERT INTO t1 (id,...) VALUES (LAST_INSERT_ID(),...);
```

On the first server, you start with the sequence table containing one record with seq_num set to 0; on the second server, you set seq_num to 1. This way, the first server will always have even ID values, and the second will have odd ID values.

Replication Internals

So far, we have provided some quick setup and troubleshooting instructions for users with relatively simple replication needs. In this section, we will discuss MySQL replication in more depth, for those who are trying to solve more complex problems or who need to understand how replication work internally. This section will also be useful if you are just curious about how things work.

Masters and Slaves

In MySQL replication, each server plays the role of the master or the slave. The roles are not mutually exclusive. A master can double as a slave, and vice versa. A server can be a slave to another master, and at the same time can have slaves of its own. One master can have an unlimited (from the replication implementation point of view) number of slaves. In practice, network bandwidth and thread library limitations on the master (each slave connection requires the creation of a new thread) impose their own limits on the number of slaves, which on most systems would be about 1000. At this point, multimaster replication is not supported, although it will probably be developed some time in 2003.

The slave server needs to know how to connect to the master in order to be a slave. The slave server is instructed about how to connect to its master when the DBA executes the CHANGE MASTER TO command during slave setup.

Server IDs

Each server is identified by a unique number, which is set with the server-id configuration option. This number is similar to the idea of an IP address—it uniquely identifies the server among its network peers.

Binary Logs

The characteristic quality of the master is that it keeps a special update log called the *binary log*. The binary log is a logical log; it is called binary to distinguish it from the textual update log from prereplication days (version 3.23.14 and earlier). Each time the master restarts or the FLUSH LOGS command is issued, the master rotates the logs, closing the current log and moving to the new one.

The name and location of the logs are controlled by the log-bin option. By default, the logs are stored in the data directory of the server (specified by datadir) and are named `hostname`-bin.001, `hostname`-bin.002, `hostname`-bin.003, and so on, where `hostname` is the name of the host or the output of the hostname command. When the sequence reaches .999, another digit is added to the name of the next log: the extensions .1000, .1001, .1002, and so forth will be used. The maximum number of the logs is thus not restricted to 999, but instead by the maximum value of unsigned long, which is 4294967295 on a 32-bit system (sufficient for most uses).

The master server keeps tracks of the logs—both the current log and the logs that have been rotated out in a special index file, the name and location of which are controlled by the log-bin-index command. By default, the name of the

file is `hostname`.index; it is located in the data directory. Knowing the names and the sequence of the logs allows the master to respond to the slave binary log dump requests, giving it a continuous feed of all the updates since the period with which the slave considers itself current. The period (or replication coordinates) is identified by the log name and the log position (or offset) on the master corresponding to the state of the dataset. With each update on the master, the log position increases. When the log rotates, the position is reset to the value 4, which signifies that you should skip the first four bytes at the beginning of every log to get to where the update information is stored. The first four bytes are the magic number—a signature indicating the file type to the system.

The log consists of binary log events. Each event entry has a header and a body. The fixed-length header specifies the length and type of the event, and a few other parameters every event must have depending on the MySQL version. The body is variable length, and its contents depend on the type of event. Events signify the startup or shutdown of the server, a query, the setting of an auto-increment value, the log rotation, and a few other things. You can learn more about the format of event entries by studying sql/log_event.h and sql/log_event.cc files in the MySQL distribution.

The log can be dumped into human-readable plain text format by executing the mysqlbinlog command that comes in the MySQL distribution. For example, the command

```
mysqlbinlog /var/lib/mysql/mysql-bin.018
```

produces the following output:

```
# at 4
#021214 15:13:53 server id  1   Start: binlog v 1, server v 3.23.52-
log created 021214 15:13:53
# at 73
#021216 14:55:11 server id  1   Query   thread_id=1     exec_time=0
error_code=0
use test;
SET TIMESTAMP=1040075711;
drop table t1;
# at 115
#021216 14:55:19 server id  1   Query   thread_id=1     exec_time=0
error_code=0
SET TIMESTAMP=1040075719;
create table t1 (n int);
# at 167
#021216 14:55:29 server id  1   Query   thread_id=1     exec_time=0
error_code=0
SET TIMESTAMP=1040075729;
insert into t1 values (1),(2),(3);
```

To see an entry at a particular offset in the log, you can use the -j option. For example, this command

```
mysqlbinlog -j 115 /var/lib/mysql/mysql-bin.018 | head
```

produces the following output:

```
# at 115
#021216 14:55:19 server id  1   Query   thread_id=1     exec_time=0
error_code=0
use test;
SET TIMESTAMP=1040075719;
create table t1 (n int);
# at 167
#021216 14:55:29 server id  1   Query   thread_id=1     exec_time=0
error_code=0
SET TIMESTAMP=1040075729;
insert into t1 values (1),(2),(3);
```

In addition to sql/log_event.h and sql/log_event.cc, developers interested in binary logging internals should also look at log.cc and mysqld.cc, and search for LOG_BIN as an entry point to begin your studies.

Conclusion

MySQL has a simple, easy-to-set-up replication that is helpful in solving many enterprise problems. It does lack some features present in more expensive database packages, but in many cases there are application-specific workarounds that are able to at least partially make up for the lack of the feature. In the hands of a clever DBA, MySQL replication is a powerful tool.

CHAPTER 17

Backup and Table Maintenance

The purpose of this chapter is to provide guidance for the DBA trying to establish some order on a system. Two things are a must to reach this target—establishing a proper backup plan, and maintaining your tables.

Physical Backup

Each MyISAM table is stored as three files: .frm (table definition), .MYI (statistical information and keys), and .MYD (data file). InnoDB tables are a bit more complex. The table definition file (.frm) is the same as with MyISAM. However, the data and the index are stored in the table space(s) specified by the innodb_data_file_path server configuration parameter. A database, both with MyISAM and with InnoDB, is simply a subdirectory in the data directory.

MySQL data can be backed up by a regular file system copy if proper precautions are taken. Both InnoDB and MyISAM tables can be backed up on the file-system level if the server is down. If the server is running, MyISAM tables can be copied if the tables have been flushed to disk and locked. This procedure can be performed with the mysqlhotcopy script originally written by Tim Bunce, which is part of the MySQL distribution. It its simplest form, mysqlhotcopy can be used as follows:

```
mysqlhotcopy -u root -p secret db_to_backup
/var/database/backups/db_to_backup
```

The above command will back up the database db_to_backup into the directory /var/database/backupts/db_to_backup. The connection to the local server will

be made with the user name root and the password secret. mysqlhotcopy supports an number of options. Full listing can be obtained with mysqlhotcopy -help (Listing 17.1).

```
/usr/bin/mysqlhotcopy Ver 1.16

Usage: /usr/bin/mysqlhotcopy db_name[./table_regex/] [new_db_name |
directory]

  -?, --help           display this helpscreen and exit
  -u, --user=#         user for database login if not current user
  -p, --password=#     password to use when connecting to server
  -P, --port=#         port to use when connecting to local server
  -S, --socket=#       socket to use when connecting to local server

  --allowold           don't abort if target already exists (rename it
_old)
  --keepold            don't delete previous (now renamed) target when
done
  --noindices          don't include full index files in copy
  --method=#           method for copy (only "cp" currently supported)

  -q, --quiet          be silent except for errors
  --debug              enable debug
  -n, --dryrun         report actions without doing them

  --regexp=#           copy all databases with names matching regexp
  --suffix=#           suffix for names of copied databases
  --checkpoint=#       insert checkpoint entry into specified db.table
  --flushlog           flush logs once all tables are locked
  --resetmaster        reset the binlog once all tables are locked
  --resetslave         reset the master.info once all tables are
locked
  --tmpdir=#        temporary directory (instead of /tmp)
  --record_log_pos=#   record slave and master status in specified
db.table

  Try 'perldoc /usr/bin/mysqlhotcopy for more complete documentation'
```

Listing 17.1 mysqlhotcopy help output.

As suggested by the help message, you can read a full description of each option by typing perldoc /usr/bin/mysqlhotcopy.

To restore the backup taken with mysqlhotcopy, simply copy the backup files into the data directory.

As convenient as mysqlhotcopy is, it cannot back up InnoDB tables. To be able to perform a file system backup of InnoDB tables, you need the InnoDB Hot

Backup utility commercially available from Innobase Oy, the company that develops the InnoDB handler. The tool documentation, as well as the information on how to purchase it, is available at www.innodb.com/hotbackup.html.

Logical Backup

Backup on the SQL level is not as efficient in terms of time and disk space as the physical backup, but it does have its own set of advantages. SQL-level backup can be restored on a different (non-MySQL) database server. You can also manually edit the backup data if there is a need. For example, you may want to add or remove keys, or exclude certain records before restoring. If you have accidentally deleted a record, you can restore it by finding the corresponding INSERT statement in the logical backup and executing it manually.

With InnoDB tables, logical backup is the only available backup option for a running server if you lack the means or are not willing to spend the money to buy the InnoDB Hot Backup tool. Logical backup can be performed with the mysqldump utility, which is included in the MySQL distribution. Below are a few examples of mysqldump usage:

```
mysqldump -u root -p secret -h db1 -A --opt > db1.sql
```

Connects to server db1 with user name root and password secret. All tables in all databases are being dumped (from -A) . Optimizations for a large dump to speed up the restore and make sure we do not run out of memory will be enabled (–opt). The output is being redirected to file db1.sql.

```
mysqldump -u root -p secret -h db1 --opt sales > sales.sql
```

Backups up all tables in the sales database on server db1. The dump is redirected to the file sales.sql.

```
mysqldump -u root -p secret -h db1 --opt personnel employee payroll
> personnel.sql
```

Backs up employee and payroll tables in the personnel database on server db1. The dump is redirected to personnel.sql.

```
mysqldump -u root -p secret -h db1 -d --add-drop-table sales > sales.sql
```

Dumps the table definitions without the data in the sales database on server db1 redirecting the output to sales.sql.

```
mysqldump -u root -p secret -h db1 --opt  -B sales personnel | mysql -u
root -h db2 s
```

Copy the sales and the personnel database from server db1 to server db2. Although this is not the fastest way to do it (physical copying is much faster), it

is very convenient. If you add the thinking and command typing time into the overall performance measurement, a reasonably small database can be copied much faster between servers with the above method than with mysqlhotcopy.

A full listing of mysqldump options can be obtained by running mysqldump --help (Listing 17.2).

```
mysqldump  Ver 8.14 Distrib 3.23.41, for pc-linux-gnu (i686)
By Igor Romanenko, Monty, Jani & Sinisa
This software comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to modify and redistribute it under the
GPL license

Dumping definition and data mysql database or table
Usage: mysqldump [OPTIONS] database [tables]
OR     mysqldump [OPTIONS] --databases [OPTIONS] DB1 [DB2 DB3...]
OR     mysqldump [OPTIONS] --all-databases [OPTIONS]

  -A, --all-databases   Dump all the databases. This will be same as
--databases with all databases selected.
  -a, --all             Include all MySQL specific create options.
  -#, --debug=...       Output debug log. Often this is
'd:t:o,filename`.
  --character-sets-dir=... Directory where character sets are
  -?, --help            Display this help message and exit.
  -B, --databases       To dump several databases. Note the difference
in usage; In this case no tables are given. All name arguments are regarded
as databasenames. 'USE db_name;' will be included in the output
  -c, --complete-insert Use complete insert statements.
  -C, --compress        Use compression in server/client protocol.
  --default-character-set=...
                        Set the default character set
  -e, --extended-insert Allows utilization of the new, much faster
                        INSERT syntax.
  --add-drop-table    Add a 'drop table' before each create.
  --add-locks         Add locks around insert statements.
  --allow-keywords    Allow creation of column names that are keywords.
  --delayed-insert    Insert rows with INSERT DELAYED.
  -F, --flush-logs    Flush logs file in server before starting dump.
  -f, --force         Continue even if we get an sql-error.
  -h, --host=...      Connect to host.
  -l, --lock-tables   Lock all tables for read.
  -n, --no-create-db  'CREATE DATABASE /*!32312 IF NOT EXISTS*/
db_name;' will not be put in the output. The above line will be added
otherwise, if --databases or --all-databases option was given.
  -t, --no-create-info  Don't write table creation info.
```

Listing 17.2 mysqldump help output. (continues)

```
  -d, --no-data       No row information.
  -O, --set-variable var=option   give a variable a value.
  --help lists variables
  --opt                 Same as --add-drop-table --add-locks --all
                        --extended-insert --quick --lock-tables
  -p, --password[=...]  Password to use when connecting to server.
                        If password is not given it's solicited on the
tty.

  -P, --port=...      Port number to use for connection.
  -q, --quick         Don't buffer query, dump directly to stdout.
  -Q, --quote-names   Quote table and column names with `
  -r, --result-file=..Direct output to a given file. This option
should be used in MSDOS, because it prevents new line '\n' from being
converted to '\n\r' (newline + carriage return).
  -S, --socket=.      Socket file to use for connection.
  --tables            Overrides option --databases (-B).
  -T, --tab=...       Creates tab separated textfile for each table
to given path. (creates .sql and .txt files). NOTE: This only works if
mysqldump is run on the same machine as the mysqld daemon.
  -u, --user=#        User for login if not current user.
  -v, --verbose       Print info about the various stages.
  -V, --version       Output version information and exit.
  -w, --where=        dump only selected records; QUOTES mandatory!
  EXAMPLES: "--where=user='jimf'" "-wuserid>1" "-wuserid<1"
  Use -T (--tab=...) with --fields-...
  --fields-terminated-by=...  Fields in the textfile are terminated by...
  --fields-enclosed-by=...  Fields in the importfile are enclosed by
...
  --fields-optionally-enclosed-by=...  Fields in the i.file are opt.
enclosed by ...
  --fields-escaped-by=...   Fields in the i.file are escaped by ...
  --lines-terminated-by=...  Lines in the i.file are terminated by ...

Default options are read from the following files in the given order:
/etc/my.cnf /var/lib/mysql/my.cnf ~/.my.cnf
The following groups are read: mysqldump client
The following options may be given as the first argument:
--print-defaults    Print the program argument list and exit
--no-defaults               Don't read default options from any options file
--defaults-file=#   Only read default options from the given file #
--defaults-extra-file=# Read this file after the global files are read

Possible variables for option --set-variable (-O) are:
max_allowed_packet    current value: 16776192
net_buffer_length     current value: 1047551
```

Listing 17.2 mysqldump help output. (continued)

The backup taken with mysqldump can be restored on a fresh installation of MySQL by feeding the dump to the standard command line client; for example:

```
mysql -u root -p secret < db1.sql
```

Incremental Backup

In addition to a regular (e.g., nightly) physical and logical backup, it is also advisable to perform frequent incremental backups. Incremental backups put very little demand on system resources. If performed frequently, incremental backups greatly limit the amount of data you can lose in the case of a catastrophic failure.

To perform incremental backups with MySQL, you should enable the log-bin option on the server being backed up. Set up a cron job that will periodically connect to the server, execute FLUSH LOGS, and then archive the logs created since the previous run. Once the logs have been archived, dispose of them with the PURGE MASTER LOGS TO command, giving it the first log you are not going to purge as a string argument. If you have replication slaves, make sure they are done processing the logs you are about to purge. For a more detailed discussion of PURGE MASTER LOGS TO, see Chapter 16.

To restore an incremental backup, you must first restore the full physical or logical backup. You will then need mysqlbinlog, a binary log dump utility included in the MySQL distribution. Use mysqlbinlog to process all of the backup logs in the correct sequence, piping the output to the standard command line client. For example:

```
mysqlbinlog master-bin.001 master-bin.002 master-bin.003 master-bin.004
| mysql -u root -p secret
```

One disadvantage of the incremental backup is that if the amount of log data becomes large or the queries in it are slow, restoring it will take a long time. Another problem is that temporary table operations are not fully supported, and updates with user variables are not supported at all. If you plan to use incremental backup, it is recommended that you disable binary logging for the creation of temporary tables with SET SQL_LOG_BIN=0.

Backup through Replication

If you can afford to buy another machine, you can back up your data by replicating it. The slave thread can be periodically stopped (or you can just take the slave completely offline), and the data on the slave can be backed up to tape or

other archival media without disturbing the live system. This method allows for more frequent full backups. It also provides you with a hot spare server that you can bring online with very little downtime and data loss.

The disadvantages of this backup method are the cost of an extra machine and the few limitations on what kind of queries you can run on the master. See the replications caveats section in Chapter 16 for details.

Table Maintenance

With MyISAM tables, the most common sources of trouble are data fragmentation and index corruption. Record fragmentation can occur when the record is dynamic length, and you have a high volume of insertions and deletions. Fragmented records cause deteriorated performance. Another form of fragmentation is unfilled holes from deleted records. This can happen if you delete a large number of records and do not insert new ones to occupy the freed space. To address the fragmentation issues on MyISAM tables, you should periodically run the OPTIMIZE TABLE command. Altering the table, repairing it (with REPAIR TABLE), or dumping it out with mysqldump and reloading the data will also defragment it.

Theoretically, MyISAM table corruption should never happen, but in reality it does. Corruption may happen if the server is not shut down cleanly (e.g., somebody kills it with signal 9, the power is cut off, or the operating system panics), the operating system has a bug, the hardware is faulty, or MySQL itself hits a bug and fatally crashes. When a corrupted table is discovered, the server will give an error message to the client that says "Error from table handler".

To check if the table is corrupted, use the CHECK TABLE command. For example:

```
CHECK TABLE account;
```

It is also possible to specify a method of checking as an argument after the name of the table; for example:

```
CHECK TABLE account QUICK;
```

CHECK TABLE supports several checking methods: FAST (check only if the table was not closed properly), QUICK (just check the index file), CHANGED (check only if changed since last check), MEDIUM (check the index file and perform some basic sanity checks in the data file), and EXTENDED (check both the index and the data file as thoroughly as possible). FAST and CHANGED options are used primarily during automated checks. The default method is MEDIUM.

If you discover a corrupt table, you can repair it with the REPAIR TABLE command. For example:

```
REPAIR TABLE account;
```

Just like CHECK TABLE, REPAIR TABLE supports method options. For example:

```
REPAIR TABLE account QUICK;
```

REPAIR TABLE methods are QUICK (repair the index without looking at the data file), MEDIUM (use data file, create keys by sorting), and EXTENDED (use data file, update keys once per record insertion as opposed to creating them by sort). MySQL 4.0.2 and later supports the USE_FRM option, in which case only the data file and the table definition file are being used.

Both CHECK TABLE and REPAIR TABLE produce a table-like output with four columns: Table, Op, Msg_type, and Msg_text. For example:

```
Table     Op       Msg_type   Msg_text
test.t2   repair   status     OK
```

Both commands can be invoked from any client language using the exact same semantics as you would when executing SELECT.

It is also possible to check a table offline with myisamchk. It is recommended that you check your physical backup this way immediately after you have copied the data. Offline repair can be done with myisamchk -r. When using myisamchk, give it the path to the directory where the table files are located, followed by the name of the table. For example:

```
myisamchk -r /var/data/mysql/sales/account
```

will repair a table with the name account residing in the directory /var/data/mysql/sales/.

myisamchk is capable of performing several other operations, documented in the output of myisamchk --help (Listing 17.3):

```
myisamchk  Ver 1.53 for pc-linux-gnu at i686
By Monty, for your professional use
This software comes with NO WARRANTY: see the PUBLIC for details.

Description, check and repair of ISAM tables.
Used without options all tables on the command will be checked for
errors
Usage: myisamchk [OPTIONS] tables[.MYI]
```

Listing 17.3 myisamchk output. (continues)

```
Global options:
  -#, --debug=...     Output debug log. Often this is 'd:t:o,filename'
  -?, --help          Display this help and exit.
  -O, --set-variable var=option
                      Change the value of a variable.
  -s, --silent        Only print errors.  One can use two -s to make
myisamchk very silent
  -v, --verbose       Print more information. This can be used with
--describe and --check. Use many -v for more verbosity!
  -V, --version       Print version and exit.
  -w, --wait          Wait if table is locked.

Check options (check is the default action for myisamchk):
  -c, --check         Check table for errors
  -e, --extend-check  Check the table VERY throughly.  Only use this
in extreme cases as myisamchk should normally be able to find out if
the table is ok even without this switch
  -F, --fast          Check only tables that hasn't been closed
properly
  -C, --check-only-changed
                      Check only tables that has changed since last
check
  -f, --force         Restart with -r if there are any errors in the
table. States will be updated as with --update-state
  -i, --information   Print statistics information about table that is
checked
  -m, --medium-check  Faster than extended-check, but only finds
99.99% of all errors.  Should be good enough for most cases
  -U  --update-state  Mark tables as crashed if you find any errors
  -T, --read-only     Don't mark table as checked

Repair options (When using -r or -o)
  -B, --backup        Make a backup of the .MYD file as 'filename-time.BAK'
  -D, --data-file-length=#  Max length of data file (when recreating data
file when it's full)
  -e, --extend-check  Try to recover every possible row from the data
file. Normally this will also find a lot of garbage rows; Don't use this
option if you are not totally desperate.
  -f, --force         Overwrite old temporary files.
  -k, --keys-used=#   Tell MyISAM to update only some specific keys. # is a
bit mask of which keys to use. This can be used to get faster inserts!
  -l, --no-symlinks   Do not follow symbolic links. Normally myisamchk
repairs the table a symlink points at.
  -r, --recover       Can fix almost anything except unique keys that
aren't unique.
  -n, --sort-recover  Force recovering with sorting even if the
temporary file would be very big.
```

Listing 17.3 myisamchk output. (continues)

```
  -o, --safe-recover  Uses old recovery method; Slower than '-r' but
can handle a couple of cases where '-r' reports that it can't fix the
data file.
  --character-sets-dir=...
                      Directory where character sets are
  --set-character-set=name
                      Change the character set used by the index
  -t, --tmpdir=path   Path for temporary files
  -q, --quick         Faster repair by not modifying the data file.
                      One can give a second '-q' to force myisamchk to
modify the original datafile in case of duplicate keys
  -u, --unpack        Unpack file packed with myisampack.

Other actions:
  -a, --analyze       Analyze distribution of keys. Will make some
joins in MySQL faster.  You can check the calculated distribution by using
'
--describe --verbose table_name'.
  -d, --description   Prints some information about table.
  -A, --set-auto-increment[=value]
                      Force auto_increment to start at this or higher value.
If no value is given, then sets the next auto_increment
value to the highest used value for the auto key + 1.
  -S, --sort-index    Sort index blocks.  This speeds up 'read-next' in
applications
  -R, --sort-records=#
                      Sort records according to an index.  This makes
your data much more localized and may speed up things (It may be VERY slow to
do a sort the first time!)

Default options are read from the following files in the given order:
/etc/my.cnf /var/lib/mysql/my.cnf ~/.my.cnf
The following groups are read: myisamchk
The following options may be given as the first argument:
--print-defaults      Print the program argument list and exit
--no-defaults         Don't read default options from any options file
--defaults-file=#     Only read default options from the given file #
--defaults-extra-file=# Read this file after the global files are read

Possible variables for option --set-variable (-O) are:
key_buffer_size       current value: 520192
read_buffer_size      current value: 262136
write_buffer_size     current value: 262136
sort_buffer_size      current value: 2097144
sort_key_blocks       current value: 16
decode_bits           current value: 9
```

Listing 17.3 myisamchk output. (continued)

To perform a full check of all tables in a database or on the entire server, you can use the mysqlcheck utility included in the MySQL distribution. Below are some examples of mysqlcheck use:

```
mysqlcheck -A -C --auto-repair -u root -p secret -h db1
```

Connect to host db1 as user root with the password secret. Check all tables that have changed since the last time they were checked or that were not closed properly. Repair tables that failed the check.

```
mysqlcheck -C --auto-repair -u root -p secret -h db1 sales
```

Check all tables, repairing when necessary, on server db1.

Although it is possible to repair corrupted MyISAM tables, you should rarely need to. If you are experiencing frequent and unexplained corruptions, you should investigate the cause of the problem immediately. First, check your hardware and ensure that your operating system is using the latest patches. With Linux, watch out for non-standard kernel patches and new hardware drivers, as well as the known buggy ones. With other operating systems, ensure that all virtual memory and I/O functions are operating correctly. After the initial hardware and operating system sanity checks, if the corruption persists, chances are you have found a bug. If you are able to produce a test case that demonstrates the problem, chances are the bug will be fixed in the next release of MySQL. However, if you cannot reproduce it, your best bet is to try converting your tables to InnoDB.

MyISAM tables that you are willing to make read-only can be compressed with myisampack. For example:

```
myisampack  /var/lib/mysql/personnel/employee
```

will compress the employee table residing in the personnel database assuming that the data directory is /var/lib/mysql. A table can be packed even without shutting the server down if you lock it. For example:

```
LOCK TABLES personnel.employee READ;
FLUSH TABLE personnel.employee;
/* run myisampack here */
UNLOCK TABLES;
```

Compression is done on a per-record basis and will be particularly effective if you have long records.

As far as the issues of corruption are concerned, InnoDB tables do not require much maintenance. Part of the reason is that they are transactional, and even if the server crashes, in most cases the table space is brought into a consistent state during recovery. Another factor is that InnoDB tables have more safety checks. We have noticed that InnoDB tables have fewer stability problems than MyISAM tables, especially in applications with large tables.

NOTE

CHECK TABLE is supported for InnoDB tables, but REPAIR TABLE is not.

MyISAM tables, especially the ones with many keys, should periodically be processed by ANALYZE TABLE. For example:

```
ANALYZE TABLE t1;
```

Using this command updates the statistics about key distribution, which helps the optimizer make better decisions concerning which key to use.

Some queries may perform better if the records in the data file are in a certain order (for both MyISAM and InnoDB tables). If you have such queries, you may want to experiment with ALTER TABLE ... ORDER BY command, for example:

```
ALTER TABLE book ORDER BY author;
```

This will physically sort records in the *book* table in the order of the value of the author column. The ORDER BY expression does not have to be a key—it can be anything MySQL is capable of evaluating.

CHAPTER 18

Exploring MySQL Server Internals

One of the great advantages of MySQL is that its source is publicly available. This means that the opportunities for customization are limited only by your programming skills. This task is not for the timid, though. MySQL source is not easy to understand at first glance, and many very good programmers have been frustrated in the past trying to do it. However, the challenge can be conquered. This chapter serves as the first steppingstone for the brave souls who dare to defy the beast.

Getting Started

There could be several reasons why you want to look at the source of MySQL. You may be simply interested in learning how it works. Or you may want to fix a bug, or write and maintain a patch. Perhaps you would like to add a small feature or try porting MySQL to a new platform. The nature of your goal will affect your method of getting the source. If you just want to fix something in the current version and do not plan to keep up with the current development or upgrade for a long time, a regular source distribution is the way to go. However, if you plan to introduce a change and then maintain it, keeping up with all the fixes and updates from MySQL AB, you should use the BitKeeper source tree.

BitKeeper is a revision control tool produced by BitMover, Inc. that MySQL AB uses to manage MySQL source. The tool is available from www.bitmover.com. If you do not plan to make any internal proprietary modification to the MySQL source, the Open Logging license will allow you to use it for free. Open Logging

means that whenever you commit your changes, the description is forwarded to the central BitMover repository and logged there. You can find the details of the Open Logging license at www.bitmover.com/Sales.Licencing.Free.html. If the Open Logging license does not meet your needs, you can obtain a commercial license from BitMover. If you're interested, send a request to sales@bitmover.com.

If you plan to use the BitKeeper source tree, download BitKeeper from www.bitmover.com and install it on your system. Once you get it up and running, it is recommended that you run bk helptool and study the basic BitKeeper concepts and commands. One of the strengths of BitKeeper is a very gentle learning curve—the concepts are easy to grasp, and the commands are intuitive.

If you do not expect to use the BitKeeper tree and prefer to just use the raw source distribution, see Chapter 3 for the source build instructions.

To access the files in a remote BitKeeper repository, you need to clone it, or in other words, make a duplicate copy of it locally. A group of changed files is called a changeset. Once a group of files have been modified, the user will *commit* the changeset. The committed changes can optionally be *push*ed to the central repository if no other user has *push*ed his or her changes that the current local repository does not have. Otherwise, those changes must be *pull*ed first.

Now let's get the tree. If you want to be on the bleeding edge of the MySQL 4.0 branch, type

```
bk clone bk://work.mysql.com:7001
```

To start with a release version as opposed to mid-release, type

```
bk clone -rmysql-version bk://work.mysql.com:7001 mysql
```

replacing *version* with the actual version you are trying to get. For example:

```
bk clone -rmysql-4.0.2 bk://work.mysql.com:7001 mysql
```

For 3.23, the hot mid-release tree is available with

```
bk clone bk://work.mysql.com:7000 mysql
```

and to get the release version:

```
bk clone -rmysql-version bk://work.mysql.com:7000 mysql
```

replacing *version* with the actual version. For example:

```
bk clone -rmysql-3.23.52 bk://work.mysql.com:7000 mysql
```

There is a small chance that the name of the repository host and the ports can change in the future. The most current information on accessing the BitKeeper tree is available at www.mysql.com/doc/en/Installing_source_tree.html.

The clone command creates a directory called mysql with the MySQL source plus the internal revision control BitKeeper files. Your next step is to change the current directory to the root of the tree:

```
cd mysql
```

And check out the files:

```
bk -r get -Sq
```

Before you edit a file, you must use bk edit to notify BitKeeper that you are going to do so. You can tell it that you are going to edit everything by using bk -r edit.

The first thing you should do—unless you want the MySQL world to know about every modification you make in your tree—is edit BitKeeper/triggers/post-commit and change the mail addresses to your company's internal ones, or get rid of the script altogether with bk rm BitKeeper/triggers/post-commit. Otherwise, every time you commit a changeset, your changes will be e-mailed to internals@lists.mysql.com and to an internal MySQL developer alias.

We recommend that you subscribe to internals@lists.mysql.com. The primary reason is that it notifies you of updates made to the tree so that you know when to pull the changes. You can also follow some of the development discussions. To subscribe, visit www.mysql.com/documentation/lists.php and follow the instructions on the Web page.

To be able to compile MySQL from the BitKeeper tree, you need a few extra tools in addition to the ones required for a regular source build: autoconf, automake, and bison. If you do not have those tools installed already, you can obtain them from ftp://ftp.gnu.org/pub/gnu/. You need autoconf to generate the configure script and a few supplementary files; automake generates Makefile.in files used by the configure script to generate Makefiles; and you need bison to generate the SQL parser code.

Now that we have taken care of the logistics, let's get down to business and do the build. In the tree you see the BUILD directory, which contains scripts for building MySQL on several different platforms and architectures. As of this writing, the following scripts are available:

- compile-alpha
- compile-alpha-ccc
- compile-alpha-cxx
- compile-alpha-debug
- compile-ia64-debug-max
- compile-pentium

- compile-pentium-debug
- compile-pentium-debug-max
- compile-pentium-debug-no-bdb
- compile-pentium-debug-openssl
- compile-pentium-gcov
- compile-pentium-gprof
- compile-pentium-max
- compile-pentium-myodbc
- compile-pentium-mysqlfs-debug
- compile-pentium-pgcc
- compile-solaris-sparc
- compile-solaris-sparc-debug
- compile-solaris-sparc-fortre
- compile-solaris-sparc-purify

The most commonly used are compile-pentium, compile-pentium-debug, compile-pentium-max, compile-pentuim-debug-max, compile-pentium-max, compile-solaris-sparc, and compile-solaris-sparc-debug. These should work out of the box on a sanely configured system. The others may or may not work, and may require some tweaking to get them to work on a particular system. If none of the available scripts fit your needs, you can write your own by using compile-pentium (or any other compiler) as a template to start with.

Begin the compilation by running the appropriate build script. For example:

```
BUILD/compile-pentium-debug
```

This builds a debugging version of MySQL on an x86 Linux or FreeBSD system. After the build finishes, you should run the new binary through the test suite to make sure that you are starting out with a problem-free binary (or at least, no problems that the test suite checks for):

```
cd mysql-test
./mysql-test-run
```

If all tests pass, you can start the new development. Otherwise, it is time to debug. If you have gdb installed and are running X Windows, you can run the failed test in a debugger by typing

```
./mysql-test-run --gdb failed_test_name
```

replacing failed_test_name with the actual name of the test that failed. For a non-replication test, you will see just one xterm window with a debugger

prompt ready to run the test instance of the server with all arguments preloaded. For a replication test, there will be two windows: one for the master and one for the slave.

Tour of the Source

Before we begin the tour, we need to mention that we use the MySQL 4.0 source as a reference. However, there have been no major source tree reorganizations since 3.23, so most of what we say here will apply to the 3.23 source. Let's begin by briefly commenting on each source-related subdirectory of interest at the root of the tree:

- **include:** Header files.
- **mysql-test:** The test suite.
- **sql-bench:** The MySQL benchmark test suite written in Perl.
- **strings:** The MySQL string library.
- **libmysql_r:** An empty directory used for building the thread-safe client library.
- **libmysqld:** The library for stand-alone MySQL applications with access to server functionality. See www.mysql.com/doc/en/libmysqld_overview.html for details. As of this writing, this is still a work in progress.
- **vio:** Stands for Virtual I/O. A portability/convenience network routine wrapper library, it was originally created to facilitate support for Secure Socket Layer (SSL) connections.
- **mysys:** Portability routines for file system operations, implementations of various algorithms and data structures, and the ISAM/MyISAM key cache—in other words, Monty's magic tool case.
- **myisammrg:** Implementation of the MyISAM merge table handler, which allows several MyISAM tables of the same structure to be viewed as one.
- **merge:** The non-MyISAM-specific part of the merge table handler support code. It can be potentially reused for creating merge tables with other table handlers.
- **client:** Client utilities such as mysql, mysqladmin, mysqldump, and mysqlcheck.
- **readline:** GNU readline library for the command-line client.
- **innobase:** The InnoDB table handler implementation.
- **dbug:** The debugging library.
- **heap:** The HEAP (in-memory) table handler implementation.

- **isam:** The ISAM (3.22 compatibility) table handler implementation.
- **tools:** The directory for threaded client code that requires linking against libmysqlclient_r.
- **os2:** OS/2 port-specific code.
- **zlib:** The compression library zlib code.
- **extra:** A collection of miscellaneous helper utilities.
- **sql:** The SQL parser, query optimizer, server socket code, table handler interface classes, table lock manager, table cache, query cache, replication implementation, logging code, DNS cache, and a few other things.
- **regexp:** The GNU regular expressions library.

Execution Flow

Now let's jump into the code by following the basic execution flow. We descend down the call hierarchy and periodically explore our surroundings. Our tour is not comprehensive, but we hope it will be a good starting point for you on the way to understanding MySQL source.

You may find it helpful to follow this discussion in the source of MySQL. To make this tour more exciting, start mysqld in a debugger and set a breakpoint in main() or in handle_one_connection(), or perhaps somewhere further down the call hierarchy. Then run a simple query hitting the breakpoints and stepping into functions we discuss here. Also do some of your own research and examine different variables and structures, and step into the functions you find interesting.

When the server is launched, execution begins in main(), which is defined in sql/mysqld.cc. main() performs various initializations; then calls load_defaults() (defined in mysys/default.c) to read the options from my.cnf, followed by more initializations and a call to get_options() (defined in sql/mysqld.cc as well) to parse the command-line arguments. More initializations follow; then open the logs with a call to open_log() (defined in sql/mysqld.cc) once per log.

After initializations of various components, we finally get to some meat. main() transfers control (on most systems through a direct call, but on Windows NT by creating a thread) to handle_connections_sockets(), which is also defined in sql/mysqld.cc.

Next, handle_connections_sockets() loops in the standard POSIX select()/accept()/invoke client handler loop until the server is shut down. The client handling takes place in two stages. handle_connection_sockets() instantiates the THD object, which is defined in sql/sql_class.h and acts as a connection descriptor. Then create_new_thread() is invoked with the THD object as

an argument. The top part of create_new_thread() checks to see if the configuration limits allow another connection and then authenticates the client. If everything is fine, control is transferred to handle_one_connection(). If there is a waiting thread in the thread cache, the control transfer happens by waking it up. Otherwise, a new thread is created with handle_one_connection() as the start routine. handle_one_connection() is defined in sql/sql_parse.cc.

At this point, handle_one_connection() performs some initializations, and then enters the command loop:

```
while (!net->error && net->vio != 0 && !thd->killed)
   {
     if (do_command(thd))
           break;
   }
```

In other words, handle_one_connection() reads loops and executes a command until you get COM_QUIT (sent to the server by the client API call mysql_close()), the network connection breaks down, or the connection thread is terminated with the KILL command or during the server shutdown.

The do_command() method, defined also in sql_parse.cc, in turn performs some initializations, reads a MySQL client-server protocol packet with a call to my_net_read()(defined in sql/net_serv.cc), and invokes dispatch_command() (also defined in sql/sql_parse.cc). dispatch_command() performs some basic initializations and then enters a long switch() statement on the value of the command.

You can view the full listing of all available commands in include/mysql_com.h in the enum enum_server_command. They correspond to the Command field in the output of SHOW PROCESSLIST. As of this writing, the following commands are available in the 4.0 tree:

- **COM_SLEEP:** The default command used mostly for status reporting in SHOW PROCESSLIST.
- **COM_QUIT:** Terminates the connection, sent by the client API call mysql_close().
- **COM_INIT_DB:** Selects a new default database, sent by the client API call mysql_select_db().
- **COM_QUERY:** Executes a query, sent by the client API call mysql_real_query().
- **COM_FIELD_LIST:** Lists fields in a table, sent by the client API call mysql_list_fields().
- **COM_CREATE_DB:** Creates a database, sent by the client API call mysql_create_db().

- **COM_DROP_DB:** Drops a database, sent by the client API call mysql_drop_db().
- **COM_REFRESH:** The equivalent of several combined FLUSH [TABLES | HOSTS | LOGS | PRIVILEGES | MASTER | SLAVE] commands based on the argument bitmask. Sent by the client API call mysql_refresh().
- **COM_SHUTDOWN:** Shuts down the server.
- **COM_STATISTICS:** Sends a short server statistics report string to the client. Sent by the client API call mysql_stat().
- **COM_PROCESS_INFO:** The equivalent of SHOW PROCESSLIST. Sent by the client API call mysql_list_processes().
- **COM_CONNECT:** The Command field of the THD object is set to this value during the authentication handshake stage.
- **COM_PROCESS_KILL:** Kills a connection thread; the equivalent of KILL. Sent by the client API call mysql_kill().
- **COM_DEBUG:** Dumps some debugging information into the error log. The level of detail depends on the server compilation options. Sent by the client API call mysql_debug().
- **COM_PING:** Responds with the OK packet to the client, reassuring it that the server is still alive. Sent by the client API call mysql_ping().
- **COM_TIME:** A special value to accommodate for Monty's clever slow log filter hack.
- **COM_DELAYED_INSERT:** An internal value used for showing delayed insert threads in SHOW PROCESSLIST.
- **COM_CHANGE_USER:** Changes the current user to a different one. Sent by the client API call mysql_change_user().
- **COM_BINLOG_DUMP:** Dumps the replication binary update log. Used by the slave server and the mysqlbinlog utility to get the log feed from the master over the network.
- **COM_TABLE_DUMP:** Dumps the table contents. Used by the slave server to fetch a table from the master.
- **COM_CONNECT_OUT:** Used by the slave thread to log the connection establishment to the master.
- **COM_REGISTER_SLAVE:** Reports the slave location to the master.

Of all the commands, the most interesting one is COM_QUERY. Let's take a look at the execution path in this direction. It is handled in the case COM_QUERY statement, which does some white space pruning, and then passes the query on to the parser with a call to mysql_parse(), which is also defined in sql/sql_parse.cc.

The mysql_parse() method starts by calling mysql_init_query(), which performs some initializations. Then you initialize thd->query_length and proceed to call query_cache_send_result_to_client(). Successful return or a system error means there is nothing else to do. However, if the call reports a cache miss, you actually need to parse and execute the query. You first call yyparse(), which is located in sql_yacc.cc (generated by bison from sql_yacc.yy), and which contains the SQL grammar definition along with actions that execute when a certain grammar element is encountered.

All parsing happens in yyparse(), which calls yylex() (defined in sql_lex.cc). Note that unlike many other projects that have a parser, MySQL does not use lex (or flex) to generate the lexical scanner, but instead has its own human-coded scanner aided by a generated static hash. The human-coded part of the lexical scanner is in sql/sql_lex.cc. The generated hash is written to lex_hash.h during builds if it is not present or if sql/lex.h has been updated from lex.h by the sql/gen_lex_hash utility (which is produced by compiling sql/gen_lex_hash.cc, the source of the hash generator).

Thus, lex.h defines all the tokens recognized by MySQL as something special. They are divided into two groups. The symbols array contains the operators, command keywords, and special modifiers. The sql_functions array contains the names of all functions you can use with MySQL in a query. Note that the lexical scanner works with the grammar in such a way that you can use any keyword as a table or a database name. For a keyword to be used this way, you must list it in the keyword rule in sql/sql_yacc.yy. In theory, you modify the parser in this way:

```
CREATE TABLE create(n INT);
```

The only reason it gives a syntax error is that to keep the parser ANSI-SQL compliant, not because it would be difficult to parse.

The yyparse() method fills out a structure of type LEX, which is defined in sql/sql_lex.h. It has a lot of members to accommodate for the rich variety of syntax that MySQL supports. We direct your attention to the most influential, so to speak.

THD* thd is a pointer to the current connection thread descriptor. Most functions in the sql directory require a THD pointer as an argument.

enum_sql_command sql_command determines what kind of query it is—for example, SQLCOM_SELECT, SQLCOM_INSERT, or SQLCOM_UPDATE. You can view the full listing of all possible values in the definition of enum_sql_command type in sql/sql_lex.h. The command type is later used in the switch statement in mysql_execute_command().

SELECT_LEX select_lex contains what you might actually call a parse tree, or perhaps a better term would be parse mini-forest, parse grove, or perhaps parse garden. The reason for the vegetation terminology is that several trees are encapsulated in the structure. We examine the following key members of SELECT_LEX:

- **Item *where:** The root of the parse tree for the WHERE clause.
- **Item* having:** The root of the parse tree for the HAVING clause.
- **SQL_LIST order_list:** The list of the roots of the ORDER BY expression parse trees.
- **SQL_LIST group_list:** The list of the roots of the GROUP BY expression parse trees.
- **SQL_LIST table_list:** The list of the tables in the FROM clause.
- **List<Item> item_list:** The list of the roots of the column expression parse trees after SELECT.

The Item class and its subclasses (always named Item_*, defined in sql/item_*.h, and implemented in sql/item_*.cc) are the nodes of an expression parse tree. Expressions are evaluated with a call to Item::val(), Item::val_int(), or Item::val_str(), depending on the type context on the root node of the tree.

SQL_LIST is a list of generic pointers. order_list and group_list will have pointers to type ORDER defined in sql/table.h, while table_list will have pointers to type TABLE_LIST, defined also in sql/table.h.

Now let's return from yyparse() and its structures to mysql_parse() and continue our tour down the execution path. Next, mysql_execute_command() is called. At the very top, check to see if you are in the slave thread; if so, the replication rules tell you to exclude the query. If not, you move forward, increment the appropriate command counter (based on the value of lex->sql_command), and hit a very long but nevertheless very important switch statement.

There are many cases, one for each SQL command type. However, each of them is handled in a rather similar fashion. First, check access privileges with the check_access() call defined in sql/sql_parse.cc. If the query is using tables, you need to open and lock them with a call to

open_and_lock_tables(), defined in sql_base.cc. open_and_lock_tables(), is an important function, so let's take a closer look at it.

open_and_lock_tables() is actually a convenience wrapper that calls open_tables(), and if that succeeds, it then calls lock_tables(). Both open_tables() and lock_tables() are defined in sql/sql_base.cc. open_tables(), in turn, simply iterates through the list of tables the query will be using, opening each with a call to open_table()(also defined in sql_base.cc), and dealing

MySQL Memory Management

While we are talking about query parsing, this would be a good place to mention how MySQL manages memory. It uses two methods. Larger blocks are allocated by calling my_malloc(), defined in mysys/my_malloc.c. If the server is compiled with the -DSAFEMALLOC flag, my_malloc() is replaced with #define in include/my_sys.h to _mymalloc(), which in turn is defined in mysys/safemalloc.c. Without SAFEMALLOC, my_malloc() is just a rather straightforward wrapper around the standard malloc() call. However, with SAFEMALLOC, my_malloc() stores some extra information in the allocated block that will help track down leaks and other memory errors. Blocks allocated with my_malloc() must be freed with my_free().

Smaller memory allocations—especially if they are done in a loop or for the purpose of creating a complex object hierarchy—are done from a memory pool. A memory pool is represented by the MEM_ROOT structure, defined in include/my_alloc.h. Memory pool functions are defined in mysys/my_alloc.c. A memory pool is initialized with init_alloc_root() and released with free_root(). Memory from a pool can be allocated with alloc_root(), memdup_root(), or strmake_root().

One area where the memory pool allocation comes in handy is managing the parse tree. To avoid the nightmare of having to traverse the tree and freeing each node in the correct order so you do not cut off the branch you are sitting on, you should instead always allocate the parse tree nodes (or, in Monty's terminology, items) from the thread memory pool. (See the mem_root member of THD in sql/sql_class.h.) Then you can dispose of the tree by simply freeing the pool.

Objects that need to have the memory containing them allocated from the thread memory pool are handled by being subclasses of Sql_alloc, defined in sql/sql_list.h. Sql_alloc implements the new operator, equating it to a call to sql_alloc(),defined in sql/thr_malloc.c. sql_alloc() is a convenience wrapper to call alloc_root() with the mem_root of the current thread descriptor as an argument. The delete operator is also implemented in a simple way—it does nothing. It indeed has nothing to do—all memory from the pool will be freed at the end of the query with a call to free_root().

with a few bumps along the road that may have been thrown in your path by a concurrent execution of FLUSH TABLES in some other thread.

open_table() first tries to find the table entry in the table cache with a call to hash_search() (defined in mysys/hash.c), passing it &open_cache as an argument. If that fails, it will call open_unireg_entry() (defined in sql/sql_base.cc), which is basically a safety/convenience wrapper around openfrm() (defined in sql/table.cc).

openfrm() mainly does three things. First, it parses the table definition file (.frm). As soon as the table type is known in the parsing process, it instantiates the table handler object. The pointer to this object is stored in the file member of the TABLE structure (defined as struct st_table in sql/table.h) and then aliased to TABLE with a typedef in sql/handler.h by calling get_new_handler() (defined in sql/handler.cc). Then after some more parsing, openfrm() calls the ha_open() method of the handler object, which performs the handler-specific initializations.

ha_open() is a method of the abstract handler class defined in sql/handler.h, and is implemented in sql/handler.cc. While ha_open() is not a virtual method, it calls handler::open(), which is pure virtual. All pure virtual methods of the handler class, and the relevant virtual but not pure, are implemented or reimplemented in the specific handler subclass. Table 18.1 shows the available handle interface classes and the files where they are defined and implemented (all listed files are in the sql directory).

Table 18.1 Handle Interface Classes

CLASS	DEFINITION FILE	IMPLEMENTATION FILE
ha_myisam	ha_myisam.h	ha_myisam.cc
ha_innobase	ha_innodb.h	ha_innodb.cc
ha_heap	ha_heap.h	ha_heap.cc
ha_isam	ha_isam.h	ha_isam.cc
ha_myisammrg	ha_myisammrg.h	ha_myisammrg.cc
ha_isammrg	ha_isammrg.h	ha_isammrg.cc
ha_berkeley	ha_berkeley.h	ha_berkeley.cc

The methods of the handler classes are later used to retrieve existing records from a table, insert new ones, and update the old ones.

Now let's go back up the call stack ladder all the way up to open_and_lock_tables(). If the opening was successful, you can proceed to lock the tables. You call lock_tables(), which in turn calls mysql_lock_tables() after some initializations, which takes you into sql/lock.cc, the implementation of MySQL table lock manager. The code looks rather complex, but the idea is quite simple—for MyISAM and HEAP tables, if nobody holds the lock, grant it to the requesting thread. If somebody holds the lock already, enqueue and suspend the thread so it will wait for its turn. InnoDB table lock requests essentially bypass the lock manager—the locks are always granted on the table lock manager level and are being dealt with inside the InnoDB handler.

Having checked the access and then opened and locked the tables, you could initialize some variables and perform some other checks, and then call the function that does the core part of the work for the given command. Table 18.2 lists some common commands, their handler functions, and the file where the handler function is defined for each (all the files are in the sql directory).

Table 18.2 Common Commands and Their Handler Functions

COMMAND	FUNCTION	FILE
SQLCOM_SELECT	handle_select()	sql_select.cc
SQLCOM_CREATE_TABLE	mysql_create_table()	sql_table.cc
SQLCOM_ALTER_TABLE	mysql_alter_table()	sql_table.cc
SQLCOM_UPDATE	mysql_update()	sql_update.cc
SQLCOM_INSERT	mysql_insert()	sql_insert.cc
SQLCOM_DELETE	mysql_delete()	sql_delete.cc
SQLCOM_DROP_TABLE	mysql_rm_table()	sql_table.cc
SQLCOM_SHOW_PROCESSLIST	mysqld_list_processes()	sql_show.cc
SQLCOM_SHOW_STATUS	mysqld_show()	sql_show.cc
SQLCOM_SHOW_VARIABLES	mysqld_show()	sql_show.cc

After mysql_execute_command(), the possible execution paths widely diverge. In this tour we visit the most difficult one: SQLCOM_SELECT. Others will be left as an exercise for the reader. If you can handle the thorny path of SQLCOM_SELECT, you will find other execution paths quite manageable to follow.

Let's descend into sql/sql_select.cc. The entry point is handle_select(), which is a wrapper around mysql_select(). Note the last argument of handle_select(), which eventually gets passed to mysql_select(). It is a pointer to the select_result class, which is defined in sql_class.h. select_result is an abstract class that defines an interface to handle the output of a SELECT query. It has several subclasses, also defined in sql/sql_class.h:

- **select_send:** Sends the data directly to the client.
- **select_export:** Sends the data to a file with column and line terminators.
- **select_dump:** Sends just one record of the data into a file with no delimiters or terminators.
- **select_insert:** Inserts the selected records into another table.

- **select_create:** A subclass of select_insert with the additional task of creating a table in which to insert the records.
- **select_union:** Used for processing the results of each individual SELECT when performing UNION.
- **multi_update:** Used for storing the matching record references when processing a multi-table UPDATE.
- **multi_delete:** Used for storing the matching record references when processing a multi-table DELETE.

These classes are implemented in sql/sql_class.cc.

Now let's continue with mysql_select(). mysql_select() begins with six initialization calls of the setup_* family: setup_tables(), setup_fields(), setup_conds(), setup_order(), setup_group(), and a little later on, setup_ftfuncs(). setup_tables(), setup_fields(), setup_conds(), and setup_ftfuncs() are defined in sql/sql_base.cc. setup_group() and setup_order() are defined in sql/sql_select.cc. All of these functions have a rather self-explanatory name in this context with the exception of setup_conds(). The term cond refers to the WHERE clause condition expression parse tree or the parse tree of the outer join ON clause. The initialization job performed is rather complicated, but it can be summarized as finding and connecting the necessary links among the variety of structures associated with the different elements of a SELECT query.

At the very top, mysql_select() defines a number of local variables to be used later on. Let's take a closer look at the one that is holding the whole thing together: JOIN join. The JOIN class is defined in sql/sql_select.h. No implementation is provided because it has no methods. You could call JOIN the SELECT query descriptor object. As the query is being processed, all the relevant information about the state of the work is stored under JOIN.

Probably the most essential part of the JOIN descriptor is join_tab, an array of JOIN_TAB structures. JOIN_TAB is also defined in sql/sql_select.h and stores the information about a table pertaining to the process of retrieving the records for the join. Note that conceptually, MySQL treats selecting from just one table as a degenerated join; thus, every SELECT query becomes a join, even if it uses only one table.

mysql_select() continues with a few more initializations and a call to setup_procedure(), defined in sql/procedure.cc. Procedure in this context is a possibly user-written extension that intercepts and modifies the results of a SELECT. The most well-known procedure is ANALYSE(), which helps you determine the best type for your column if you do SELECT col_name FROM tbl_name PROCEDURE ANALYSE(). It is implemented in sql/sql_analyse.cc, and the procedure interface is defined in sql/procedure.h.

At this point, you see a massive initialization of join members, followed by a call to the prepare() method of the select_result object we discussed earlier. Now let's move on to optimize_cond(), defined also in sql/sql_select.cc. optimize_cond() prunes the WHERE clause tree, making it as simple as possible for the optimizer to deal with. If you notice that your WHERE clause is always false (cond_value == Item::COND_FALSE), you call return_zero_rows() (which is also defined in sql/sql_select.cc), clean up, and return.

Next, you check to see if you can optimize a COUNT(*), MIN(), or MAX() in a query without GROUP BY. Indeed, the row count could be read from the table descriptor if there is no WHERE clause. Or if you can use a key capable of range lookup, you can quickly pull the minimum or the maximum by reading the index. All of this logic is in opt_sum_query(), defined in sql/opt_sum.cc. Perhaps a terminology clarification would be helpful here. For one reason or another, Monty decided to call the aggregate functions "summary." Therefore, he used the suffix sum in all variables and filenames that had to do with aggregate functions.

Let's move on to call get_sort_by_table(), also defined in sql/sql_select.cc. The ultimate question is whether you can avoid having to create a temporary table if you have GROUP BY or ORDER BY. You would definitely have to do that if the ORDER BY and GROUP BY expressions are such that you cannot just read one table in a certain order and get the correct order for your query. get_sort_by_table() tells you if such a table exists.

Your next step is a call to make_join_statistics(), defined in sql/sql_select.cc. make_join_statistics() is aided by a number of helper subroutines (update_ref_ and_keys(), set_position(), create_ref_for_key(), join_read_const_table(), make_select(), get_quick_record_count(), find_best_ combination()) and determines the join processing strategy—which tables are going to use which keys, if any, and in what order the tables are going to be joined. The helper functions are all defined in sql/sql_select.cc, except for make_select(), which is defined in sql/opt_range.cc.

While the entire flow of execution of make_join_statistics() is beyond the scope of our source code tour, we should mention a few highlights. update_ref_and_keys() figures out what keys could possibly be used for each table based on the table structure and the WHERE clause. set_position() shuffles a table into the given join order position in the join table list after the const tables. (A const table is a table that you know in advance will yield no more than one record for the query.) create_ref_for_key() fiddles with the keys and tries to figure out the best way to use them. join_read_const_table() is invoked to read a const table for the purpose of being able to make better optimization decisions, and possibly even short-circuit the execution and provide the result

early without ever performing a true join. make_select() instantiates and initializes the SQL_SELECT object, which acts as a descriptor for the key range optimization used in the subsequent call to get_quick_record_count().

get_quick_record_count() performs some structure initializations, determines if the key range lookup optimization is possible, and estimates how many records you would have to read from a table if you used a key range. In Monty's terminology, "quick" stands for "read records using a key range." You can only guess why he called it "quick"—possibly because reading a key range is quicker than scanning the whole table. find_best_combination() actually makes the decision as to which key to use for each table, and what order they should be joined in.

Let's now return from make_join_statistics() to mysql_select(). You tell the select_result object to

1. Get ready for the data with a call to select_result::initialize_tables().
2. Perform a check to see if you have already obtained the entire result in make_join_statistics() when you read the const tables.
3. Short-circuit the execution with an error if it looks like you will have to examine more records than the max_join_size setting allows.
4. Unlock the const tables (unless they have been locked with LOCK TABLES) since you do not need to use them any more.
5. Fix up the WHERE clause for an outer join to be equivalent to 1 (or always true) if you have an outer join without WHERE.

Let's proceed with a call to make_select() to instantiate and initialize the key range optimization descriptor SQL_SELECT, and then call make_join_select(). make_join_select(), which is defined in sql/sql_select.cc, further prepares and refines the structures describing the query execution plan. Then, remove_const() is invoked to simplify the ORDER BY expression and to decide if you can deliver the result by simply reading the records in the order of a certain key. You then make an attempt to optimize SELECT DISCTINCT ... ORDER BY with a call to create_distinct_group() (defined in sql/sql_select.cc), followed by another call to remove_const()—this time to optimize GROUP BY and a few other GROUP BY and ORDER BY strategic preparations.

Next comes the decision as to whether you should use a temporary table. The temporary table will be used if

- DISTINCT is used in the query and you have gotten that far without optimizing it.
- You have ORDER BY or GROUP BY on fields that are not in the first table in the join order.

- GROUP BY and ORDER BY expressions have not been simplified by this point to make one automatically take care of the other.
- The user has explicitly requested the use of a temporary table with the SQL_BUFFER_RESULT option.

Now you call make_join_readinfo(), defined also in sql/sql_select.cc. make_join_readinfo() sets the function pointers in the JOIN_TAB elements of the join descriptor for each table that will be called to read the first and the next record. The functions pointed to are wrappers around the direct handler object method calls. All of them are defined in sql/sql_select.cc, and their names start with join_. Usually, the name of the function also includes the word read—for example, join_read_key(), join_read_last(), join_read_first().

A few more optimization adjustments and structure fix-ups, and the query plan is ready. You check to see if all the user wants is to know the query plan and not the query result (EXPLAIN SELECT). If that's the case, you call select_describe() and goto err to the bottom to do the cleanup and return.

Now is the time to do the real work of actually reading the records. Let's first perform the initial full-text search with a call to init_ftfuncs() (defined in sql/sql_base.cc).Then create a temporary table if necessary with a call to create_tmp_table() (defined in sql/sql_select.cc). The conditional block for if (need_tmp_table) is quite lengthy. The following applies only inside this block.

If the first table needs to be traversed in a sorted order on something other than a key, you call create_sort_index(). This function prepares for a call to filesort() (defined in sql/filesort.cc); calls filesort(), which will bring back the list of record positions in the correct sort order); and then sets things up in the JOIN_TAB descriptor so that the actual records will be read in the correct order in do_select(). do_select() is defined in sql/sql_select.cc and is the function that actually joins all the tables.

Perform a few more SELECT DISTINCT optimizations, and then populate the temporary table with a call to do_select(). You perform a few more field manipulations to ensure that the relevant field objects now point to the newly created temporary table with calls to change_to_use_tmp_fields() or change_refs_to_tmp_fields(), depending on the context. (Both functions are defined in sql/sql_select.cc.) If you have a procedure, you call update_refs() on the procedure object. In the case of non-cooperating GROUP BY and ORDER BY, and in a few other difficult situations, you are forced to use another temporary table. This is handled in a few dozen subsequent lines.

You are not using the old JOIN structure any longer, so you can release it with a call to join_free() (defined in sql/sql_select.cc) and set it up to read the temporary table data with a call to make_simple_join() (defined in

sql/sql_select.cc). This way, when you are done with the conditional and call do_select() at the end, the temporary table data will be read and sent to the client instead of the actual query tables. You finish off the JOIN structure setup with calls to calc_group_buffer() and count_field_types(). Now you have finished with the temporary table conditional block.

Now if you still have an unresolved ORDER BY or GROUP BY, you have some work to do. You perform more JOIN shuffling that culminates in a call to create_sort_index(), which will set up the JOIN structure to read the records in the sorted order. Now you can proceed with the long-awaited call to do_select(). It reads the data and disposes with it according to the classification of the select_result object. select_result is an abstract class, but as we have discussed earlier, it has a number of subclasses. If the object is of type select_send, which is the case if you arrive at mysql_select() through SQLCOM_SELECT, the data will go to the client.

After you perform some cleanup, you have finished. You successfully return to the caller, and then all the way back to the do_command() loop in handle_one_connection(). This completes our tour. Next we offer some practical examples of how you can extend the MySQL source.

General Guidelines for Extending the Source

If you plan to make your own addition to the MySQL code base, it is recommended, and in some cases required, that you adhere to the following guidelines:

- Write thread-safe code. This means, in particular, that if you use a global variable, you must protect it with a mutex. If you call a library function that you are not certain is thread safe, you must also use a mutex.
- For memory allocations, prefer THD::alloc or alloc_root() over my_malloc(). Use my_malloc() only for big chunks. This improves performance under high concurrency. The reason you should use my_malloc() instead of malloc() is that the uniform interface this provides makes memory debugging easier. Blocks allocated with my_malloc() should be freed with my_free(), and alloc_root() allocations are released when the pool is freed with free_root().
- Use mysys library calls for file operations. This solves a lot of porting issues and allows you to take advantage of any global file I/O optimizations.

- The mysys library also contains a collection of tools such as quick sort, radix sort, command-line and configuration file-option parsing, hash, dynamic array, soundex routine, file path manipulations, red-black tree, and a few others. You are encouraged to examine the code, understand it, and reuse it your additions.
- The strings library contains a large variety of string routines you may find useful. Using string routines from the strings library also ensures a greater degree of portability for your code.
- Do not use exceptions. Because of the poor stability record with exceptions in threads on several platforms, they are not used anywhere in MySQL code, and the default compilation flags disable them.
- Do not use C++ streams, STL classes, or any other C++-specific calls or objects that will require linking in libstdc++. Whenever possible, the compilation flags are chosen to be such that libstdc++ is not linked in.
- Implement the new and delete operators in your classes to use my_malloc() or alloc_root(). This allows more refined control over failed memory allocations and also makes it easier to get rid of libstdc++ dependency.
- Overall, try to avoid creating external library dependencies. This will simplify maintenance and facilitate porting.

Adding a New Native SQL Function

SQL functions in MySQL are implemented by subclassing the Item class, implementing the relevant methods (constructor(s), val_int(), val(), val_str(), fix_length_and_dec(), and optionally some others), and then hooking it up to the parser. Let's demonstrate the process through an example. We add two functions: WORD_COUNT() (to count the number of words in a text column) and REVERSE_PHRASE() (to reverse the order of words in a sentence).

First, you need to make a couple of decisions. You'll name the classes that implement your functions—Item_func_word_count and Item_func_reverse_phrase()—to be consistent with the names of other MySQL functions. You'll also put your definition and implementation code into new separate files–sql/item_custom.cc and sql/item_custom.h—to facilitate maintenance. Note that you do not have to create the new files; you could have put the code, for example, into sql/item_strfunc.h and sql/item_strfunc.cc in the existing source.

Now you are ready to go to work. As we list the source files, the comments will provide an explanation of what is happening.

First, create sql/item_custom.h, shown in Listing 18.1.

```
#ifndef ITEM_CUSTOM_H
#define ITEM_CUSTOM_H

#ifdef __GNUC__
#pragma interface
#endif

// Sub-class Item_int_func because the return result is always
// an integer
class Item_func_word_count: public Item_int_func
{
protected:
 /* used as a temporary buffer */
  String tmp_value;

public:
  /* constructor */
  Item_func_word_count(Item* a): Item_int_func(a) {}

  /* calculate the return value */
  longlong val_int();

  /* initialization function to set the maximum length and decimal
     precision
  */
  void fix_length_and_dec() { max_length=10; }

  /* provide function name for debugging code */
  const char* func_name() const { return "word_count"; }

};

// In this case we sub-class Item_str_func as the return value is a
// string
class Item_func_reverse_phrase: public Item_str_func
{
public:
  /* constructor */
  Item_func_reverse_phrase(Item *a): Item_str_func(a) {}

  /* compute the string value */
  String* val_str(String*);

  /* initialization function to set the maximum length and decimal
     precision
  */
  void fix_length_and_dec()
```

Listing 18.1 sql/item_custom.h. (continues)

```
    {
     /* maximum length the same as the argument */
      max_length = args[0]->max_length;
    }

  /* provide function name for debugging code */
  const char* func_name() const { return "reverse_phrase";}
};

#endif
```

Listing 18.1 sql/item_custom.h. (continued)

Now that you have defined the Item_func_word_count and Item_func_reverse_phrase classes, let's implement them. Create the sql/item_custom.cc file, shown in Listing 18.2.

```
#ifdef __GNUC__
#pragma implementation
#endif

#include "mysql_priv.h"

/* helper function */
static void reverse_str(char* start, char* end);

longlong Item_func_word_count::val_int()
{
  /*
   Evaluate and store the argument. In this case we have only one
   argument. If we had more, we would access them as args[1],
   args[2], args[3], etc
  */
  String* res = args[0]->val_str(&tmp_value);

  /* deal with SQL NULL argument value */
  if (!res)
  {
    null_value=1;
    return 0;
  }

  /* not NULL return value */
  null_value=0;

  /* set initial word count to 0 */
```

Listing 18.2 The sql/item_custom.cc file. (continues)

```
  longlong word_count = 0;

  /* flag to keep track if we are on a word as we traverse the string,
     initially we are not on a word
  */
  bool on_word = 0;

  /* point at the start of the evaluated argument string */
  const char* s = res->ptr();

  /* set a sentry at the end */
  const char* s_end = s + res->length();

  /* now iterate through the string */
  for (;s<s_end;s++)
  {
    /* If we see a space, we are not on a word any longer */
    if (isspace(*s))
    {
      if (on_word)
          on_word = 0;
    }
    /* Otherwise, if we see a non-space character, and we were not on
a word, we increment the counter and flip the on_word flag
    */
    else
    {
      if (!on_word)
      {
        word_count++;
            on_word = 1;
      }
    }
  }
  return word_count;
}

/* helper function */
static void reverse_str(char* start, char* end)
{
  while (start < end)
  {
    char tmp = *start;
    *start++ = *--end;
    *end = tmp;
  }
}
```

Listing 18.2 The sql/item_custom.cc file. (continues)

```
String* Item_func_reverse_phrase::val_str(String* str)
{
  /* Evaluate and store the argument. Note that the buffer comes to us
from the caller. We reuse that buffer, modify the argument in place,
and return it to the caller.
   */
  String* res = args[0]->val_str(str);

  /* deal with SQL NULL argument value */
  if (!res)
  {
    null_value=1;
    return 0;
  }

  /* not NULL return value */
  null_value=0;

  /* flag to keep track if we are on a word as we traverse the string,
     initially we are not on a word
  */
  bool on_word = 0;

  /* point at the start of the evaluated argument string */
  char* s = res->c_ptr();

  /* set a sentry at the end */
  char* s_end = s + res->length();
  char* word_start = 0;

  /* now iterate through the string */
  for (;s<s_end;s++)
  {
    /* If we see a space, we are not on a word any longer.
       We reverse the word.
    */
    if (isspace(*s))
    {
      if (on_word)
      {
          on_word = 0;
          reverse_str(word_start,s);
      }
    }
    /* Otherwise, if we see a non-space character, and we were not on
a word, we flip the on_word flag and remember the start of the word.
```

Listing 18.2 The sql/item_custom.cc file. (continues)

```
    */
    else
    {
      if (!on_word)
      {
        word_start = s;
           on_word = 1;
      }
    }
  }
  /* reverse the last word */
  if (word_start)
    reverse_str(word_start,s);

  /* now reverse the entire string */
  reverse_str(res->c_ptr(),s);

  /* we are now ready to return the result */
  return res;
}

/* wrappers for constructors used by the parser */
Item *create_func_word_count(Item* a)
{
  return new Item_func_word_count(a);
}

Item *create_func_reverse_phrase(Item* a)
{
  return new Item_func_reverse_phrase(a);
}
```

Listing 18.2 The sql/item_custom.cc file. (continued)

Both item_custom.h and item_custom.cc are available in the mysqld directory on the book Web site (www.wiley.com/compbooks/pachev).

Now you need to hook up your additions to the parser. Add the following two function declaration lines to sql/item_create.h. You can add them to the list of create_func_* functions (in alphabetical order) or at the very end:

```
Item *create_func_word_count(Item* a);
Item *create_func_reverse_phrase(Item* a);
```

Next, you need to edit sql/lex.h. Find the functions array, and add to it the following two structures. Again it does not matter where; reasonable choices are either at the end or alphabetically within the array:

```
 { "WORD_COUNT",       SYM(FUNC_ARG1),0,
CREATE_FUNC(create_func_word_count)},
```

```
{ "REVERSE_PHRASE",   SYM(FUNC_ARG1),0,
   CREATE_FUNC(create_func_reverse_phrase)}
```

The first member of the structure, as you may have guessed, is the name of the function as it is going to appear in the SQL syntax. Note that there is an alternative way to add a function to the parser. You can put 0 instead of the CREATE_FUNC() macro, replace SYM(FUNC_ARG1) with SYM(SOME_UNIQUE_SYM_NAME), add token SOME_UNIQUE_SYMNAME to sql/sql_yacc.yy, and then add a clause that will call the constructor of your Item to the simple_expr rule in the same file. The latter method is necessary when the function falls into the category of a simple 0-, 1-, 2-, or 3-argument function.

You then "legitimize" your item_custom.* addition to the source by adding #include "item_custom.h" in sql/item.h after all the other item_*.h include statements. Then edit sql/Makefile.am by adding item_custom.h to noinst_HEADERS and item_custom.cc to mysqld_SOURCES. Now you are ready to compile.

The first time, you have to run automake to regenerate sql/Makefile.in and then run ./configure to regenerate sql/Makefile. This is automatically taken care of if you run the BUILD/compile-* scripts. If you have already built the source, in theory you should be able to just type make in the sql directory and it should work. But sometimes you run into Makefile rule glitches, and dependencies do not get updated as they should. So as a rule, if you change Makefile.am, it is a good idea to have a clean start with your favorite BUILD/compile-* script. However, if your source has been updated and you just want to recompile it, usually make in the sql directory is sufficient.

Once the compilation finishes successfully, you are ready to test. Change to the mysql-test directory and put the test case script shown here into t/custom.test:

```
drop table if exists t1;
create table t1 (s text);
insert into t1 values ("Sally spawns C-shells, stack smashed, she
sees core"),
 ("If you want completion hash, don't forget to type /bin/bash"),
 ("Once again I get SIGBUS, how I'm sick of C++!"),
 ("It is not the C++ that's to blame for your SIGBUS, look at this
big dirty hack, it will surely corrupt the stack"),
 (NULL),
 ("");
select s, reverse_phrase(s),word_count(s) from t1;
drop table t1;
```

Run the following command to record the result:

```
./mysql-test-run --local --record custom
```

If you do not have any fatal bugs in the code that will crash the server on the test, the command produces the following output:

```
Installing Test Databases
Removing Stale Files
Installing Master Databases
020828  5:02:09  ../sql/mysqld: Shutdown Complete

Installing Slave Databases
020828  5:02:09  ../sql/mysqld: Shutdown Complete

Starting MySQL daemon
Loading Standard Test Databases
Starting Tests

 TEST                     USER   SYSTEM  ELAPSED        RESULT
-----------------------------------------------------------------
custom                    0.01     0.02     0.03       [ pass ]
-----------------------------------------------------------------

Ending Tests
Shutting-down MySQL daemon

Master shutdown finished
Slave shutdown finished
All 1 tests were successful.
```

Note that pass in the record mode does not mean the test works correctly. It means that there are no fatal errors, the test was able to run to completion, and the output was successfully recorded. Now study the recorded result:

```
cat r/custom.result
```

which gives you the following:

```
s       reverse_phrase(s)       word_count(s)
Sally spawns C-shells, stack smashed, she sees core     core sees
she smashed, stack C-shells,
spawns Sally     8
If you want completion hash, don't forget to type /bin/bash
/bin/bash type to forget don't hash,
completion want you If     10
Once again I get SIGBUS, how I'm sick of C++!   C++! of sick I'm how
SIGBUS, get I again
Once   10
It is not the C++ that's to blame for your SIGBUS, look at this big
dirty
hack, it will sure corrupt the stack   stack the corrupt sure will
it hack,
dirty big this at look SIGBUS, your for blame to that's C++ the not
is It      23

NULL    NULL    NULL
                0
```

This is exactly what we anticipated. If it had been different, you would have had two options. You could manually edit the result to make it what it is supposed to be, and then run the subsequent tests without the --record flag (but with --local, which tells mysql-test-run to start a special instance of the server on a nonstandard port). Or you could keep fixing up the source and running with --record on each iteration, and visually examine the result until it begins to look right.

If the test crashes in the record stage, or if the result is incorrect and a brief visual examination of the code does not reveal why, it might be the time to try running the code through the debugger. Type ./mysql-test-run --local --gdb custom and wait for an xterm window with the debugger prompt to appear. Then the ball is in your court. If you need a tutorial on how to use gdb, type info gdb at the shell prompt.

If you make some modifications in the code and would like to test if your function is still working, you should execute ./mysql-test-run --local custom. If everything is fine, the test will be reported as pass. If there is a problem, the test will report as fail, and you will see the output of diff -c between the expected and the actual result. You may also want to make sure that your changes have not broken the rest of the code by running the full regression test suite with ./mysql-test-run. Note that if you are running all tests, you do not need to add --local because the option will be added automatically.

Our previous examples show how to add a simple function returning an integer or a string accepting one string argument. This is by no means the full gamut of the functions that you could add to MySQL. If you would like to add a function that is significantly different from the ones we have studied here, it is recommended that you follow these steps:

1. Identify an already existing function that is semantically identical, or at least similar, to the one you are trying to implement.
2. Find its entry in the functions array in sql/lex.h.
3. If it is using the CREATE_FUNC() macro, track down the corresponding Item_* class in item_create.cc. Otherwise, check the symbol it is using, and track down the Item_* class in the simple_expr rule in sql_yacc.yy.
4. Use grep Item_func_my_func item_*.h item_*.cc to locate the files where the class is defined and implemented.
5. Open the found files and study the implementation.

One group of functions that may be of particular interest to you is the aggregate functions. To learn about them, study the definition and implementation of Item_sum_sum and other classes in item_sum.h and item_sum.cc.

Adding a UDF

Unlike a native function, which is integrated directly into the code base and becomes a full-fledged member of the function family, a user-defined function (UDF) is compiled as a separate module and is loaded at runtime. For you to be able to use a UDF, the mysqld binary must be capable of loading a shared library. This requirement creates some interesting implications on various platforms. On Linux, a statically linked server binary cannot load a UDF that is referencing internal symbols of the server. A dynamically linked server binary does not have any restrictions. On other platforms, the limitations and requirements for a UDF have not yet been thoroughly explored by the MySQL development team as of this writing, but it is expected that a dynamically linked binary should be able to load a UDF. The rule of thumb, therefore, is that if you plan to use a UDF, you should link the server binary dynamically.

Adding a UDF is documented in quite a bit of detail in the online manual at www.mysql.com/doc/en/Adding_UDF.html. An example of a UDF function is also provided with the source in sql/udf_example.cc (the documentation in the comments at the top is even better than the one in the manual). We will, however, complement the online documentation and the source example with one of our own by reimplementing WORD_COUNT() as a UDF.

First, create a file (in an arbitrary directory on the file system) called udf_word_count.cc (see Listing 18.3).

```
#include <mysql.h>
#include <ctype.h>

/* We need to use the C mangling-free symbols so that dlopen() will be
able to find them. This is accomplished with extern "C" declaration
*/
extern "C" {

/* initializer */
my_bool word_count_init(UDF_INIT* udf, UDF_ARGS* args, char* message);

/* computation */
long long word_count(UDF_INIT* udf, UDF_ARGS* args, char* is_null,
                                 char* error);

/* clean-up */
void word_count_deinit(UDF_INIT* udf);
}
```

Listing 18.3 The udf_word_count.cc file. (continues)

```
/* In the initializer we just check that we have one and only one
   argument, and that its type is correct
*/
my_bool word_count_init(UDF_INIT* udf, UDF_ARGS* args, char* message)
{
  if (args->arg_count != 1 || args->arg_type[0] != STRING_RESULT)
  {
    strcpy(message, "word_count() works only on strings");
    return 1;
  }
  return 0;
}

/* This is where the actual computation happens */
long long word_count(UDF_INIT* udf, UDF_ARGS* args, char* is_null,
                             char* error)
{
  /* set initial word count to 0 */
  long long word_count = 0;

  /* flag to keep track if we are on a word as we traverse the string,
     initially we are not on a word
  */
  bool on_word = 0;

  /* point at the start of the argument  */
  const char* s = args->args[0];

  /* set a sentry at the end */
  const char* s_end = s + args->lengths[0];

  /* now iterate through the string */
  for (;s<s_end;s++)
  {
    /* If we see a space, we are not on a word any longer */
    if (isspace(*s))
    {
      if (on_word)
          on_word = 0;
    }
    /* Otherwise, if we see a non-space character, and we were not on
a word, we increment the counter and flip the on_word flag
    */
    else
    {
      if (!on_word)
      {
```

Listing 18.3 The udf_word_count.cc file. (continues)

```
        word_count++;
             on_word = 1;
       }
     }
   }
   return word_count;
}

/* Clean-up routine is rather boring. Nothing to do */
void word_count_deinit(UDF_INIT* udf)
{
}
```

Listing 18.3 The udf_word_count.cc file. (continued)

The udf_word_count.cc file is available at the book Web site in the mysqld directory.

Compile the file in Listing 18.3 with the following command if you are using GCC on Linux (other systems and compilers may require slight modifications to the command line):

```
gcc -shared -o udf_word_count.so -I/usr/include/mysql
udf_word_count.c
```

The -I argument should be the directory containing the mysql.h include file, which is a part of the client API distribution and can also be found within MySQL source code in the include/ directory.

Having compiled the UDF library, copy it to a library directory that is visible to the MySQL server binary. There are several approaches to make this happen. The path of least resistance is to copy it into /usr/lib or /lib. A more systematic approach would be to create a special directory for UDF libraries and then edit the safe_mysqld script by adding the new directory to LD_LIBRARY_PATH. Note that just setting LD_LIBRARY_PATH before you launch safe_mysqld will not work because safe_mysqld resets the variable.

If LD_LIBRARY_PATH had to be fixed, restart the server for the change to take effect. Once you are sure that the server will see your library, execute the following command:

```
CREATE FUNCTION word_count RETURNS INTERGER SONAME
'udf_word_count.so';
```

To be able to execute the above command, the user you connect as must have the INSERT privilege on the mysql database. If you want to update the UDF library, replace the old version with the new one and execute the following commands:

```
DROP FUNCTION word_count;
CREATE FUNCTION word_count RETURNS INTERGER SONAME
'udf_word_count.so';
```

To be able to drop a function, you need to have the DELETE privilege on the mysql database.

The function can now be invoked just like a regular SQL function, for example:

```
mysql> SELECT WORD_COUNT("How much wood would a woodchuck chuck if a
woodchuck could chuck wood? He'd chuck all that a woodchuck could if a
woodchuck could chuck
wood") as COUNT;
+-------+
| COUNT |
+-------+
|    26 |
+-------+
1 row in set (0.00 sec)
```

Adding a New Table Handler

One task MySQL is very well suited for is providing an SQL interface to low-level data storage systems. This can be accomplished by adding your own table handler that will define some basic data storage- and retrieval-operations for MySQL in terms of what your low-level storage engine already knows how to do.

Adding a new handler is a much more complex task than adding a new function or writing a UDF. In the history of MySQL, the most efficiently integrated third-party handler (among the ones that I have seen integrated) is the InnoDB handler, and it took Heikki about a month to do it. We can't provide a full example or a very comprehensive set of directions on how to add a new handler; we have to restrict ourselves to a set of basic guidelines instead. We encourage you to start with the information we provide and gain the rest of the knowledge through a more in-depth study of the recommended source files.

A set of directions follows to get you started. All the files we mention reside in the sql directory unless stated otherwise.

First, take care of the parser. Add the symbol entry for your table handler to the symbols array in lex.h, define the corresponding token with the %token directive, extend the table_types rule, and update the keyword rule in sql_yacc.yy with the name of your table handler (follow the syntax prototype established by code for other table handlers). Next, add the type for your table handler to enum db_type (defined in handler.h). Be sure to add your table type identifier to the ha_table_type array in handler.cc.

Now move on to the compilation setup. Create a .h and a .cc file for your handler. It is recommended that you follow the naming convention used with other handlers—start the filenames and the handler classname with ha_ . Add the .h filename to noinst_HEADERS and the .cc filename to mysqld_SOURCES in Makefile.am. Also, in Makefile.am, add the path to the low-level interface library to LDADD, and add -I with the path to the directory containing the include files for the low-level interface to INCLUDES.

In the .h file, define a class for your handler that will publicly inherit from handler. Add the include statement for the .h file to handler.cc after the inclusion of mysql_priv.h. Modify get_new_handler(), adding the case for your db_type and returning a new instance of your handler class.

In the class in addition to the constructor, the following pure virtual methods have to be implemented:

- **open():** Called when the table is opened.
- **close():** Called when the table is closed.
- **write_row():** Inserts a new record.
- **update_row():** Updates a record.
- **delete_row():** Deletes a record.
- **index_read():** Reads one record based on the key value. An index number is supplied as an argument.
- **index_read_idx():** Reads one record based on the key value. The currently active key is used.
- **index_next():** Reads the next key value and repositions the index cursor.
- **index_prev():** Reads the previous key value and repositions the index cursor.
- **index_first():** Positions the index cursor at the start of the index and reads the first value.
- **index_last():** Positions the index cursor at the end of the index and reads the last value.
- **rnd_init():** Prepares for a sequential table scan, positioning the scan cursor on the first record.
- **rnd_next():** Reads the next record during a sequential table scan.
- **rnd_pos():** Reads a record from the table at the given offset.
- **position():** Records the current scan cursor position.
- **info():** Retrieves statistical information about the table.

- **extra():** Passes a hint or an explicit directive to the low-level handler on how to perform subsequent operations.
- **reset():** Resets the low-level handler to the initial state. It is not currently used for anything valuable and can return 0 without doing anything in the implementation. It will probably be removed in the future.
- **external_lock():** Originally the method was created to obtain external file system locks on the table files when locking the tables if MySQL was running without —skip-locking options. Heikki decided that he could take advantage of this in the InnoDB handler to perform some initializations. The function should probably be renamed to something else—for example, init_on_lock()—and it could happen by the time you read this book.
- **table_type():** Returns the name of the table type as a string.
- **bas_ext():** Returns an array of file extensions used by the table handlers. It is used only in error messages. It can return an arbitrary array of strings if the handler is not using a file with the table name in it plus the file extension.
- **table_flags():** Returns the capability bitmask for the table handler, with each bit indicating that a certain functionality is supported or in some cases not supported, or a certain peculiarity is present.
- **max_record_length():** Returns the maximum length of a record the handler supports.
- **max_keys():** Returns the maximum length of a key the handler supports.
- **max_key_parts():** Returns the maximum number of parts in a composite key the handler supports.
- **create():** Creates the table on the low level based on the definition structure produced by the parser.
- **store_lock():** Remembers the lock structure for the lock acquired in the table lock manager.

Additionally, you may want to implement other virtual methods of handler that are not pure. We list them next with a brief description of each:

- **scan_time():** Specifies the amount of time (in arbitrary units) it takes to scan the whole table. The reason for the unit being arbitrary is that it will be compared with the return value of read_time() (described next), so the unit does not matter as long as it is the same as in read_time().
- **read_time():** Specifies the amount of time (in the same units as the return value of read_time()) it would take to read the given number of rows with an index. This value is compared with the return value of scan_time() to determine whether it is better to scan or to use a key.

- **fast_key_read():** You can reimplement this method to return 1 for safety reasons. It is possible that the method will be removed in the future.
- **keys_to_use_for_scanning():** Returns a value that affects the decision in some cases whether a key will be used for ORDER BY. If you want to encourage the optimizer to use a key for ORDER BY when in doubt, return (key_map)~0. This method was added to help the optimizer make better decisions on queries with InnoDB tables.
- **has_transactions():** Reimplement this method only if your storage engine supports transactions, and return 1 in that case.
- **extra_rec_buf_length():** This method exists as a workaround for the BDB handler, which wants to have more than table->reclength bytes reserved in the record manipulation buffer per record. The extra space is needed for the hidden primary key. The method returns the amount of extra space needed. Usually, you would not need to reimplement this.
- **estimate_number_of_rows():** Returns the upper bound of the number of records in the table. It is used to make the decision on the join order when the table is being scanned.
- **index_init():** Changes the active index. For MyISAM and HEAP, the default is sufficient (it just changes handler::active_index). InnoDB and BDB handlers require more sophisticated operations.
- **index_end():** This method is needed only in the BDB handler to inform it that you have finished using the keys in the table so that it can clean up before you unlock the table. If your handler does not require this, the default implementation—which does nothing—should be fine.
- **index_next_same():** Reads the next value in the key only if it is the same as the previous value read and advances the index cursor on success.
- **index_read_last():** Reads the last entry with the matching value for the given key. Implementing this method will help the optimizer come up with a more efficient query plan.
- **ft_init():** This is a full-text initialization method called before reading the first record from the table with a full-text key. Leave unimplemented if your storage engine does not support full-text keys.
- **ft_init_ext():** This is a full-text initialization that takes place before the optimizer starts processing the parsed SQL query. Leave unimplemented if your storage engine does not support full-text keys.
- **ft_read():** Reads a record based on a full-text key. Leave unimplemented if your storage engine does not support full-text keys.
- **rnd_end():** Notifies the handler that you are done with non-indexed (random) reads and gives it an opportunity to clean up. So far, only BDB handlers need to do this.

- **rnd_first():** Reads just the first record in a table. The default implementation is a sequence of rnd_init(); rnd_next(); and rnd_end(). The method is called to read one record from a system table (a table containing one record or none at all). All handlers so far have found the default implementation sufficient.
- **restart_rnd_next():** Moves the scan cursor to the saved position. It is needed only if your handler can be used with a temporary table.
- **record_in_range():** Returns an estimate of how many records there are in the given key range. The value is used by the optimizer when deciding which key to use. The default implementation returns 10 regardless of the range.
- **row_position():** Returns the data file offset corresponding to the current position of the index scan cursor. It is currently used only by the full-text MATCH AGAINST function.
- **extra_opt():** A special version of extra() that allows the caller to pass an argument to the handler, which is usually the size of some cache. This is called to optimize multirow inserts and LOAD DATA INFILE.
- **unlock_row():** The original intent of this method was to unlock the currently locked row for the handlers that are capable of row-level locking. However, it currently is not implemented by any of the existing handlers, although it is called several times from the query processing code.
- **start_stmt():** Provides per-table notification of the transaction start. It is currently needed only for the BDB handler.
- **delete_all_rows():** Deletes all records from the table, leaving it empty. You should reimplement this method if it provides a faster way to delete all records from a table in your storage engine than to iterate through all of them and delete them one by one.
- **get_auto_increment():** Gets the current value of the auto-increment field. The default implementation reads the last value auto-increment key.
- **update_create_info():** Updates the table structure information. It is used during ALTER TABLE handling in mysql_alter_table() in sql_table.cc.
- **check():** Called during processing of CHECK TABLE to check the table for corruption. By default, it returns HA_ADMIN_NOT_IMPLEMENTED.
- **repair():** Called during processing of REPAIR TABLE to repair a possibly corrupted table. It is legal to call repair on a table that is not corrupted. By default, it returns HA_ADMIN_NOT_IMPLEMENTED.
- **check_and_repair():** Checks the table for corruption, and repairs it if needed. It is called when opening a table that has not been cleanly closed if the myisam-recover option is enabled.

- **optimize():** Called during processing of OPTIMIZE TABLE to defragment and otherwise improve the table. By default, it returns HA_ADMIN_NOT_IMPLEMENTED.
- **analyze():** Called during processing of ANALYZE TABLE to update key distribution statistics. By default, it returns HA_ADMIN_NOT_ IMPLEMENTED.
- **backup():** Backs up table files to a directory. It is implemented only for MyISAM tables and used in handling the BACKUP TABLE command.
- **restore():** Restores the table backed up with the BACKUP TABLE command. It is used in processing the RESTORE TABLE command.
- **dump():** Dumps the table in binary format across the network. It is used by the master to send a copy of the table to the slave. It is currently implemented only for MyISAM tables.
- **deactivate_non_unique_index():** Temporarily disables non-unique index updates to speed up data import. It is used in handling ALTER TABLE ... DISABLE_KEYS and to speed up LOAD DATA INFILE and multirow inserts. The index is reenabled and updated with a call to activate_all_index().
- **activate_all_index():** Re-enables all keys and updates them based on the inserted data. It will be called to finalize a multirow insert and LOAD DATA INFILE with disabled keys, or through the invocation of ALTER TABLE ... ENABLE KEYS.
- **net_read_dump():** Reads the binary table dump sent by dump() and recreates the table on the slave. It is currently implemented only for the MyISAM handler.
- **update_table_comment():** Updates the comment field on the table with its stored definition. It is used in ALTER TABLE.
- **append_create_info():** Appends extra information that cannot normally be deduced from the regular table structures and that is available only to the handlers via the output of SHOW CREATE TABLE command.
- **get_foreign_key_create_info():** Obtains the string used for creating the foreign keys from the handler if the handler supports them. Currently, InnoDB is the only handler that supports foreign keys.
- **init_table_handler_for_HANDLER():** Initializes the table to prepare it for the use of the HANDLER command. The HANDLER command is a new feature in MySQL 4.0 that allows you to access the records by iterating through a key instead of using a SELECT, UPDATE, or DELETE command.
- **free_foreign_key_create_info():** Frees the string returned by get_ foreign_key_create_info(). It is currently implemented only in the InnoDB handler.

- **index_flags():** Returns the capability bitmask for the given index, with each bit indicating that a certain functionality is supported.
- **max_key_part_length():** Returns the maximum length of a key part in a composite key.
- **min_record_length():** Returns the length of the shortest record supported by the handler.
- **low_byte_first():** Reports whether the integers in the internal representation are in the big-endian or the little-endian byte order. The default is 1 (little-endian).
- **is_crashed():** Checks if the table was cleanly closed before the last shutdown.
- **auto_repair():** Reports whether the handler has the auto-repair capability.
- **rename_table():** Changes the name of the table. The default implementation goes through the table file extensions reported by bas_ext(), appending each to the table name and renaming the resulting file according to the name of the new table. If your storage engine does not store the table in a set of files unique to the given table or if the name of the file is not related to the name of the table, you should reimplement this method.
- **delete_table():** Does the handler-specific part of dropping the table. It is called when processing the DROP TABLE command.
- **lock_count():** Reports how many tables are actually locked by the handler. Normally the value is 1, which is what the default implementation returns. However, the MERGE table handler may lock more than one table (see ha_myisammrg.cc for details).

Once you have implemented all pure virtual methods, along with the non-pure virtual methods appropriate for your storage interface, you will be able to create tables of your own custom type with CREATE TABLE tbl_name (...) TYPE=*your_custom_type;* and perform regular SQL queries on them.

However, as we have mentioned earlier, the information in this chapter is just a set of general directions and guidelines. To actually be able to complete the task, you have to delve into some specific issues that are beyond the scope of this book. We can point you to a few places in the source that you can study to gain that information. We recommend you study the following (all files are in the sql directory):

- handler.h and handler.cc can help you become more familiar with the handler members and default implementations of the virtual methods that are not pure.

- ha_myisam.h, ha_myisam.cc, ha_heap.h, ha_heap.cc, ha_innodb.h, ha_innodb.cc, ha_berkeley.h, ha_berkeley.cc, ha_myisammrg.h and ha_myisammrg.cc include functional handler implementation examples. You may want to follow the engine-specific calls into the sources where they are implemented. To facilitate the task of finding them, you may want to use a tool like Doxygen, or if you do not want to take the time to install and learn a new tool, a simple grep can take you a long way.
- Pay particular attention to the comments in ha_innodb.cc at the top of each method implementation. They provide additional insights into the difficult issues you may have to deal with.
- sql_select.cc, sql_table.cc, filesort.cc, and sql_base.cc show you the context of handler method invocations. Those are not the only files where handler methods are invoked, but that is where the bulk of the calls originates.
- table.h familiarizes you with members of struct st_table (later typedefed to TABLE).The most important member you want to follow throughout the source is record, which is an array of three one-record buffer pointers. Those three one-record buffers are used very extensively while interfacing with the handlers.
- table.cc shows the details of how MySQL opens a table and initializes the TABLE structure in openfrm(). Particularly look for the initialization of file with a call to get_new_handler().
- sql_base.cc shows how the record is assembled in fill_record(). Then study item.h and item.cc to see the definition and implementation of Item_field. Taking the code further apart, take a look at field.h and field.cc and study the definition and implementation of Field along with its subclasses. Field is responsible for converting the in-memory processing field format to the storage format and vice versa.

Maintaining Your Code Modifications

If you modify the source of MySQL, you will still likely want to keep up with the new features and bug fixes in the newer versions of MySQL. Fortunately, BitKeeper makes it very convenient to do so. The additional benefit of using BitKeeper is having your own changes managed by a very powerful version-control system. Let's examine some of the basic BitKeeper operations involved in keeping your code base up-to-date.

Once you have reached some point of consistency in your code modification process (e.g., it compiles and passes some simple tests, or you have reached the point where you believe the code is stable enough to put into production), it is

time to commit your changes. To do this, you run bk citool from anywhere within the source directory (all BitKeeper commands we mention in this section must be run from within the tree unless otherwise stated). You will see a dialog window showing all of the changes you have made for each file in diff format, and prompting you to comment on each file as well as the entire set of changes. After you have entered the comments, click the Commit button twice (each time with a single click). The changes will be committed, and the dialog window will disappear.

To synchronize your source with the central repository in its current state, execute bk pull. In many cases, this may not be quite exactly what you want to do, since the centralized repository is more often than not in a mid-release state. Although all MySQL AB developers are required to run their code through a regression test suite before they push their local changes into the main repository, sometimes mistakes are made, and the mid-release tree could contain a bug. Additionally, developers are required to run the test suite only on their development machine, which means they can unknowingly introduce a bug that simply does not manifest itself on their platform but is present on others. Mid-release commits are also not tested with Purify, which means that the chances of having a memory leak or some other bad code error are higher. For those reasons, it is preferable to synchronize with release snapshots rather than an arbitrary mid-release point.

To synchronize with a certain release version of MySQL, you would have to do a small maneuver, at least until BitKeeper begins to support pulling up to the given changeset as opposed to pulling everything. First, clone the snapshot of the main tree corresponding to the version you are interested in:

```
bk clone -rmysql-version bk://work.mysql.com:7001
/path/to/bkdir/mysql-version
```

This command is for the 4.0 tree. For 3.23, change the port from 7001 to 7000. Replace *version* with the actual version number (without the alpha, beta, or gamma suffix). Replace */path/to/bkdir* with the name of an existing directory when you want the tree cloned. Then pull from the newly cloned repository into your development tree with the changes. Do so by changing into the development tree directory, and then execute

```
bk pull /path/to/bkdir/mysql-version
```

In some cases, your changes may conflict with the ones in the central repository in a way that BitKeeper would not know what to do. In this case, you would have to run bk resolve and tell BitKeeper what you want to do about the conflicts. Sometimes BitKeeper does not do the right thing when merging changes. If you suspect this might happen, you should do pk pull -i /path/to/bkdir/mysql-version and then run bk resolve.

Sometimes you may have pulled changes that break something in your code, or are undesirable in some other way. With bk sccstool, BitKeeper makes it quite easy to track down when and how this happened, and who did it. You can see the graph of changesets, and you can click on each and examine them in more detail. You can also select individual files and see for each line who modified it last and in what changeset. After you have found the trouble line, you can go to the changeset that has introduced it, read the comments, look at what else the changeset has modified, and then decide what you are going to do to fix it.

Conclusion

We have provided a basic introduction into working with the MySQL server code base. As you work with the source, you will become more familiar with it, and will see its strengths and weaknesses. Sometimes you may have a question about the code, or you may want to submit a patch, or just suggest a way to improve it. If you have anything to say about MySQL source, e-mail your comments to internals@lists.mysql.com. You do not have to subscribe to be able to post.

It is my hope that your experience with the source will be not just challenging but also enjoyable, and that you will take advantage of the opportunity to both learn and contribute.

APPENDIX A

Migration Notes

Migration to MySQL from another database is always possible, but it is not always easy. The two most difficult challenges are porting an application that depends on features that MySQL does not currently support, and being locked into a client language that does not allow the code to be easily modified to work with MySQL. The key to determining the feasibility of migration is examining the potential roadblocks to see if they exist, and if they do, whether there is a reasonable way to eliminate them. Answering the following questions will help you determine the ease with which you can migrate to MySQL:

- Does your application depend on subqueries? If it does and the subqueries are sufficiently simple that you could write a parser to detect and rewrite them to joins or sequences of queries using a temporary table, then migration might be worthwhile.
- Does your application depend on triggers? If so, is it feasible to replace them with equivalent user-level code? If you can do this, migration might be worth a try.
- Does your application depend on views? If it does, there is no easy workaround short of completely removing the dependency.
- Does your application depend on stored procedures? If yes, consider writing a parser that will translate the currently used stored procedures into equivalent client code and then incorporate it into your application.
- What language is the client written in and what connectivity standard is being used? Perl/DBI, ODBC, and JDBC are the most encouraging

answers—the porting of the client code becomes a matter of changing the data source in the connect call and perhaps fixing a few cases of dependency on some database-specific behavior. If the client is written in another language that uses a replaceable library for database operations, the porting can be accomplished by writing a special library that will translate the semantics of the old database calls into that of MySQL and replacing the old connectivity library with the new one. The feasibility of porting then is determined by how close the old database API resembles that of MySQL, and how easy it is to implement MySQL connectivity in terms of the syntax of the old database. The worst case is when you are using a scripting language that does not have any intermediate layers that you can replace. In this case, the only solution is to write an automatic code converter if the application is too big; otherwise, you would need to rewrite the code from scratch.

In the past, the lack of transaction support has been a stumbling block on the migration path. With the availability of InnoDB tables, this stumbling block now is largely removed, although you may occasionally experience some portability issues when InnoDB uses a different paradigm from the database the code was originally written in. These issues can be dealt with on a case-by-case basis. It is recommended that if your code makes heavy use of transactions that you purchase MySQL support contract with InnoDB support at https://order.mysql.com. This way, you are likely to have any potential problems solved quickly.

Many potential users ask whether MySQL will be able to handle the load and/or the data volume after migration from another database. In most cases, MySQL will improve your application's performance. Occasionally, you may run into a query that was optimized for your old database and needs to be rewritten for MySQL. These issues can be addressed on a case-by-case basis using suggestions found in Chapter 15, "Analyzing and Improving Server Performance."

To actually perform a migration to MySQL, follow these general steps:

- Export the data from the database you have been using into a set of SQL statements that will re-create the tables and populate them with the data. The actual sequence of the set depends on the database you are using. You should consult the documentation for more specific instructions.
- Use MySQL's GRANT command to create users and give them appropriate privileges.
- If the SQL export does not include CREATE DATABASE statements, then create the needed databases.
- Import the data into MySQL using mysqldump.

- Adapt your client code to connect to MySQL. In some cases (Peri/DBI, ODBC, JDBC), this will be as simple as changing the DSN value. In other cases, some ingenuity may be required. The basic principle is to substitute the original API layer with the one that will translate the old calls to their MySQL equivalents.
- Address the MySQL missing features issues. For example, rewrite the subqueries as joins or a sequence of queries using a temporary table.

Test your new application setup and deal with any remaining portability issues as they arise.

APPENDIX B

Troubleshooting Notes

Troubleshooting any system or application is a complex task. It is impossible to list every potential issue and give its solution in this book. Instead, this appendix explains several concepts that will help you determine the category of issue you face. Then you will be able to make efficient use of a variety of resources; first and foremost is the troubleshooting section of the MySQL online manual (www.mysql.com/doc/en/Problems.html) to find your solution.

Issues with MySQL can be categorized into three groups: functionality, stability, and performance.

Functionality

MySQL functionality issues usually manifest themselves as queries result sets that are not quite what you expected. The two obvious reasons for unexpected query results are bugs and incorrect expectations. The first thing you should do is isolate the problem using the smallest possible test case: a sequence of SQL statements that demonstrate the aberrant result on any installation of MySQL (not just on your system). In other words, the test case must be fully contained and should not depend on any previously created users, tables, or databases, other than those created during a default installation of MySQL.

Once you have the test case, it is recommended that you consult the SQL standards to verify that your query is written correctly for the results you expect,

especially if the query is not trivial. If your query is structured correctly, then MySQL's behavior may not be correct. If you believe this is the case, verify that the incorrect behavior happens with the most recent version of MySQL. If the upgrade does not fix the bug, e-mail the test case to bugs@lists.mysql.com. Make sure to use mysqlbug script to compose the bug report. If large tables are required to duplicate the bug, upload them to ftp://support.mysql.com/pub/mysql/secret.

Bugs reported with a test case that MySQL developers can duplicate are usually fixed in the next release. You also will frequently receive a recommendation on how to work around the bug to deal with the problem in the meantime.

Stability

There are two main symptoms of stability issues: queries returning an error message from handler with an error code for the corrupted table and a message about a lost connection during query. If such messages appear in your client applications, you should first examine the error log, which often contains more details about the problem. Usually you will see a message saying that MySQL has crashed.

MySQL server may crash for two reasons: an internal bug, or a problem with the hardware or operating system. The critical step to diagnosing the problem is finding the query that crashed the server. General logging happens before the query is executed, so the trouble query will always be logged. Sometimes you may see the query in the "post-mortem" stack trace diagnostic message. If you do not, enabling the log option will enable you to detect the culprit (it will most likely be the last query in the log right before the crash, and if not, very close to the end of the log).

Once you have identified a potentially troublesome query or a set of suspects, you should run each on a development server with the same data as the installation where the crash occurred to see if you can duplicate the result. If you are able to duplicate the crash, try it also on the most recent version of MySQL, and if it still happens, report the bug with a full text case to bugs@lists.mysql.com with mysqlbug. Until the fix is provided, replace the trouble query with the one that is functionally equivalent, but will not crash the server.

If you are unable to duplicate the crash by running the suspect queries, there are two possibilities: a hardware and operating system problem, or a concurrency bug. At this point, troubleshooting becomes an art more than a science. You have to evaluate the complexity of the query, the maturity of the server code that it depends on, the level of concurrency on the server at the moment of the crash, and a few other factors to make a guess as to the most likely cause.

If you see frequent crashes on simple queries like SELECT * FROM tbl WHERE key1='val' or INSERT INTO t1 (a,b,c) VALUES ('a', 'b','c') , it is very likely that the problem is in the operating system or the hardware.

Verify hardware integrity first: Examine your RAM, disk controller, disk drives, and CPU fans. If feasible, replace these hardware components to eliminate suspects. If the hardware checks out, the cause is most likely your operating system. At this point, MySQL AB has only anectodal evidence with regard to the level of stability of different operating systems, but hopefully in the near future you will find a database of operating systems with numeric data on their stability in interaction with MySQL at www.mysql.com. Usually the operating system problem is a bug in a disk I/O device driver, the virtual memory implementation, the file system, or the generic I/O code.

In very rare cases, your problem may be an internal concurrency bug. Patterns of behavior that suggest this include the following: crashes that occur only on complex queries, crashes from a collection of queries that follow a distinct pattern, queries that use a special feature (e.g., concurrent or delayed insert), a special command is always invoked around the time of the crash (e.g., FLUSH TABLES), or you can regularly identify some other suspicious activity around the time of the crash. Concurrency bugs are not easy to duplicate, and the bug that cannot be duplicated is nearly impossible to fix. Such bugs can be detected only during a thorough code audit. In this case, the practical solution is to artificially serialize the trouble queries, (e.g., using the GET_LOCK()/RELEASE_LOCK() SQL functions) until a better solution is found.

Sometimes instability may result from a compiler bug or library problems. In this case, it might be worthwhile to recompile MySQL with a different compiler or different options, and/or update the system libraries to the latest patch level.

Performance

If you are experiencing slow performance with MySQL, it is recommended that you first read Chapter 15, "Analyzing and Improving Server Performance." You may also find Chapter 12, "Writing the Client for Optimal Performance" quite helpful. The most common sources of performance issues include the following:

- Inefficient queries. To solve this, you must optimize your schema (add keys, change column types, etc.), and/or rewrite the queries to make better use of the schema.
- Unnecessary queries. Cache the query result in the application whenever possible using generated static instead of dynamic HTML pages; if your

application is difficult to optimize, use the query cache feature available since MySQL version 4.0.1.

- Performance bugs in the operating system. These are difficult to diagnose, but you can try running a similar load on a different operating system and compare the results. Solutions to this problem are usually not very easy. You can try applying the latest set of operating system patches, or use a different operating system. In some cases, it might be possible to track down the problem to a specific system call or a sequence of calls and work around it.
- Performance bugs in the shared libraries used by MySQL server binary. The thread library is usually the source of the problem. To solve, update the system libraries to the latest patch level, or use a statically linked binary that was built on a problem-free system.
- Bugs in the compiler used to build MySQL binary that affect performance. You can simultaneously diagnose and solve these problems by recompiling MySQL with a different compiler and/or different flags.
- Performance bugs in the MySQL server code itself. These are also difficult to diagnose. You would need to look at the frequently executed queries, time them, and notice that a particular query is taking much longer to execute than it should under those conditions. Sometimes you can diagnose these bugs by running EXPLAIN on a query and noticing that it is not using a key that it should or that it is using the wrong key. If this is the case, you may be able to build a workaround by giving the USE KEY hint to the optimizer. Once you have a test case, report it with mysqlbug to bugs@lists.mysql.com.

Overall, the general troubleshooting principle is that if MySQL does not work right, do not put up with it. If a limitation is not stated in the manual at www.mysql.com/doc/, assume it should not exist, and if you are hitting it, it is a bug. MySQL AB is committed to quality; if you press an issue, there will be a positive response. If you have a support contract, the response will be quicker, but even if you do not, if you are willing to spend the time to clearly demonstrate that something is wrong, MySQL AB will respond with a fix.

APPENDIX C

SQL Problem Solving Notes

Writing an SQL query to solve a particular problem is often not a trivial task. If you are accustomed to using subselects, the MySQL lack of sub-select support may present you with more of a challenge. On the other hand, some features unique to MySQL can make your work easier. This appendix examines several common SQL problems and provides solutions with the SQL syntax understood by MySQL version 3.23.

For all problems discussed in this appendix, I assume the schema shown in Listing C.1.

```
CREATE TABLE employee
(
  id  INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
 first_name VARCHAR(15) NOT NULL,
last_name VARCHAR(15) NOT NULL,
 ssn CHAR(9) NOT NULL,
 position ENUM ('CEO', 'CFO', 'CTO', 'Sales rep', 'Manager',
'Developer', 'Receptionist') NOT NULL,
salary DECIMAL(7,2) NOT NULL,
 supervisor_id INT NOT NULL,
UNIQUE KEY(ssn),
KEY(supervisor_id),
KEY(last_name,first_name),
KEY(salary)
);
```

Listing C.1 Sample schema. (continues)

```
CREATE TABLE payroll_transaction
(
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  date DATE NOT NULL,
  amount DECIMAL(7,2) NOT NULL,
  employee_id INT NOT NULL,
  description TEXT NOT NULL,
  KEY(date),
  KEY(employee_id),
  KEY(employee_id,date),
  KEY(date,amount),
  KEY(amount)
);

CREATE TABLE task
(
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
 description TEXT NOT NULL,
 deadline DATE NOT NULL,
 priority ENUM('Urgent', 'High', 'Medium', 'Low') NOT NULL,
 KEY(name(10)),
 KEY(deadline),
 KEY(priority,deadline)
);

CREATE TABLE task_log
(
  task_id INT NOT NULL,
  start TIMESTAMP NOT NULL,
  end TIMESTAMP NOT NULL,
  employee_id INT NOT NULL,
  work_description TEXT NOT NULL,
  PRIMARY KEY (employee_id,task_id,start,end)
);

CREATE TABLE product
(
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  price DECIMAL(7,2) NOT NULL,
  category ENUM('Software','Hardware','Consulting') NOT NULL,
  description TEXT NOT NULL,
  KEY(price),
  KEY(category)
);
```

Listing C.1 Sample schema. (continues)

```
CREATE TABLE customer
(
  id  INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  region ENUM('North America', 'South America', 'Europe', 'Asia',
'Africa', 'Pacific') NOT NULL,
  KEY(name),
  KEY(region)
);

CREATE TABLE sale
(
  id  INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  deal_amount DECIMAL(7,2) NOT NULL,
  num_units INT NOT NULL,
  product_id INT NOT NULL,
  customer_id INT NOT NULL,
  description TEXT NOT NULL,
  KEY(product_id),
  KEY(deal_amount)
);
```

Listing C.1 Sample schema. (continued)

Problem 1

Display the total number of hours worked by each employee for each week during the current year (assume the task is in the current year if the year of its end timestamp is current, and count the task in a week by the end timestamp, as well). For each employee, show their id, first name, and last name.

Solution

```
SELECT employee.id, employee.first_name, employee.last_name,
  FROM_DAYS(TO_DAYS(end) - WEEKDAY(end))
  AS week_start,
  SUM((UNIX_TIMESTAMP(end) - UNIX_TIMESTAMP(start))/3600)
  AS hours
FROM employee, task_log
WHERE employee.id = task_log.employee_id AND
  YEAR(end) = YEAR(CURRENT_DATE())
GROUP BY employee_id,week_start;
```

Comments

We perform a join of employee and task_log on employee_id key. This solution also uses the appropriate GROUP BY to take advantage of the rich date

arithmetic capabilities of MySQL to calculate the number of hours worked and the start of the week for each entry.

Problem 2

Display the top ten highest paying employees, showing their first name, last name, position, and salary.

Solution

```
SELECT first_name,last_name,position,salary
FROM employee
ORDER BY salary DESC
LIMIT 5;
```

Comments:

We take advantage of MySQL LIMIT clause in combination with ORDER BY DESC.

Problem 3

Display all employees that have not been paid over the last three months, showing their id, first name, and last name.

Solution

```
SELECT  employee.id,employee.first_name,employee.last_name
FROM employee LEFT JOIN payroll_transaction
ON employee.id = payroll_transaction.employee_id AND
  date >   DATE_SUB(CURRENT_DATE(), INTERVAL 3 MONTH)
WHERE payroll_transaction.id IS NULL ;
```

Comments:

We perform a left join, and again take advantage of the rich collection of the date and time manipulation functions in MySQL. Note that the correct syntax to test for equality to NULL is IS NULL, rather than = NULL, which is a common mistake.

Problem 4

Display every employees' first and last names, their positions, their salaries, the first and last name of their immediate supervisor, the supervisors' positions, and their salaries. If there is no supervisor, display *NULL* values in the appropriate columns.

Solution

```
SELECT worker.first_name,worker.last_name,worker.position,
worker.salary,supervisor.first_name,supervisor.last_name,
supervisor.position,supervisor.salary
FROM employee AS worker LEFT JOIN employee AS supervisor
ON worker.supervisor_id = supervisor.id;
```

Comments

Here we use the self-join technique. Because we need to deal with the case of an employee without a supervisor, we perform a left join.

Problem 5

Display a breakdown in sales by region and by product category showing the subtotals for each region and for each category in one query.

Solution

```
SELECT
  SUM(IF(product.category ='Software', sale.deal_amount,0)) AS
Software_sales,
  SUM(IF(product.category ='Hardware', sale.deal_amount,0)) AS
Hardware_sales,
  SUM(IF(product.category ='Consulting', sale.deal_amount,0)) AS
Consulting_sales,
  SUM(IF(customer.region ='North America', sale.deal_amount,0)) AS
North_America_sales,
  SUM(IF(customer.region ='South America', sale.deal_amount,0)) AS
 South_America_sales,
  SUM(IF(customer.region ='Africa', sale.deal_amount,0)) AS
Africa_sales,
  SUM(IF(customer.region ='Europe', sale.deal_amount,0)) AS
Europe_sales,
  SUM(IF(customer.region ='Pacific', sale.deal_amount,0)) AS
Pacific_sales
FROM sale,product,customer
WHERE sale.product_id = product.id AND sale.customer_id =
customer.id;
```

Comments

We emulate the cube cross-sections with a clever IF() trick. The same trick can solve a number of difficult data analysis problems of similar nature. Although the query is quite lengthy, it can be generated in the application using a rather straightforward algorithm.

Problem 6

Give a $400 bonus to each employee who has spent more than 40 hours working on an urgent task over the last month.

Solution

```
INSERT INTO payroll_transaction (date,amount,employee_id,description)
SELECT CURRENT_DATE(),400.00,employee_id, 'Urgent task bonus'
FROM  task,task_log
WHERE task.id = task_log.task_id AND task.priority = 'Urgent' AND
   task_log.end >= CONCAT(YEAR(CURRENT_DATE()),'-
',MONTH(CURRENT_DATE()),
     '-1')
GROUP BY employee_id
HAVING SUM(UNIX_TIMESTAMP(end) - UNIX_TIMESTAMP(start)) > 40*3600;
```

Comments

We perform a join with GROUP BY and take advantage of the liberal MySQL syntax that will allow the use of an aggregate function in HAVING referencing the columns that are not being selected. We use the INSERT INTO ... SELECT construct to insert the selected data directly into the payroll_transaction table. Also note the use of the time functions and CONCAT.

Problem 7

Display the first and the last name along with the employee id for all employees that have worked on urgent tasks; list them in alphabetical order by last name, and for employees with matching last names, further alphabetize them by first name.

Solution

```
SELECT DISTINCT employee.id, employee.first_name,employee.last_name
FROM employee,task_log,task
WHERE employee.id = task_log.employee_id AND task.id =
task_log.task_id AND task.priority = 'Urgent' ORDER BY
employee.last_name, employee.first_name;
```

Comments

We perform a join of three tables containing the data needed to answer the question. DISTINCT operator helps us remove duplicate rows.

Problem 8

Display all employees who have subordinates; list their subordinates alphabetically by last name and then by first name for matching last names. For each employee, show the first and last name, id, and position. List the supervisor employees in the order of the last name and then first name for matching last names.

Solution

```
SELECT supervisor.id,supervisor.first_name,supervisor.last_name,
  worker.id,worker.first_name,worker.last_name
FROM employee AS supervisor, employee AS worker
WHERE worker.supervisor_id = supervisor.id
ORDER BY supervisor.last_name,supervisor.first_name,
worker.last_name,worker.first_name;
```

Comments

We again perform self-join of employee using a different alias name for each instance, and order the results with ORDER BY.

Problem 9

Show the names of all customers who have made at least one deal with a volume discount, and indicate the number of volume discount deals for each customer.

Solution

```
SELECT customer.name, COUNT(*)  AS num_deals
FROM customer,sale,product
WHERE customer.id = sale.customer_id AND product.id =
sale.product_id AND
  product.price > sale.deal_amount/sale.num_units
GROUP BY customer.name
HAVING num_deals > 0;
```

Comments

We use a three-table join with GROUP BY, the discount conditional in the WHERE clause, and a filter in HAVING to eliminate the customers who have not received a discount.

Problem 10

Find all employees who share the same first and last name with another employee. For each, print their id and first and last name, alphabetized by last name and then by first for matching last names.

Solution

```
SELECT DISTINCT a.id,a.first_name,a.last_name
FROM employee AS a, employee AS b
WHERE a.first_name = b.first_name AND a.last_name = b.last_name AND
a.id <> b.id
ORDER BY a.last_name, a.first_name;
```

Comments

We again use self-join and look for suspects with the appropriate filter expression in the WHERE clause. As there could be duplicate entries in the case of three or more identical first and last name combinations, we need to weed those out with DISTINCT. To polish off the output, we use ORDER BY.

APPENDIX D

Online Resources

It would be fair to say that MySQL would not have become what it is today without a strong online community supporting it. The concept of an online community is at the very root of MySQL's existence. As you would expect, there are numerous online resources of MySQL-related information; I've listed some of the most useful ones here.

MySQL AB

MySQL AB's primary Web site is www.mysql.com. Specific areas of the site you will probably find useful are the following:

www.mysql.com/doc: MySQL online manual.

www.mysql.com/downloads: MySQL main download page.

www.mysql.com/documentation/lists.html: Information about MySQL mailing lists.

http://order.mysql.com/: Place to order commercial support for MySQL from MySQL AB.

mysql@lists.mysql.com: General MySQL mailing list. Subscription is not required to post. Make sure to follow the posting guidelines from Chapter 1.

www.mysql.com/doc/en/C.html: MySQL client C API documentation.

Third-Party Sites

MySQL users have contributed many excellent resources to the community. Following are some of the most useful ones.

www.mysqldeveloper.com: This MySQL community support Web site is not affiliated with MySQL AB. The site contains a dynamic FAQ, a rearranged version of the MySQL online manual, daily BitKeeper snapshot builds, mailing list archives, tutorials, and live chat. This resource is frequently updated.

www.php.net/manual/en/ref.mysql.php: MySQL client PHP API documentation.

http://dbi.perl.org/doc/index.html: Perl DBI documentation.

http://java.sun.com/products/jdbc/: JDBC information.

www.freshmeat.net: Lists open-source related software, including many packages that work with MySQL.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T