1

MySQL

It was not too long ago that you were simply out of luck if you wanted to build an application backed by a robust database on a small budget. Your choices were quite simple: shell out thousands—or even tens or hundreds of thousands—of dollars on Oracle or Sybase or build the application against “toy” databases like Access and FileMakerPro. Thanks to databases like mSQL, ProgreSQL, and MySQL, however, you now have a variety of choices that suit different needs. This book, of course, is the story of MySQL.

The History of MySQL

This story actually goes back to 1979 when MySQL’s inventor, Michael Widenius (a.k.a. Monty) developed an in-house database tool called UNIREG for managing databases. UNIREG is a tty interface builder that uses a low-level connection to an ISAM storage with indexing. In the years that have followed, UNIREG has been rewritten in several different languages and extended to handle big databases. It is still available today, but it is largely supplanted by MySQL.

The Swedish company TcX* began developing Web-based applications in 1994 and used UNIREG to support this effort. Unfortunately, UNIREG created too much overhead to be successful in dynamically generating Web pages. TcX thus began looking at alternatives.

TcX looked at SQL and mSQL. mSQL was a cheaply available database management system that gave away its source code with database licenses—almost Open Source. At the time, mSQL was still in its 1.x releases and had even fewer features

* For most of its existence, TcX had a single employee: Monty.

Copyright © 2001 O’Reilly & Associates, Inc.

than the currently available version. Most important to Monty, it did not support any indices. mSQL's performance was therefore poor in comparison to UNIREG.

Monty contacted David Hughes, the author of mSQL, to see if Hughes would be interested in connecting mSQL to UNIREG's B+ ISAM handler to provide indexing to mSQL. Hughes was already well on his way to mSQL 2, however, and had his indexing infrastructure in place. TcX decided to create a database server that was more compatible with its requirements.

TcX was smart enough not to try to reinvent the wheel. It built upon UNIREG and capitalized on the growing number of third-party mSQL utilities by writing an API into its system that was, at least initially, practically identical to the mSQL API. As a result, an mSQL user who wanted to move to TcX's more feature-rich database server would only have to make trivial changes to any existing code. The code supporting this new database, however, was completely original.

By May 1995, TcX had a database that met its internal needs—MySQL 1.0. A business partner, David Axmark at Detron HB, began pressing TcX to release this server on the Internet and follow a business model pioneered by Aladdin's L. Peter Deutsch. Specifically, this business model enabled TcX developers to work on projects of their own choosing and release the results as free software. Commercial support for the software generated enough income to create a comfortable lifestyle. The result is a very flexible copyright that makes MySQL "more free" than mSQL. Eventually, Monty released MySQL under the GPL so that MySQL is now "free as in speech" and "free as in beer".

As for the name MySQL, Monty says, "It is not perfectly clear where the name MySQL derives from. TcX's base directory and a large amount of their libraries and tools have had the prefix 'my' for well over 10 years. However, my daughter (some years younger) is also named My. So which of the two gave its name to MySQL is still a mystery."

A few years ago, TcX evolved into the company MySQL AB at *http://www.mysql.com*. This change better enables its commercial control of the development and support of MySQL. MySQL AB is a Swedish company run by MySQL's core developers. MySQL AB owns the copyright to MySQL as well as the trademark "MySQL". Since the initial Internet release of MySQL, it has been ported to a host of Unix operating systems (including Linux, FreeBSD, and Mac OS X), Win32, and OS/2. MySQL AB estimates that MySQL runs on about 500,000 severs.

MySQL Design

Working from the legacy of mSQL, TcX decided MySQL had to be at least as fast as mSQL with a much greater feature set. At that time, mSQL defined database per-

Copyright © 2001 O'Reilly & Associates, Inc.

www.mysql.com vs. www.mysql.org

As this book is being written, MySQL AB has no control over the mysql.org domain. Until June 2001, mysql.org was owned by someone else who had simply pointed it to the mysql.com domain. On June 4, 2001, a company called NuSphere bought the mysql.org domain and put up a different site. MySQL AB requested the domain be transferred to them, but NuSphere refused.

MySQL AB claims that NuSphere is distributing non-GPL, proprietary code with MySQL in violation of MySQL's GPL license in addition to hijacking the MySQL trademark. NuSphere, on the other hand, claims it is not violating the GPL and that MySQL signed a contract to allow NuSphere to use the MySQL trademark. Both sides have sued each other, and the matter is now before the courts.

Regardless of where your sympathies lie on the issue or the eventual outcome, the official source of MySQL is MySQL AB at *www.mysql.com*.

formance, so TcX's goal was no small task. MySQL's specific design goals were speed, robustness, and ease of use. To get this sort of performance, TcX decided to make MySQL a multithreaded database engine. A multithreaded application performs many tasks at the same time just as if multiple instances of that application were running simultaneously. Fortunately, multithreaded applications do not pay the very expensive cost of starting up new processes.

In being multithreaded, MySQL has many advantages. A separate thread handles each incoming connection with an extra thread always running in order to manage the connections. Multiple clients can perform read operations simultaneously without impacting one another. Write operations, on the other hand, only hold up other clients that need access to the tables being updated. While any thread is writing to a table, all other threads requesting access to that table simply wait until the table is free. Your client can perform any allowed operation without concern for other concurrent connections. The connection-managing thread prevents other threads from reading or writing to a table in the middle of an update.

Figure 1-1 illustrates the multithreaded nature of a MySQL database server. Another advantage of this architecture is inherent to all multithreaded applications: even though the threads share the same process space, they execute individually. Because of this separation, multiprocessor machines can spread the load of each of the threads across the many CPUs.

In addition to the performance gains introduced by multithreading, MySQL has a richer subset of SQL than mSQL. MySQL supports over a dozen data types and additionally supports SQL functions. Your application can access these functions through ANSI SQL statements.

Copyright © 2001 O'Reilly & Associates, Inc.

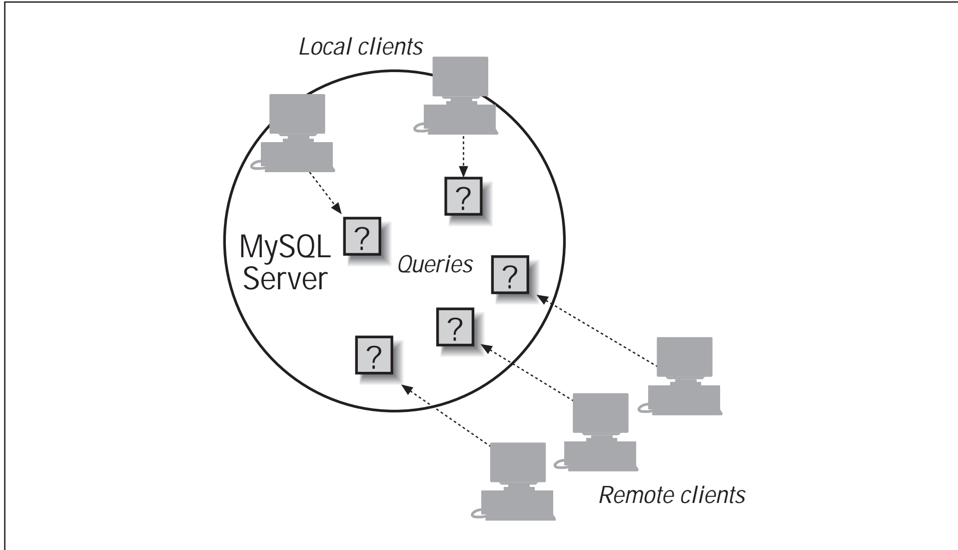


Figure 1-1. The client/sever design of MySQL

The Great Transaction Debate

Transactions are a relatively new feature of the MySQL database engine. It is, in fact, a feature that is not present unless you set up your tables to support it. A lot of people wondered for a long time what use MySQL was without transactions and why they would set up tables without support for transactions. The answer is one word: performance.

The minute you introduce transactions into the picture, the database takes a performance hit. Transactions add the overhead of complex locking and transaction logging. The complex locking includes support for something called transaction isolation levels. We will discuss transaction isolation levels in Chapter 9. Basically, however, increasing transaction isolation levels require an increasing amount of work by the database to support the same functionality. The more work the database has to do for a task, the slower it performs that task.

Transactions are important for applications that support complex rules for the concurrent updating data. They prevent concurrent updates from leaving the database in an inconsistent state at any point in its life. The most common use for MySQL, on the other hand, is to drive dynamic Web content. This use is "heavy read" in that 80% or more of the work is reading from the database. In such an environment, support for transactions is generally not needed. Performance, however, is.

Copyright © 2001 O'Reilly & Associates, Inc.

MySQL actually extends ANSI SQL with a few features. These features include new functions (`ENCRYPT`, `WEEKDAY`, `IF`, and others), the ability to increment fields (`AUTO_INCREMENT` and `LAST_INSERT_ID`), and case sensitivity.

MySQL did intentionally omit some SQL features found in the major database engines. For the longest time, transactions support was the most notable omission. The latest releases of MySQL, however, now provide some support for transactions. Stored procedures, another notable omission, are scheduled for a future release. Finally, MySQL does not support most additions to the SQL standard as of SQL3. The most important SQL3 feature missing from MySQL is support for object-oriented data types.

Since 1996, MySQL AB has been using MySQL internally in an environment with more than 40 databases containing 10,000 tables. Of these 10,000 tables, more than 500 contain over seven million records—about 100 GB of data.

MySQL Features

We have already mentioned multithreading as a key feature to support MySQL's performance design goals. It is the core feature around which MySQL is built. Other features include:

Openness
: MySQL is open in every sense of the term. Its SQL dialect uses ANSI SQL2 as its foundation. The database engine runs on countless platforms, including Windows 2000, Mac OS X, Linux, FreeBSD, and Solaris. If no binary is available for your platform, you have access to the source to compile to that platform.

Application support
: MySQL has an API for just about any programming language. Specifically, you can write database applications that access MySQL in C, C++, Eiffel, Java, Perl, PHP, Python, and Tcl. In this book, we cover C, C++, Java, Perl, and PHP.

Cross-database joins
: You can construct MySQL queries that can join tables from different databases.

Outer join support
: MySQL supports both left and right outer joins using both ANSI and ODBC syntax.

Internationalization
: MySQL supports several different character sets, including ISO-8859-1, Big5, and Shift-JIS. It also supports sorting for different character sets and can be customized easily. Error messages can also be provided in different languages.

Copyright © 2001 O'Reilly & Associates, Inc.

Above all else, MySQL is cheap and it's fast. Other features of MySQL may attract you, but you are probably looking in the wrong direction if cost and performance are not issues for you. The other relational databases fall into two categories:

- Other low-cost database engines like mSQL, PostgresSQL, and InstantDB.
- Commercial vendors like Oracle, Microsoft, and Sybase.

MySQL compares well with other free database engines. It blows them away, however, in terms of performance. In fact, mSQL does not compare with MySQL on any level. InstantDB compares reasonably on a feature level, but MySQL is still much faster. PostgresSQL has some cool SQL3 features, but it carries the bloat of the commercial database engines. If you are looking at the low-cost database engines, then you probably want PostgresSQL if you are using advanced SQL3 features and MySQL if you are doing anything else.

Oddly enough, comparing MySQL with Oracle or some other commercial database is a lot like comparing MySQL with PostgresSQL. The commercial database engines support just about every feature you can think of, but all those features come at a performance cost. None of these database engines can compete with MySQL for read-heavy database applications. They certainly cannot compete on price. They only really compete in terms of SQL3 feature set and commercial support. MySQL AB is working to close the gap on both counts.

Like many applications, MySQL has a test suite that verifies that a newly compiled system does indeed support all of the features it is supposed to support without breaking. The MySQL team calls its test suite "crash-me" because one of its features is to try to crash MySQL.

Somewhere along the way, someone noticed that "crash-me" was a portable program. Not only could it work on different operating systems, but you could use it to test different database engines. Since that discovery, "crash-me" has evolved from a simple test suite into a comparison program. The tests encompass all of standard SQL as well as extensions offered by many servers. In addition, the program tests the reliability of the server under stress. A complete test run gives a thorough picture of the capabilities of the database engine being tested.

You can use "crash-me" to compare two or more database engines online. The "crash-me" page is *http://www.mysql.com/information/crash-me.php*.

What You Get

MySQL is a relational database management system. It includes not only a server process to manage databases, it also provides tools for accessing the databases and building applications against those databases. Among these tools are:

Copyright © 2001 O'Reilly & Associates, Inc.

mysql
: Executes SQL against MySQL. You can also use it to execute SQL commands stored in a file.

mysqlaccess
: Manages users.

mysqladmin
: Enables you to manage the database server, including the creation and deletion of databases.

mysqld
: The actual MySQL server process.

mysqldump
: Dumps the definition and contents of a MySQL database or table to a file.

mysqlhotcopy
: Performs a hot backup of a MySQL database.

mysqlimport
: Imports data in different file formats into a MySQL table

mysqlshow
: Shows information about the MySQL server and any objects like databases and tables in that server.

safe_mysqld
: Safely starts up the *mysqld* process on a UNIX machine.

Over the course of this book, we will go into the details of each of these tools. How you use these tools and this book will depend on the way you want to use MySQL.

Are you a database administrator (DBA) responsible for the MySQL runtime environment? The chief concerns of a DBA is the installation, maintenance, security, and performance of MySQL. We tackle these issues in Part II.

Are you an application architect responsible for the design of solid database applications? Architects focus on data modeling and database application architecture. We address the impact of MySQL on these issues in the first few chapters of Part III.

Are you a database application developer responsible for building applications that rely on a database? A database application developer needs tools for providing their applications with data from MySQL. Most of Part III covers the various programming APIs that support application interaction with MySQL.

No matter who you are, you need to know the language spoken by MySQL—SQL. Like most database engines, MySQL supports the American National Standards

Copyright © 2001 O'Reilly & Associates, Inc.

Institute (ANSI) SQL2 standard with proprietary extensions. Chapter 4 is a comprehensive tutorial on MySQL's dialect of SQL. The details of the language are covered in several chapters in Part IV.

Copyright © 2001 O'Reilly & Associates, Inc.

2

Introduction to Relational Databases

Large corporate computing shops have been using complex and expensive database products for years. These full-featured, heavily optimized software systems are the only way for a big organization to manage its volumes of corporate information.

Home computer users haven't traditionally needed database products at all. They house their data—addresses, to-do lists, etc.—on their systems in small files or in specialized off-the-shelf spreadsheet and phonebook applications.

A new category of computer users who fall between these two extremes has come into play. These persons maintain moderate-sized information sets required for small organizations, such as new businesses or nonprofit organizations. Alternatively, such users may be a geographically isolated part of a larger company. Or they be individuals interested in maintaining complex personal data, such as a list of songs from favorite bands that can be served up on a personal web page. If you are the kind of person who wants a database, who is willing to do some work to set one up, but who does not want to spend six figures on a product and a fleet of programmers to maintain it, this book is for you.

This book introduces you to the world of small-scale database development through one popular database product: MySQL. We start by introducing you to relational databases and MySQL. We then proceed to show you how to get up and running with MySQL and how to administer it. The rest of the book covers the use of MySQL to design, build and support the type of applications important to users like you.

What Is a Database?

A *database* is, simply put, a collection of data. An example of a non-electronic database is the public library. The library stores books, periodicals, and other documents. When you need to locate some data at the library, you search through the card catalog or the periodicals index, or maybe you even ask the librarian. Another similar example is the unsorted

Copyright © 2001 O'Reilly & Associates, Inc.

pile of papers you might find on your desk. When you need to find something, you rifle through the stack until you find the scrap of paper you are looking for. This database works (or maybe it doesn't) because the size of the database is incredibly small. A stack of papers certainly would not work with a larger set of data, such as the collections in the library. In the library, without the card catalog, periodicals index, and librarian, the library would still be a database; it would just be an unusable database. A database therefore generally requires some sort of organization to be of value. Your pile of papers would be much more reliable if you had some sort of filing system (then maybe you would not have lost that phone number!). So, restating our definition, we will define a database as an *organized* collection of data.

The library and the stack of papers have many similarities. They are both databases of documents. It makes no sense, however, to combine them because your papers are only interesting to you and the library contains documents of general interest. Both databases have a specific purpose and they are organized according to that purpose. We will therefore amend our definition a bit further: a database is a collection of data that is *organized* and stored according to some *purpose*.

Traditional paper-based databases have many disadvantages. They require a tremendous amount of physical space. Libraries occupy entire buildings and searching a library is relatively slow. Anyone who has spent time in a library knows that it can consume a non-trivial amount of time to find the information you seek. Libraries are also tedious to maintain and an inordinate amount of time is spent keeping the catalogs and shelves consistent. Electronic storage of a database helps to address these issues.

MySQL is not a database, per se. It is computer software that enables a user to create, maintain, and manage electronic databases. This category of software is known as a Database Management System (DBMS). A DBMS acts as a broker between the physical database and the users of that database.

When you first began managing electronic information, you almost certainly used a flat file such as a spreadsheet file. The file system is the electronic version of the pile of papers on your desk. You likely came to the conclusion that this sort of ad hoc electronic database didn't meet your needs any more. A DBMS is the logical next step for your database needs, and MySQL is the first stepping stone into the world of relational database management systems.

What Is a Relational Database?

According to our definition, a database is an organized collection of data. A relational database organizes data into tables and represents relationships between those tables. These relationships allow you to combine data from multiple tables to provide different "views" of the data.It is probably easier to illustrate the concepts of tables and relationships than try to explain them. Table 1-1 is an example of a table that might appear in a book database.

Table 1-2: A Table of Books (continued)

 Copyright © 2001 O'Reilly & Associates, Inc.

ISBN	Title	Author
0-446-67424-9	L.A. Confidential	James Ellroy
0-201-54239-X	An Introduction to Database Systems	C.J. Date
0-87685-086-7	Post Office	Charles Bukowski
0-941423-38-7	The Man with the Golden Arm	Nelson Algren

Table 1-3 and Table 1-5 demonstrate two tables that might appear in an NBA database.

Table 1-4: A Table of NBA Teams (continued)

Team #	Name	Coach
1	Sacramento Kings	Rick Adelman
2	Minnesota Timberwolves	Flip Saunders
3	L.A. Lakers	Phil Jackson
4	Portland Trailblazers	Mike Dunleavy

Table 1-6: A Table of NBA Players (continued)

Name	Position	Team #
Vlade Divac	Center	1
Kevin Garnett	Forward	2
Kobe Bryant	Guard	3
Rasheed Wallace	Forward	4
Damon Stoudamire	Guard	4
Shaquille O'Neal	Center	3

We'll get into the specifics about tables later on, but you should note a few things about these examples. Each table has a name, several columns, and rows containing data for each of the columns. A relational database represents all of your data in tables just like this and provides you with retrieval operations that generate new tables from existing ones. As a result, the user sees the entire database in the form of tables.

Also note that the "Team #" column appears in both tables. It encodes a relationship between a player and a team. By linking the "Team #" columns you can determine that Vlade Divac plays for the Sacramento Kings. You could also figure out all the players on the Portland Trailblazers. This linking of tables is called a "relational join", or "join" for

Copyright © 2001 O'Reilly & Associates, Inc.

short.A DBMS for a relational system is often called a Relational Database Management System (RDBMS). MySQL is an example of an RDBMS.

Where does SQL fit into all of this? We need some way to interact with the database. We need to define tables and retrieve, add, update, or delete data. SQL (Structured Query Language) is a computer language used to express database operations for data organized in a relational form.

> SQL is commonly pronounced "Sequel". MySQL is pronounced "MySequel".

SQL is the industry standard language that most database programmers speak, and it is used by most RDBMS packages. As the name indicates, MySQL is a SQL database engine. However, it only supports a subset of the current SQL standard, SQL2. We will discuss exactly how MySQL support for SQL differs from the standard in later chapters.

Applications and Databases

According to our definition, a database is an organized collection of data that serves some purpose. Simply having a DBMS is not sufficient to give your database purpose. How you use your data defines its purpose. Imagine a library where nobody ever reads the books. There would not be much point in storing and organizing all those books if they're never used. Now, imagine a library where you could not change or add to the collection. The utility of the library as a database would decrease over time since obsolete books could never be replaced and new books could never be added. In short, a library exists so that people may read the books and find the information they seek.

Databases exist so that people can interact with them. In the case of electronic databases, the interaction occurs not directly with the database, but instead indirectly through software applications. Before the emergence of the World Wide Web, databases typically were used by large corporations to support various business functions: accounting and financials, shipping and inventory control, manufacturing planning, human resources, and so on. The web and more complex home computing tasks have helped move the need for database applications outside the realm of the large corporation.

Databases and the Web

The area in which databases have experienced the most explosive growth—an area where MySQL excels—is in web application development. As the demand for more complex and robust web applications grows, so does the need for databases. A database backend can support many critical functions on the web. Virtually any web content can be driven by a database.

Consider the example of a catalog retailer who wants to publish on the web and accept orders online. If the contents of the catalog are entered directly into one or more HTML files, someone has to hand edit the files each time a new item is added to the catalog or a price is changed. If the catalog information is instead stored in a relational database, it becomes possible to publish real-time catalog updates simply by changing the product or price data in the database. It also becomes possible to integrate the online catalog with

 Copyright © 2001 O'Reilly & Associates, Inc.

existing electronic order-processing systems. Using a database to drive such a web site thus has obvious advantages for both the retailer and the customer.

Here’s how a web page typically interacts with a database. The database is on your web server or another machine that your server can talk to (a good DBMS makes this kind of distributed responsibility easy). You put a form on one web page that the user fills in with a query or data to submit. When the user submits the form to your server, the server runs a program that you’ve written to extract the data from the form. These programs are most often written as CGI scripts and Java servlets, but they can also be implemented by embedding programming commands right inside the HTML page. We will look at all of these methods in this book.

Now your program knows what the user is asking for or wishes to add to the database. The program issues an SQL query or update, and the database magically takes care of the rest. Any results obtained from the database can be formatted by your program into a new HTML page to send back to the user.

Summary

MySQL is an SQL-based Relational Database Management System (RDMS). Relational databases are organized into tables and relationships between the tables. These tables are further broken up into columns and rows. Each row in a table represents one record of data. Each column represents one “piece” of data for each record. Some columns are used to represent a relationship between two tables, thus storing that relationship a another “piece” of data. SQL stands for **S**tructured **Q**uery **L**anguage. This is a standard language that is used to insert, retrieve, update and delete data from your relational database. It is also used to define the structure of your database tables.

That’s all there is to it. The basic concepts behind relational databases are quite simple. Therein lies the power: these simple building blocks – tables, columns, rows, relationships, and SQL – can be combined to create powerful applications.

Copyright © 2001 O’Reilly & Associates, Inc.

3

Installation

This chapter describes how to download and install MySQL. MySQL is available for wide variety of target operating systems. In this chapter, we provide an overview of how to install MySQL binary and source distributions for Unix (Solaris and Linux). Instructions for installation onto Win32 systems are also provided.

Getting prepared

Before you begin installing MySQL, you must answer a couple of questions.

1. Which version do I want to install?

This is typically a decision between the latest stable release and the latest development release. In general, we recommended that you go with the latest stable release unless you need specific features in a development release that are not available in the stable release.

Presently this comes down to a choice between MySQL 3.23 and MySQL-Max 3.23. MySQL-Max is a beta release of the MySQL software with support for transactions (via BerkeleyDB and InnoDB tables). The standard MySQL binary does not include support for these types of tables.

2. Do I want to install a binary or source distribution?

In general, we recommend that you install a binary distribution if one is available for your platform. In most cases a binary distribution will be easier to install than a source distribution.

We recommend that whenever possible you install from a binary distribution. In general this is the fastest and most reliable way to get MySQL up and running. The MySQL team and contributors have to great lengths to ensure that the binary distributions on their site

Copyright © 2001 O'Reilly & Associates, Inc.

are built with the best possible options. However, you may encounter cases where you need to build your MySQL distribution from scratch. For example,

There are a few reasons that you would need to install a source distribution:

- You are not able to locate a binary distribution for your target system
- You want to configure MySQL with some combination of options that is not available in any of the binary distributions
- You want to optimize your installation of MySQL by modifying compiler options or by using a different compiler
- You need to apply a bug fix patch

Downloading the Software

With the answers to those questions in mind, you can complete the first step in installing MySQL. That is to download the distribution. The best place to obtain MySQL source or binary distributions is from the MySQL downloads page, http://www.mysql.com/downloads or from one of the many mirror sites which can be found at http://www.mysql.com/downloads/mirrors.html.

Unix Installation

MySQL is available on a wide variety of UNIX platforms. Here go over the steps necessary to install binary and source distributions on Solaris and Linux. These can also be used as a general guide to installation on other operating systems, which should be very similar to our examples.

Installing a binary (tarball) distribution

In order to install a binary distribution, you will need the tar utility and the GNU gunzip utility.

> Solaris tar is known to have problems with some of the long filenames in the MySQL binary distribution. In order to successfully unpack the binary distribution on a Solaris system, you may need to obtain GNU gtar. A binary distribution version of this is available at www.mysql.com/downloads/os-solaris.html.

The binary distributions are all named using the following convention: mysql-<VERSION>-<OS>.tar.gz. <VERSION> is a number representing the version of the software contained in the distribution <OS> is the operating system the binary distribution is built for. Binary distributions named mysql-max-<VERSION>-

 Copyright © 2001 O'Reilly & Associates, Inc.

<OS>.tar.gz contain a version of MySQL compiled with support for transaction –safe tables.

Assume for this example, that we have chosen to install MySQL 3.23.40 on an Sun Solaris server. Also assume the distribution file mysql-3.23.40-sun-solaris2.7-sparc.tar.gz has been downloaded into the /tmp directory.

We recommend that you create a user and group for MySQL administration. This user should be used to run the mysql server, and to perform administrative tasks. It is possible to run the server as `root`, but is it not recommended.

The first step is to create a user that will be used to run the MySQL server. On Solaris and Linux, this can be done with the `useradd` and `groupadd` utilities. In our example, we create a user called "mysql". In practice, you can choose any username and/or that you like.

```
$ groupadd mysql
$ useradd -g mysql mysql
```

Select the desired location for the mysql software and change your current directory to that location. In this example, we install into `/usr/local`.

> `/usr/local` is the standard install location that is assumed by the MySQL software. You can, of course, install it wherever you like. If you choose to install in a location other than `/usr/local`, you will need to modify some of scripts provided by MySQL. See the MySQL installation instructions at *http://www.mysql.org/documentation* for more details.

```
$ cd /usr/local
```

Now, unpack the software.

```
$ gunzip -c /tmp/mysql-3.23.40-sun-solaris2.7-sparc.tar.gz | tar -xf -
```

> On a Solaris server, you may need to use GNU tar:
>
> ```
> $ gunzip -c /tmp/mysql-3.23.40-sun-solaris2.7-
> sparc.tar.gz | gtar -xf -
> ```

You should now see one directory.

```
$ ls -l
total 1
drwxr-xr-x  28 user     users        1024 Jul 18 14:29 mysql-3.23.40-sun-
solaris2.7-sparc/
```

The next step is to create a symbolic link so that the installation can be referred to as `/usr/local/mysql`.

```
$ ln -s mysql-3.23.40-sun-solaris2.7-sparc mysql
$ ls -l
.
.
lrwxrwxrwx   1 user     users          31 Jul 26 18:32 mysql -> mysql-3.23.40-sun-
solaris2.7-sparc/
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
drwxr-xr-x  12 user     users        1024 Jul 18 17:07 mysql-3.23.40-sun-
solaris2.7-sparc/
.
```

Now, lets go into the `mysql` directory and have a look around.

```
$ cd mysql
$ ls -l
total 4476
-rw-r--r--   1 user     users       19076 Jul 18 14:21 COPYING
-rw-r--r--   1 user     users       28011 Jul 18 14:21 COPYING.LIB
-rw-r--r--   1 user     users      122213 Jul 18 14:19 ChangeLog
-rw-r--r--   1 user     users       14842 Jul 18 14:21 INSTALL-BINARY
-rw-r--r--   1 user     users        1976 Jul 18 14:19 README
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 bin/
-rwxr-xr-x   1 user     users         773 Jul 18 17:07 configure*
drwxr-x---   4 user     users        1024 Jul 26 18:27 data/
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 include/
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 lib/
-rw-r--r--   1 user     users     2321255 Jul 18 14:21 manual.html
-rw-r--r--   1 user     users     1956858 Jul 18 14:21 manual.txt
-rw-r--r--   1 user     users       80487 Jul 18 14:21 manual_toc.html
drwxr-xr-x   6 user     users        1024 Jul 18 17:07 mysql-test/
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 scripts/
drwxr-xr-x   3 user     users        1024 Jul 18 17:07 share/
drwxr-xr-x   7 user     users        1024 Jul 18 17:07 sql-bench/
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 support-files/
drwxr-xr-x   2 user     users        1024 Jul 18 17:07 tests/
```

The software is now installed. We have a few set-up tasks left to do. Run `scripts/mysql_install_db` to create the MySQL grant tables:

```
$ scripts/mysql_install_db
Preparing db table
Preparing host table
Preparing user table
Preparing func table
Preparing tables_priv table
Preparing columns_priv table
Installing all prepared tables
010726 19:40:05  ./bin/mysqld: Shutdown Complete
.
.
.
```

Set up the ownership of the binaries so they are owned by root and in the MySQL adminstrator group that you created earlier (in our case, mysql).

```
$ chown -R root /usr/local/mysql
$ chgrp -R mysql /usr/local/mysql
```

Set the ownership of the data directories to the MySQL administrative user you created earlier (for this example, mysql).

```
$ chown -R mysql /usr/local/mysql/data
```

MySQL is now installed and ready to go. To start the server run safe_mysqld:

```
$ bin/safe_mysqld -user=mysql &
```

 Copyright © 2001 O'Reilly & Associates, Inc.

If you would like to have MySQL server start automatically at server boot, you can copy `support-files/mysql.server` script to the appropriate location on your system. See the script for more details.

Installing a binary RPM (RedHat Package Manager) Distribution

The recommended way to install MySQL on an Intel Linux system is via RPM (RedHat Package Manager). Several RPM files are available for download.

Filename	Description
`MySQL-<VERSION>.i386.rpm`	The MySQL server software
`MySQL-client-<VERSION>.i386.rpm`	The MySQL client software
`MySQL-bench-<VERSION>.i386.rpm`	MySQL tests and benchmarks. This requires the perl and msql-mysql-modules RPMs.
`MySQL-devel-<VERSION>.i386.rpm`	Libraries and includes files for compiling other MySQL clients.
`MySQL-shared-<VERSION>.i386.rpm`	MySQL client shared libraries.

The procedure for installing a RPM distribution is simple. First, obtain the RPM(s) you wish to install. Second, use the rpm utility to install.

Assume for this example that we will install all of the RPM packages for version 3.23.40 on an Intel Linux system. Also assume RPM files `MySQL-3.23.40-1.i386.rpm`, `MySQL-client-3.23.40-1.i386.rpm`, `MySQL-devel-3.23.40-1.i386.rpm`., `MySQL-bench-3.23.40-1.i386.rpm` and `MySQL-shared-3.23.40-1.i386.rpm` have been downloaded to `/tmp`.

Installing them is as simple as executing this sequence of commands:

```
$ rpm -i /tmp/MySQL-3.23.40-1.i386.rpm
$ rpm -i /tmp/MySQL-client-3.23.40-1.i386.rpm
$ rpm -i /tmp/MySQL-devel-3.23.40-1.i386.rpm
$ rpm -i /tmp/MySQL-bench-3.23.40-1.i386.rpm
$ rpm -i /tmp/MySQL-shared-3.23.40-1.i386.rpm
```

You don't need to install all of them. At a minimum, you'll need the MySQL and MySQL-client packages.

The RPM will creates the appropriate entries in `'/etc/rc.d/'` to automatically start and stop the server at system boot and shutdown. The RPM also starts the mysql server , so after the RPM install is complete, you are ready to start using MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

The RPM distributions place the files in different locations than the "tarball" distribution. To examine an RPM to determine where the files were placed, use the RPM query option.

```
$ rpm -qpl MySQL-<VERSION>.i386.rpm
```

If you wish to determine the location but have discarded the RPM files already, you can query the RPM database.

```
$ rpm -ql MySQL-<VERSION>
```

Another thing to note: the RPM places data in `/var/lib/data` instead of `/usr/local/data`.

Installing from a source distribution

Installing from a source distribution is very different from installing a binary distribution. Since you will be building the software from source code, you will need a full set of tools:

- GNU `gunzip`
- `tar` or GNU `tar`.
- An ANSI C++ compiler. GNU gcc 2.95.2 (or higher) is recommended. egcs 1.0.2/egcs 2.91.66, SGI C++ and SunPro C++ are known to work.
- make. Gnu make is recommended.

Compiling from source is an inherently involved process with many possible variations depending upon your operating system, your desired configuration, your toolset, etc. As a result, we provide an overview of the process to get you started. However, we assume that you are experienced with building software from source. If you encounter problems building or installing MySQL, please refer to the full MySQL install documentation set at *http://www.mysql.com/documentation*.

The source distributions are named using the following convention: mysql-<VERSION>.tar.gz. There is not a special MySQL-Max version of the MySQL source as all versions are compiled from the same code base.

For our example, assume that mysql-3.23.40.tar.gz has been already downloaded to `/tmp`.

Just as with the binary install, the first step is to create a user that will be used to run the MySQL server.

```
$ groupadd mysql
$ useradd -g mysql mysql
```

In your filesystem, move to the location where you would like to unpack the source. Unpack the bundle.

 Copyright © 2001 O'Reilly & Associates, Inc.

```
$ gunzip -c /tmp/mysql-3.23.40.tar.gz | tar -xf -
```

Move into the newly created mysql directory. You must configure and build MySQL from this location.

```
$ cd mysql-3.23.40
```

Now, use the `configure` script to configure your build. We use the prefix option to set our install location to /usr/local/mysql.

```
$ ./configure --prefix=/usr/local/mysql
```

---

configure offers a host of options that you can use to control how your build is set up. For more help on what's available, run

```
$ ./configure --help
```

Also, check the full install documentation at *http://www.mysql.com/documentation* for a list of commonly used configure options.

Configure may take a few minutes to complete. Next, we build the binaries.

```
$ make
```

If all went well, you now have binary version of MySQL. The last thing you need to do is install it.

```
$ make install
```

The software is now installed. We have a few set-up tasks left to do. Run `mysql_install_db` to create the MySQL grant tables.

```
$ cd /usr/local/mysql
$ scripts/mysql_install_db
Preparing db table
Preparing host table
Preparing user table
Preparing func table
Preparing tables_priv table
Preparing columns_priv table
Installing all prepared tables
010726 19:40:05  ./bin/mysqld: Shutdown Complete
.
.
.
```

Set up the ownership of the binaries so they are owned by root and in the MySQL adminstrator group that you created earlier (in our case, mysql).

```
$ chown -R root /usr/local/mysql
$ chgrp -R mysql /usr/local/mysql
```

Set the ownership of the data directories to the MySQL administrative user you created earlier (for this example, mysql).

```
$ chown -R mysql /usr/local/mysql/data
```

MySQL is now installed and ready to go. To start the server run safe_mysqld:

Copyright © 2001 O'Reilly & Associates, Inc.

```
$ bin/safe_mysqld –user=mysql &
```

If you would like to have MySQL server start automatically at server boot, you can copy `support-files/mysql.server` script to the appropriate location on your system. See the script for more details.

Windows Installation

The distributions for Windows can be found in the same place as the distributions for Unix: at *http://www.mysql.com/downloads* or at one of the mirror sites. Windows installation is simply a matter of downloading the mysql-<VERSION>.zip, unzipping it, and running the setup program

The default install location for MySQL Windows is c:\mysql. The installer will allow you to change the location, however if you choose to do so, you may need to modify some configuration files to get everything working correctly. Refer to the full MySQL installation documentation at *http://www.mysql.com/documentation* for more information.

The installer will give you the choice between a typical, compact and custom install. We recommend the typical install unless you wish to modify the list of components that are installed. In that case, use the custom install.

The Windows binary distribution contains several servers for you to choose from.

Server Name	**Description**
`Mysqld`	Debug binary with memory allocation checking, symbolic link support and transactional table support (InnoDB and BDB).
`mysqld-opt`	Optimized binary with NO support for transactional tables.
`mysqld-nt`	Optimized binary with support for NT named pipes.
`mysqld-max`	Optimized binary with support for transactional tables.
`mysqld-max-nt`	Optimized binary with support for transactional tables and NT named pipes.

Once you have the software installed, the next step is to start the server. Though the binaries are the same, the procedure for running the server is different depending on whether you are using Windows 95/98 or Windows NT/2000. Each of these is covered separately.

Copyright © 2001 O'Reilly & Associates, Inc.

Starting MySQL on Windows 95/98

In order to run MySQL on a Wndows 95/98 system, you'll need to have TCP/IP support installed. This can be found on your Windows CD-ROM if you haven't installed it already.

> If you are running Windows 95, you need to make sure you have the right version of Winsock. MySQL requires Winsock 2. Obtain the latest and greatest Winsock from *http://www.microsoft.com*.

You will need to choose (from the list above) which server you would like to run. Note that you can run the '`-nt`' binaries, but you don't get any benefit from it, since named pipes are not supported on Window 95/98. Assume for our example, we have decided to run `mysql-opt.` To get the server started, open up an MS-DOS window and type:

```
C:\> c:\mysql\bin\mysqld-opt
```

To stop the server, in an MS-DOS window type:

```
C:\> c:\mysql\bin\mysqladmin –u root shutdown
```

Starting MySQL on Windows NT/2000

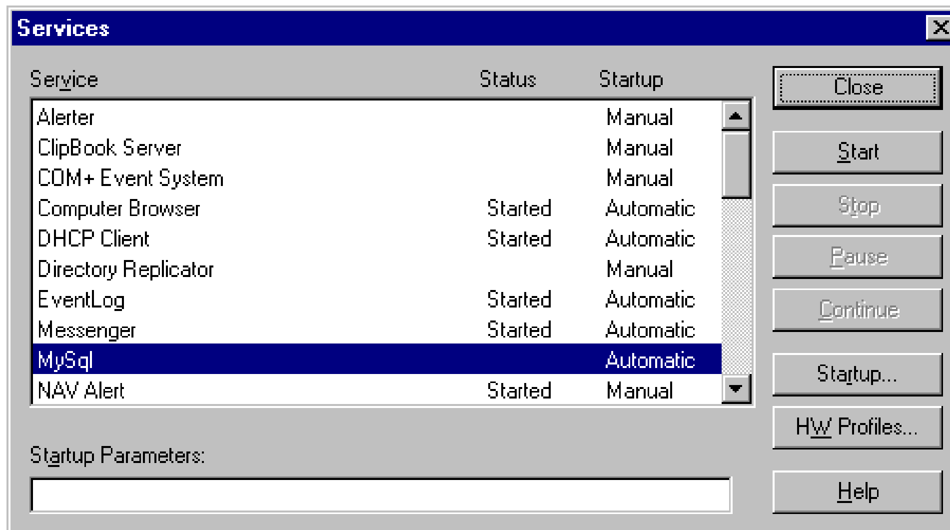
On Windows NT/2000, you'll need at least service pack 3 to get the right level of TCP/IP support for MySQL.

We recommend that you run the MySQL server as an NT service. To install it as a service, open up an MS-DOS window and type:

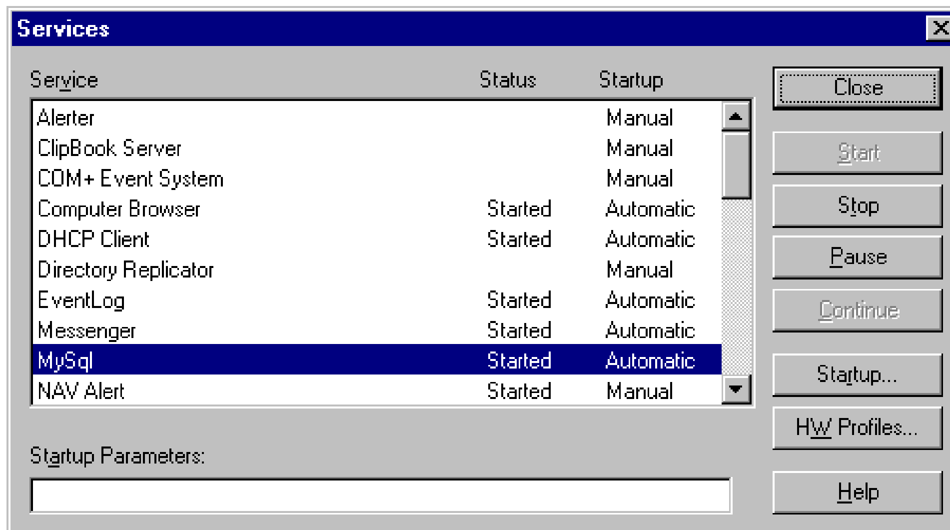
```
C:\> c:\mysql\bin\mysqld-nt –install
```

This will create an NT service called 'MySQL'. This service is now available from from Services control panel. To access this, open up your control panel and double-click on the "Services" icon. You will see a MySQL service.

Copyright © 2001 O'Reilly & Associates, Inc.



You can start now MySQL by clicking on the "Start" button. If you would like to change the command line options for the MySQL service you can type them in the "Startup Parameters" text box before starting the service. After the service has started, the status shows as "Started".



To stop the service, press "Stop. You can also start and stop the service from an MS-DOS prompt using the net start and net stop commands. To start it this way, open an MS-DOS window, and type

```
C:\> net start mysql
The MySql service is starting.
The MySql service was started successfully.
```

To stop it again, type

```
C:\> net stop mysql
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
The MySql service is stopping............
The MySql service was stopped successfully.
```

Wrapping up

If all went well, you've successfully installed MySQL. Now what? We recommend that you take a look at Chapter 5 "Database Adminstration". Here we cover the basics of how to configure and run your server. After that, you'll be all set to start developing applications MySQL.

If you had problems getting MySQL to install, please refer to the full MySQL documentation at *http://www.mysql.com/documentation* for help. There you will find the most up-to-date information as well as more details about installation steps for other operating systems.

4

SQL According to MySQL

The Structured Query Language (SQL) is the language used to read and write to MySQL databases. Using SQL, you can search for data, enter new data, modify data, or delete data. SQL is simply the most fundamental tool you will need for your interactions with MySQL. Even if you are using some application or graphical user interface to access the database, somewhere under the covers that application is generating SQL.

SQL is a sort of "natural" language. In other words, an SQL statement should read—at least on the surface—like a sentence of English text. This approach has both benefits and drawbacks, but the end result is a language very unlike traditional programming languages such as C, Java, or Perl.

SQL Basics

SQL* is "structured" in the sense that it follows a very specific set of rules. A computer program can easily parse a formulated SQL query. In fact, the O'Reilly book *lex & yacc* by John Levine, Tony Mason, and Doug Brown implements a SQL grammar to demonstrate the process of writing a program to interpret language! A *query* is a fully-specified command sent to the database server, which then performs the requested action. Below is an example of an SQL query:

```
SELECT name FROM people WHERE name LIKE 'Stac%'
```

As you can see, this statement reads almost like a form of broken English: "Select names from a list of people where the names are like Stac." SQL uses very few of

* Pronounced either "sequel" or "ess-que-ell." Certain people get very religious about the pronunciation of SQL. Ignore them. It is important to note, however, that the "SQL" in MySQL is properly pronounced "ess-que-ell."

Copyright © 2001 O'Reilly & Associates, Inc.

the formatting and special characters that are typically associated with computer languages. Consider, for example, "$++;($*++/$|);$&$^,,;$!" in Perl versus "SELECT value FROM table" in SQL.

The SQL Story

IBM invented SQL in the 1970s shortly after Dr. E. F. Codd first invented the concept of a relational database. From the beginning, SQL was an easy to learn, yet powerful language. It resembles a natural language such as English, so that it might be less daunting to a nontechnical person. In the 1970s, even more than today, this advantage was an important one.

There were no casual hackers in the early 1970s. No one grew up learning BASIC or building web pages in HTML. The people programming computers were people who knew everything about how a computer worked. SQL was aimed at the army of nontechnical accountants and business and administrative staff that would benefit from being able to access the power of a relational database.

SQL was so popular with its target audience, in fact, that in the 1980s the Oracle corporation launched the world's first publicly available commercial SQL system. Oracle SQL was a huge hit and spawned an entire industry built around SQL. Sybase, Informix, Microsoft, and several other companies have since come forward with their implementations of a SQL-based Relational Database Management System (RDBMS).

At the time Oracle and its first competitors hit the scene, SQL was still brand new and there was no standard. It was not until 1989 that the ANSI standards body issued the first public SQL standard. These days it is referred to as SQL89. This new standard, unfortunately, did not go far enough into defining the technical structure of the language. Thus, even though the various commercial SQL languages were drawing closer together, differences in syntax still made it non-trivial to switch among implementations. It was not until 1992 that the ANSI SQL standard came into its own.

The 1992 standard is called both SQL92 and SQL2. The SQL2 standard expanded the language to accommodate as many of the proprietary extensions added by the commercial implementations as was possible. Most cross-DBMS tools have standardized on SQL2 as the way in which they talk to relational databases. Due to the extensive nature of the SQL2 standard, however, relational databases that implement the full standard are very complex and very resource intensive.

Copyright © 2001 O'Reilly & Associates, Inc.

SQL2 is not the last word on the SQL standard. With the growing popularity of object-oriented database management systems (OODBMS) and object-relational database management systems (ORDBMS), there has been increasing pressure to capture support for object-oriented database access in the SQL standard. The recent SQL3 standard is the answer to this problem.

When MySQL came along, it took a new approach to the business of database server development. Instead of manufacturing another giant RDBMS and risk having nothing more to offer than the big guys, Monty created a small, fast implementation of the most commonly used SQL functionality. Over the years, that basic functionality has grown to support just about anything you might want to do with 80% of database applications.

The Design of SQL

As we mentioned earlier, SQL resembles a human language more than a computer language. SQL accomplishes this resemblance by having a simple, defined imperative structure. Much like an English sentence, individual SQL commands, called "queries," can be broken down into language parts. Consider the following examples:

```
CREATE   TABLE             people (name CHAR(10))
verb     object            adjective phrase

INSERT   INTO people       VALUES ('me')
verb     indirect object   direct object

SELECT   name              FROM people             WHERE name LIKE '%e'
verb     direct object     indirect object         adj. phrase
```

Most implementations of SQL, including MySQL, are case-insensitive. Specifically, it does not matter how you type SQL keywords as long as the spelling is correct. The CREATE example from above could just as well appeared:

```
cREatE TAblE people (name cHaR(10))
```

The case-insensitivity only extends to SQL keywords.* In MySQL, names of databases, tables, and columns are case-sensitive. This case-sensitivity is not necessarily true for all database engines. Thus, if you are writing an application that should work against all databases, you should act as if names are case-sensitive.

* For the sake of readability, we capitalize all SQL keywords in this book. We recommend this convention as a solid "best practice" technique.

Copyright © 2001 O'Reilly & Associates, Inc.

This first element of an SQL query is always a verb. The verb expresses the action you wish the database engine to take. While the rest of the statement varies from verb to verb, they all follow the same general format: you name the object upon which you are acting and then describe the data you are using for the action. For example, the query CREATE TABLE people (name CHAR(10)) uses the verb CREATE, followed by the object TABLE. The rest of the query describes the table to be created.

An SQL query originates with a client—the application that provides the façade through which a user interacts with the database. The client constructs a query based on user actions and sends the query to the SQL server. The server then must process the query and perform whatever action was specified. Once the server has done its job, it returns some value or set of values to the client.

Because the primary focus of SQL is to communicate actions to the database server, it does not have the flexibility of a general-purpose language. Most of the functionality of SQL concerns input to and output from the database: adding, changing, deleting, and reading data. SQL provides other functionality, but always with an eye towards how it can be used to manipulate the data within the database.

Sending SQL to MySQL

You can send SQL to MySQL using a variety of mechanisms. The most common way is through some programming API from Part III. For the purposes of this chapter, however, we recommend you use the command line tool *mysql*. When you run this program at the command line, it prompts you for SQL to enter:

```
[09:04pm] carthage$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.22.29

Type 'help' for help.

mysql>
```

The *mysql* command above says to connect to the MySQL server on the local machine as the user root with the client prompting you for a password. Another option, the -h option, enables you to connect to MySQL servers on remote machines:

```
[09:04pm] carthage$ mysql -u root -h db.imaginary.com -p
```

There is absolutely no relationship between UNIX or Windows 2000 user names and MySQL user names. Users have to be added to MySQL independently of the host on which they reside. No one therefore has an account on a clean MySQL install except root. As a general rules, you should never connect to MySQL as root

Copyright © 2001 O'Reilly & Associates, Inc.

except when performing database administration tasks. If you have a clean installation of MySQL that you can afford to throw away, then it is useful to connect as root for the purposes of this chapter so that you may create and drop databases. Otherwise, you will have to connect to MySQL as whatever user name has been assigned to you.

You can enter your SQL commands all on a single line, or you can split them across multiple lines. MySQL patiently waits for a semi-colon before executing the SQL you enter:

```
mysql> SELECT book_number
    -> FROM book
    -> ;
+-------------+
| book_number |
+-------------+
|           1 |
|           2 |
|           3 |
+-------------+
3 rows in set (0.00 sec)
```

With the *mysql* command line, you generally get a command history depending on how it was compiled. If it is compiled into your *mysql* client, you can use the up and down arrows on your keyboard to navigate through past SQL commands you have executed. For more information on the *mysql* tool, see Chapter 20.

Database Creation

In order to get started using MySQL, you need to create a database to use. First, let's take a look at the databases that come with a clean MySQL installation using the `SHOW DATABASES` command. On a clean install of MySQL 3.23.40 on Mac OS X, the following tables already exist:

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.37 sec)

mysql>
```

The first database, `mysql`, is MySQL's system database. You will learn more about it in Chapter 5. The second table is a play table you can use to learn MySQL and run tests against. You may find other databases on your server if you are not deal-

Copyright © 2001 O'Reilly & Associates, Inc.

ing with a clean install. For right now, however, we want to create a new database to illustrate the use of the MySQL CREATE statement:

```
CREATE DATABASE TEMPDB;
```

And then to work with the new database TEMPDB:

```
USE TEMPDB;
```

Finally, you can delete that database by issuing the DROP DATABASE command:

```
DROP DATABASE TEMPDB;
```

You will find as you explore SQL that you create new things using the CREATE statement and destroy things using the DROP statement just as we used them here.

Table Management

You should now feel comfortable connecting to a database on a MySQL server. For the rest of the chapter, you can use the test database that comes with MySQL or your own play database. Using the SHOW command, you can display a list of tables in the current database in a similar manner to the way you used it to show databases. In a brand new install, the test database has no tables. The following shows the output of the SHOW TABLES command when connected to the mysql system database:

```
mysql> SHOW TABLES;
+-----------------+
| Tables_in_mysql |
+-----------------+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----------------+
6 rows in set (0.00 sec)
```

To get a look at the what one of these tables looks like, you can use the DESCRIBE command:

```
mysql> DESCRIBE db;
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
13 rows in set (0.36 sec)
```

This output describes each column in the table with its data type, whether or not it can contain null values, what kind of key it is, any default values, and extra information. If all of this means nothing to you, don't worry. We will describe each of these elements as the chapter progresses.

You should now be ready to create your first table. You will, of course, want to connect back to the `test` database since you definitely do not want to be adding tables to the `mysql` database. The *table*, a structured container of data, is the most basic concept of a relational database. Before you can begin adding data to a table, you must define the table's structure. Consider the following layout:

```
+---------------------------------+
|             people              |
+-------------+-------------------+
| name        | char(10) not null |
| address     | text(100)         |
| id          | int               |
+-------------+-------------------+
```

Not only does the table contain the names of the columns, but it also contains the types of each field as well as any additional information the fields may have. A field's data type specified what kind of data the field can hold. SQL data types are similar to data types in other programming languages. The full SQL standard allows for a large range of data types. MySQL implements most of them as well as a few MySQL-specific types.

The general syntax for table creation is:

```
CREATE TABLE table_name (column_name1 type [modifiers]
                      [, column_name2 type [modifiers]]
)
```

What constitutes a valid identifier—a name for a table or column—varies from DBMS to DBMS. MySQL allows up to 64 characters in an identifier, supports the character '$' in identifiers, and lets identifiers start with a valid number. More important, however, MySQL considers any valid letter for your local character set to be a valid letter for identifiers.

Copyright © 2001 O'Reilly & Associates, Inc.

A *column* is the individual unit of data within a table. A table may have any number of columns, but large tables may be inefficient. This is where good database design, discussed in Chapter 8, *Database Design*, becomes an important skill. By creating properly normalized tables, you can "join" tables to perform a single search from data housed in more than one table. We discuss the mechanics of a join later in the chapter.

Consider the following create statement:

```
CREATE TABLE USER (
    USER_ID    BIGINT UNSIGNED NOT NULL PRIMARY KEY,
    USER_NAME  CHAR(10)        NOT NULL,
    LAST_NAME  VARCHAR(30),
    FIRST_NAME VARCHAR(30),
    OFFICE     CHAR(2)         NOT NULL DEFAULT 'NY');
```

This statement creates a table called `USER` with four columns: `USER_ID`, `USER_NAME`, `LAST_NAME`, `FIRST_NAME`, and `OFFICE`. After each column name comes the data type for that column followed by any modifiers. We will discuss data types and the `PRIMARY KEY` modifier later in this chapter.

The NOT NULL modifier indicates that the column may not contain any null values. If you try to assign a null value to that column, your SQL will generate an error. Actually, there are a couple of exceptions to this rule. First, if the column is `AUTO_INCREMENT`, a null value will cause a value to be automatically generated. We cover auto-incrementing later in the chapter. The second exception is for columns that specify default values like the `OFFICE` column. In this case, the `OFFICE` column will be assigned a value of 'NY' when a null value is assigned to the column.

Like most things in life, destruction is much easier than creation. The command to drop a table from the database is:

```
DROP TABLE table_name
```

This command will completely remove all traces of that table from the database. MySQL will remove all data within the destroyed table from existence. If you have no backups of the table, you absolutely cannot recover from this action. The moral of this story is to always keep backups and be very careful about dropping tables. You will thank yourself for it some day.

With MySQL, you can specify more than one table to delete by separating the table names with commas. For example, `DROP TABLE` *`people, animals, plants`* would delete the three named tables. You can also use the `IF EXISTS` modifier to avoid an error should the table not exist when you try to drop it. This modifier is useful for huge scripts designed to create a database and all its tables. Before the create, you do a `DROP TABLE` *`table_name`* `IF EXISTS`.

Copyright © 2001 O'Reilly & Associates, Inc.

MySQL Data Types

In a table, each column has a type. As we mentioned earlier, a SQL data type is similar to a data type in traditional programming languages. While many languages define a bare-minimum set of types necessary for completeness, SQL goes out of its way to provide types such as `MONEY` and `DATE` that will be useful to every day users. You could store a `MONEY` type in a more basic numeric type, but having a type specifically dedicated to the nuances of money processing helps add to SQL's ease of use—one of SQL's primary goals.

Chapter 17, *MySQL Data Types*, provides a full reference of SQL types supported by MySQL. Table 4-1 is an abbreviated listing of the most common types.

Table 4-1. Common MySQL Data Types (see Chapter 17 for a full list)

Data Type	Description
INT	An integer value. MySQL allows an INT to be either signed or unsigned.
REAL	A floating point value. This type offers a greater range and precision than the INT type, but it does not have the exactness of an INT.
CHAR(*length*)	A fixed-length character value. No CHAR fields can hold strings greater in length than the specified value. Fields of lesser length are padded with spaces. This type is likely the most commonly used type in any SQL implementation.
TEXT(*length*)	A variable length character value.
DATE	A standard date value. The DATE type stores arbitrary dates for the past, present, and future. MySQL is Y2K compliant in its date storage.
TIME	A standard time value. This type stores the time of day independent of a particular date. When used together with a date, a specific date and time can be stored. MySQL additionally supplies a DATETIME type that will store date and time together in one field.

MySQL supports the UNSIGNED attribute for all numeric types. This modifier forces the column to accept only positive (unsigned) numbers. Unsigned fields have an upper limit that is double that of their signed counterparts. An unsigned TINYINT—MySQL's single byte numeric type—has a range of 0 to 255 instead of the -127 to 127 range of its signed counterpart.

MySQL provides more types than those mentioned above. In day-to-day programming, however, you will find yourself using mostly these types. The size of the data you wish to store plays a large role the design of your MySQL tables.

Copyright © 2001 O'Reilly & Associates, Inc.

Numeric Types

Before you create a table, you should have a good idea of what kind of data you wish to store in the table. Beyond obvious decisions about whether your data is character-based or numeric, you should know the approximate size of the data to be stored. If it is a numeric field, what is its maximum possible value? What is its minimum possible value? Could that change in the future? If the minimum is always positive, you should consider an unsigned type. You should always choose the smallest numeric type that can support your largest conceivable value. If, for example, we had a field that represented the population of a state, we would use an unsigned `INT` field. No state can have a negative population. Furthermore, in order for an unsigned `INT` field not to be able to hold a number representing a state's population, that state's population would have to be roughly the population of the entire Earth.

Character Types

Managing character types is a little more complicated. Not only do you have to worry about the minimum and maximum string lengths, but you also have to worry about the average size, the amount of variation likely, and the need for indexing. For our current purposes, an *index* is a field or combination of fields on which you plan to search—basically, the fields in your `WHERE` clause. Indexing is, however, much more complicated than this simplistic description, and we will cover indexing later in the chapter. The important fact to note here is that indexing on character fields works best when the field is fixed length. If there is little—or, preferably, no—variation in the length of your character-based fields, then a CHAR type is likely the right answer. An example of a good candidate for a CHAR field is a country code. The ISO provides a comprehensive list of standard two-character representations of country codes (US for the U.S.A., FR for France, etc.).* Since these codes are always exactly two characters, a CHAR(2) is always the right answer for this field.

A value does not need to be invariant in its length to be a candidate for a CHAR field. It should, however, have very little variance. Phone numbers, for example, can be stored safely in a CHAR(13) field even though phone number length varies from nation to nation. The variance simply is not that great, so there is no value to making a phone number field variable in length. The important thing to keep in mind with a CHAR field is that no matter how big the actual string being stored is,

* Don't be lulled into believing states/provinces work this way. If you want to write an application that works in an international environment and stores state/province codes, make sure to make it a CHAR(3) since Australia uses three-character state codes. Also note that there is a 3-character ISO country-code standard.

Copyright © 2001 O'Reilly & Associates, Inc.

the field always takes up exactly the number of characters specified as the field's size—no more, no less. Any difference between the length of the text being stored and the length of the field is made up by padding the value with spaces. While the few potential extra characters being wasted on a subset of the phone number data is not anything to worry about, you do not want to be wasting much more. Variable-length text fields meet this need.

A good, common example of a field that demands a variable-length data type is a web URL. Most web addresses can fit into a relatively small amount of space—*http://www.ora.com*, *http://www.imaginary.com*, *http://www.mysql.com*—and consequentially do not represent a problem. Occasionally, however, you will run into web addresses like:

> *http://www.winespectator.com/Wine/Spectator/_notes|5527293926834323221480431354?Xv11=&Xr5=&Xv1=&type-region-search-code=&Xa14=flora+springs&Xv4=.*

If you construct a `CHAR` field large enough to hold that URL, you will be wasting a significant amount of space for most every other URL being stored. Variable-length fields let you define a field length that can store the odd, long-length value while not wasting all that space for the common, short-length values.

Variable-length text fields in MySQL use precisely the minimum storage space required to store an individual field. A `VARCHAR(255)` column that holds the string "hello world," for example, only takes up twelve bytes (one byte for each character plus an extra byte to store the length).

In opposition to the ANSI standard, `VARCHAR` in MySQL fields are not padded. Any extra spaces are removed from a value before it is stored.

You cannot store strings whose lengths are greater than the field length you have specified. With a `VARCHAR(4)` field, you can store at most a string with 4 characters. If you attempt to store the string "happy birthday," MySQL will truncate the string to "happ." The downside is that there is no way to store the odd string that exceeds your designated field size. Table 4-2 shows the storage space required to

Copyright © 2001 O'Reilly & Associates, Inc.

store the 144 character Wine Spectator URL shown above along with an average-sized 30 character URL.

Table 4-2. The Storage Space Required by the Different MySQL Character Types

Data Type	Storage for a 144 Character String	Storage for a 30 Character String	Maximum String Size
CHAR(150)	150	150	255
VARCHAR(150)	145	31	255
TINYTEXT(150)	145	31	255
TEXT(150)	146	32	65535
MEDIUMTEXT(150)	147	33	16777215
LONGTEXT(150)	148	34	4294967295

In this table, you will note that storage requirements grow one byte at a time for variable-length types of `MEDIUM_TEXT` and `LONGTEXT`. This is because `TEXT` uses an extra byte to store the potentially greater length of the text it contains. Similarly, `MEDIUM_TEXT` uses an extra two bytes over `VARCHAR` and `LONGTEXT` an extra three bytes.

If, after years of uptime with your database, you find that the world has changed and a field that once comfortably existed as a `VARCHAR(25)` now must be able to hold strings as long as 30 characters, you are not out of luck. MySQL provides a command called `ALTER TABLE` that enables you to redefine a field type without losing any data.

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

Binary Data Types

MySQL provides a set of binary data types that closely mirror their character counterparts. The MySQL binary types are `CHAR BINARY`, `VARCHAR BINARY`, `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB`. The practical distinction between character types and their binary counterparts is the concept of encoding. *Binary data* is basically just a chunk of data that MySQL makes no effort to interpret. *Character data*, on the other hand, is assumed to represent textual data from human alphabets. It thus is encoded and sorted based on rules appropriate to the character set in question. In the case of installations on an ASCII system, MySQL sorts binary in a case-insensitive, ASCII order.

Enumerations and Sets

MySQL provides two other special kinds of types. The `ENUM` type allows you specify at table creation a list of possible values that can be inserted into that field. For

Copyright © 2001 O'Reilly & Associates, Inc.

example, if you had a column named fruit into which you wanted to allow only "apple," "orange," "kiwi," or "banana," you would assign this column the type ENUM:

```
CREATE TABLE meal(meal_id INT NOT NULL PRIMARY KEY,
                  fruit ENUM('apple', 'orange', 'kiwi',
                             'banana'))
```

When you insert a value into that column, it must be one of the specified fruits. Because MySQL knows ahead of time what valid values are for the column, it can abstract them to some underlying numeric type. In other words, instead of storing "apple" in the column as a string, it stores it as a single byte number. You just use "apple" when you call the table or when you view results from the table.

The MySQL SET type works in the same way, except it lets you store multiple values in a field at the same time.

Other Kinds of Data

Every piece of data you will ever encounter can be stored using numeric or character types. Technically, you could even store numbers as character types. Just because you can do so, however, does not mean that you should do so. Consider, for example, storing money in the database. You could store that as an INT or a REAL. While a REAL might seem more intuitive—money requires decimal places, after all—an INT fields actually makes more sense. With floating point values like REAL fields, it is often impossible to capture a number with a specific decimal value. If, for example, you insert the number 0.43 to represent $0.43, MySQL may store that as 0.42999998. This small difference can be problematic when applied to a large number of mathematical operations. By storing the number as an INT and inserting the decimal into the right place, you can be certain that the value represents exactly what you intend it to represent.

Isn't all of that a major pain? Wouldn't it be nice if MySQL provided some sort of data type specifically suited to money values? MySQL provides special data types to handle special kinds of data. MONEY is an example of one of these kinds of data. DATE is another.

Indexing

While MySQL has better performance than any of the larger database servers, some problems still call for careful database design. For instance, if we had a table with millions of rows of data, a search for a specific row would take a long time. Most database engines enable you to help it in these searches through a tool called an index.

Copyright © 2001 O'Reilly & Associates, Inc.

Indices help the database store data in a way that makes for quicker searches. Unfortunately, you sacrifice disk space and modification speed for the benefit of quicker searches. The most efficient use of indices is to create an index for columns on which you tend to search the most. MySQL supports the following syntax for index creation:

```
CREATE INDEX index_name ON tablename (column1,
                                      column2,
                                      ...,
                                      columnN)
```

MySQL also lets you create an index at the same time you create a table using the following syntax:

```
CREATE TABLE materials (id          INT      NOT NULL,
                        name        CHAR(50) NOT NULL,
                        resistance INT,
                        melting_pt REAL,
                        INDEX index1 (id, name),
                        UNIQUE INDEX index2 (name))
```

The previous example creates two indices for the table. The first index—named `index1`—consists of both the `id` and `name` fields. The second index includes only the `name` field and specifies that values for the name field must always be unique. If you try to insert a field with a `name` held by a row already in the database, the insert will fail. All fields declared in a unique index must be declared as being `NOT NULL`.

Even though we created an index for `name` by itself, we did not create an index for just `id`. If we did want such an index, we would not need to create it—it is already there. When an index contains more than one column (for example: `name`, `rank`, and `serial_number`), MySQL reads the columns in order from left to right. Because of the structure of the index MySQL uses, any subset of the columns from left to right are automatically created as indices within the "main" index. For example, `name` by itself and `name` and `rank` together are both "free" indices created when you create the index `name`, `rank`, `serial_number`. An index of `rank` by itself or `name` and `serial_number` together, however, is not created unless you explicitly create it yourself.

MySQL also supports the ANSI SQL semantics of a special index called a primary key. In MySQL, a primary key is a unique key with the name `PRIMARY`. By calling a column a primary key at creation, you are naming it as a unique index that will support table joins. The following example creates a cities table with a primary key of `id`.

```
CREATE TABLE cities (id      INT  NOT NULL PRIMARY KEY,
                     name    VARCHAR(100),
                     pop     MEDIUMINT,
                     founded DATE)
```

Copyright © 2001 O'Reilly & Associates, Inc.

Before you create a table, you should determine which fields, if any, should be keys. As we mentioned above, any fields which will be supporting joins are good candidates for primary keys. See Chapter 8 for a detailed discussion on how to design your tables with good primary keys.

Though MySQL supports the ANSI syntax for foreign keys, it does not actually use them to perform integrity checking in the database. This is an sue where the introduction of a feature would cause a slowdown in performance with little real benefit. Applications themselves should generally worry about foreign key integrity.

Managing Data

The first thing you do with a newly created table is add data to it. With the data in place, you may want to make changes and eventually remove it.

Inserts

Adding data to a table is one of the more straightforward concepts in SQL. You have already seen several examples of it in this book. MySQL supports the standard SQL INSERT syntax:

```
INSERT INTO table_name (column1, column2, ..., columnN)
VALUES (value1, value2, ..., valueN)
```

When inserting data into numeric fields, you can insert the value as is; for all other fields, you must wrap them in single quotes. For example, to insert a row of data into a table of addresses, you might issue the following command:

```
INSERT INTO addresses (name, address, city, state, phone, age)
VALUES('Irving Forbush', '123 Mockingbird Lane', 'Corbin', 'KY',
       '(800) 555-1234', 26)
```

In addition, the escape character—‘\’ by default—enables you to escape single quotes and other literal instances of the escape character:

```
# Insert info for the directory Stacie's Directory which
# is in c:\Personal\Stacie
INSERT INTO files (description, location)
VALUES ('Stacie\'s Directory', 'C:\\Personal\\Stacie')
```

MySQL allows you to leave out the column names as long as you specify a value for every single column in the table in the exact same order they were specified in the table’s `CREATE` call. If you want to use the default values for a column, however, you must specify the names of the columns for which you intend to insert non-default data. If you do not have a default value set up for a column and that

Copyright © 2001 O’Reilly & Associates, Inc.

column is NOT NULL, you must include that column in the INSERT statement with a non-NULL value. If the earlier files table had contained a column called size, then the default value would be used. MySQL allows you to specify a custom default value in the table's CREATE.

Newer versions of MySQL support a nonstandard INSERT call for inserting multiple rows at once:

```
INSERT INTO foods VALUES (NULL, 'Oranges', 133, 0, 2, 39),
                         (NULL, 'Bananas', 122, 0, 4, 29),
                         (NULL, 'Liver', 232, 3, 15, 10)
```

While these nonstandard syntaxes supported by MySQL are useful for quick system administration tasks, you should not use them when writing database applications unless you really need the speed benefit they offer. As a general rule, you should stick as close to the ANSI SQL2 standard as MySQL will let you. By doing so, you are making certain that your application can run against any other database in the future. Being flexible is especially critical for people with mid-range database needs because such users generally hope one day to become people with high-end database needs.

Another non-standard syntax supported by MySQL is where you specify the column name and value together:

```
INSERT INTO book SET title='The Vampire Lestat', author='Anne Rice';
```

Another approach to inserting data is by using the data from some other table (or group of tables) to populate your new table. For example:

```
INSERT INTO foods (name, fat)
SELECT food_name, fat_grams FROM recipes
```

You should note that the number of columns in the INSERT matches the number of columns in the SELECT. In addition, the data types for the INSERT columns must match the data types for the corresponding SELECT columns. Finally, the SELECT clause in an INSERT statement cannot contain an ORDER BY modifier and cannot be selected from the same table where the INSERT is occurring.

Sequence Generation

The best kind of primary key is one that has absolutely no meaning in the database except to act as a primary key. The best way to achieve this is to make a numeric primary key that increments every time you insert a new row. Looking at the cities table shown earlier, the first city you insert would have an id of 1, the second 2, the third 3, and so on. In order to successfully manage this sequencing

Copyright © 2001 O'Reilly & Associates, Inc.

of a primary key, you need some way to guarantee that a number can be read and incremented by one and only one client at a time.

When you create a table in MySQL, you can specify at most one column as being AUTO_INCREMENT. When you do this, you can automatically have this column insert the highest current value for that column + 1 when you insert a row and specify NULL or 0 for that row's value. The AUTO_INCREMENT row must be indexed. The following command creates the cities table with the id field being AUTO_INCREMENT:

```
CREATE TABLE cities (id      INT  NOT NULL PRIMARY KEY AUTO_INCREMENT,
                     name    VARCHAR(100),
                     pop     MEDIUMINT,
                     founded DATE)
```

The first time you insert a row, the id field for your first row will be 1 so long as you use NULL or 0 for that field in the INSERT statement. For example, this command takes advantage of the AUTO_INCREMENT feature:

```
INSERT INTO cities (id, name, pop)
VALUES (NULL, 'Houston', 3000000)
```

If no other values are in that table when you issue this command, MySQL will set this field to 1, not NULL (remember, it cannot be NULL). If other values are present in the table, the value inserted will be one greater than the largest current value for id.

Another way to implement sequences is by referring to the value returned by the LAST_INSERT_ID() function:

```
UPDATE table SET id=LAST_INSERT_ID (id+1);
```

Updates

The insertion of new rows into a database is just the start of data management. Unless your database is read-only, you will probably also need to make periodic changes to the data. The standard SQL modification statement looks like this:

```
UPDATE table_name
SET column1=value1, column2=value2, ..., columnN=valueN
[WHERE clause]
```

In addition to assigning literal values to a column, you can also assign calculate the values. You can even calculate the value based on a value in another column:

```
UPDATE years
SET end_year = begin_year+5
```

This command sets the value in the end_year column equal to the value in the begin_year column plus 5 for each row in that table.

Copyright © 2001 O'Reilly & Associates, Inc.

The WHERE Clause

You probably noted something earlier called the WHERE clause. In SQL, a WHERE clause enables you to pick out specific rows in a table by specifying a value that must be matched by the column in question. For example:

```
UPDATE bands
SET lead_singer = 'Ian Anderson'
WHERE band_name = 'Jethro Tull'
```

This UPDATE specifies that you should only change the lead_singer column for the row where band_name is identical to "Jethro Tull." If the column in question is not a unique index, that WHERE clause may match multiple rows. Many SQL commands employ WHERE clauses to help pick out the rows on which you wish to operate. Because the columns in the WHERE clause are columns on which you are searching, you should generally have indices created around whatever combinations you commonly use. We discuss the kinds of comparisons you can perform in the WHERE clause later in the chapter.

Deletes

Deleting data is a very straightforward operation. You simply specify the table from which you want to delete followed by a WHERE clause that identifies the rows you want to delete:

```
DELETE FROM table_name [WHERE clause]
```

As with other commands that accept a WHERE clause, the WHERE clause is optional. In the event you leave out the WHERE clause, you will delete all of the records in the table! Of all destructive commands in SQL, this is the easiest one to issue mistakenly.

Queries

The last common SQL command used is the one that enables you to view the data in the database: SELECT. This action is by far the most common action performed in SQL. While data entry and modifications do happen on occasion, most databases spend the vast majority of their lives serving up data for reading. The general form of the SELECT statement is as follows:

```
SELECT column1, column2, ..., columnN
FROM table1, table2, ..., tableN
[WHERE clause]
```

This syntax is certainly the most common way in which you will retrieve data from any SQL database. Of course, there are variations for performing complex and powerful queries. We cover the full range of the SELECT syntax in Chapter 16. The simplest form is this:

Copyright © 2001 O'Reilly & Associates, Inc.

```
SELECT 1;
```

This simple, though completely useless query returns a result set with a single row containing a single column with the value of 1. A more useful version of this query might be something like:

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
1 row in set (0.01 sec)
```

The expression `DATABASE()` is a MySQL function that returns the value of the current database in use. We will cover functions in more detail later in the chapter. Nevertheless, you can see how simple SQL can provide a quick and dirty way of finding out important information.

Most of time, however, you will want to use slightly more complex queries that help you pull data from a table in the database. The first part of a `SELECT` statement enumerates the columns you wish to retrieve. You may specify a "*" to say that you want to select all columns. The `FROM` clause specifies which tables those columns come from. The `WHERE` clause identifies the specific rows to be used and enables you to specify how to join two tables.

Joins

Joins put the "relational" in relational databases. Specifically, a join enables you to match a row from one table up with a row in another table. The basic form of a join is what you may hear sometimes described as an *inner join*. Joining tables is a matter of specifying equality in columns from two tables:

```
SELECT book.title, author.name
FROM author, book
WHERE book.author = author.id
```

Consider a database where the book table looks like Table 4-3.

Table 4-3. A book Table

ID	Title	Author	Pages
1	The Green Mile	4	894
2	Guards, Guards!	2	302
3	Imzadi	3	354
4	Gold	1	405
5	Howling Mad	3	294

Copyright © 2001 O'Reilly & Associates, Inc.

And the author table looks like Table 4-4.

Table 4-4. An author Table

ID	Name	Citizen
1	Isaac Asimov	US
2	Terry Pratchet	UK
3	Peter David	US
4	Stephen King	US
5	Neil Gaiman	UK

An inner join creates a virtual table by combining the fields of both tables for rows that satisfy the query in both tables. In our example, the query specifies that the `author` field of the `book` table must be identical to the `id` field of the `author` table. The query's result would thus look like Table 4-5.

Table 4-5. Query Results Based on an Inner Join

Book Title	Author Name
The Green Mile	Stephen King
Guards, Guards!	Terry Pratchet
Imzadi	Peter David
Gold	Isaac Asimov
Howling Mad	Peter David

Neil Gaiman is nowhere to be found in these results. He is left out because there is no value for his `author.id` value found in the `author` column of the `book` table. An inner join only contains those rows that exactly match the query. We will discuss the concept of an outer join later in the chapter for situations where we would be interested in the fact that we have an author in the database who does not have a book in the database.

Aliasing

When you use column names that are fully qualified with their table and column name, the names can grow to be quite unwieldy. In addition, when referencing SQL functions, which will be discussed later in the chapter, you will likely find it cumbersome to refer to the same function more than once within a statement. The aliased name, usually shorter and more descriptive, can be used anywhere in the same SQL statement in place of the longer name. For example:

```
# A column alias
SELECT long_field_names_are_annoying AS myfield
FROM table_name
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
WHERE myfield = 'Joe'
# A table alias under MySQL
SELECT people.names, tests.score
FROM tests, really_long_people_table_name AS people
# A table alias under mSQL
SELECT people.names, tests.score
FROM tests, really_long_people_table_name=people
```

Ordering and Grouping

The results you get back from a select are, by default, indeterminate in the order they will appear. Fortunately, SQL provides some tools for imposing order on this seemingly random list: ordering and grouping.

Basic Ordering

You can tell a database that it should order any results you see by a certain column. For example, if you specify that a query should order the results by `last_name`, then the results will appear alphabetized according to the `last_name` value. Ordering comes in the form of the `ORDER BY` clause:

```
SELECT last_name, first_name, age
FROM people
ORDER BY last_name, first_name
```

In this situation, we are ordering by two columns. You can order by any number of columns, but the columns must be named in the `SELECT` clause. If we had failed to select the `last_name` above, we could not have ordered by the `last_name` field.

If you want to see things in reverse order, add the `DESC` keyword:

```
ORDER BY last_name DESC
```

The `DESC` keyword applies only to the field that comes right before it. If you are sorting on multiple fields, only the field right before `DESC` is reversed; the others occur in ascending order.

Localized Sorting

Sorting is actually a very complex problem for applications that need to be able to run on computers all over the world. The rules for sorting strings vary from alphabet to alphabet, even when two alphabets use mostly the same symbols. MySQL handles the problem of sorting by making it dependent on the character set of the MySQL engine. Out of the box, the default character set is ISO-8859-1 (Latin-1). MySQL uses the sorting rules for Swedish and Finnish with ISO-8859-1.

To change the sorting rules, you change the character set. First, you need to make sure the character set is compiled into the server when you compile MySQL.

Copyright © 2001 O’Reilly & Associates, Inc.

Chapter 3 contains detailed instructions on installing MySQL with specific character sets, including those you define yourself. With the proper character set compiled into the server, you can change the default character set by launching the server with the argument *--default-character-set=*CHARSET.

Because of the simplicity of the English alphabet, the use of a single set of sorting rules MySQL associates with ISO-8859-1 does not affect English sorting. Unfortunately, different languages can have different sorting rules even though they share the same character sets. Swedish and German, for example, both use the ISO-8859-1 character set. Swedish sorts 'ä' after 'z', while German sorts 'ä' before 'a'. The default rules therefore fail German users.

MySQL lets you address this problem by creating custom character sets. When you compile the driver, you can compile in support for whatever character sets you desire as long as you have a configuration file for that character set. This file contains the characters that make up the character set and the rules for sorting them. You can write your own as well as use the ones that come with MySQL.

The real problem here is that MySQL incorrectly associates sorting rules with character sets. A character set is nothing more than a grouping of characters with a related purpose. Nothing about the ISO-8859-1 character set implies sorting for Swedes, Italians, Germans, or anyone else. when working with MySQL, however, you just need to remember that sorting rules are directly tied to the character set.

Grouping

Grouping lets you group rows with a similar value into a single row in order to operate on them together. You usually do this to perform aggregate functions on the results. We will go into functions a little later in the chapter.

Consider the following:

```
mysql> SELECT name, rank, salary FROM people;
+--------------+----------+--------+
| name         | rank     | salary |
+--------------+----------+--------+
| Jack Smith   | Private  |  23000 |
| Jane Walker  | General  | 125000 |
| June Sanders | Private  |  22000 |
| John Barker  | Sargeant |  45000 |
| Jim Castle   | Sargeant |  38000 |
+--------------+----------+--------+
5 rows in set (0.01 sec)
```

If you group the results by rank, the output changes:

```
mysql> SELECT rank FROM people GROUP BY rank;
+----------+
| rank     |
+----------+
| General  |
| Private  |
| Sargeant |
+----------+
3 rows in set (0.01 sec)
```

Now that you have the output grouped, you can finally find out the average salary for each rank. Again, we will discuss more on the functions you see in this example later in the chapter.

```
mysql> SELECT rank, AVG(salary) FROM people GROUP BY rank;
+----------+-------------+
| rank     | AVG(salary) |
+----------+-------------+
| General  | 125000.0000 |
| Private  |  22500.0000 |
| Sargeant |  41500.0000 |
+----------+-------------+
3 rows in set (0.04 sec)
```

The power of ordering and grouping combined with the utility of SQL functions enables you to do a great deal of data manipulation even before you retrieve the data from the server. You should take great care not to rely too heavily on this power. While it may seem like an efficiency gain to place as much processing load as possible onto the database server, it is not really the case. Your client application is dedicated to the needs of a particular client, while the server is being shared by many clients. Because of the greater amount of work a server already has to do, it is almost always more efficient to place as little load as possible on the database server. MySQL may be the fastest database around, but you do not want to waste that speed on processing that a client application is better equipped to manage.

If you know that a lot of clients will be asking for the same summary information often (for instance, data on a particular rank in our previous example), just create a new table containing that information and keep it up to date as the original tables change. This is similar to caching and is a common database programming technique.

Limiting Results

Sometimes an application is looking for only the first few rows that match a query. Limiting queries can help avoid problems bogging down the network with unwanted results. MySQL enables an application to limit the number of results through a `LIMIT` clause in a query:

Copyright © 2001 O'Reilly & Associates, Inc.

```
SELECT * FROM people ORDER BY name LIMIT 10;
```

Of course, to get the last 10 people from the table, you can use the DESC keyword. If you want people from the middle, however, you have to get a bit trickier. To accomplish this task, you need to specify the number of the first record you want to see (record 0 is the first record, 1 the second) and the number of rows you want to see:

```
SELECT * FROM people ORDER BY name LIMIT 19, 30;
```

This sample displays records 20 through 49. The 19 in the LIMIT clause tells MySQL to start with the twentieth record. The thirty then tells MySQL to return the next 30 records.

SQL Operators

So far, we have basically used the = operator for the obvious task of verifying that two values in a WHERE clause equal one another. Other fairly basic operations include <>, >, <, <=, and >=. One special thing to note is that MySQL allows you to use either <> or != for "not equal". Table 4-6 contains a full set of simple SQL operators.

Table 4-6. . The Simple SQL Operators Supported by MySQL

Operator	Context	Description
+	Arithmetic	Addition
-	Arithmetic	Subtraction
*	Arithmetic	Multiplication
/	Arithmetic	Division
=	Comparison	Equal
<> or !=	Comparison	Not equal
<	Comparison	Less than
>	Comparison	Greater than
<=	Comparison	Less than or equal to
>=	Comparison	Greater than or equal to
AND	Logical	And
OR	Logical	Or
NOT	Logical	Negation

MySQL operators have the following rules of precedence:

1. BINARY
2. NOT

Copyright © 2001 O'Reilly & Associates, Inc.

3. - (unary minus)
4. * / %
5. + -
6. << >>
7. &
8. |
9. < <= > >= = <=> <> IN IS LIKE REGEXP
10. BETWEEN
11. AND
12. OR

Logical Operators

SQL's logical operators—AND, OR, and NOT—let you build more dynamic WHERE clauses. The AND and OR operators specifically let you add multiple criteria to a query:

```
SELECT USER_NAME
FROM USER
WHERE AGE > 18 AND STATUS = 'RESIDENT';
```

This sample query provides a list of all users who are residents and are old enough to vote.

You can build increasingly complex queries through the use of parentheses. The parentheses tell MySQL which comparisons to evaluate first:

```
SELECT USER_NAME
FROM USER
WHERE (AGE > 18 AND STATUS = 'RESIDENT')
OR (AGE > 18 AND STATUS = 'APPLICANT');
```

In this more complex query, we are looking for anyone currently eligible to vote as well as people who might be eligible in the near future. Finally, you can use the NOT operator to negate an entire expression:

```
SELECT USER_NAME
FROM USER
WHERE NOT (AGE > 18 AND STATUS = 'RESIDENT');
```

Null's Idiosyncrasies

Null is a tricky concept for most people new to databases to understand. As in other programming languages, null is not a value, but an absence of a value. This concept is useful, for example, if you have a customer profiling database that grad-

Copyright © 2001 O'Reilly & Associates, Inc.

ually gathers information about your customers as they offer it. When you first create the record, for example, you may not know how many pets they have. You want that column to hold NULL instead of 0 so you can tell the difference between customers with no pets and customers whose pet ownership is unknown to you.

The concept of null gets a little funny when you use it in SQL calculations. Many programming languages use null as simply another kind of value. In Java, the following syntax evaluates to true when the variable is null and false when it is not:

```
str == null
```

The similar expression in SQL, `COL = NULL`, is neither true nor false—it is always `NULL`, no matter what the value of the `COL` column. The following query will therefor not act as you would expect:

```
SELECT TITLE FROM BOOK WHERE AUTHOR = NULL;
```

This query will always provide an empty result set, even when you have `AUTHOR` columns with `NULL` values. To test for "nullness", you should use the `IS NULL` and `IS NOT NULL` operators:

```
SELECT TITLE FROM BOOK WHERE AUTHOR IS NULL;
```

MySQL provides a special operator to use when you are not sure if you might be dealing with null values called the null-safe operator <=>. It will return true if both sides are null or if both sides are not null:

```
mysql> SELECT 1 <=> NULL, NULL <=> NULL, 1 <=> 1;
+------------+---------------+---------+
| 1 <=> NULL | NULL <=> NULL | 1 <=> 1 |
+------------+---------------+---------+
|          0 |             1 |       1 |
+------------+---------------+---------+
1 row in set (0.00 sec)
```

Membership Tests

Sometimes applications need to be able to check if a value is a member of a set of values or within a particular range. The IN operator helps with the former:

```
SELECT TITLE FROM BOOK WHERE AUTHOR IN ('Stephen King', 'Richard Bachman');
```

This query will return the titles of all books where Stephen King is the author.* Similarly, you can check for all books not written by him by using the `NOT IN` operator.

To determine if a value is in a particular range, an application should use the BETWEEN operator:

* Richard Bachman is a pseudonym used by Stephen King for some of his books.

Copyright © 2001 O'Reilly & Associates, Inc.

```
SELECT TITLE FROM BOOK WHERE BOOK_ID BETWEEN 1  AND 100;
```

Both of these simple examples could, of course, be replicated with the more basic operators. The Stephen King check, for example, could have been done by using the = operator and an OR. The check on book ID's also could have been done with an OR clause using the >= and <= or > and <. As your queries get more complex, however, these operators can help you build both readable and better performing queries than those you might create with the basic operators.

Pattern Matching

We provided a peek at ANSI SQL pattern matching earlier in the chapter with the query:

```
SELECT name FROM people WHERE name LIKE 'Stac%'
```

Using the LIKE operator, we compared a column value (name) to an incomplete literal ('Stac%'). MySQL supports the ability to place special characters into string literals that match like wild cards. The '%' character, for example, matches any arbitrary number of characters, including no character at all. The above SELECT statement would therefore match 'Stacey', 'Stacie', 'Stacy', and even 'Stac'. The character '_' matches any single character. 'Stac_y' would match only 'Stacey'. 'Stac__' would match 'Stacie' and 'Stacey', but not 'Stacy' or 'Stac'.

Pattern matching expressions should never be used with the basic comparison operators. Instead, they should be used only with the LIKE and NOT LIKE operators. It is also important to remember that these comparisons are case-insensitive.

MySQL supports a non-ANSI kind of pattern matching that is actually much more powerful using the same kind of expressions that Perl programmers and *grep* users are accustomed to. MySQL refers to these as extended regular expressions. Instead of LIKE and NOT LIKE, these operators must be used with the REGEXP and NOT REGEXP operators.* Table 4-7 contains a list of the supported extended regular expression patterns.

Table 4-7. . MySQL Extended Regular Expressions

Pattern	Description	Examples
.	Matches any single character.	*Stac..* matches any value containing the characters "Stac" followed by two characters of any value.

* MySQL provides synonyms for these operators: RLIKE and NOT RLIKE.

Copyright © 2001 O'Reilly & Associates, Inc.

Table 4-7. . MySQL Extended Regular Expressions

Pattern	Description	Examples
[]	Matches any character in the brackets. You can also match a range of characters.	*[Ss]tacey* matches values containing both "Stacey" and "stacey". *[a-zA-Z]* matches values containing one instance of any character in the English (unaccented) portion of the Roman alphabet.
*	Matches zero or more instances of the character that precedes it.	*Ap*le* matches values containing "Aple", "Apple", "Appple", etc. *Los .*es* matches values containing the string "Los " and "es" with anything in between. *[0-9]** matches values containing any arbitrary number.
^	What follows must come at the beginning of the value.	*^Stacey* matches values that start with "Stacey".
$	What precedes it must end the value.	*cheese$* matches any value ending in the string "cheese"

You should note a couple of important facts about extended regular expressions. First, unlike basic pattern matching, MySQL extended regular expressions are case sensitive. They also do not require a match for the entire string. The pattern simply needs to occur somewhere within the value. Consider the following example:

```
mysql> SELECT * FROM BOOK;
+---------+-----------------------------------------+----------------+
| BOOK_ID | TITLE                                   | AUTHOR         |
+---------+-----------------------------------------+----------------+
|       1 | Database Programming with JDBC and Java | George Reese   |
|       2 | JavaServer Pages                        | Hans Bergsten  |
|       3 | Java Distributed Computing              | Jim Farley     |
+---------+-----------------------------------------+----------------+
3 rows in set (0.01 sec)
```

In this table, we have three books from O'Reilly's Java series. The interesting thing about the Java series is that all books begin with or end with the word "Java". The first sample query checks for any titles `LIKE 'Java'`:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java';
Empty set (0.01 sec)
```

Because `LIKE` looks for an exact match of the pattern specified, no rows match—none of the titles are exactly 'Java'. To find out which books start with the word 'Java' using simple patterns, we need to add a '%' sign:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java%';
+----------------------------+
| TITLE                      |
+----------------------------+
| JavaServer Pages           |
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
| Java Distributed Computing |
+----------------------------+
2 rows in set (0.00 sec)
```

This query had two matches because only two of the books had titles that matched 'Java%' exactly. The extended regular expression matches, however, are not exact matches. They simply look for the expression anywhere within the compared value:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java';
+------------------------------------------+
| TITLE                                    |
+------------------------------------------+
| Database Programming with JDBC and Java |
| JavaServer Pages                         |
| Java Distributed Computing               |
+------------------------------------------+
3 rows in set (0.06 sec)
```

By simple changing the operator from LIKE to REGEXP, we changed how it matches things. 'Java' appears somewhere in each of the titles, so the query returns all of the titles. In order to find only the titles that start with the word 'Java' using extended regular expressions, we need to specify that we are interested in the start:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP '^Java';
+----------------------------+
| TITLE                      |
+----------------------------+
| JavaServer Pages           |
| Java Distributed Computing |
+----------------------------+
2 rows in set (0.01 sec)
```

The same thing applies to finding 'Java' at the end:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java$';
+------------------------------------------+
| TITLE                                    |
+------------------------------------------+
| Database Programming with JDBC and Java |
+------------------------------------------+
1 row in set (0.00 sec)
```

The extended regular expression syntax is definitely much more complex than the simple pattern matching of ANSI SQL. In addition to the burden of extra complexity, you also should consider the fact that MySQL extended regular expressions do not work in most other databases. When you need complex pattern matching, however, they provide you with power that is simply unsupportable by simple pattern matching.

Copyright © 2001 O'Reilly & Associates, Inc.

Advanced Features

Using the SQL presented thus far in this chapter should handle 90% of your database programming needs. On occasion, however, you will need some extra power not available in the basic SQL functionality. We close out the chapter with a discussion of a few of these features.

Transactions

MySQL recently introduced transactions and thus SQL for executing statements in a transactional context. By default, MySQL is in a state called autocommit. Autocommit mode means that any SQL you send to MySQL is executed immediately. In some cases, however, you may want to execute two or more SQL statements together as a single unit of work.

A transfer between to bank accounts is the perfect example of such a transaction. The bank system needs to make sure that the debit of the first account and the credit to the second account occur as a single unit of work. If they were treated separately, the server could in theory crash between the debit and the credit. The result would be that you would lose that money!

By making sure the two statements occur as a single unit of work, transactions ensure that the first statement can be "rolled back" in the event the second statement fails. To use transactions in MySQL, you first need to create a table using a transactional table type such as BDB or InnoDB. If your MySQL install was not compiled with support for these table types, you cannot use transactions. The SQL to create a transactional table is:

```
CREATE TABLE ACCOUNT (
    ACCOUNT_ID BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    BALANCE    DOUBLE)
TYPE = BDB;
```

For a transaction against a transactional table to work, you need to turn off autocommit. You can do this through the command:

```
SET AUTOCOMMIT=0;
```

Now you are ready to begin using MySQL transactions. Transactions start with the `BEGIN` command:

```
BEGIN;
```

Your mysql client is now in a transactional context with respect to the server. Any change you make to a transactional table will not be made permanent until you commit it. Changes to non-transactional tables, however, will still take place immediately. In the case of the account transfer, we issue the following statements:

Copyright © 2001 O'Reilly & Associates, Inc.

```
UPDATE ACCOUNT SET BALANCE = 50.25 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = 100.25 WHERE ACCOUNT_ID = 2;
```

Once done with any changes, you complete the transaction using the `COMMIT` command:

```
COMMIT;
```

The true advantage of transactions, of course, comes into play should an error occur in executing the second statement. To abort the entire transaction before a commit, issue the `ROLLBACK` command:

```
ROLLBACK;
```

Of course, it would be useful if MySQL performed the actual math. It can do just that so long as you store the values you want with a SELECT call:

```
SELECT @FIRST := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
SELECT @SECOND := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 2;
UPDATE ACCOUNT SET BALANCE = @FIRST - 25.00 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = @SECOND + 25.00 WHERE ACCOUNT_ID = 2;
```

In addition to issuing the `COMMIT` command, a handful of other commands will automatically end any current transaction as if a `COMMIT` had been issued. These commands are:

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

Chapter 9 covers some of the more intricate details of using transactions in database applications.

Table Locking

Table locking is the poor man's transaction. In short, MySQL lets you lock down any table so that only a single client can use it. Unlike transactions, you are not limited by the type of the table. You cannot, however, rollback any actions taken against a locked table.

Locking has two basic functions:

Copyright © 2001 O'Reilly & Associates, Inc.

1. To enable multiple statements to execute against a single table as one unit of work.
2. To enable multiple updates to occur faster since under some circumstances.

MySQL supports three kinds of locks, read, read local, and write. Both kinds of read locks lock the table for reading by a client and all other clients. As long as the lock is in place, no one can write to the locked tables. Read and read local locks differ in that read local allows a client to execute non-conflicting INSERT statements so long as no changes to the MySQL files from outside of MySQL will occur while the lock as held. If changes might occur by agents outside of MySQL, then a read lock is required.

A write lock locks the specified tables against all access, read or write, by any other client. To lock a table, use the following command:

```
LOCK TABLES ACCOUNT WRITE;
```

Now that the ACCOUNT table is locked, you can read from it and then modify the data behind it and be certain that no one else will change the data you read between your read and write operations.

```
SELECT @BAL:=BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = @BAL * 0.03 WHERE ACCOUNT_ID = 1;
```

Finally, you need to release the locks:

```
UNLOCK TABLES;
```

Functions

Functions in SQL are similar to functions in other programming languages like C and Perl. The function takes zero or more arguments and returns some value. For example, the function SQRT(16) returns 4. Within a MySQL SELECT statement, functions may be used in either of two places:

As a value to be retrieved

This form involves a function in the place of a column in the list of columns to be retrieved. The return value of the function, evaluated for each selected row, is part of the returned result set as if it were a column in the database.* For example:

```
# Select the name of each event as well as the date of the event
# formatted in a human-readable form for all events more
# recent than the given time. The FROM_UnixTIME() function
# transforms a standard Unix time value into a human
# readable form.
SELECT name, FROM_UnixTIME(date)
```

* You can use the aliasing you used earlier in the chapter to give the resulting columns "friendly" names.

Copyright © 2001 O'Reilly & Associates, Inc.

```
FROM events
WHERE time > 90534323

# Select the title of a paper, the full text of the paper,
# and the length (in bytes) of the full text for all
# papers authored by Stacie Sheldon.
# The LENGTH() function returns the character length of
# a given string.
SELECT title, text, LENGTH(text)
FROM papers
WHERE author = 'Stacie Sheldon'
```

As part of a `WHERE` clause

This form involves a function in the place of a constant when evaluating a WHERE clause. The value of the function is used for comparison for each row of the table. For example:

```
# Randomly select the name of an entry from a pool of 35
# entries. The RAND() function generates a random number
# between 0 and 1 (multiplied by 34 to make it between 0
# and 34 and incremented by 1 to make it between 1 and
# 35). The ROUND() function returns the given number
# rounded to the nearest integer, resulting in a whole
# number between 1 and 35, which should match one of
# the ID numbers in the table.
SELECT name
FROM entries
WHERE id = ROUND( (RAND()*34) + 1 )

# You may use functions in both the value list and the
# WHERE clause. This example selects the name and date
# of each event less than a day old. The UNIX_TIMESTAMP()
# function, with no arguments, returns the current time
# in Unix format.
SELECT name, FROM_UnixTIME(date)
FROM events
WHERE time > (Unix_TIMESTAMP() - (60 * 60 * 24) )

# You may also use the value of a table field within
# a function. This example returns the name of anyone
# who used their name as their password. The ENCRYPT()
# function returns a Unix password-style encryption
# of the given string using the supplied 2-character salt.
# The LEFT() function returns the left-most n characters
# of the given string.
SELECT name
FROM people
WHERE password = ENCRYPT(name, LEFT(name, 2))
```

Date Functions

The most common functions you will use are likely to be the MySQL functions that enable you to manipulate dates. You already saw some of these functions above

Copyright © 2001 O'Reilly & Associates, Inc.

for translating a UNIX-style date into a human-readable form of the date. MySQL, of course, provides more powerful functions for doing things like calculating the time between two dates:

```
SELECT TO_DAYS(NOW()) - TO_DAYS('2000-12-31');
```

This example provides the number of days that have passed in this millennium. The `NOW()` function, of course, returns the `DATETIME` representing the moment in time when the command is executed. Less obviously, the `TO_DAYS()` function returns the number of days since the year 1 B.C.* represented by the specified `DATE` or `DATETIME`.

Not everyone likes to see dates formatted the way MySQL provides them by default. Fortunately, MySQL lets you format dates to your own liking using the DATE_FORMAT function. It takes a DATE or DATETIME and a format string indicating how you want the date formatted:

```
mysql> SELECT DATE_FORMAT('1969-02-17', '%W, %M %D, %Y');
+--------------------------------------------+
| DATE_FORMAT('1969-02-17', '%W, %M %D, %Y') |
+--------------------------------------------+
| Monday, February 17th, 1969                |
+--------------------------------------------+
1 row in set (0.39 sec)
```

Chapter 16 contains a full list of valid tokens for the `DATE_FORMAT()` function.

String Functions

In addition to date functions, you are likely to make use of string functions. We saw one such function above: the `LENGTH()` function. This function naturally provides the number of characters in the string. The most common string function you are likely to use, however, is the `TRIM()` function that helps remove spaces from columns that may be padded with spaces.

One interesting function is the SOUNDEX() function. It translates a word into its soundex representation. The soundex representation is a way of representing the sound of a string so that you can compare two strings to see if they sound alike:

```
mysql> SELECT SOUNDEX('too');
+----------------+
| SOUNDEX('too') |
+----------------+
| T000           |
+----------------+
1 row in set (0.42 sec)
```

* MySQL is actually incapable of representing this date. Valid date ranges in MySQL are from the January 1, 1000 to December 31, 9999. There is no support in MySQL for alternative calendaring systems.

Copyright © 2001 O'Reilly & Associates, Inc.

```
mysql> SELECT SOUNDEX('two');
+----------------+
| soundex('two') |
+----------------+
| T000           |
+----------------+
1 row in set (0.00 sec)
```

Outer Joins

MySQL supports a more powerful joining than the simple inner joins we have used so far. Specifically, MySQL supports something called a *left outer join* (also known as simply *outer join*). This type of join is similar to an inner join, except that it includes data in the first column named that does not match any in the second column. If you remember our author and book tables from earlier in the chapter, you will remember that our join would not list any authors who did not have a book in our database. It is common that you may want to show entries from one table that have no corresponding data in the table to which you are joining. That is where an outer join comes into play:

```
SELECT book.title, author.name
FROM author
LEFT JOIN book ON book.author = author.id
```

Note that a outer join uses the keyword ON instead of WHERE. The results of our query would look like this:

```
+----------------+----------------+
| book.title     | author.name    |
+----------------+----------------+
| The Green Mile | Stephen King   |
| Guards, Guards!| Terry Pratchett|
| Imzadi         | Peter David    |
| Gold           | Isaac Asimov   |
| Howling Mad    | Peter David    |
| NULL           | Neil Gaiman    |
+----------------+----------------+
```

MySQL takes this concept one step further through the use of a natural outer join. A natural outer join will combine the rows from two tables where the two tables have identical column names with identical types and the values in those columns are identical:

```
SELECT my_prod.name
FROM my_prod
NATURAL LEFT JOIN their_prod
```

Copyright © 2001 O'Reilly & Associates, Inc.

Batch Processing

Batch loading is the act of loading a lot of data into or pulling a lot of data out of MySQL all at once. MySQL supports two manners of batch loading.

Command Line Loads

The simplest kind of batch load is where you stick all of your SQL commands in a file and then send the contents of that file to MySQL:

```
mysql -h somehost -u uid -p < filename
```

In other words, you are using the command line to pipe the SQL commands into a *mysql* command line. The examples that come with this book contain several SQL command files that you can load into MySQL in this manner before you run the examples.

The LOAD Command

The LOAD command enables you to load data from a file containing only data (no SQL commands). For example, if you had a file containing the names of all the books in your collection with one book on each line and the title and author separated by a tab, you could use the following command to load that data into your book table:

```
LOAD DATA LOCAL INFILE 'books.dat' INTO TABLE BOOK;
```

This command assumes that the file *books.dat* has one line for each database record to be inserted. It further assumes that there is a value for every column in the table or `\N` for null values. So, if the `BOOK` table has 3 columns, then each line of books.dat should have three tab-separated values.

The `LOCAL`* keyword tells the mysql command line to look for the file on the same machine as the client. Without it, it will try to look for the file on the server. Of course, if you are trying to load something on the server, you need to be granted the special file privilege. Finally, keep in mind that non-local loads refer to files relative to the installation directory of MySQL.

If you have a comma-separated value file like an Excel file, you can change the delimiter of the LOAD command:

```
LOAD DATA LOCAL INFILE 'books.dat'
INTO TABLE BOOK
FIELDS TERMINATED BY ',';
```

If a file contains values that would cause duplicate records in the database, you can use the `REPLACE` and `IGNORE` keywords to dictate the correct behavior.

* Reading from files local to the client is available only to users of MySQL 3.22.15 and later.

Copyright © 2001 O'Reilly & Associates, Inc.

REPLACE will cause the values from the file to replace the ones in the database, where the IGNORE keyword will cause the duplicate values to be ignored. The default behavior is to ignore duplicates.

Pulling Data from MySQL

Finally, MySQL provides a tool for pulling the results of a SELECT from the database and sticking it into a file:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS TERMINATED BY ','
FROM BOOK;
```

This query puts all rows in the BOOK table into the file *books.dat*. You could then use this file to load into an Excel spreadsheet or another database. Because this file is created on the server, it is created relative to the base directory for the database in use. On an Mac OS X basic installation, for example, this file is created as */usr/local/var/test/test.dat*.

A more complex version of this command enables you to put quotes (or any other character) around fields:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS ENCLOSED BY '"' TERMINATED BY ','
FROM BOOK;
```

Of course, you probably want only the string fields (CHAR, VARCHAR, etc.) enclosed in quotes. You can accomplish this by adding the OPTIONALLY keyword:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS OPTIONALLY ENCLOSED BY '"' TERMINATED BY ','
FROM BOOK;
```

Chapter 16 contains a full range of options for loading and extracting data from MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

5

MySQL Database Administration

Introduction

For the most part, MySQL is low maintenance software. Once you have it installed and set up, there aren't a lot of administrative demands. Nonetheless, it is not maintenance free. This chapter provides an overview of the most typical administrative tasks. These range from configuring your server, to backing it up and running periodic maintenance on it.

MySQL Configuration

As an administrator, you will need to understand how the MySQL server and clients are configured and how to modify that configuration.

The configuration of mysqld, the MySQL server, is controlled from the command line or from one or more options files. The options files simply provide an convenient way for you to specify command line options. The full set of options for the mysqld server and other utilities are documented in Chapter 20 – MySQL Programs and Utilities.

An options file might look like this:

```
# Example mysql options file.
#
# These options go to all clients
[client]

password       = my_password
port            = 3306
socket          = /var/lib/mysql/mysql.sock
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
# These options go to the mysqld server
[mysqld]
port            = 3306
socket          = /var/lib/mysql/mysql.sock
skip-locking
set-variable    = max_allowed_packet=1M
```

These seems straightforward, but lets spend a few moments to dissect this file. The first three lines look like:

```
# Example mysql options file.
#
# These options go to all clients
```

Lines starting with the pound (#) character are comment lines, and are ignored. You may also use a semi-colon (;) to indicate a comment.

The next line,

```
[client]
```

is puzzling. This specifies a "group". All options following this line apply to the group mentioned. In our case, we have specified that the following options apply to the "client" group, meaning that the following options will apply to all MySQL client programs. Let look at the next section. The line

```
password        = my_password
```

is equivalent to the command line option `--password=my_password`. Since we specified a group of `client`, this option will be passed to all client programs.

> Specifying the password in the client group is an ideal way to specify your password to all clients so you don't have to type it every time you connect. However, if you do this, make some the option file is not readable by others.

In general, any command line option `--option=value` can be specified with a line in the format `option=value`. The next two lines

```
port            = 3306
socket          = /var/lib/mysql/mysql.sock
```

provide two more examples of this. Next we encounter

```
# These options go to the mysqld server
[mysqld]
```

which specifies a group of `mysqld`. All options following this line will apply only to the `mysqld` program, and no others. A group specification continues to apply until the end of the file is reached or another group is encountered. Following this we have some more option settings which look similar to the client section. However, there are a few differences worth mentioning. The line

```
skip-locking
```

 Copyright © 2001 O'Reilly & Associates, Inc.

is an option even though it is not in the option=value format. This is equivalent to the command line option --skip-locking. So, any command line option --option can be specified by simply listing the option name.

What about this line?

```
set-variable    = max_allowed_packet=1M
```

This is the same as specifying --set-variable max_allowed_packet=1M on the command line. This is the format you use to set a mysqld variable. (See Chapter for more information on system variables.)

So know you know how to set up a configuration file for MySQL. But once you've created one, how does MySQL find it? On UNIX, MySQL reads options from the following locations:

1. /etc/my.cnf
   This is the global options file. This is read by all MySQL installations on this host.
2. DATADIR/my.cnf
   This provides a server specific options file. DATADIR is the MySQL data directory.
3. --defaults-extra-file
   A command line option specifying an additional options file to be read.
4. $HOME/.my.cnf
   A user specific options file. This is particularly useful for customizing the operation of clients.

> On Windows, the files are read from
>
> 1. My.ini in the windows system folder
> 2. C:\my.cnf
> 3. C:\mysql\data\my.cnf
> 4. --defaults-extra-file

Option files are read in the order specified above. If any option appears in multiple files, the last one takes precedence over the others. Options specified on the command line take precedence over options specified in an option file. Using this scheme, you can provide a default behavior in /etc/my.cnf that can be overridden for a particular server or by an individual user.

> Some options can be specified using Environment variables. In this case, options specified in an option file or on the command line take precedence over the Environment variables. See Chapter 18 for more information on MySQL Environment variables.

Copyright © 2001 O'Reilly & Associates, Inc.

Starting and Stopping the Server

One of the first things you'll want to do as administrator is to get MySQL set up so that it will automatically start when the OS boots, and shutdown cleanly along with the OS. On UNIX, you have a couple of options: the mysql.server script or safe_mysqld.

mysql.server

The `mysql.server` script is intended for use with the SVR4 style startup/shutdown mechanism. It is available in the `support-files` directory of your installation (usually `/usr/local/mysql/support-files`). When properly installed, it will automatically start and stop MySQL for you along with the OS.

> If you installed MySQL on Linux using the RPM package, `mysql.server` may have already been installed on your system. The RPM installer renames `mysql.server` to `mysql` when it copies it to `/etc/rc.d/init.d`. If the file `/etc/rc.d/init.d/mysql` is already there, you are already set up to automatically start and stop MySQL.

The procedure for installing mysql.server on a RedHat Linux system is as follows (you must be logged in as the `root` user to do this):

```
# cp mysql.server /etc/rc.d/init.d
```

This copies the the mysql.server into the init.d location.

```
# ln -s /etc/rc.d/init.d/mysql.server /etc/rc.d/rc3.d/S99mysql
```

This sets it up so that you'll start the MySQL server when you enter run level three, which corresponds to the directory `rc3.d`. Run level three is typically used for full multi-user mode. If your system uses a different run level for multi-user mode, you need to link that directory instead.

```
# ln -s /etc/rc.d/init.d/mysql.server /etc/rc.d/rc0.d/S01mysql
```

Run level zero (`rc0.d`) is system halt. Scripts linked here will be called when the system shuts down. Now you are all set. You may want to test it by rebooting your server.

The above example assumes RedHat Linux as the target system. The location of the init files varies depending upon which UNIX you use. For example, Solaris places them in `/etc/init.d`. You'll need to tailor the commands to the location of the init files on your system.

safe_msyqld

On a non-SVR4 UNIX system, you can use `safe_mysqld` to automatically start MySQL at boot. `safe_mysqld` is found in the bin directory of your MySQL installation (usually `/usr/local/mysql/bin`).

Copyright © 2001 O'Reilly & Associates, Inc.

In order to use it, you'll need to figure out how your system starts processes at boot time. Often, there is an /etc/rc.local file which can be modified to call safe_mysqld. If you are unsure what file to modify, talk to your system administrator to get help. Once you have identified the file, adding a section to invoke safe_mysqld.

It is also possible to start the MySQL server by directly invoking mysqld. This is NOT recommended however. Safe_mysqld and mysql.server were designed for this purpose and have a number of advantages, such as:

- determining the location of the server and invoking it with the right options
- logging run-time information to a log file
- monitoring the server and restarting it if necessary

Windows NT/2000

You can automatically start and stop the MySQL server under Windows NT or 2000 if it is installed as an NT Service. To do this, open up a MS-DOS window, and type

```
C:\mysql\bin\mysqld-nt --install
```

This is covered in greater detail in Chapter 3 – Installation. Please refer to that chapter if you need more information about installing MySQL as a service.

Once MySQL is installed as a service, it can be controlled like any other NT service.

Log Files

The MySQL server can produce a number of log files that you might find helpful:

- Error Log
- Query Log
- Binary Log
- Slow Query Log

By default, the log files are written to the data directory. The details of these are each outlined below.

Another Log file is available: The update Log. This has been obsoleted by the Binary log. If you are still using the update log (invoked by

option --log-update), we recommend that you switch to that format instead.

The Error Log

The Error Log contains the redirected output from the safe_mysqld script. On UNIX, it is a file called *hostname*.err. In windows, it is called mysql.err.

This file contains an entry for everytime the server is started and stopped, including an entry for every time the server is restarted because the server died. Critical errors and warnings about tables that need to be checked or repaired also appear here.

The Query Log

The query log contains the details of all connections and queries to the MySQL server. This can be useful for debugging a client application. It will log the SQL commands exactly as they are received by the server.

The query log can be enabled using the --log[=file] option. If no filename is given, it defaults to hostname.log. If no directory is given, it defaults to the data directory.

The Binary Log

The binary log writes contains all SQL commands that update data. Only statements that actually change data are logged. So for example, if I perform a delete that doesn't affect any rows, it will not be logged. Update statements that set a column to the same value are not logged either. Updates are logged in execution order.

The binary log is very useful for journaling transactions since the last backup. If, for example, you backup your database once per day, and your database crashes in the middle of the day. You can restore the database up the last completed transaction, by

1. Restoring the database (see the section in this chapter for more information on database backup and restore)
2. Applying the transactions from all binary logs since the last backup

The binary log can be enabled using the –log-bin[=file] option. If no filename is provided, it defaults to *hostname*-bin. If no directory is geiven, it defaults to the data directory. MySQL appends a numeric index to the filename, so the actual filename ends up being *hostname*-bin.*number*. The index is used for rotating the files. MySQL will rotate to the next index:

- When the server is restarted
- When the server is refreshed (with mysqladmin refresh)

 Copyright © 2001 O'Reilly & Associates, Inc.

- When the logs are flushed (with mysqladmin flush-logs, or SQL "FLUSH LOGS")

MySQL also creates an index file which contains a list of all used binary log files. By default this is named hostname-bin.index. If you wish, you may specify the name and or location of the index file with the `--log-bin-index[=file]` option.

In order to read a binary log, you'll need a utility called `mysqlbinlog`. The example below illustrates the workings of the binary log. Assume we've started MySQL on a host called `odin` and specified `log-bin` in our global configuration file, `/etc/my.cnf`. In the data directory, we'll see:

```
$ cd /usr/local/mysql/data
$ ls -l
.
.
-rw-rw----   1 mysql    mysql          73 Aug  5 17:06 odin-bin.001
-rw-rw----   1 mysql    mysql          15 Aug  5 17:06 odin-bin.index
.
.
```

If we inspect `odin-bin.index`, we see

```
$ cat odin-bin.index
./odin-bin.001
$
```

Lets use `myslbinlog` to read the binary log:

```
$ mysqlbinlog odin-bin.001
# at 4
#010805 17:06:00 server id  1   Start: binlog v 1, server v 3.23.40-log created
010805 17:06:00
$
```

After we do an update to a database, and look again, we get some more information.

```
$ mysql
mysql> use test
.
.
mysql> insert into test (object_id, object_title) values (1, "test");
Query OK, 1 row affected (0.02 sec)
mysql> quit
Bye
$ mysqlbinlog odin-bin.001
# at 4
#010805 17:06:00 server id  1   Start: binlog v 1, server v 3.23.40-log created
010805 17:06:00
# at 73
#010805 17:39:38 server id  1   Query   thread_id=2     exec_time=0
error_code=0
use test;
SET TIMESTAMP=997058378;
insert into test (object_id, object_title) values (1, "test");
$
```

Now, lets flush the logs to see what happens.

```
$ mysqladmin -uroot -pblueshoes flush-logs
```

We now have new file called `odin-bin.002` and our index file has been updated.

Copyright © 2001 O'Reilly & Associates, Inc.

```
$ ls -l
.
.
-rw-rw----   1 mysql    mysql         203 Aug  5 17:45 odin-bin.001
-rw-rw----   1 mysql    mysql          73 Aug  5 17:45 odin-bin.002
-rw-rw----   1 mysql    mysql          30 Aug  5 17:45 odin-bin.index
.
.
.
$ cat odin-bin.index
./odin-bin.001
./odin-bin.002
$
```

The binary logs can be played back into a server by piping the output from mysqlbinlog to a mysql session. For example

```
$ mysqlbinlog odin-bin.001 | mysql ...
```

There are a few other options you can use to control the binary logs. The option `--binlog-do-db=dbname` tells MySQL to only log updates for the specified database. The option `--binlog-ignore-db=dbname` tells MySQL to ignore the specified database for the purposes of binary logging.

The Slow Query Log

The slow query log contains all SQL commands that took longer than the variable long_query_time This can be used to identify problem queries, and expose parts of your database or application that need tuning.

The slow query log is enabled with the `–log-slow-queries[=file]` option. If no filename is provided, it defaults to *`hostname`*`-slow.log`. If no directory is given, it defaults to the data directory. The long_query_time can be set using the `--set-variable long_query_time=time` query (`time` is specified in seconds).

Log Rotation

No matter which log files you choose to enable, you'll have to worry about maintaining them so they don't fill up a file system.

If you are running RedHat Linux, you can use `mysql-log-rotate` for this. It can be found in the support-files directory of your installation. This uses the logrotate utility to automatically rotate your error log for you. For more information on logrotate, read the man page or refer to your RedHat documentation.

To install `mysql-log-rotate` on Linux, simply copy it to `/etc/logrotate.d`. You may wish to edit the script to rotate other logs that you have enabled as well. By default it only rotates the query log.

If you installed MySQL on Linux using the RPM package, `mysql-log-rotate` may have already been installed on your system. The

Copyright © 2001 O'Reilly & Associates, Inc.

RPM installer renames `mysql-log-rotate` to `mysql` when it copies it to `/etc/logrotate.d/`. If the file `/etc/logrotate.d/mysql` is already there, it has already been installed.

On systems other than RedHat, you will have to devise you own scripts for rotating the logs. Depending on which logs you have enabled, and how you want them located, the scripts could range from very simple to very complex. In general, the procedure is to copy the logfile(s) out of the way, and use `mysqladmin flush-logs`.

Unfortunately, none of these techniques work for the error log. Since it is written from the safe_mysql script, the flush logs command does not flush it. Also note that safe_mysqld will continue to append to it on successive restarts. You may want to modify the startup or shutdown scripts for your MySQL server to maintain the error log.

Database Backup

A good backup strategy is by far the most important thing you can develop as an administrator. In particular, you'll be really glad you have good backups if you ever have a system crash and need to restore your databases with as little data loss as possible. Also if you ever accidentally delete a table or destroy a database, those backups will come in very handy.

Every site is different, so it is very difficult to give specific recommendations on what you should. You need to think about your installation and your needs. In this section, we present some general backup principles that you can adopt, and we cover the technical details of performing the backups. You will have to turn this information into a coherent strategy for your installation.

In general there are a number of backups:

- Store your backups on a different device (either on another disk or perhaps a tape device) than the database, if possible. If your disk crashes, you'll be really happy to have the backups in a different place. If you are doing binary logging, store the binary logs with the backups.
- Make sure you have enough disk space for the backups to complete.
- Use binary logging, if appropriate, so you can restore your database with minimal loss of data. If you choose not to use binary logging, you will only be able to recover your database to the point of your last backup. Depending upon your application, a backup without binary logs might be useless.
- Keep an adequate number of archived backups
- Test your backups, before an emergency occurs

Copyright © 2001 O'Reilly & Associates, Inc.

mysqldump

Mysqldump is the MySQL utility provided for dumping databases. It basically generates an SQL script containing the commands (CREATE TABLE, INSERT, etc.) necessary to rebuild the database from scratch. The main advantages of this approach over direct copy (mysqlhotcopy) is that output is in a portable ASCII format which can be used across hardware and operating system to rebuild a database. Also since the output is a SQL script, it is possible to recover individual tables.

To use mysqldump to backup your database,we recommend that you use the `–opt` option. This turns on `–quick, --add-drop-table, --add-locks , --extended-insert ,` and `--lock-tables`. This should give you the fastest possible dump of your database.

> Be aware that this locks all the tables, so your database will essentially be offline while you are doing this.

So your command will look something like this:

```
$ mysqldump --opt test > /usr/backups/testdb
```

If you are using binary logging, you will also want to specify `--flush-logs`, so the binary logs get checkpointed at the time of the backup.

```
$ mysqldump --flush-logs --opt test > /usr/backups/testdb
```

mysqldump has a number of other options that you can use to customize your backup. For a list of all the options available for mysqldump, type mysqldump –help, or refer to Chapter 20, MySQL Programs and Utilities.

mysqlhotcopy

mysqlhotcopy is a perl script that uses a combination of LOCK TABLES, FLUSH TABLES and cp to perform a fast backup of the database. It simply copies the raw database files database files to another location. Since it is only doing a file copy, it is much faster than mysqldump. But, since the copy is in native format, the backup is not portable to other hardware or operating systems. Also, mysqlhotcopy can only be run on the same host as the database, whereas mysqldump can be executed remotely.

To run, mysqlhotcopy, type

```
$ mysqlhotcopy test /usr/backups
```

This will create a new directory in the /usr/backups directory which has a copy of all the datafiles in your database.

If you are using binary logging, you will also want to specify `--flushlog`, so the binary logs get checkpointed at the time of the backup.

Copyright © 2001 O'Reilly & Associates, Inc.

Database Recovery

Individual recovery scenarios vary widely, ranging from disk hardware failures to corrupted data files to accidentally dropped tables, and many points in between. In this section, we provide a general overview of recovery procedures.

In general, you need two things to perform a database recovery: your backup files and you're your binary logs. In general, performing a recovery consists of

- Restoring the database from the last backup
- Applying the binary logs to bring the system completely up to date

If you don't have binary logging enabled, the best you will be able to do is to restore the system to the last full backup.

Recovering from mysqldump files

Assume for this example, we are recovering a database named test.

Bring up the mysql server and reload the database using the mysqldump files.

```
$ cat test.dump | mysql
```

This will bring the database back to the state it was at the last backup.

Apply the binary logs to bring the system up to date. Use the `–one-database` mysql option to filter out SQL commands that apply to other databases. You only want to apply the binary logs that were created since your last backup. For each binary log file, type

```
$ mysqlbinlog host-bin.xxx | mysql --one-database=testdb
```

> Sometimes you will need to massage the output from the mysqlbinlog program before sending it through mysql. If you are recovering from a mistaken drop table statement, for example, you will need to remove this from the output of mysqlbinlog, otherwise you'll drop the table again!

Recovering from mysqlhotcopy files

Reload the database by copying the database files from the backup location to the mysql data location. Make sure the mysql server is down when you do this. Assume for this example, the database is backed up in /var/backup/test and the mysql data location is /usr/local/mysql/data.

```
$ cp -r /var/backup/test /usr/local/mysql/data
```

This will bring the database back to the state it was at the last backup.

Copyright © 2001 O'Reilly & Associates, Inc.

Now, bring the mysql server up and apply the binary logs to bring the system up to date. Refer to the previous section for an example of how to do this.

Table Maintenance and Crash Recovery

Database tables can get out of whack when a write to the data file is not complete for some reason. This can happen due to a variety of reasons, such as a power failure or a non-graceful shutdown of the MySQL server.

MySQL provides two mechanisms for detecting for and repair table errors: `myisamchk/isamchk` and `mysqlcheck`.. It is a wise practice to perform these checks regularly. Early detection may increase your chances of successfully recovering from errors.

Mysqlcheck is new with version 3.23.38 of MySQL. The main difference between myisamchk/isamchk and mysqlcheck is that mysqlcheck allows you to check or repair tables while the server is running. Myisamchk/isamchk require that the server not be running.

Checking a table

If you suspect errors on a table, the first thing you'll want to do is use one of the utilities to check it out.

Myisamchk and isamchk are quite similar. They provide the exact same functions. The only difference is that myisamchk is used on MyISAM tables, and isamchk is used on ISAM tables.

Mysqlcheck can only be used with MyISAM tables.

You can tell what kind of table you are dealing with by looking at the extension of the data file. An extension of ".MYI" tells you it is MyISAM table and. ".ISM" indicates an ISAM table. So, myisamchk is only used with .MYI files, and isamchk with .ISM files. Simple enough?

Lets assume we have a database called test with two tables, `table1` which is an ISAM table, and `table2` which is a MyISAM table.

The first step is to check your table, using the appropriate utility. If you are using myisamchk or isamchk, make sure you MySQL server is not running to prevent the server from writing to the file while you are reading it.

```
$ myisamchk table2.MYI
Data records:       0   Deleted blocks:       0
- check file-size
- check key delete-chain
- check record delete-chain
- check index reference
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
$ isamchk table1.ISM
Checking ISAM file: table1.ISM
Data records:       0   Deleted blocks:       0
- check file-size
- check delete-chain
- check index reference
$ mysqlcheck table2.MYI
test.table2                                     OK
```

The default method is usually adequate for detecting errors. However, if no errors are reported but you still suspect damage, you can perform and extended check using the `--extend-check` option with `myisamchk/isamchk` or the `--extend option` with `mysqlcheck`. This will take a long time, but is very thorough. If the extended check doesn't report any errors, you are in good shape.

Repairing a table

If the check reported errors on a table, you can try to repair them.

If you are using myisamchk or isamchk, make sure your MySQL server is not running when you attempt the repair. Also, it is a good idea to back up the data files before attempting a repair operation, in case something goes haywire.

With myisamchk/isamchk, you want to first try the `--recover` option.

```
$ isamchk --recover table1.ISM
$ myisamchk -recover table2.MYI
```

If this fails for some reason, they you can try `--safe-recover`, a slower recovery method which can fix a some errors `--recover` cannot..

```
$ isamchk --safe-recover table1.ISM
$ myisamchk --safe-recover table2.MYI
```

With `mysqlcheck`,your only recovery option is `--repair.`

```
$ mysqlcheck --repair test table2
```

If these operations fail, your only remaining option is restore the table from your back ups and binary logs. See the section on Database Backup and Recovery for more information about this.

Scheduled Table Checking

We recommend that you take steps to perform regularly schedule table checks on your database files. This can be done by wrapping `isamchk/myisamchk/mysqlcheck` commands into a script that is executed periodically from `cron` or some other scheduling software.

You also may want to modify your system boot procedure to check tables at boot time. This is especially useful if the system is rebooting after a system crash.

Copyright © 2001 O'Reilly & Associates, Inc.

6

Performance Tuning

Introduction

Performance tuning is an important part of every significant development effort. In general, MySQL is designed with speed in mind. However, there are a number of factors that can impact application and/or database performance. The focus of this chapter will be to introduce you to some of the principles of performance tuning and some of the tools you have at your disposal.

Performance Tuning Methodology

When performance tuning a MySQL application, there are four main areas that you consider: the application, the database server, the operating system and the hardware. These should be ranked in terms of “bang for the buck.” For example, adding memory or upgrading your processor will usually improve the performance of your application(s), but you should be able to get greater gains for less cost if you tune your application code and database server first. In addition, any performance tuning on the MySQL server will apply to all applications using that server. Characteristics that are advantageous for one particular application, may not improve that performance of another. Based on these factors, as a general methodology, we recommend that you look at application tuning issues in the following order:

1. SQL Query tuning
2. Database server tuning
3. Operating system

Copyright © 2001 O’Reilly & Associates, Inc.

4. Hardware

Detailed coverage of operating system and hardware tuning are beyond the scope of this book. That is not to say that these are unimportant parts of the performance equation. However, the wide variety of Oses and hardware types make it impractical to cover these topics in adequate detail.

Application Performance Tuning

There are really two parts to applicaton performance tuning: host application tuning (i.e. C/C++, Java, Perl, etc.) and SQL query tuning. First we will look at host application considerations. Good application design and programming practices are crucial to getting good performance from you MySQL application. No amount of query tuning can make up for inefficient code. Following are few guidelines you can follow in your applications to optimize your performance:

- Normalize your database

  Elimination of redundancy from your database is critical for performance. Read Chapter 8 for more details on database design and normalization.

- Let the MySQL server do what it does well

  This seems obvious, but it is frequently overlooked. For example, if you need to retrieve a set of rows from a table, you could write a loop in you host application to retrieve the rows like this

```
for (int i = 0; i++; i< keymax) {
    … select * from foobar where key=i;
    process the row
}
```

  The problem with this approach is that MySQL has the overhead of parsing, optimizing and executing the same query multiple times. If you let MySQL retrieve all the rows at once, you let MySQL do what it is good at.

```
… select * from foobar where key < keymax;
for each row {
    process the row
}
```

- Denormalize your database where appropriate

  Sometimes performance demands require that you denormalize your database. A classic example of this is a nightly report which summarizes some information. These types of reports often require sifting of large quantities of data to produce the summaries. In this situation, you can create a "redundant" table which is updated with the latest summary information on a periodic basis. This summary table can be then be used as basis for your report.

- Use persistent connections or connection pooling if possible

  Connecting and disconnecting from the database has an overhead associated with it. In general you want to reduce the number of connections and disconnections to a minimum. In particular, this can be a problem with web applications where each

 Copyright © 2001 O'Reilly & Associates, Inc.

time a page is requested, the CGI or PHP script connects to the database to retrieve the relevant information. By using persistent connections or a connection pool, you will bypass connect/disconnect overhead, and you application will perform better.

SQL Query Tuning

The data in your database is stored as data on your disk.. Retrieving and updating data in your database is ultimately a series of disk input/output operations (I/Os). The goal of SQL query tuning is to reduce the number of I/Os to a minimum. Your main weapon for tuning your queries is the index.

In the absence of indexes on your database tables, all retrievals will require that all the data in all of the involved tables be scanned. To illustrate this, consider the following example

```
SELECT NAME FROM EMPLOYEE WHERE SSN = 999999999
```

Assume for this example that we have a table named "EMPLOYEE" with a number of columns including "NAME" and "SSN". Also, assume this table has no indexes.

WE know the SSN should be unique (that is, for each record in the table, SSN will have a unique value), and we expect to get one row in return. However, MySQL doesn't know this, and when the above query is executed MySQL has to scan the entire table to find all the records that match the where clause (SSN = 999999999). If we have a thousand rows in the EMPLOYEE table, MySQL has to read each of those rows. This operation is linear with the number of rows in the table.

What happens if we add an index on the SSN column of the EMPLOYEE table? This gives MySQL some more information. When we execute above query, it can consult the index first to find the matching SSNs. Since the index is sorted by SSN and is organized into a tree structure, it can find the matching records very quickly. After it finds the matching records, it simply has to read the NAME data for each match. This operation is logarithmic with the number of rows in the table – a signifigant improvement over the unindexed table.

Just as in this example, most MySQL query tuning boils down to a process of ensuring that you have the right indexes on your tables and that they are being used correctly by MySQL.

Index Guidelines

We've established that proper indexing of your tables is crucial to the performance of your application. On first glace, It may be tempting then to index every single column in every table of your database. After all, it should improve performance, right? Actually, indexes have some downsides too.

Copyright © 2001 O'Reilly & Associates, Inc.

Each time you write to a table (i.e. INSERT, UPDATE or DELETE) with one or more indexes, MySQL has to update each of the indexes at the same time. So each index adds overhead to all write operations. In addition, each index adds to the size of your database. You will only gain a performance benefit from an index if its column(s) are referenced in a where clause. If an index is never used, it is not worth incurring the cost of maintaining it.

With these tradeoffs in mind, here are some guidelines for index creation:

- **Try to index all columns referenced in a WHERE clause**

  As a general goal, you want any column that is referenced in a where clause to be indexed. However, this is not always true. If columns are compared or joined using the ‘<’, ‘<=’, ‘=’, ‘>=’, ‘>’ and BETWEEN operations, the index will be used. Use of a function on a column in a where clause will defeat an index on that column. So for example

  ```
  SELECT * FROM EMPLOYEE WHERE LEFT(NAME, 6) = "FOOBAR"
  ```

  would not be able to take advantage of an index on the NAME column. The LIKE operator will use an index if the there is a literal prefix in the pattern. For example

  ```
  SELECT * FROM EMPLOYEE WHERE NAME LIKE "FOOBAR%"
  ```

  would use an index, but

  ```
  SELECT * FROM EMPLOYEE WHERE NAME LIKE "%FOOBAR"
  ```

  would not.

- Use unique indexes where possible

  If you know data in an index is unique, such as a primary key or an alternate key, use a unique index. These are even more beneficial for performance than regular indexes.

- Take advantage of multi-column indexes

  Well designed multi-column indexes can reduce the total number of indexes needed. MySQL will use a left prefix of a multi-column index if applicable. Say, for example, you have an employee table with the columns first_name and last_name. If you know that last_name is always used in queries while first_name is only used sometimes, you can create a multi-column index with last_name as the first column and first_name as second column. With this kind of index, all queries with last_name or last_name and first_name in the where clause will use the index.

  Poorly designed multi-column indexes may end up either not being used at all or being used infrequently. From the example above, queries will only first_name in the where clause will NOT use the index.

  Having a strong understand of your application and the query scenarios is invaluable in determining what the right set of multi-column indexes are. Always verify your results with “EXPLAIN SELECT”(see below).

- Consider not indexing some columns

Copyright © 2001 O’Reilly & Associates, Inc.

Sometimes performing a full table scan is faster than having to read the index and the data table. This is especially true for cases where the indexed column contains a small set of evenly distributed data. The classic example of this is gender, which has two values (male and female) that are evenly split. Selecting by gender will require you to read roughly half of the rows. It might be faster to do a full table scan in this case. As always, test your application to see what works best for you.

- ALWAYS that your indexes are being used as expected using the "EXPLAIN SELECT" command on your queries. The use of EXPLAIN SELECT is detailed in the next section.

EXPLAIN SELECT

What do you do if you have a problem query and you can't see where an index is needed? And how do you verify that all your indexes are being used as expected? Are your tables being joined in the most logical order? MySQL provides a tool that will help answer these questions: the EXPLAIN SELECT command. EXPLAIN SELECT provides information about each table in your query including which indexes will be used and how the tables will be joined. The output from EXPLAIN SELECT includes the following columns.

<table>
<tr><td>Table</td><td>The database table to which the EXPLAIN SELECT output refers.</td></tr>
<tr><td>Type</td><td>The join type. Possible join types are listed below, ranked from most desirable to least desirable.
<table>
<tr><td>system</td><td>The table has only row (i.e. it is a system table). This is a special case of the const join type. See that for more information.</td></tr>
<tr><td>Const</td><td>The table has at most one matching row, so it can be read once and treated as a constant for remainder of query optimization. These are fast because they are read once.</td></tr>
<tr><td>Eq_ref</td><td>No more than one row will be read from this table for each combination of rows from the previous tables. This is used when all columns of an index are used the query and the index is UNIQUE or a PRIMARY KEY.</td></tr>
<tr><td>Ref</td><td>All matching rows will be read from this table for each combination of rows from the previous tables. This is used when an index is not UNIQUE or a PRIMARY key, or if a left subset of index columns are used in the query.</td></tr>
<tr><td>Range</td><td>Only rows that are in a given range will be retrieved from this table, using an index to select the rows.</td></tr>
<tr><td>Index</td><td>A full scan of the index will be performed for each combination of rows from the previous tables. This is the</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td><table>
<tr><td></td><td>same as an ALL join type except that only the index is scanned.</td></tr>
<tr><td>ALL</td><td>A full scan of the table will be performed for each combination of rows from the previous tables. ALL joins should be avoided by adding an index.</td></tr>
</table></td></tr>
<tr><td>Possible_keys</td><td>possible_keys lists which indexes MySQL could use tofind the rows in this table. When there are no relevant indexes, possible_keys is NULL. This indicates that you may be able to improve the performance of your query by adding an index.</td></tr>
<tr><td>Key</td><td>Key lists the actual index that MySQL choose. Key is NULL if no index was chosen.</td></tr>
<tr><td>Key_len</td><td>Key_len lists the length of the index that MySQL choose. This also indicates how many parts of a multi-column index MySQL choose to use.</td></tr>
<tr><td>Ref</td><td>Ref lists which columns or constants are used to select rows from this table.</td></tr>
<tr><td>Rows</td><td>Rows lists the number of rows that MySQL thinks it will have to examine from this table in order to execute the query.</td></tr>
<tr><td>Extra</td><td>Extra lists more information about how a query is resolved.<table>
<tr><td>Distinct</td><td>After MySQL has found the first matching row, it will stop searching in this table.</td></tr>
<tr><td>Not exists</td><td>MySQL was able to do a left join optimization of the query.</td></tr>
<tr><td>Range checked for each record (index map: #)</td><td>MySQL was not able to identify a suitable index to use. For each combination of rows from the</td></tr>
<tr><td>Using filesort</td><td>MySQL has to sort the rows before retrieving the data.</td></tr>
<tr><td>Using index</td><td>All needed information is available in the index, so MySQL doesn’t need to read any data</td></tr>
<tr><td>Using temporary</td><td>MySQL has to create a temporary table to resolve the query. This occurs if you use</td></tr>
</table></td></tr>
</table>

 Copyright © 2001 O’Reilly & Associates, Inc.

		query. This occurs if you use ORDER BY and GROUP BY on different sets of columns.
	Where used	The WHERE clause will be used to restrict the rows returned from this table.

Let's go through a detailed example to look at how to EXPLAIN SELECT to optimize a query.

Assume for this example, that we have a STATE table which includes data about all fifty of the U.S. States.

```
mysql> describe state;
+------------+----------+------+-----+---------+-------+
| Field      | Type     | Null | Key | Default | Extra |
+------------+----------+------+-----+---------+-------+
| state_id   | int(11)  |      |     | 0       |       |
| state_cd   | char(2)  |      |     |         |       |
| state_name | char(30) |      |     |         |       |
+------------+----------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Now, suppose we want to get the state name for California. For this, we would write a query like

```
select state_name from state where state_cd = 'CA';
```

> Even though SELECT queries are referred to in this section, these guidelines apply to UPDATE and DELETE statements as well. INSERT statements don't need to be optimized unless they are INSERT … SELECT statements.

Now, lets run EXPLAIN SELECT to see what we can discover about how the query will be executed.

```
mysql> explain select state_name from state where state_cd = 'CA';
+-------+------+---------------+------+---------+------+------+------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra      |
+-------+------+---------------+------+---------+------+------+------------+
| state | ALL  | NULL          | NULL |    NULL | NULL |   50 | where used |
+-------+------+---------------+------+---------+------+------+------------+
1 row in set (0.00 sec)
```

This tells us that MySQL will scan all rows in the MySQL table to satisfy the query. This is indicated by the join type of "ALL". The rows column tells us that MySQL estimates it will have to read fifty rows to satisfy the query, which is what we would expect since there are fifty states. How can we improve upon this? Since state_cd is being used in a where clause of we should put an index on it.

```
mysql> create index st_idx on state (state_cd);
.
.
mysql> explain select state_name from state where state_cd = 'CA';
+-------+------+---------------+--------+---------+-------+------+------------+
| table | type | possible_keys | key    | key_len | ref   | rows | Extra      |
+-------+------+---------------+--------+---------+-------+------+------------+
| state | ref  | st_idx        | st_idx |       2 | const |    1 | where used |
```

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

Now we can see from the key column that that MySQL has decided to use the index that we created, and that it will only read one row to satisfy the query. The only possible improvement we could make upon this would be to use a unique index instead, since we know that each state codes is unique.

```
mysql> create unique index st_idx on state (state_cd);
.
.
mysql> explain select state_name from state where state_cd = 'CA';
+-------+-------+---------------+--------+---------+-------+------+-------+
| table | type  | possible_keys | key    | key_len | ref   | rows | Extra |
+-------+-------+---------------+--------+---------+-------+------+-------+
| state | const | st_idx        | st_idx |       2 | const |    1 |       |
+-------+-------+---------------+--------+---------+-------+------+-------+
```

With the unique index in place, MySQL uses a "const" join type. We won't be able to improve upon that! Now for a more complicated example with some joins.

Suppose in addition to the STATE table, we also have a CITY table that looks like this:

```
mysql> describe city;
+-----------+----------+------+-----+---------+-------+
| Field     | Type     | Null | Key | Default | Extra |
+-----------+----------+------+-----+---------+-------+
| city_id   | int(11)  |      |     | 0       |       |
| city_name | char(30) |      |     |         |       |
| state_cd  | char(2)  |      |     |         |       |
+-----------+----------+------+-----+---------+-------+
```

Assume CITY has fifty cities for each STATE for a total of twenty-five hundred. Also, assume for this example, we're back to our original STATE table without any indexes.

Now let's ask what state San Francisco is in?

```
mysql> select state_name from state, city where city_name = "San Francisco" and
    -> state.state_cd = city.state_cd;
```

What does EXPLAIN tell us about this query?

```
mysql> explain select state_name from state, city where city_name =
    -> "San Francisco" and state.state_cd = city.state_cd;
+-------+------+---------------+------+---------+------+------+------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra      |
+-------+------+---------------+------+---------+------+------+------------+
| state | ALL  | NULL          | NULL |    NULL | NULL |   50 |            |
| city  | ALL  | NULL          | NULL |    NULL | NULL | 2500 | where used |
+-------+------+---------------+------+---------+------+------+------------+
```

The numbers in the "rows" column tell us that MySQL is going to read each row in the the state table. It will then read each of the 2500 city rows and look for a city named "San Francisco". This means that it will read a total of 125,000 (50 x 2500) rows before it can satisfy the query. This is obviously not ideal, but we should be able to improve it with some indexes. First, lets create our state_cd index.

```
mysql> create unique index st_cd on state (state_cd);
```

How does our query look with that?

```
mysql> explain select state_name from state, city where city_name =
    -> "San Francisco" and state.state_cd = city.state_cd;
+-------+--------+---------------+-------+---------+---------------+------+------------+
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
| table | type   | possible_keys | key    | key_len | ref           | rows | Extra      |
+-------+--------+---------------+--------+---------+---------------+------+------------+
| city  | ALL    | NULL          | NULL   |    NULL | NULL          | 2500 | where used |
| state | eq_ref | st_idx        | st_idx|       2 | city.state_cd |    1 | where used |
+-------+--------+---------------+--------+---------+---------------+------+------------+
```

That helps quite a bit. Now MySQL will only read one state for each city. If we add an index on the city_name column, that should do away with the "ALL" join type for the city table.

```
mysql> create index city_idx on city (city_name);
.
.
mysql> explain select state_name from state, city where city_name =
    -> "San Francisco" and state.state_cd = city.state_cd;
+-------+------+---------------+----------+---------+---------------+------+------------+
| table | type | possible_keys | key      | key_len | ref           | rows | Extra      |
+-------+------+---------------+----------+---------+---------------+------+------------+
| city  | ref  | city_idx      | city_idx |      30 | const         |    1 | where used |
| state | ref  | st_idx        | st_idx   |       2 | city.state_cd |    1 | where used |
+-------+------+---------------+----------+---------+---------------+------+------------+
```

By adding two indexes, we have gone from 125,000 rows read to two. This illustrates what a dramatic difference indexes can make. Now, what happens if we try a different query: what are all the cities in California?

```
mysql> explain select city_name from city, state where city.state_cd
    -> = state.state_cd and state.state_cd = 'CA';
+-------+------+---------------+--------+---------+-------+------+-------------------------+
| table | type | possible_keys | key    | key_len | ref   | rows | Extra                   |
+-------+------+---------------+--------+---------+-------+------+-------------------------+
| state | ref  | st_idx        | st_idx |       2 | const |    1 | where used; Using index |
| city  | ALL  | NULL          | NULL   |    NULL | NULL  | 2500 | where used              |
+-------+------+---------------+--------+---------+-------+------+-------------------------+
```

Again, we have a problem because MySQL plans to scan all 2500 cities. This is because it can't properly join on the state_cd code column with out an index in the city table. So lets add it.

```
create index city_st_idx on city (state_cd);
.
.
mysql> explain select city_name from city, state where city.state_cd
    -> = state.state_cd and state.state_cd = 'CA';
+-------+------+---------------+-------------+---------+-------+------+----------------------
--+
| table | type | possible_keys | key         | key_len | ref   | rows | Extra
|
+-------+------+---------------+-------------+---------+-------+------+----------------------
--+
| state | ref  | st_idx        | st_idx      |       2 | const |    1 | where used; Using
index |
| city  | ref  | city_st_idx   | city_st_idx |       2 | const |   49 | where used
|
+-------+------+---------------+-------------+---------+-------+------+----------------------
--+
```

With that index, MySQL only has to read roughly 50 rows in to satisfy the query. This is exactly what we would expect, since California has fifty cities in this database.

Other Options

MySQL is not always perfect when optimizing a query. Sometimes it just wont choose the index that it should. What can you do in this situation?

Isamchk/Myisamchk can help. MySQL assumes that values in an index are distributed evenly. Isamchk/Myisamchk –analyze will read a table and generate a histogram of data distribution for each column. MySQL uses this to make a more intelligent decision about what indexes to use. See Chapter 18 more information on using Isamchk/Myisamchk.

Another option is to use the USE INDEX/IGNORE INDEX in your query. This will give MySQL specific instructions about which indexes to use or not use. See the Chapter ?? for more information about this.

Tuning the MySQL Server

There are a number of settings you can tweak at the MySQL server level to influence application performance. One thing to keep in mind when tuning a server is that server behavior will affect all the applications using that server. An improvement for one application may have a detrimental effect for other applications on the same server.

There are a number of variables that can be modified in the MySQL server which may improve your performance. A full reference on these parameters can be found in Chapter 20, or by typing mysqld –help.

In general, when tuning MySQL, the two most important variables are key_buffer_size and table_cache.

- Table_cache

  Table_cache controls the size of the MySQL table cache. Increasing this parameter allows MySQL to have more tables open simultaneously without opening and closing files.

- Key_buffer_size

  Key_buffer_size controls the size of the buffer used to hold indexes. Increasing this will improve index creation and modification, and will allow MySQL to hold more index values in memory.

OS/Hardware Considerations

A full discussion of Hardware and/or OS tuning is beyond the scope of this book. However, here are a few things to consider:

- Many of the traditional hardware upgrades can help MySQL perform better. Increasing the memory in your system, gives you more to allocate to MySQL caches and buffers.

 Copyright © 2001 O'Reilly & Associates, Inc.

- Intelligently distributing your databases across multiple physical devices can also help.
- Static binaries are faster. You can configure MySQL to link statically instead of dynamically when you build it.

Copyright © 2001 O'Reilly & Associates, Inc.

Security

One of the most important aspects of both database administration and application design is also one of the most overlooked: security. While there are dozens of definitions of security that can apply to computer software, we can sum up our needs simply. Security is the ability to allow authorized users to access data while preventing access from unauthorized users. As simple as this definition is, it provides flexibility through the terms 'users' and 'access'. Different kinds of security allow different kinds of users different kinds of access.

In this chapter we will examine the three main areas where security is important with regards to MySQL: MySQL data security, server security and client security. A lack of foresight in any of these areas can open up valuable data to attackers. However, with a little preventative care, your MySQL server can safely provide data in any environment.

MySQL data security

MySQL provides several mechanisms to control access to the data of the system. Looking back at the definition of security above, we can define MySQL data security be specifying meaning of 'user' and 'access' in this context.

A 'user', to MySQL, is an authenticated connection to the MySQL server. Any time a program attempts to connect to a MySQL server it must provide credentials that identify the user. Those credentials then define the users to MySQL for that connection.

To a MySQL server, 'access' is simply access to the functions the server provides. While this usually means access to the data in the server, via SQL queries, it can also mean access to administrative functions, such as setting access-rights for other users and shutting down or reloading the server.

Protecting MySQL data is the major job of the MySQL security system. The 'data' in this context can have two seperate meanings: actual data stored in the database, and information about that data (also called meta-data).

An example of actual database data would be any information stored within an actual database table. For example, consider a database named 'mydb' that had a table 'People' with the columns 'firstName' and 'lastName'. This table has two rows with the data

Copyright © 2001 O'Reilly & Associates, Inc.

'John'/'Doe' and 'Mary'/'Smith'. The actual database data in this example is 'John', 'Doe', 'Mary' and 'Smith'. This is the content that is the reason the database exists.

Meta-data, (or data about data), is the information that describes the structure of the database content. In the example above, the fact that there is one database is a piece of meta-data. That the database is named 'mydb' is also meta-data. Other meta-data includes the fact that there is one table, named 'People', containing two columns named 'firstName' and 'lastName'. The fact that there are currently two rows of data in the table is also meta-data. At first thought it might not seem like that should be meta-data since it is directly dependant on the actual data content. However, even though it is dependant on the data content, it is not actually the data content. Therefore it is meta-data.

The MySQL security schema protects both content data and meta-data. This is important because a system that protected only the data of the system would still be open for abuse. Consider an attacker that gained access to the meta-data in the above example. Simple changing the table name from 'People' to 'ConvictedCriminals' could make quite a change in the meaning of the data, even though the data has not been touched.

MySQL protects its data (and meta-data) through the use of special set of tables called 'grant tables'. These grant tables reside in the the 'mysql' database, which is a special system database that exists on every MySQL server. MySQL provides two ways of interacting with the grant tables: directly and through SQL queries.

We will cover using SQL queries to interact with the grant tables first, and then move into directly manipulating the tables. For many users the SQL interface will be all that is needed and the details of the underlying tables can be skimmed or skipped altogether. However, if you have every had any problems setting up MySQL security, of if you have complex security needs, interfacing directly with the grant tables may be necessary.

SQL Interface

Standard ANSI SQL provides two statements specifically designed for managing access to the database server: GRANT and REVOKE. MySQL supports both of these statements, with several semantic extensions that allow control over the meta-data of the system as well as the data itself.

The GRANT statement is used to provide a user access to functionality on the MySQL server. Conversely, the REVOKE statement removes access for a user. Both statements are executed the same way as other SQL statements such as SELECT, INSERT, etc.

GRANT

```
GRANT privilege [(columns)] [, privilege [(columns)] ...] ON table(s)
    TO user [IDENTIFIED BY 'password']
    [, user_name [IDENTIFIED BY 'password'] ...] [WITH GRANT OPTION]
```

The GRANT statement can be broken into three sections: what is being granted, where the grant takes effect, and who is being granted the privilege. This can be seen most

 Copyright © 2001 O'Reilly & Associates, Inc.

clearly by looking at the structure of GRANT when all of the optional attributes are remove:

```
GRANT privilege ON table(s) TO user
```

All other information given simply refines the 'what', 'where' and 'who' given in this simple format.

What

The type of privilege granted determines what abilities the user will have as a result of the GRANT statment. There are 15 privileges currently defined in MySQL:

ALL PRIVILEGES
: Despite its name, this does not grant all privileges to the user. It does grant complete control over the data, and databases, within the MySQL server. This privilege includes the following privilges: ALTER, CREATE, DELETE, DROP, INDEX, INSERT, REFERENCES, SELECT and UPDATE. It does not automatically grant FILE, PROCESS, RELOAD and SHUTDOWN. Those privileges effect the raw system state the MySQL server and must be granted explicitly. The privilege 'ALL' is provided as a synonym for 'ALL PRIVILEGES'.

ALTER
: This provides the ability to alter the structure of an existing table. In particular it allows the user to execute the ALTER SQL statement for any purpose that does not involve table indexes (see INDEX, below).

CREATE
: This provides the ability to create new databases and/or tables. In particular it allows the user to execute the CREATE SQL statement.

DELETE
: This provides the ability to delete data from a table. Note that this does not grant the ability to delete the actual tables or databases (which are meta-data), only the data itself. Specifically, this allows the users to execute the DELETE SQL statement.

DROP
: This provides the ability to remove entire tables and/or databases. Conversely to DELETE, this does not provide the ability to remove specific elements of data from the tables, only to erase the entire table or database. This grants the user the ability to execute the DROP SQL statement.

FILE
: The allows the user to (directly or indirectly) read, write, modify or delete any operating system level file on the server with the same privileges as the MySQL server process. The purpose of this privilege is to allow users to use the LOAD DATA INFILE and SELECT INTO OUTFILE SQL statements to read and write server-side data files.

: However, as stated above, a side effect of this privilege is that the user is trusted with the same operating system level rights as the MySQL server for accessing files. See the 'Server Security' section below for tips on how to secure the server against abuse of this privilege.

Copyright © 2001 O'Reilly & Associates, Inc.

INDEX
: This allows the user to create, alter and/or drop indexes on a table. This allows the user to execute the ALTER SQL statement only with regards to indexes.

INSERT
: This allows the user to insert data into a database table. In particular it allows the user to execute the INSERT SQL statement.

PROCESS
: This provides the user with the ability to view the list of MySQL process threads as well as the ability to kill any of the threads. MySQL process threads exist for each connection to the server. In addition, serveral utility threads exist to for overall server functionality. Therefore this priviliege should be careful granted as it can be used to arbitrarily terminate client connections or shutdown the entire MySQL server. This privilege allows the user to execute the SHOW PROCESSLIST and KILL SQL statements.

REFERENCES
: This privilege currently does not do anything. It is provided for SQL compatibility with servers such as Oracle that provide "foreign key" functionality.

RELOAD
: This provides the user with the ability to make the MySQL server reload information it keeps cached, such as privilege information, log files, table data, etc. In particular it allows the user to execute the FLUSH SQL statement.

SELECT
: This allows the user to read data from a database table. Specifically, it allows the user to execute the SELECT SQL statement.

SHUTDOWN
: This allows the user to completely shutdown the running MySQL server. This privilege should be granted carefully for obvious reasons.

UPDATE
: This allows the user to modify data within a database table. This does not grant the ability to add new data or remove old data, but only to change existing data. Specifically, it allows the user to execute the UPDATE SQL statement.

USAGE
: This provides the user with no privileges whatsoever. It can be used to create a user who can do nothing but connect to the server.

Multiple privileges can be specified in a single GRANT statement. In addition, there is one final privilege, GRANT, that cannot be specified with the rest of the privileges. Instead, the clause `'WITH GRANT OPTION'` must be included at the end of the GRANT statement. If this is done, any users given privileges in the statement will be able to re-GRANT those privileges to any other user.

This is a very powerful ability and should only be given to trusted users. Because of the nature of the MySQL grant system, the ability to grant new privileges is not limited to those granted initially. That is, if a user is given the GRANT ability during a GRANT statement, then the user is later given new privileges without specifying `'WITH GRANT`

Copyright © 2001 O'Reilly & Associates, Inc.

OPTION', the user will still be able to re-GRANT those new privileges. Once a user has the ability to grant a privilege, they can grant any privilege they have at any time.

Examples

```
/* Note that these are incomplete SQL statements.
   They simply illustrate the format of the first
   section of the GRANT statement */
GRANT SELECT                  -- The user can execute SELECT statements
GRANT ALL PRIVILEGES          -- The user has all privileges except for
                              -- the system-wide privileges
GRANT INSERT, UPDATE, DELETE  -- The user can execute INSERT, UPDATE
                              -- or DELETE SQL queries
GRANT SHUTDOWN                -- The user can shutdown the server
GRANT CREATE, DROP ... WITH GRANT OPTION -- The user can execute CREATE
          -- and DROP statements. In addition, the user can execute
          -- the GRANT statement and re-GRANT the CREATE and DROP
          -- privileges, as well as any other privilege the user
          -- already has. Furthermore, if the user is given new privileges, they
          -- will automatically be able to re-GRANT those privileges as well.
```

Where

Some of the privileges discussed above apply only in very specific contexts. For instance, the SHUTDOWN privilege only has meaning when shutting down the entire MySQL server. However, most of the privileges can apply in a number of places. The CREATE privilege, for example, could apply to creating a new database or a new table. Privileges can apply to the entire server, specific databases, table and even individual columns within a table. Table XX-1 shows which privileges apply in which contexts.

Privilege	Column	Table	Database	Server
ALTER		X		
CREATE		X	X	
DELETE		X		
DROP		X	X	
GRANT		X	X	X
FILE				X
INDEX		X		
INSERT	X	X		
PROCESS				X
RELOAD				X

Copyright © 2001 O'Reilly & Associates, Inc.

SELECT	X	X		
SHUTDOWN				X
UPDATE	X	X		

Every privilege granted must be granted at a specific location. For table, database and server-wide privileges, the location is specified in the ON clause of the GRANT statement. The ON clause can take several different location formats:

table_name
: This will grant the privilege for the given table name within the currently active database.

database.*
: This will grant the privilege for all tables within the given database

.
: This will grant the privilege for all tables within all databases. In other words, the privilege will be granted globally. This should be used when granting a server-level privilege such as SHUTDOWN.

*
: If there is a currently active database selected, this will grant the privilege on all tables within that database (the same as database.* using the active database). If no database is selected, this will grant the privilege globally (the same as *.*)

The one type of location that is not specified in the ON clause are column-level privileges. Column-level privileges should have the table (or tables, using the syntax above) specified in the ON clause as usual. The specific columns within the table(s) are specified after the individual privileges, in parenthesis, seperated by commas. Those privileges will this only take effect for those specific columns within the tables given in the ON clause.

Examples

```
/* As before, these are not complete GRANT statements, but only fragements
illustrating the
    portions covered so far *.
GRANT SHUTDOWN ON *.*   -- Grant SHUTDOWN globally. Since SHUTDOWN is a
                    -- server-wide privilege, this is the only context where it
                    -- makes sense.
GRANT SHUTDOWN ON * -- Same as above *if* no database is currently selected.
                    -- If a database is selected, this makes no sense as
                    -- SHUTDOWN cannot be granted for a database or it's tables.
GRANT CREATE ON mydb.*  -- Allow the user to create new tables within
                    -- the 'mydb' database.
GRANT CREATE ON *.* -- Allow the user to create new tables for any database.
                    -- In addition, allow the user to create new databases.
GRANT DROP ON mydb.* -- Allow the user to drop tables within the
                    --'mydb' database. This does not give the user permission
                    -- to drop the mydb database itself, the user can drop
                    -- every table within in, which is as effective.
GRANT DROP ON *.*   -- Allow the user to drop any table and/or any database
GRANT INSERT ON mydb.People -- Allow the user to insert new rows into
                    -- the mydb.People table.
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
GRANT SELECT (firstName, lastName) ON mydb.People -- Allow the user to execute
                    -- SELECT statements on the mydb.People as long as the
                    -- statements only read data from the 'firstName' and
                    -- 'lastName' columns.
GRANT SELECT (firstName, lastName, phone),
         INSERT (firstName, lastName), UPDATE on mydb.People
                    -- Allow the user to execute INSERT, UPDATE and SELECT
                    -- statement on the mydb.People table. The user can only
                    -- use SELECT when reading data from the firstName,
                    -- lastName or phone columns. The user can only use
                    -- INSERT to insert rows containing only firstName and
                    -- lastName data. The user can use UPDATE to change the
                    -- value of any columns in any of the rows of existing data.
```

Who

Now that we know what we are granted and where it will apply, the last step is to specify who we are granting the privileges to. This is specified by the TO clause of the grant statement. The format of the TO clause is a list of users, with optional passwords (given after the `'IDENTIFIED BY'` clause), separated by commas.

Unlike most other database servers, the format of the user within MySQL includes not only a username, but the user location as well. A valid MySQL username is any string of characters that is 16 characters or less. Any characters can be used (this allows usernames to be written in non-ASCII languages), however, standard ASCII characters are recommended as some clients cannot handle other character sets. However, if you know for sure your clients can handle the characters, there is no reason to stick to standard ASCII. If the user string contains any characters other than the standard ASCII alphanumerics it must be enclosed in quotes (either single or double quotes will do).

Note: When performing authentication, the MySQL server performs a case-insensitive check on the username. That means that if your username is defined as 'myuser', you can use 'MYUSER', 'MyUsEr', 'myuser' or any other permutation to log in. This also means that if you grant a privilege to 'myuser' and another privilege to 'MYUSER' you are really granting two privileges to the same user.

The location can be specified as a fully qualified domain name (my.server.com) or as IP addresses in standard dotted decimal notation (10.20.30.40). In addition, the SQL wildcards '%' and '_' can be used to specify a range of addresses. If a wildcard is used (or any character other than the standard ASCII alphanumerics), the location must be enclosed in quotes (single or double). In addition, if the numeric IP format is used, a netmask (also in dotted decimal notation) can be supplied after a '/' character. The netmask will be applied to any connecting user before checking it against the IP address.

For convenience, a username maybe provided without a location (and within the '@' symbol). This is equivalent to user@"%", which specified the username coming from any location.

Every user within the MySQL security system has a password. The password must be specified along with the username whenever the user connects to the MySQL server. When creating a new user using a GRANT statement, the password of the user can be specified by using the `'IDENTIFIED BY'` clause after the username/location in the TO

Copyright © 2001 O'Reilly & Associates, Inc.

clause. The password can be any number of characters, and like the username can contain any characters. If the password contains anything except the standard ASCII alphanumerics (and a good password should), it must be enclosed in quotes (single or double).

If the GRANT statement is creating a new user, and no `IDENTIFIED BY` clause is provided, the user will be created with a blank password. This creates an immediate security hole and should never be done. If the GRANT statement is modifying the privileges of an existing user, an `IDENTIFIED BY` clause will change the password of that user, and no `IDENTIFIED BY` clause will leave the user's password unchanged.

Note: When MySQL is first installed it creates a couple of default users. The first is the 'root' user. This user is the only user that initially has the ability to GRANT other privileges. In fact, the 'root' user is given all privileges and can do anything to the server. In the default installation, the root user is enabled only for localhost and can not be accessed remotely. In addition, the default user is given a blank password. This should be changed immediately after installing MySQL as anyone (on the server machine) could take complete control of your MySQL server while root has a blank password.

Beyond the root user, MySQL also creates default permissions for any user connecting via localhost. By default, MySQL forbids everything for these users. They will be able to connect to the MySQL server, but not execute any operations. All other users (that is, any user connecting remotely) are not even allowed to connect to the MySQL server by default.

Examples

```
/* Now that we've seen all of the parts of the GRANT syntax,
   we can look at some complete GRANT statements... */
GRANT ALL ON *.* TO super@"%" -- Give all privileges
               -- (except for system-wide functions) to the user 'super',
               -- when connecting from any location. If 'super' did not
               -- already exist as a user, it will be created without a
               -- password. Consider that this user has been granted
               -- complete control over all of the data on the server
               -- this would not be a good idea.
GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.People TO 'people_user'@localhost
               -- Give the ability to read, write, modify and delete
               -- information in the People table of the 'mydb' database to
               -- the user 'people_user', only if they are connecting from the
               -- same machine as the server. As in the previous examine, if
               -- 'people_user' did not already exist, it would be created
               -- with no password; a bad idea.
GRANT PROCESS, RELOAD, SHUTDOWN ON *.* TO admin@localhost IDENTIFIED BY 'pass'
               -- Give the ability to execute server-wide functions,
               -- such as killing processes, reloading cached data and
               -- shutting down the server to the user 'admin' when connected
               -- from the local machine. The password for the user is set
               -- to 'pass'.
GRANT SELECT, UPDATE(firstName, lastName) ON *
        TO joe@'%.server.com' IDENTIFIED BY 'mypass'
               -- If a current database is selected, this statement grants
               -- the ability to read data from any table in that database,
               -- as well as update any columns named 'firstName' or
               -- 'lastName' in those tables. If there is no current database,
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
                -- this statement grants those privileges on all tables in all
                -- databases in the server. Whichever is the case, the rights
                -- are granted to the user 'joe' when connected from any
                -- hostname that ends with '.server.com'. The password for the
                -- user 'joe' is set to 'mypass'.
GRANT USAGE ON *.* TO guest, dummy@"%.%" IDENTIFIED BY 'password'
                -- This grants only the ability to connect the server,
                -- no other functionality is allowed. This takes effect for
                -- the user 'guest' when connected from any location, as well as
                -- the user 'dummy' when connected from any location that
                -- contains a '.' (this eliminates the localhost). If the user
                -- 'guest' is being created, it is given a blank password. The
                -- password for the user 'dummy' is set to 'password'.
GRANT SELECT ON *.* TO joe@'10.0.0.0/255.0.0.0' -- Grant the ability to
                -- read any table in any database on the server to the user
                -- 'joe' that is connecting from any IP address starting with
                -- '10.' This is because the netmask 255.0.0.0 is applied to
                -- connecting address first, leaving only the first segment,
                -- which must match '10'.
```

REVOKE

The REVOKE statement has a structure virtually identical to GRANT.

```
REVOKE privilege [(columns)] [, privilege [(columns)] ...] ON table(s) FROM user [,
user_name ...]
```

Just as with GRANT, a 'what', 'where' and 'who' must be specified that details what privilege will be revoked from which user. All of the options and syntax of REVOKE is identical to GRANT with a couple of exceptions.

- To revoke the ability to grant privileges simple use 'GRANT' as the name of the privilege to revoke (this replaces the 'WITH GRANT OPTION' clause in the GRANT statement).
- The privilege 'ALL' actually refers to all privileges in this case, as opposed to GRANT, where 'ALL' really meant 'most of the privileges except for the really dangerous ones.' Using REVOKE with the ALL privilege will reduce a user to the level of the USAGE privilege. They will be able to connect to the server, but not perform any actions.
- There is no 'IDENTIFIED BY' clause within the 'TO' clause of REVOKE.

For each user given, the specified privileges will be immediately removed.

Examples

```
REVOKE ALL ON *.* FROM olduser -- Removes all privileges from the user 'olduser'
                        -- when connected from any location
REVOKE CREATE, DROP ON mydb.* FROM anotheruser@"%.server.com"
                        -- Removes the ability to create and drop tables in the
'mydb' database from the user
                        -- 'anotheruser' when connected from any location
                        -- ending in '.server.com' users names 'anotheruser'
                        -- connecting from any other location are not affected
                        -- by this statement.
REVOKE INSERT (phone) ON mydb.People FROM user@localhost
                        -- Removes the ability of the user to insert data
                        -- into the People table of the 'mydb' database that
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
                        -- contains data for the 'ohone' column. If the user
                        -- had INSERT privileges for any other columns of that
                        -- table, they can still insert new rows, as long
                        -- they do not give any data for the 'phone' column.
                        -- This affects the user 'user' when connected from the
                        --  same machine as the server.
REVOKE GRANT ON *.* FROM olduser -- Removes the ability to grant new
                        -- privileges from the user 'olduser' when connected
                        -- from any location.
```

Direct Interface

The SQL GRANT and REVOKE commands described above give very complete access to the MySQL security mechanism. However, sometimes it is necessary to fine tune the security settings by going directly to the security tables used internally by MySQL to store the policies. These tables are present in every MySQL server and are installed by default when the server is first set up.

While examining the GRANT SQL statement, we saw that MySQL privileges fall into four contexts: server-wide, database, table and column. Furthermore, a MySQL user contains information about both the username and the location of the user. All of this information is stored internally by MySQL in five tables within the 'mysql' database.

user - The 'main' privilege table and contains the username, user location and global privileges.

db - Database-level privileges for specific databases.

host - Location (hostname) level privileges for specific databases

tables_priv - Table level privileges for specific tables within databases

coumn_priv - Column level privileges for specific columns within tables

User

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Password        | char(16) binary |      |     |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Reload_priv     | enum('N','Y')   |      |     | N       |       |
| Shutdown_priv   | enum('N','Y')   |      |     | N       |       |
| Process_priv    | enum('N','Y')   |      |     | N       |       |
| File_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
```

 Copyright © 2001 O'Reilly & Associates, Inc.

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

The primary key if the 'user' table is a joint key including both the Host field (the location of the user) and the User field (the username of the user). This means that users within MySQL are defined by from where they connect. The user 'joe' connecting from localhost is not the same user as the user 'joe' connecting from 'my.server.com'. The Host field can contain SQL Wildcards ('%' and '_') to indicate multiple hosts.

Also contained in the 'user' table is the Password field which is a scrambled version of the password of the user. It is important to note that this is not an encrypted version of the password, as in other security systems such as Unix logins. The password here is merely scrambled and the original password can be recovered from the scrambled version.

The other fields of this table are each of the possible privileges. A value of 'Y' or 'N' in these fields determine whether the user is granted that privilege or not.

db

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The primary key of the db table is a joint key containing the Host (the location of the user), the Db field (the database) and the User field (the username of the user). The Host and Db fields can contain SQL wildcards ('%' and '_') to indicate multiple values.

The other field sof this table are each of the possible privileges applicable to databases. A value of 'Y' or 'N' in these fields determine whether the user is granted that privilege or not.

host

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
```

```
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The primary key of host table is a joint key containing both the Host field (the location of the user) and the Db field (the database). Both the Host and Db fields can contain SQL wildcards '%' and '_' to indicate a range of options.

The other field sof this table are each of the possible privileges applicable to databases. A value of 'Y' or 'N' in these fields determine whether the user is granted that privilege or not.

tables_priv

```
+-------------+---------------------------------+------+-----+---------+-------+
| Field       | Type                            | Null | Key | Default | Extra |
+-------------+---------------------------------+------+-----+---------+-------+
| Host        | char(60) binary                 |      | PRI |         |       |
| Db          | char(64) binary                 |      | PRI |         |       |
| User        | char(16) binary                 |      | PRI |         |       |
| Table_name  | char(60) binary                 |      | PRI |         |       |
| Grantor     | char(77)                        |      | MUL |         |       |
| Timestamp   | timestamp(14)                   | YES  |     | NULL    |       |
| Table_priv  | set(...)                        |      |     |         |       |
| Column_priv | set(...)                        |      |     |         |       |
+-------------+---------------------------------+------+-----+---------+-------+
```

The primary key of the 'tables_priv' table is a joint key containing the Host (location) field, Db (database) field, User (username) field and Table_name field. The Host and Db field can contain SQL wildcards ('%' and '_') to indicate multiple values. The Table_name field can contain the '*' wildcard to indicate every table within a database.

The table also contains a Grantor field that contains the name of the user that granted this particular privilege and a Timestamp that contains the time the privilege was created or last modified.

The final two columns of this table are 'Table_priv' and 'Column_priv'. The Table_priv column contains a set of privileges that are applicable to the table as a whole. The Column_priv column contains a set of privileges that are application to individual columns.

columns_priv

```
+-------------+---------------------------------+------+-----+---------+-------+
| Field       | Type                            | Null | Key | Default | Extra |
+-------------+---------------------------------+------+-----+---------+-------+
| Host        | char(60) binary                 |      | PRI |         |       |
| Db          | char(64) binary                 |      | PRI |         |       |
| User        | char(16) binary                 |      | PRI |         |       |
| Table_name  | char(64) binary                 |      | PRI |         |       |
| Column_name | char(64) binary                 |      | PRI |         |       |
| Timestamp   | timestamp(14)                   | YES  |     | NULL    |       |
| Column_priv | set(...)                        |      |     |         |       |
```

 Copyright © 2001 O'Reilly & Associates, Inc.

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

The primary key of the ‘columns_priv’ table is a joint key containing the Host (location) field, Db (database) field, User (username) field and Table_name field. The Host and Db field can contain SQL wildcards (‘%’ and ‘_’) to indicate multiple values. The Table_name field can contain the ‘*’ wildcard to indicate every table within a database.

The table also contains a Timestamp column that indicates the time the privilege was created or last modified. The Column_priv column contains a set of privileges that are application to individual columns.

These tables are consulted at two times during a session between the MySQL server and a client: during the initial connection and whenever a query is executed.

Initial Connection

Whenever a client attempts to connect to a MySQL server, the server consults the data in the ‘user’ table to determine whether is allowed to connect. The connecting user must have a location and username that matches an entry in that table.

You may recall that the values of the Host column can contain SQL wildcards. Given this, it is a distinct possibility that more than one row in the ‘user’ table may apply to a connecting user. The MySQL server will always use only one row to determine the access rights of a user. It decides which row to use by the following algorithm:

- More specific values for the ‘Host’ column are considered by less specific values. That is, host values that contain no wildcards are considered first, following by values that contain wildcards mixed with characters, with a singe ‘%’ that matches anything considered last.
- Rows that contain the same value for Host are considered by their ‘User’ value. Values of user that are not blank are considered before a blank User. Therefore a blank User is a ‘default’ that is used if no other user name matches.

Consider the following Host/User pairs:

```
root       localhost
joe        localhost
           localhost
joe        "%"
jan        "%.server.com"
mary       "%"
```

Using the basic principle of ‘most specific first’, various connection outcomes are possible.

- ‘root’ connecting from localhost - Matches the first line ‘root'/'localhost’ since both are specific.
- ‘joe’ connecting from localhost - Matches the second line ‘joe'/'localhost’ since both are specific.

- 'jane' connecting from localhost - Matches the third line ''/'localhost' since the host is specific and there is no matching user
- 'joe' connecting from 'my.server.com' - Matches the fourth line 'joe'/"%" since no specific host matches 'my.server.com' with a user 'joe', but the unspecific "%" has a user 'joe'
- 'mary' connecting from localhost - Matches the third line ''/'localhost' since the host is specific and there is no matching user. This is probably the most common mistake in MySQL access configuration. Intuitively, the line 'mary/'%" would be used since it specifies the the username 'mary'. However, Host is checked before User and a perfectly matching host (localhost) with no user takes precedence over an unspecific host ("%") with a specific.
- 'root' connecting from 'my.server.com' - No line that matches 'my.server.com' has a user that matches 'root'. Connection is denied.

If a user does not match any of the rows within the user table, the connection is rejected. If a match is made, the password is checked against the password supplied by the client. If the password matches the connection is allowed.

Query Execution

Once a client is allowed to connect to the MySQL database server, the next stage where security is checked is whenever the client attempts to execute a query or execute some function of the server. This stage of security involves all of the security tables.

The first stage of security checked here is the same 'user' table checked when the client first connected. Whichever row of this table was used to allow the user to connect also contains the 'global' rights for the user. Any privileges that are granted at this level apply to every database, table and column on the server. If the user has the necessary privilege at this level, the permission is granted and no further check is made.

If the global privileges did not grant sufficient permissions for the operation, the MySQL server then checks database-level privileges. The 'db' table is checked for the name and location of the user and the name of the database involved in the query. Just as in the case of the 'user' table, a 'most specific first' rule is used in the 'db' table. If a row of this table matches the username, host and database name, the privileges granted in that row are used.

If there is no match of username, host and database in the 'db' table, the server then looks for a row that matches the username and database but have a blank host column. If such a row exists, the server then moves the 'host' table, which can be considered an extension of the 'db' table. Within the host table, it is possible to have multiple rows for each database, by specifying different hosts. In this manner, it is possible to create rules that take effect for some locations but not for others.

The server looks to see if there is a row in the host table that has a matching location and database. If such a row is found, the user is granted any privileges that are both in this row and the corresponding row of the 'db' table. This is an important point, as a privilege that

 Copyright © 2001 O'Reilly & Associates, Inc.

is not in both places will not be granted. This allows rule to be set up where a privilege is granted to most people (using the row in the 'db' table with the blank host field) but selectively denied for certain locations (but including a row in the 'hosts' table that does not have that privilege).

If the 'db' and 'host' tables have been resulted in sufficient privileges, the query is executed. If the privilege is still not suffecient and the query is one that only effects the database, but not any tables (such as a DROP database query), then the query is forbidden. If the query does involve accessing a table, the server then moves to the 'tables_priv' table.

In the tables_priv table, the username, location, database and table name are all checked. As in the previous cases, a 'most specific first' rule is used when multiple rows match. If a matching row is found, the 'table_priv' column is checked to see if the required privileges exist. If so, the query is executed. If not, the 'column_priv' column is check to see if the required privileges are present. If they are also not there, the query is denied. If the privileges do exist in the 'column_priv' column, the server moves to the 'columns_priv' table for a final check.

When 'columns_priv' is used, each of the columns accessed in the query are checked. The username, location, database, table and column name must have a match in this table for each column used in the query. If the columns all have a match with the sufficient privileges the query is executed. If any of the columns do not have a match, or if any of the matches do not grant sufficient privileges, the query is denied.

Consider the following hypothetical excerpts from each of the privilege tables:

```
'user':
Host              User   Select_priv    Insert_priv
localhost         root       Y               Y
localhost                    N               N
"%"               joe        Y               N
localhost         joe        Y               Y
"%"               mary       Y               N
localhost         jane       N               N
"%.server.com"    john       N               N
"%"               jill       N               N

'db':
Host              Db         User         Select_priv   Insert_priv
localhost         mydb       mary             Y              N
"%.server.com"    "%"        joe              Y              Y
                  mydb       john             Y              Y

'host':
Host              Db       Select_priv   Insert_priv
localhost         mydb         Y              Y
"%"               mydb         Y              N
"%"               "%"          N              N

Tabes_priv:
Host              User      Db     Table       Table_priv         Column_priv
"%.server.com"    john      mydb   People      'Select,Insert'
localhost         jim       mydb   People      'Select'           'Insert'
```

```
columns_priv:
Host            User      Db       Tables_name   Column_name   Column_priv
localhost       jim       mydb     People        firstName     'Insert'
localhost       jim       mydb     People        lastName      'Insert'
```

Given the above security information, lets look at the outcome of a couple of SQL queries from various users:

```
'SELECT * from mydb.People'
```

- 'root' connecting from 'localhost': Succeeds -- An exact match in the 'user' table gives 'root' global Select privileges.
- 'mary' connecting from 'my.server.com': Succeeds -- A match against the wildcard "%" for host and the exact username 'mary' in the 'user' tables gives global Select privileges to 'mary'
- 'mary' connecting from 'localhost': Succeeds -- In 'user' an exact match for 'localhost' and a default (blank) user, give 'mary' no permission to Select. Therefore 'db' is consulted next, where an exact match for 'localhost' and 'mary' for the database 'mydb' results in the Select privilege for 'mary'.
- 'john' connecting from 'my.server.com': Succeeds -- In user, a wildcard match for '%.server.com' an the exact username 'john' result in no permission to Select. The 'db' table is then consulted, where no match is found for the host, db and user. However. a match is found for the db 'mydb' and user 'john' with a blank host field. Therefore, the 'host' table is consulted. There, a wildcard matches the host, with an exact match for the db 'mydb', resulting in Select privilege for 'john'.
- 'jim' connecting from 'localhost': Succeeds -- In 'user', an exact match for the host 'localhost' and the default user results in no Select privilege, so we move to 'db'. There, no match is found for the db, user and host, or for the db, user and a blank host. Therefore the 'tables_priv' table is used next. There, an exact match is found for the host, user, database and table in question resulting in the Select privilege for 'jim'.
- 'jill' connecting from 'my.server.com': Fails -- In 'user', a match is found for the wilcard host and the exact username 'jill', resulting in no Select privilege. The 'db' table is used next, where no match is found for the db, user and host; or for the db, user and a blank host. The 'tables_priv' column is consulted, resulting again in no match for the db, user, host and table. Therefore, the 'N' privilege in the 'user' table for 'jill' stands and the query is not executed.

```
'INSERT INTO mydb.People (firstName, lastName) VALUES ('john', 'doe')
```

- 'root' connecting from 'localhost': Succeeds -- An exact match in the 'user' table gives 'root' global Insert privileges.
- 'mary' connecting from 'my.server.com': Fails -- A match against the wildcard '%' and the exact username 'mary' result in no Insert privilege for 'mary'. The 'db' table is then consulted where no match is found the the host, db and user (or for the db and user and a blank host). Finally, the 'tables_priv' table is used where again no match is found for the host, db, user and table. This causes the 'N' Insert privilege from the 'user' table to stand and the query is not executed.

Copyright © 2001 O'Reilly & Associates, Inc.

- ‘mary’ connecting from ‘localhost’: Fails -- A match against the exact host ‘localhost’ and default user gives no Insert privilege. The ‘db’ table is used next, where an exact match against the host ‘localhost’ and the user ‘mary’ against results in no Insert privilege. The ‘tables_priv’ table is checked last where no match is found for the host, db, user and table. Therefore the ‘N’ Insert privilege from the ‘db’ table stands and the query is not executed.
- ‘john’ connecting from ‘my.server.com’: Succeeds -- A match against the wildcard host ‘%.server.com’ and the exact username gives no Insert privilege in the ‘user’ table. In the ‘db’ table, no match is found for the username, host and db, but a match is found for the username, db and a blank host, which causes the ‘host’ table to be used. There, a match again the wildcard host and the exact db gives no Insert privilege. The ‘tables_priv’ is checked next where a match against the wilcard host ‘%.server.com’, and an exact username, database and table result in the Insert privilege for ‘john’ and the query is executed.
- ‘jim’ connecting from ‘localhost’: Succeeds -- A match again the exact host and the default user gives no Insert privilege in the ‘user’ table. The ‘db’ table does not match the user, host and database (or the user, database and a blank host). Therefore the ‘tables_priv’ table is used, where an exact match for the username, host, database and table result in no table-wide Insert privilege, but rather a column-specific Insert privilege. This causes the ‘columns_priv’ table to be used. There, an exact match in for the username, host, database, table and both columns used in the query cause the permission to be given and the query is executed.

Server Security

While controlling access to data within MySQL is the probably the most important aspect of security with regards to MySQL, it is not the whole story. For as good as your MySQL security rules are, they can be bypassed if someone can gain access to the actual files that MySQL uses to store the data. Protecting these files, as well as protecting unwanted network access the MySQL server falls under the real of server security.

In this context, a ‘user’ is any operating system-level user on the same machine as the MySQL server. Anybody with network access to the MySQL server machine is also considered a user in this context.

The concept of ‘access’ in this context is the ability to read or write the operating system-level files used by MySQL. Also, any network connection to the MySQL server is considered access in this context.

The definition of ‘access’ we are using for server-side security involves two distinct concepts that we will deal with individually: operating-system security and network security

Copyright © 2001 O’Reilly & Associates, Inc.

Operating System Secuity

The data within a MySQL database server is only as secure as the files that contain that data. The most well designed access-rights schema will mean nothing if a user can copy the database files to another machine with different access-rights where the data can be accessed.

Therefore, any files containing data used by MySQL must be inaccessible by a normal user. However, we cannot simply make all MySQL-related files forbidden. Most of the MySQL files, such as the executable binaries and configuration files must be readable by normal users for the normal operation of MySQL.

This creates a situation where a specific set of files must be protected, while other related files must be made accessible. This situation can be resolved through proper installation and configuration of MySQL:

- MySQL data files should be in a separate directory

Whatever directory schema is used for the rest of the MySQL installation, the data files themselves should be kept in their own directory. The default MySQL installation does this by creating a 'var' directory to store the database (or optionally using a system-wide /usr/var/mysql). This allows the data files to have a seperate security setting than the rest of the MySQL installation

- The MySQL server should run as a special user and group

Since any user who has access to the MySQL data files has access to the MySQL data, the MySQL server should be run as a special user, created just for MySQL. The default MySQL installation does this by creating a 'mysql' user and a 'mysql' group. This user and this group have full access to the MySQL data and should never be used for any other purpose than running the MySQL server.

- The permissions on the data directory should be properly set

If the previous two precautions have been taken, we have a special directory just for the MySQL data files and a special user and group just for running MySQL. The last step is to make sure only the special MySQL user has access to to the MySQL data directory. This is done by setting the proper permissions for the server operating system.

On a Unix system, the data directory and all of the files underneath it should be owned by the MySQL user and group. The 'owner' read, write and execute permission should be set on the data directory and the 'owner' read and write should be set on each of the data files. No other permissions should be allowed. The default MySQL installation uses these permissions.

On a Windows system it is possible to perform this same type of security set up using user and groups. However, if the Windows is using any other filesystem except for NTFS (e.g. FAT or FAT32), any user will still be able to read the data files. In addition, MySQL on

 Copyright © 2001 O'Reilly & Associates, Inc.

Windows currently does not create the users and groups automatically, as it does on Unix. If you are installing on Windows, you must manually set up this scheme yourself.

Once a MySQL server has been secured using these steps, a local user on the same machine as the server will not be able to access the MySQL data files.

Network Security

If your MySQL server is intended only for use by local clients, network security is easy. However, most MySQL servers are used for applications that require network access. When this is the case it is important to take the proper steps to ensure network safety.

There are three main dangers to a MySQL server that involve network security. The first is an attack that would directly compromise the MySQL server itself. The second is an attack that would allow an unauthorized user to access the MySQL server. The third is an attack that would prevent the MySQL server from performing its duties.

Direct Compromise

The first form of attack is the most dangerous, but the least likely. This type of attack would grant the intruder complete control over the MySQL server process itself. Most commonly in these type of attacks, the intruder uses the process to gain further entry into the server machine, ultimately leading to root access. For this type of attack to succeed there would have to be some sort of flaw with either the MySQL network protocol or the MySQL server code itself.

The MySQL network protocol is an open protocol that can be examined by any interested party. Although the protocol currently has very little documentation, the source code is easy to follow. In addition, there have been several widely-used applications that use the protocol directly (mainly language bindings such as the MySQL JDBC driver). If there were some backdoor or flaw in the protocol, it would be very easy to spot.

An attack exploiting a bug in the MySQL server code is slightly more possible, but still very unlikely. These attacks usually take advantage of poor C programming practices that lead to the possibility of buffer overflows, allowing the attacker to execute arbitrary code. While MySQL has never undergone an official public code audit, in several years of popular use it has not gained a history of vulnerability. This does not at all mean that there aren't undiscovered exploits in the code, but it does speak well for the quality of the code. And of course, like any open source project, the source code of the MySQL server is freely available for scrutiny.

In all, the possibly of a direct compromise of the MySQL server is very slim. However, the difference between slim and none can mean the safety of your data. Any available precautions against this type of attack should always be taken. For instance, the 'libsafe' package provides system-wide protections against buffer overflow attacks. Also never run the MySQL server as the 'root' user. Should the server become compromised the damage done by the attacker can by greatly limited if they are stuck in a restricted account, such as a special user created just to run MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

Unauthorized Access

This is probably the most real and common danger to a MySQL server system. An attacker, eavesdropping on the network traffic on the server machine's network, can intercept the authentication information used by MySQL clients to connect to the server. The attacker can then create their own connection, and gain access as an authorized user.

By default MySQL performs all network communication using unencrypted data. Any one monitoring the network traffic can watch the entire conversation between the MySQL server and a client. Since the MySQL protocol is open, it is simple to decode the conversation and extract information such as the authentication data used by the client, or perhaps sensitive data straight from a query result.

One way to minimize this problem is to access MySQL over a network as little as possible. While that may seem like a useless solution, it is quite common with modern applications. With the advent of the Web application, it is common to have a situation where the MySQL 'client' is actually a Web application accessed through a Web server. In this scenario, it is usually desirable to use local (Unix socket) communication between the 'client' Web application and the MySQL server (assuming they are on the same machine). The outside world communicates with the application via the Web server, leaving the MySQL server securely in a non-networked environment.

When you have to use a network connection, the next best thing is to limit your network visibility by using a firewall or similar setup. There is no reason to expose the MySQL server to the Internet as a whole unless you have to.

Even if you do have make your MySQL server visible on an open network (like the Internet), all hope is not lost. Recent versions of MySQL have introduced the ability to use SSL encrypted communications between the MySQL server and clients. To use this feature, the SSL libraries must exist on both the MySQL server and client machines, before MySQL is installed.

It is always a good idea to configure a MySQL server to use SSL if it is available on your machine. Once a MySQL server has been equipped with SSL, it will automatically attempt to use SSL whenever contacted by a client. Only if the client is incapable of SSL does the server fall back to plain unencrypted communication. For more information about configuration the MySQL server to use SSL, see Chapter XX: Installation.

Denial of Service

The final type of network attack on a MySQL server is the most insidious. Denial of Service attacks have gained popularity among the cracker community in recent years because of its difficulty to defeat. In a denial of service attack, the attacker floods the server machine with network data. If the attacker can send data faster than the server can respond to it, network traffic to the server will come to a halt as it tries to process the data. The problem with this type of attack is that it is extremely difficult to defeat. As long as an attacker can communicate with the server machie, they can bombard the server with data in order of cause a denial of service.

 Copyright © 2001 O'Reilly & Associates, Inc.

The only real way to prevent a denial of service attack is to eliminate an attackers ability to contact to the server machine. The easiest way to do this is to limit network access to the MySQL server to only those clients that need it. If all of the users directly connecting to the MySQL server are on the same network, it is possible to configure a firewall to deny access to the MySQL server to anyone else.

Likewise, as mentioned earlier, if the only client access to MySQL is through a Web application on the same server machine, it is possible to disable MySQL network available completely, since all remote users will only be connecting directly to the Web server.

Client Security

The final area of security we will consider does not directly impact the MySQL server but is nonetheless important. This is this the realm of client security. For as good as your internal security is, if an authorized client can be hijacked, they will be able to access the MySQL server will all of the rights of that client, and the server will have no way of knowing the difference.

In the realm of the client, a 'user' is the user of the client program. That is, any operating system-level user on the same machine as the client.

The concept of 'access' for a client is the ability to execute the client program. Unless the client program performs some sort of authentication whenever it is run (such as prompting for a password), anyone who executes the client has full access to all of its abilities.

Client security is often overlooked when dealing with these matters, usually because the client is not necessarily on the same machine as the server. It is tempting to think that if your server is locked down then the possibility of malicious activity is eliminated. However, if malicious users gain access to secure clients, they also gain access to the server.

This becomes an even greater problem when considering clients on multi-use machines. For example, consider a Web server that is used by multiple sites. Depending on the setup of the server, a web application that accesses a database may provide it's authentication information to any other user on the server.

To better understand this problem, it is necessary to review how MySQL clients communicate with the server. To do this, we will consider two scenarios: a client on a single-user machine and a client on a multi-use machine separate from the MySQL server.

Scenario 1

Single-user machines are the easiest to secure, but should still not overlooked. The recent emergence of Web-based attacks such as the ILOVEU trojan can expose a machine that

Copyright © 2001 O'Reilly & Associates, Inc.

has no servers to attack. In addition, there is always the possibility of someone gaining physical access to your personal machine. Who doesn't step away occasionally without activating a screen lock?

The danger posed by an attacker gaining access to a single-user machine is that they could then use the client to access the MySQL server with the same rights as the actual user. For this to happen, the MySQL client has to have some knowledge of the users authentication (that is, the username and password). There are four ways a MySQL client can obtain the authentication information for a user:

- The client client asks for the authentication
- The client uses a configuration file for authentication
- The client has the authentication information hard-coded
- The client uses the default authentication

Prompting

In this method, the client presents a prompt to the user to enter a username and password. This method is the most secure out of all of the possibilities because the authentication information is not stored permanently anywhere on the machine. An attacker who gains access to the machine would have to know the username and password in order to use the client to access the MySQL server. This makes the machine useless to the attacker, since they could use any machine as the client if they knew the username and password before hand.

While this is the most secure method of running a MySQL client, it is often inconvenient. In fact, it can be argued that this method actually leads to decreased security for some users. The argument is that if a user is forced to type in their passwords for something they use infrequently, they are likely to forget the passwords. Therefore, in order to make remembering the passwords easier, the user will tend to pick more insecure passwords.

Whether or not this is true in practice, it is undeniable that having to type in the username and password repeated for a commonly used application can be tedious. The main factor to consider when evaluating this option is the nature of the user. If they can put up with entering the username and password whenever they want to access the client, this is definitely a secure way to go.

Configuration File

In this method, the client reads a configuration file that contains the username and password. This configuration file could be a standard MySQL configuration file or a configuration file specific to the client. The MySQL client libraries look for configuration files in the following places:

`/etc/my.cnf` (Unix)

`$HOME/.my.cnf` (Unix, where *$HOME* is the home directory of the user)

 Copyright © 2001 O'Reilly & Associates, Inc.

`%WINDOWS%\my.ini` (Windows, where *%WINDOWS%* is the system Windows directory)

`C:\my.cnf` (Windows)

In both cases (Unix and Windows), the second listed configuration file takes precedence over the first. And any client-specific configuration file will take precedence over either.

The most important thing to remember when using a configuration file for storing authentication information is that the information is stored in the configuration file as plain text. Anyone who can read the configuration file can read the authentication data. On a single-user machine, this generally means that anyone with access to the machine, locally or remotely, will have full access to this information. Since a single-user machine is meant for only one person's use, this is usually an acceptible risk. However it is still a risk. It should be assumed that anyone who has accessed the machine knows the MySQL username and password used by the client.

> If you are writing a MySQL client, you may be deciding whether use the default MySQL configuration files or your own for accessing authentication information. The advantage to using the MySQL configuration files is that you do not have to do anything. If you leave the connection information blank the client libraries will automatically use any username and password given in the standard configuration files. This makes programming the client easy, while still giving the user the option of configuring their username and password.

There are no direct advantages to using a custom configuration file. However, if your application already has a configuration file, including the MySQL options in it will let your users configure the entire application in one file. In addition, using a custom configuration file provides a weak form of 'security by obscurity'. If an attacker were looking for the MySQL authentication data, he or she would sure look in all of the standard configuration files. However, the attacker may not know about the configuration file for your application, thus protecting the data. Like any form of security by obscurity, this should not be the basis for your security plan, but rather a small 'bonus' received when using a custom configuration file.

Hard Coding

This method involves encoding the username and password directly into the application. For this to happen, the application has to be custom written (or at least, custom-compiled) for the client machine. This is rarely done for single-user machines, so this method is hardly used. However, it does remain and option.

As far as security goes this method falls between the security of prompting the user and the openness of using configuration files. It is tempting to think that this method is just as secure as prompting because the authentication data is hidden in the client binary and not visible to any user of the system. However, this is not generally the case.

Copyright © 2001 O'Reilly & Associates, Inc.

With most programming languages, text strings are stored within a binary program as a plain text string. Therefore, using methods such as the Unix string program, it is possible to examine the binary and read the authentication information. Even worse is the case where the client is written in a non-precompiled language such as Perl or Python. In this case, the username and password would be visible directly within the script file in plain text.

Default

In this method, the client does not make any attempt at all to find out the authentication information for the user. Instead it simply uses the default username and password used by the MySQL client library. On Unix, this default username is the Unix-username that owns the client process. The default password is bank.

As far as pure client-security is concerned this method is just as secure as prompting. No authentication information is stored permanently on the machine, rendering it useless to an attacker. However, for this method to be useful, the MySQL access rights have to be configured to allow a user to connect with no password.

Therefore, while this method is secure from a client perspective, it requires a compromise of the MySQL access rights that is unacceptable in most instances.

SSL

So far, when considering our single-user machine the types of attacks we have considered have involved direct access to the machine in order to execute the client. However, this is not the only type of attack that could comprise the MySQL authentication data.

If a malicious user were monitoring the network traffic coming from the machine, they would be able to eavesdrop on the communication between the machine and the MySQL server. With this access they may be able to obtain the authentication information used to connect to the server.

As mentioned in the section on server security, MySQL transmits authentication information by default as scrambled plaintext. Therefore, the actual username and password can be extracted (with some work). SSL connections should be used whenever available, as they encrypt the traffic between the client and the server, eliminating the effectiveness of eavesdropping attacks.

Scenario 2

While multi-user computers used to be the rule, the desktop computing explosion of the 1980‘s and '90‘s has made them the exception for workstations and home computers. However, for server applications, multi-user computers are still very common. There are two common ways in which a MySQL client is used within a multi-user machine:

 Copyright © 2001 O’Reilly & Associates, Inc.

- Most obviously, users of a multi-user machine can connect to the machine and directly execute MySQL clients. For instance, a user of a Unix server can log in and run a MySQL client on the command line.
- More commonly, but perhaps less obvious, is the case where users do not directly connect to a server, but rather run MySQL clients remotely (or automatically). For example, consider a Web server that hosts multiple sites. If these sites include database-driver applications, each Web application acts as a MySQL client when access the database. Therefore, a user's application acts as a proxy for the user, creating the same concerns that exist when users directly connect.

The major concern when dealing with a multi-user machine is that one user could access the authentication information for another user. Since MySQL uses the same methods to retrieve authentication information on a multi-user machine as on a single-user machine, we have the same four possibilities to consider. However, the implications of some of these possibilities are different in a mutli-user environment:

Prompting

Just like in the single-user scenario, prompting the user for the username and password is the most secure method of securing that information. However, there are a couple of additional considerations to be aware of when in a multi-user environment.

Firstly, it may be somewhat easier for an attacker to intercept the username and password of the user while they are typing it in. On a single-user machine, the attacker must be able to intercept the network traffic remotely. In the case of a multi-user machine, the attacker has much easier access to the network traffic, and could possibly even intercept the keystrokes of the user locally without even looking at the network traffic.

Secondly, there is always the possibility that a malicious user replaces the MySQL client with a trojan copy that works just like the real client, except that it also captures the username and password.

While both of these possibilities exist, they involve the concentrated effort of a malicious attacker. Prompting is still the surest way to keep authentication safe from casual snoopers. In addition, there are several steps you can take to minimize the effectiveness of the above attacks:

- As mentioned in in the first scenario, use SSL connections to MySQL whenever possible. This eliminates the danger of network snooping as the authentication information will not be decipherable within the encrypted data.
- Also use encryption (such as SSH) when connection to the multi-user machine remotely. Even if your client is secure, if you are using plain telnet to connect to the machine, an attacker can read your username and password as you type it in.
- Make sure the device permissions are properly set on the machine. While you may not have control over this if you are not the systems administrator of the machine, it is important to check to see of the terminal devices can be read by any user or only

Copyright © 2001 O'Reilly & Associates, Inc.

the owner. If any user can read the terminal device, a user logged in at the same time as you can read your keystrokes as you type them.

- Make sure your path is secure. A common method of tricking users into running trojan programs is to put a program with the same name as the real client in a commonly used directory, such as '/tmp'. If your path checks the current directory first, it will run the trojan instead of the real client whenever you are in '/tmp'. The current directory should always be the last entry in the path, if it's there at all.
- Make sure the client remains untouched. The systems administrator of the machine should keep a list of checksums for all of the executable programs on the machine. This list should be periodically checked with the actual binaries to make sure they haven't changed. Again, if you don't have system administrator access to the machine, the best you can do is bug the admin to make sure this happens.

> The method of prompting only applies to the first type of user described above, that is, a user directly connecting to the machine (such as via SSH). Users running applications on a shared server (such as a Web server) cannot use prompting, naturally, since they are not connected to the machine.

Configuration File

Storing the authentication information within a configuration file has the same benefits as in the case of the single-user machine and fewer drawbacks. On a single-user machine, there is always the threat that someone could access the machine and look at the configuration files.

For a multi-user machine this is actually less of a problem if proper precautions are taken. Since the MySQL client will be executed by a logged-in user of the machine, it is possible to set the permissions of the configuration file so that only that user can read it. Any other user of the machine will not be able to access the configuration file. Therefore, to get the authentication information, an attacker would have to be able to become the logged-in user. At this point, the problem becomes one of system security and not one of MySQL security.

This makes properly secured configuration files a very good choice for multi-user machines. However, this becomes tricky when considering the two different types of multi-user access mentioned above.

For users directly accessing the machine (such as via SSH), everything is fine. Each user can have their own configuration file, which is only readable by that user. For users running applications, especially Web applications, on a shared server, however, the situation becomes tricky.

Most Web servers run applications (such as CGI programs) as the same user that owns the web server process itself. That is, if the web server is running as user 'www', and a CGI program is owned by user 'joebob', the web server will run the program as user 'www'.

Copyright © 2001 O'Reilly & Associates, Inc.

Therefore, any configuration file used by the Web application must be readable by ‘www’.

This creates a problem if applications belonging to more than one user are on the Web server. If each user has to make their configuration file readable by ‘www’, then any user’s web application can read any other users configuration file. A malicious user can create a web application which reads other users’ configuration files and retrieves the usernames and passwords.

One solution to this is to configure the web server to execute web applications as the user who owns the executable, rather than the user that owns the web server process. In other words, using our previous example, the web server would execute a web application belonging to the user ‘joebob’ as the user ‘joebob'.

This brings back the per-user security of the applications, and allows them to read configuration files that are private to each user. However, it also opens up security risks on the server. If a user were to obtain access to the server via a flaw in a web application, they would gain access as the user who owned the web application. In the previous model where all applications ran as the web server user, the attacker would only gain access as that user, which should have limited access to the rest of the machine.

Besides the security implications of running the web applications as the individual user, this method simply doesn’t work well with some applications. Environments that share all of the applications in a single process space (such as Java servlets and Apache mod_perl) must use the same user for each application (because only one actual process is being used). In these cases, every web application will always have access to the configuration files of any other web application.

Hard Coding

Hard coding the authentication information into the application has the same implications with multi-user servers as it does with single-user machines. These implications become even more apparent in a multi-user environment. Unless the application binary is somehow encrypted (which is not very practical with an executable binary), any user on the machine will be able to examine the binary and extract the username and password.

Therefore, while this method will keep out casual snoopers, it should not be used in a multi-user environment where any of the other users are not trusted.

Default

Using the default MySQL authentication provides the same benefits and problems in a multi-user environment as on a single-user machine. It creates great client security, as the username and password are not stored anywhere on the machine, but it requires that the user have a blank password, eliminating the usefulness of MySQL authentication.

Copyright © 2001 O’Reilly & Associates, Inc.

Final Note

A section on data security cannot be considered complete with a mention of the two most common attackers of all: user error and natural disaster. Many times more data is lost through equipment failure or software design error than is lost via malicious intruders. Fortunately, protecting against these types of incidents is much more straightforward than the steps needed to protect against a determined human attacker. Please read Chapter XX: Administration and make sure your MySQL data is frequently backed up, and replicated if necessary. Accidents always happen.

Copyright © 2001 O'Reilly & Associates, Inc.

8

Database Design

Once you install your DBMS software on your computer, it can be very tempting to just jump right into creating a database without much thought or planning. As with any software development, this kind of ad hoc approach works with only the simplest of problems. If you expect your database to support any kind of complexity, some planning and design will definitely save you time in the long run. You will need to take a look at what details are important to good database design.

Database Design Primer

Suppose you have a large collection of compact discs and you want to create a database to track them. The first step is to determine what the data that you are going to store is about. One good way to start is to think about why you want to store the data in the first place. In our case, we most likely want to be able to look up CDs by artist, title, and song. Since we want to look up those items, we know they must be included in the database. In addition, it is often useful to simply list items that should be tracked. One possible list might include: CD title, record label, band name, song title. As a starting point, we will store the data in the table shown in Table 2-1.

Table 2-1: A CD Database Made Up of a Single Table (continued)

Band Name	**CD Title**	**Record Label**	**Songs**
Stevie Wonder	Talking Book	Motown	You Are the Sunshine of My Life, Maybe Your Baby, Superstition, . . .
Miles Davis Quintet	Miles Smiles	Columbia	Orbits, Circle, . . .
Wayne Shorter	Speak No Evil	Blue Note	Witch Hunt, Fee-Fi-Fo-Fum
Herbie Hancock	Headhunters	Columbia	Chameleon, Watermelon

Copyright © 2001 O’Reilly & Associates, Inc.

			Man, . . .
Herbie Hancock	Maiden Voyage	Blue Note	Maiden Voyage

(For brevity's sake, we have left out most of the songs.) At first glance, this table seems like it will meet our needs since we are storing all of the data we need. Upon closer inspection, however, we find several problems. Take the example of Herbie Hancock. "Band Name" is repeated twice: once for each CD. This repetition is a problem for several reasons. First, when entering data in the database, we end up typing the same name over and over. Second, and more important, if any of the data changes, we have to update it in multiple places. For example, what if "Herbie" were misspelled? We would have to update the data in each of the two rows. The same problem would occur if the name Herbie Hancock changes in the future (à la Jefferson Airplane or John Cougar). As we add more Herbie Hancock CDs to our collection, we add to the amount of effort required to maintain data consistency.

Another problem with the single CD table lies in the way it stores songs. We are storing them in the CD table as a list of songs in a single column. We will run into all sorts of problems if we want to use this data meaningfully. Imagine having to enter and maintain that list. And what if we want to store the length of the songs as well? What if we want to perform a search by song title? It quickly becomes clear that storing the songs in this fashion is undesirable.

This is where database design comes into play. One of the main purposes of database design is to eliminate redundancy from the database. To accomplish this task, we use a technique called *normalization*. Before we start with normalization, let's start with some fundamental relational database concepts. A data model is a diagram that illustrates your database design. It is made up of three main elements: entities, attributes, and relationships. For now, let's focus on entities and attributes; we will take a look at relationships later.

Database Entities

An *entity* is a thing or object of importance about which data must be captured. All "things" are not entities, only those things about which you need to capture information. Information about an entity is captured in the form of attributes and/or relationships. If something is a candidate for being an entity and it has no attributes or relationships, it is not really an entity. Database entities appear in a data model as a box with a title. The title is the name of the entity.

Entity Attributes

An *attribute* describes information about an entity that must be captured. Each entity has zero or more attributes that describe it, and each attribute describes exactly one entity. Each entity instance (row in the table) has exactly one value, possibly `NULL`, for each of its attributes. An attribute value can be numeric, a character string, date, time, or some other basic data value. In the first step of database design, logical data modeling, we do not worry about how the attributes will be stored.

 Copyright © 2001 O'Reilly & Associates, Inc.

NULL provides the basis for the problem of dealing with missing information. It is specifically used for the case in which you lack a certain piece of information. As an example, consider the situation where a CD does not list the song lengths of each of its tracks. Each song has a length, but you cannot tell from the case what that length is. You do not want to store the length as zero, since that would be incorrect. Instead, you store the length as NULL. If you are thinking you could store it as zero and use zero to mean "unknown", you are falling into one of the same traps that led to one of the Y2K problems. Not only did old systems store years as two digits, but they often gave a special meaning to 9-9-99.

Our example database refers to a number of things: the CD, the CD title, the band name, the songs, and the record label. Which of these are entities and which are attributes?

Data Model

Notice that we capture several pieces of data (CD title, band name, etc.) about each CD, and we absolutely cannot describe a CD without those items. CD is therefore one of those things we want to capture data about and is likely an entity. To start a data model, we will diagram it as an entity. Figure 2-1 shows our sole entity in a data model.

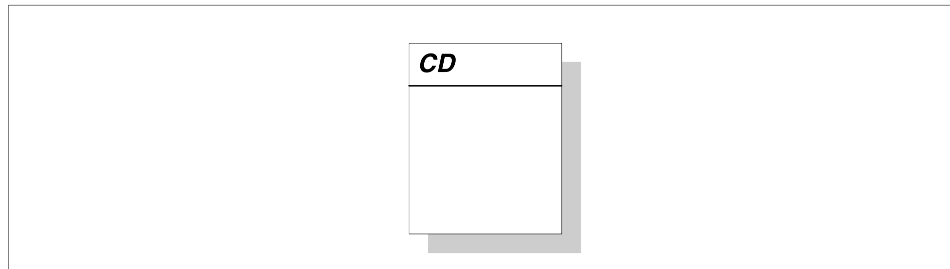


Figure 2-1: The CD entity in a data model

By common entity naming conventions, an entity name must be singular. We therefore call the table where we store CDs "CD" and not "CDs." We use this convention because each entity names an instance. For example, the "San Francisco 49ers" is an instance of "Football Team," not "Football Teams."

At first glance, it appears that the rest of the database describes a CD. This would indicate that they are attributes of CD. Figure 2-3 adds them to the CD entity in Figure 2-1. In a data model, attributes appear as names listed in their entity's box.

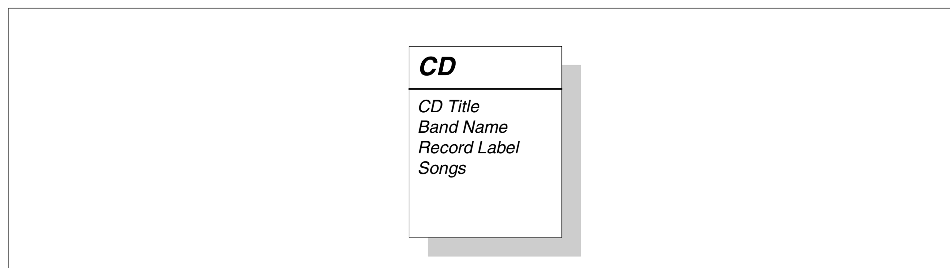


Figure 2-2: The CD entity with its attributes

This diagram is simple, but we are not done yet. In fact, we have only just begun. Earlier, we discussed how the purpose of data modeling is to eliminate redundancy using a technique called normalization. We have a nice diagram for our database, but we have not gotten rid of the redundancy as we set out to do. It is now time to normalize our database.

Normalization

E.F. Codd, then a researcher for IBM, first presented the concept of database normalization in several important papers written in the 1970s. The aim of normalization remains the same today: to eradicate certain undesirable characteristics from a database design. Specifically, the goal is to remove certain kinds of data redundancy and therefore avoid update anomalies. Update anomalies are difficulties with the insert, update, and delete operations on a database due to the data structure. Normalization additionally aids in the production of a design that is a high-quality representation of the real world; thus normalization increases the clarity of the data model.

As an example, say we misspelled "Herbie Hancock" in our database and we want to update it. We would have to visit each CD by Herbie Hancock and fix the artist's name. If the updates are controlled by an application which enables us to edit only one record at a time, we end up having to edit many rows. It would be much more desirable to have the name "Herbie Hancock" stored only once so we have to maintain it in just one place.

First Normal Form (1NF)

The general concept of normalization is broken up into several "normal forms." An entity is said to be in the first normal form when all attributes are single-valued. To apply the first normal form to an entity, we have to verify that each attribute in the entity has a single value for each instance of the entity. If any attribute has repeating values, it is not in 1NF.

A quick look back at our database reveals that we have repeating values in the Songs attribute, so the `CD` is clearly not in 1NF. To remedy this problem, an entity with repeating values indicates that we have missed at least one other entity. One way to discover other entities is to look at each attribute and ask the question "What thing does this describe?"

What does `Song` describe? It lists the songs on the CD. So `Song` is another "thing" that we capture data about and is probably an entity. We will add it to our diagram and give it a `Song Name` attribute. To complete the `Song` entity, we need to ask if there is more about a `Song` that we would like to capture. We identified earlier song length as something we might want to capture. Figure 2-3 shows the new data model.

Copyright © 2001 O'Reilly & Associates, Inc.

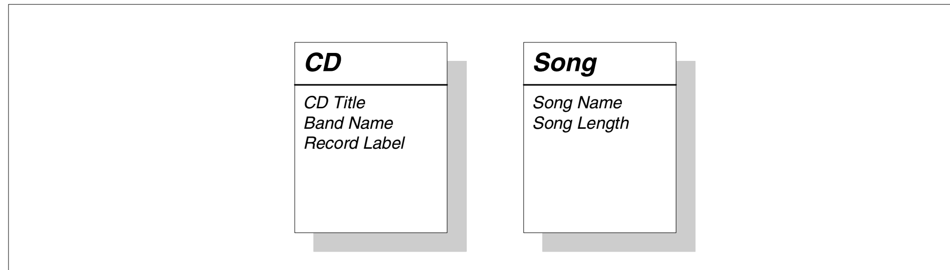


Figure 2-3: A data model with CD and Song entities

Now that the `Song Name` and `Song Length` are attributes in a `Song` entity, we have a data model with two entities in 1NF. None of their attributes contain multiple values. Unfortunately, we have not shown any way of relating a `CD` to a `Song`.

The Unique Identifier

Before discussing relationships, we need to impose one more rule on entities. Each entity must have a unique identifier—we'll call it the ID. An *ID* is an attribute of an entity that meets the following rules:

- It is unique across all instances of the entity.
- It has a non-`NULL` value for each instance of the entity, for the entire lifetime of the instance.
- It has a value that never changes for the entire lifetime of the instance.

The ID is very important because it gives us a way to know which instance of an entity we are dealing with. Identifier selection is critical because it is also used to model relationships. If, after you've selected an ID for an entity, you find that it doesn't meet one of the above rules, this could affect your entire data model.

Novice data modelers often make the mistake of choosing attributes that should not be identifiers and making them identifiers. If, for example, you have a `Person` entity, it might be tempting to use the `Name` attribute as the identifier because all people have a name and that name never changes. But what if a person marries? What if the person decides to legally change his name? What if you misspelled the name when you first entered it? If any of these events causes a name change, the third rule of identifiers is violated. Worse, is a name really ever unique? Unless you can guarantee with 100% certainty that the `Name` is unique, you will be violating the first rule. Finally, you do know that all `Person` instances have non-`NULL` names. But are you certain that you will always know the name of a `Person` when you first enter information about them in the database? Depending on your application processes, you may not know the name of a `Person` when a record is first created. The lesson to be learned is that there are many problems with taking a non-identifying attribute and making it one.

The solution to the identifier problem is to invent an identifying attribute that has no other meaning except to serve as an identifying attribute. Because this attribute is invented and completely unrelated to the entity, we have full control over it and guarantee that it meets the rules of unique identifiers. Figure 2-4 adds invented ID attributes to each of our entities. A unique identifier is diagrammed as an underlined attribute.

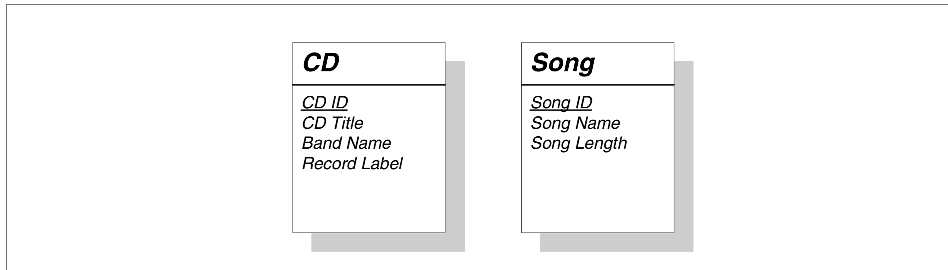


Figure 2-4: The CD and Song entities with their unique identifiers

Relationships

The identifiers in our entities enable us to model their relationships. A relationship describes a binary association between two entities. A relationship may also exist between an entity and itself. Such a relationship is called a *recursive relationship*. Each entity within a relationship describes and is described by the other. Each side of the relationship has two components: a name and a degree.

Each side of the relationship has a name that describes the relationship. Take two hypothetical entities, an `Employee` and a `Department`. One possible relationship between the two is that an `Employee` is "assigned to" a `Department`. That `Department` is "responsible for" an `Employee`. The `Employee` side of the relationship is thus named "assigned to" and the `Department` side "responsible for."

Degree, also referred to as cardinality, states how many instances of the describing entity must describe one instance of the described entity. Degree is expressed using two different values: "one and only one" (1) and "one or many" (M). An employee is assigned to one department at a time, so `Employee` has a one and only one relationship with `Department`. In the other direction, a department is responsible for many employees. We therefore say `Department` has a "one or many" relationship with `Employee`. As a result a `Department` could have exactly one `Employee`.

It is sometimes helpful to express a relationship verbally. One way of doing this is to plug the various components of a direction of the relationship into this formula:

> *entity1* has [one and only one | one or many] *entity2*

Using this formula, `Employee` and `Department` would be expressed like so:

> Each `Employee` must be assigned to one and only one `Department`.
> Each `Department` may be responsible for one or many `Employees`.

We can use this formula to describe the entities in our data model. A `CD` contains one or many `Songs` and a `Song` is contained on one and only one `CD`. In our data model, this relationship can be shown by drawing a line between the two entities. Degree is expressed with a straight line for "one and only one" relationships or "crows feet" for "one or many" relationships. Figure 2-5 illustrates these conventions.

 Copyright © 2001 O'Reilly & Associates, Inc.

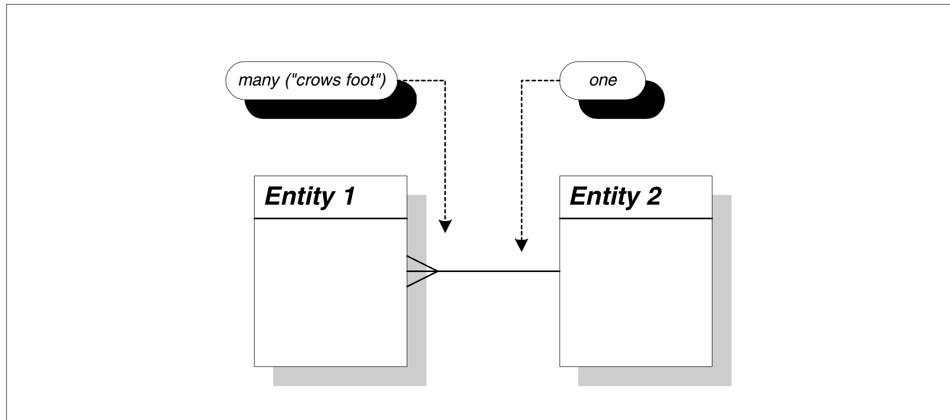


Figure 2-5: Anatomy of a relationship

How does this apply to the relationship between `Song` and `CD`? In reality, a `Song` can be contained on many `CD`s, but we ignore this for the purposes of this example. Figure 2-6 shows the data model with the relationships in place.

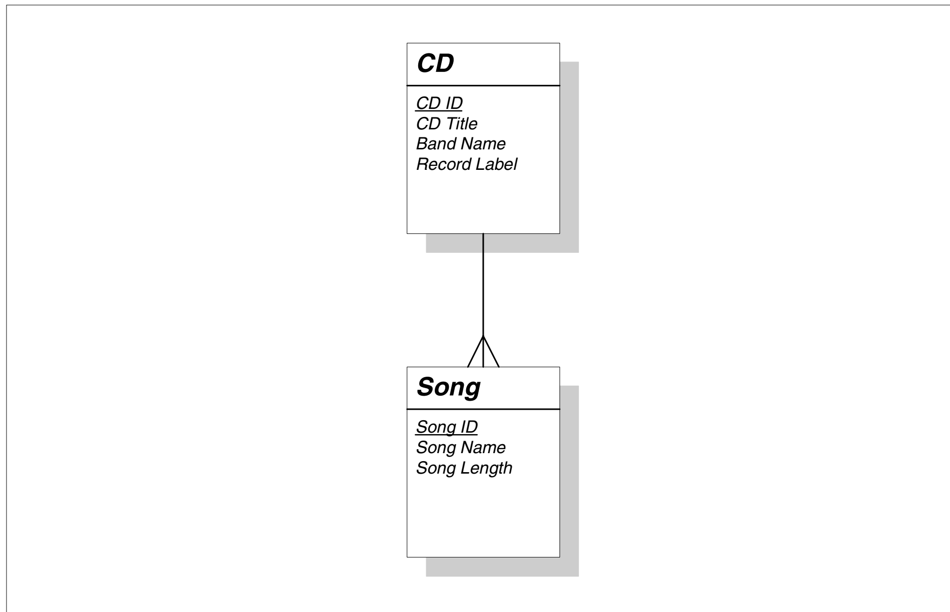


Figure 2-6: CD-Song relationship

With these relationships firmly in place, we can go back to the normalization process and improve upon the design. So far, we have normalized repeating song values into a new entity, Song, and modeled the relationship between it and the `CD` entity.

Second Normal Form (2NF)

An entity is said to be in the second normal form if it is already in 1NF and all non-identifying attributes are dependent on the entity's entire unique identifier. If any attribute is not

dependent entirely on the entity's unique identifier, that attribute has been misplaced and must be removed. Normalize these attributes either by finding the entity where it belongs or by creating an additional entity where the attribute should be placed.

In our example, "Herbie Hancock" is the `Band Name` for two different CDs. This fact illustrates that `Band Name` is not entirely dependent on `CD ID`. This duplication is a problem because if, for example, we had misspelled "Herbie Hancock," we would have to update the value in multiple places. We thus have a sign that `Band Name` should be part of a new entity with some relationship to `CD`. As before, we resolve this problem by asking the question: "What does a band name describe"? It describes a band, or more generally, an artist. Artist is yet another thing we are capturing data about and is therefore probably an entity. We will add it to our diagram with `Band Name` as an attribute. Since all artists may not be bands, we will rename the attribute `Artist Name`. Figure 2-7 shows the new state of the model.

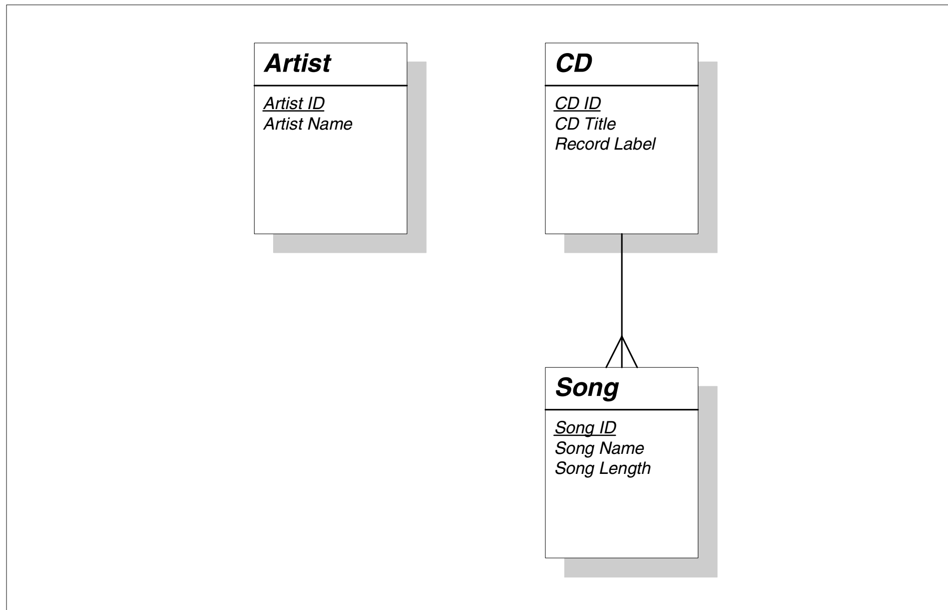


Figure 2-7: The data model with the new Artist entity

Of course, the relationships for the new Artist table are missing. We know that each `Artist` has one or many `CD`s. Each `CD` could have one or many `Artist`s. We model this in Figure 2-8.

 Copyright © 2001 O'Reilly & Associates, Inc.

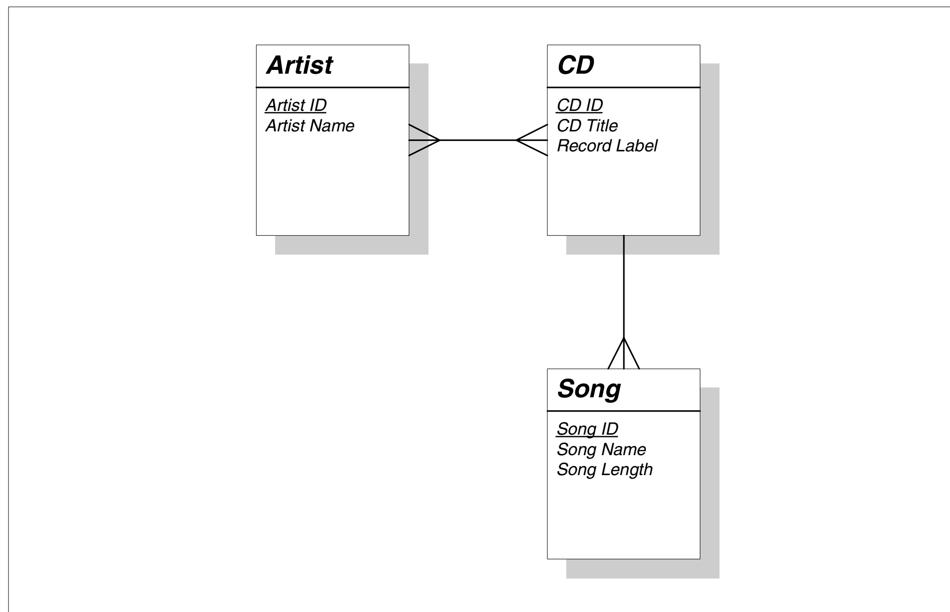


Figure 2-8: The Artist relationships in the data model

We originally had the `Band Name` attribute in the `CD` entity. It thus seemed natural to make `Artist` directly related to `CD`. But is this really correct? On closer inspection, it would seem that there should be a direct relationship between an `Artist` and a `Song`. Each `Artist` has one or more `Song`s. Each `Song` is performed by one and only one `Artist`. The true relationship appears in Figure 2-9.

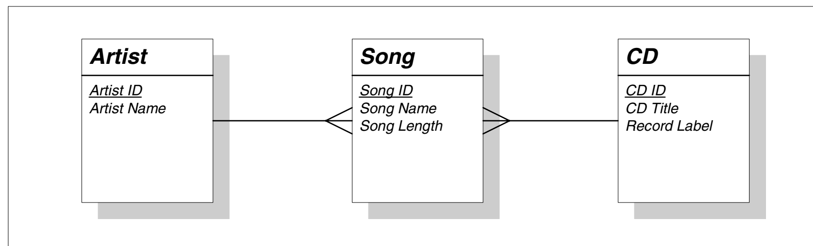


Figure 2-18: The real relationship between Artist and the rest of our data model

Not only does this make more sense than a relationship between `Artist` and `CD`, but it also addresses the issue of compilation `CD`s.

Kinds of Relationships

When modeling the relationship between entities, it is important to determine both directions of the relationship. After both sides of the relationship have been determined, we end up with three main kinds of relationships. If both sides of the relationship have a degree of one and only one, the relationship is called a “one-to-one” or “1-to-1” relationship. As we will find out later, one-to-one relationships are rare. We do not have one in our data model.

Copyright © 2001 O’Reilly & Associates, Inc.

If one of the sides has a degree of “one or many” and the other side has a degree of “one and only one,” the relationship is a “one-to-many” or “1-to-M” relationship. All of the relationships in our current data model are one-to-many relationships. This is to be expected since one-to-many relationships are the most common.

The final kind of relationship is where both sides of the relationship are “one or many” relationships. These kinds of relationships are called “many-to-many” or “M-to-M” relationships. In an earlier version of our data model, the `Artist-CD` relationship was a many-to-many relationship.

Refining Relationships

As we noted earlier, one-to-one relationships are quite rare. In fact, if you encounter one during your data modeling, you should take a closer look at your design. A one-to-one relationship may imply that two entities are really the same entity. If they do turn out to be the same entity, they should be folded into a single entity.

Many-to-many relationships are more common than one-to-one relationships. In these relationships, there is often some data we want to capture about the relationship. For example, take a look at the earlier version of our data model in Figure 2-15 that had the many-to-many relationship between `Artist` and `CD`. What data might we want to capture about that relationship? An `Artist` has a relationship with a `CD` because an `Artist` has one or more `Songs` on that `CD`. The data model in Figure 2-17 is actually another representation of this many-to-many relationship.

All many-to-many relationships should be resolved using the following technique:

1. Create a new entity (sometimes referred to as a *junction entity*). Name it appropriately. If you cannot think of an appropriate name for the junction entity, name it by combining the names of the two related entities (e.g., `ArtistCD`). In our data model, `Song` is a junction entity for the `Artist-CD` relationship.
2. Relate the new entity to the two original entities. Each of the original entities should have a one-to-many relationship with the junction entity.
3. If the new entity does not have an obvious unique identifier, inherit the identifying attributes from the original entities into the junction entity and make them together the unique identifier for the new entity.

In almost all cases, you will find additional attributes that belong in the new junction entity. If not, the many-to-many relationship still needs to be resolved, otherwise you will have a problem translating your data model into a physical schema.

More 2NF

Our data model is still not in 2NF. The value of the `Record Label` attribute has only one value for each `CD`, but we see the same `Record Label` in multiple `CD`s. This situation is similar to the one we saw with `Band Name`. As with `Band Name`, this duplication indicates that `Record Label` should be part of its own entity. Each `Record Label` releases one or many `CD`s. Each `CD` is released by one and only one `Record Label`. Figure 2-10 models this relationship.

Copyright © 2001 O’Reilly & Associates, Inc.

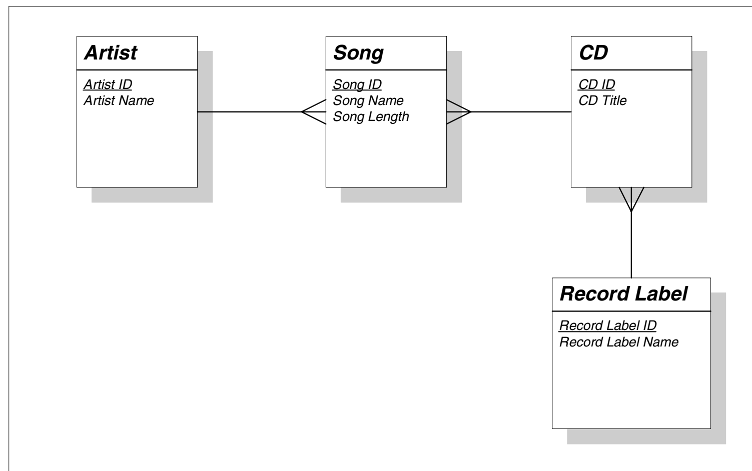


Figure 2-10: Our data model in the second normal form

Third Normal Form (3NF)

An entity is said to be in the third normal form if it is already in 2NF and no non-identifying attributes are dependent on any other non-identifying attributes. Attributes that are dependent on other non-identifying attributes are normalized by moving both the dependent attribute and the attribute on which it is dependent into a new entity.

If we wanted to track `Record Label` address information, we would have a problem for 3NF. The `Record Label` entity with address data would have `State Name` and `State Abbreviation` attributes. Though we really do not need this information to track `CD` data, we will add it to our data model for the sake of our example. Figure 2-11 shows address data in the `Record Label` entity.

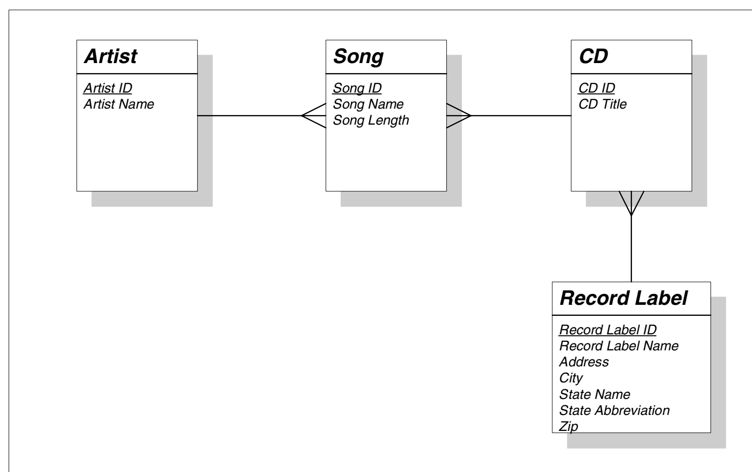


Figure 2-11: Record Label address information in our CD database

The values of `State Name` and `State Abbreviation` would conform to 1NF because they have only one value per record in the `Record Label` entity. The problem here is that `State Name` and `State Abbreviation` are dependent on each other.

Copyright © 2001 O'Reilly & Associates, Inc.

In other words, if we change the `State Abbreviation` for a particular `Record Label`—from MN to CA—we also have to change the `State Name`—from Minnesota to California. We would normalize this by creating a `State` entity with `State Name` and `State Abbreviation` attributes. Figure 2-12 shows how to relate this new entity to the `Record Label` entity.

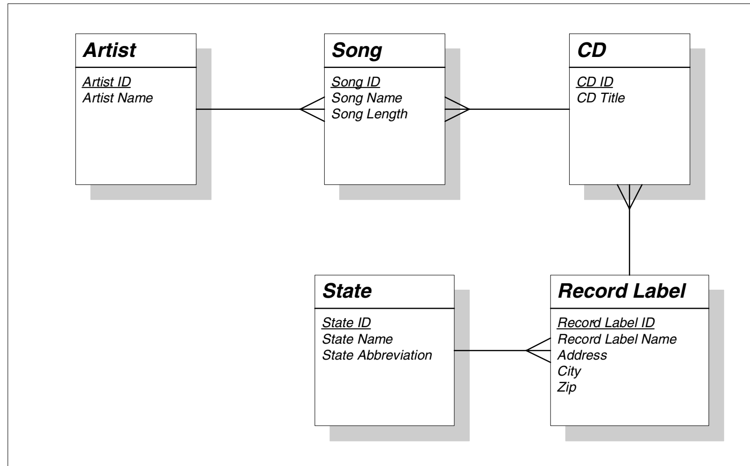


Figure 2-12: Our data model in the third normal form

Now that we are in 3NF, we can say that our data model is normalized. There are other normal forms which have some value from a database design standpoint, but these are beyond the scope of this book. For most design purposes, the third normal form is sufficient to guarantee a proper design.

A Logical Data Modeling Methodology

We now have a completed logical data model. Let's review the process we went through to get here.

1. Identify and model the entities.
2. Identify and model the relationships between the entities.
3. Identify and model the attributes.
4. Identify unique identifiers for each entity.
5. Normalize.

In practice, the process is rarely so linear. As shown in the example, it is often tempting and appropriate to jump around between entities, relationships, attributes, and unique identifiers. It is not as important that you follow a strict process as it is that you discover and capture all of the information necessary to correctly model the system.

The data model we created in this chapter is quite simple. We covered an approach to creating such a model which is in-line with the type and complexity of databases you are likely to encounter in developing MySQL or mSQL databases. We did not cover a whole

 Copyright © 2001 O'Reilly & Associates, Inc.

host of design techniques and concepts that are not so important to small-scale database design, but these can be found in any text dedicated to database design.

Physical Database Design

What was the point in creating the logical data model? You want to create a database to store data about CDs. The data model is only an intermediate step along the way. Ultimately, you would like to end up with a MySQL database where you can store data. How do you get there? Physical database design translates your logical data model into a set of SQL statements that define your MySQL database.

Since MySQL is a relational database systems, it is relatively easy to translate from a logical data model, such as the one we described earlier, into a physical MySQL database. Here are the rules for translation:

1. Entities become tables in the physical database.
2. Attributes become columns in the physical database. Choose an appropriate datatype for each of the columns.
3. Unique identifiers become columns that are not allowed to have NULLs. These are called *primary keys* in the physical database. You may also choose to create a unique index on the identifiers to enforce uniqueness.
4. Relationships are modeled as *foreign keys*.

Tables and columns

If we apply the first three rules to our data model—minus the Record Label address information—we will end up with the physical database described in Table 2-2. This does not include the foreign keys yet.

Table 2-2: Physical Table Definitions for the CD Database

Table	Column	Datatype	Notes
CD	CD_ID	INT	primary key
	CD_TITLE	VARCHAR(50)	
ARTIST	ARTIST_ID	INT	primary key
	ARTIST_NAME	VARCHAR(50)	
SONG	SONG_ID	INT	primary key
	SONG_NAME	VARCHAR(50)	
	SONG_LENGTH	TIME	
RECORD_LABEL	RECORD_LABEL_ID	INT	primary key
	RECORD_LABEL_NAME	VARCHAR(50)	

Copyright © 2001 O’Reilly & Associates, Inc.

The first thing you may notice is that all of the spaces are gone from the entity names in our physical schema. This is because these names need to translate into SQL calls to create these tables. Table names should thus conform to SQL naming rules. Another thing to notice is that we made all primary keys of type `INT`. Because these attributes are complete inventions on our part, they can be of any index-able datatype.[1] The fact that they are of type `INT` here is almost purely arbitrary. It is *almost* arbitrary because it is actually faster to search on numeric fields in many database engines and hence numeric fields make good primary keys. However, we could have chosen `CHAR` as the type for the primary key fields and everything would work just fine. The bottom line is that this choice should be driven by your criteria for choosing identifiers.

CD_TITLE, ARTIST_NAME, SONG_NAME and RECORD_LABEL_NAME are set to be of type `VARCHAR` with a length of 50. The length has been chosen arbitrarily for the sake of this example. In reality, you should do some analysis of sample data to determine the length of text fields. If you set them too short, you may end up with a database that is not able to capture all the data you need to store.

SONG_LENGTH is set to type `TIME` which can store elapsed time.

Foreign Keys

We now have a starting point for a physical schema. We haven't yet translated the relationships into the physical data model. As we discussed earlier, once you have refined your data model, you should have all 1-to-1 and 1-to-M relationships—the M-to-M relationships were resolved via junction tables. We model relationships by adding a foreign key to one of the tables involved in the relationship. A foreign key is the unique identifier or primary key of the table on the other side of the relationship.

The most common relationship is the 1-to-M relationship. This relationship is mapped by placing the primary key from the "one" side of the relationship into the table on the "many" side. In our example, this rule means that we need to do the following:

- Place a `RECORD_LABEL_ID` column in the `CD` table.
- Place a `CD_ID` column in the `SONG` table.
- Place an `ARTIST_ID` column in the `SONG` table.

Table 2-3 shows the new schema.

Table 2-3: The Physical Data Model for the CD Database (continued)

Table	Column	Datatype	Notes

1 Later in this book, we will cover the datatypes supported by MySQL and mSQL. Each database engine has different rules about which datatypes can be indexible. Neither database, for example, allows indices to be created on whole `TEXT` fields. It would therefore be inappropriate to have a primary key column be of type `TEXT`.

 Copyright © 2001 O'Reilly & Associates, Inc.

CD	CD_ID	INT	primary key
	CD_TITLE	VARCHAR(50)	
	RECORD_LABEL_ID	INT	foreign key
ARTIST	ARTIST_ID	INT	primary key
	ARTIST_ID	VARCHAR(50)	
SONG	SONG_ID	INT	primary key
	SONG_NAME	VARCHAR(50)	
	SONG_LENGTH	TIME	
	CD_ID	INT	foreign key
	ARTIST_ID	INT	foreign key
RECORD_LABEL	RECORD_LABEL_ID	INT	primary key
	RECORD_LABEL_NAME	VARCHAR(50)	

We do not have any 1-to-1 relationships in this data model. If we did have such a relationship, it should be mapped by picking one of the tables and giving it a foreign key column that matches the primary key from the other table. In theory, it does not matter which table you choose, but practical considerations may dictate which column makes the most sense as a foreign key. Another way to handle 1-to-1 relationships is to simply combine both entities into a single table. In that case, you have to pick a primary key from one of the tables to be the primary key of the combined table.

We now have a complete physical database schema ready to go. The last remaining task is to translate that schema into SQL. For each table in the schema, you write one `CREATE TABLE` statement. Typically, you will choose to create unique indices on the primary keys to enforce uniqueness.

Example 2-1 is an example SQL script for creating the example database in MySQL.

Example 2-1: An Example Script for Creating the CD Database in MySQL

```
CREATE TABLE CD (CD_ID INT NOT NULL,
                 RECORD_LABEL_ID INT,
                 CD_TITLE TEXT,
                 PRIMARY KEY (CD_ID))

CREATE TABLE ARTIST (ARTIST_ID INT NOT NULL,
                     ARTIST_NAME  TEXT,
                     PRIMARY KEY (ARTIST_ID))

CREATE TABLE SONG (SONG_ID INT NOT NULL,
                   SONG_NAME TEXT,
                   SONG_LENGTH TEXT,
                   CD_ID INT,
                   ARTIST_ID INT,
                   PRIMARY KEY (SONG_ID))
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
CREATE TABLE RECORD_LABEL (RECORD_LABEL_ID INT NOT NULL,
                           RECORD_LABEL_NAME TEXT,
                           PRIMARY KEY(RECORD_LABEL_ID))
```

Note that the "FOREIGN KEY" reference is not used in the script. This is because MySQL does not support FOREIGN KEY constraints. MySQL will allow you to embed them in your CREATE TABLE statements but they will be ignored for the purposes of enforcing them.

Data models are meant to be database independent. You can therefore take the techniques and the data model we have generated in this chapter and apply them not only to MySQL, but to Oracle, Sybase or any other relational database engine. In the following chapters, we will discuss the details of how you can use your new database design knowledge to build applications.

Copyright © 2001 O'Reilly & Associates, Inc.

9

Database Applications

We have spent the entire book so far discussing the database as if it exists in some sort of vacuum. It serves its purpose only when being used by other applications. We should therefore take a look at how the database relates to the other elements of a database application before exploring the details of database application development in various languages. This detour examines conceptual issues important not only to programming with MySQL, but also to programming with any relational database engine. Our look at database programming covers such complex issues as understanding the basic architectures common to Web-oriented database applications and how to map complex programming models into a relational database.

Architecture

Architecture describes how the different components of a complex application relate to one another. A simple Web application using Perl to generate dynamic content has the architecture shown in Figure 9-1. This architecture describes four components: the Web browser, the Web server, the Perl CGI engine, and the MySQL database.

FIGURE9-1.BMP

Figure 9-1. . The architecture of a simple Web application

Architecture is the starting point for the design of any application. It helps you identify at a high level all of the relevant technologies and what standards those technologies will use to integrate. The architecture in Figure 9-1, for example, shows the Web browser talking to the server using HTTP.

Copyright © 2001 O'Reilly & Associates, Inc.

As we will cover in the later chapters of this section, MySQL exposes itself through a variety of APIs tailored to specific programming languages. Java applications access MySQL through JDBC; Python applications through the Python DB-API, etc. The architecture above clearly shows to any observer that the application in question will use the Perl DBI API to access MySQL.

There are numerous architectures used in database applications. In this chapter, we will cover the three most common architectures: client/server, distributed, and Web. Though one could argue that they are all variations on a theme, they do represent three very different philosophical approaches to building database applications.

Client/Server Architecture

At its simplest, the client/server architecture is about dividing up application processing into two or more logically distinct pieces. The database makes up half of the client/server architecture. The database is the 'server'; any application that uses that data is a 'client.' In many cases, the client and server reside on separate machines; in most cases, the client application is some sort of user-friendly interface to the database. Figure 9-2 provides a graphical representation of a simple client/server system.

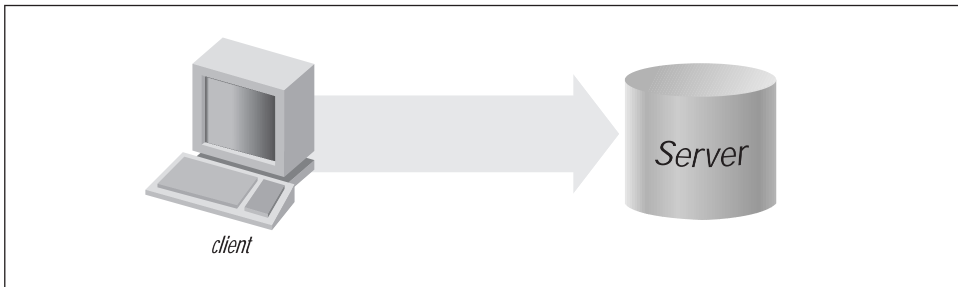


Figure 9-2. The client/server architecture

You have probably seen this sort of architecture all over the Internet. The Web, for example, is a giant client/server application in which the Web browser is the client and the Web server is the server. In this scenario, the server is not a relational database server, but instead a specialized file server. The essential quality of a server is that it serves data in some format to a client.

Application Logic

Because client/server specifically calls out components for user interface and data processing, actual application processing is left up to the programmer to integrate. In other words, client/server does not provide an obvious place for a banking application to do interest calculations. Some client/server applications place this

Copyright © 2001 O'Reilly & Associates, Inc.

kind of processing in the database in the form of stored procedures; others put it in the client with the user interface controls. In general, there is no right answer to this question.

Under MySQL, the right answer currently is to put the processing in the client due to the lack of stored procedure support in MySQL. Stored procedures are on the MySQL to-do list, and—perhaps even by the time you read this book—stored procedures will eventually be a viable place for application logic in a client/server configuration. Whether or not MySQL has stored procedures, however, MySQL is rarely used in a client/server environment. It is instead much more likely to be used with the Web architecture we will describe later in this chapter.

Fat and Thin Clients

It used to be that there were two kinds of clients: fat clients and thin clients. A fat client was a client in a client/server applications that included application processing; a thin client was one that simply had user interface logic. With the advent of Web applications, we now have the term ultra-thin to add to the list. An ultra-thin client is any client that has only display logic. Controller logic—what happens when you press "Submit"—happens elsewhere. In short, an ultra-thin client is a Web form.

The advantage of an ultra-thin client is that it makes real the concept of a ubiquitous client. As long as you can describe the application layout to a client using some sort of markup language, the client can paint the UI for a user without the programmer needing to know the details of the underlying platform. When the UI needs to respond to a user action, it sends information about the action to another component in the architecture to respond to the action. Client/server, of course, has no such component.

Distributed Application Architecture

The distributed application architecture provides a logical place where application logic is supposed to occur, but it does not provide a place for UI controller logic. Figure 9-3 shows the layout of an application under the distributed application architecture.

FIGURE9-3.BMP

Figure 9-3. . The distributed application architecture

As you can see, this architecture is basically the client/server architecture with a special place for application logic—the middle tier. This small difference, however, represents a major philosophical shift from the client/server design. It says, in short, that it is important to separate application logic from other kinds of logic.

Copyright © 2001 O'Reilly & Associates, Inc.

In fact, placing application logic in the database or in the user interface is a good way to hinder your application's ability to grow with changing demands. If, for example, you need a simple change to your data model, you will have to make significant changes to your stored procedures if your application logic is in the database. A change to application logic in the UI, on the other hand, forces you to touch the UI code as well and thus risk adding bugs to systems that have nothing to do with the changes you are introducing.

The distributed application architecture thus does two things. First, it provides a home for application processing so that it does not get mixed in with database or user interface code. The second thing it does, however, is make the user interface independent of the underlying data model. Under this architecture, changes to the data model affect only how the middle tier gets data from and puts it into the database. The client has no knowledge of this logic and thus does not care about such changes.

The distributed application architecture introduces two truly critical elements. First of all, the application logic tier enables the reuse of application logic by multiple clients. Specifically, by calling out the application logic with well-defined integration points, it is possible to reuse that logic with user interfaces not conceived when the application logic was written.

The second, not so obvious thing this architecture brings to applications is the ability to provide easy support for fail-over and scalability. The components in this architecture are logical components, meaning that they can be spread out across multiple actual instances. When a database or an application server introduces clustering, it can act and behave as a single tier while spreading processing across multiple physical machines. If one of those machines goes down, the middle-tier itself is still up and running.

Complex transactions are a hallmark of distributed applications. For that reason, MySQL today makes a poor backend for this architecture. As support for transactions in MySQL matures, this state of affairs may change.

Web Architecture

The Web architecture is another step in evolution that appears only slightly different than the distributed application architecture. It makes a true ultra-thin client possible by providing only display information in the form of HTML to a client. All controller logic occurs in a new component, the Web server. Figure 9-4 illustrates the Web architecture.

FIGURE9-4.BMP

Figure 9-4. . The Web application architecture

Copyright © 2001 O'Reilly & Associates, Inc.

The controller comes in many different forms, depending on what technologies you are using. PHP, CGI, JSP, ASP, ColdFusion, and WebObjects are all examples of technologies for processing user events. Some of these technologies even break things up further into content creation and controller logic. Using a content management system like OpenMarket, for example, your JSP is nothing more than a tool for dynamically building your HTML. The actual controller logic is passed off to a servlet action handler that performs any application server interaction.

The focus of this book will be the Web architecture since it is the most common architecture in which MySQL is used. We will use both the vision of the Web architecture shown in Figure 9-4 and a simpler one in which the application logic is embedded with controller logic in the Web server. The simpler architecture is mostly relevant to MySQL applications since MySQL performs best for heavy read applications—applications without complex application logic.

Connections and Transactions

Whatever architecture you are using, the focus of this book lies at the point where your application talks to the database. As a database programmer, you need to worry about how you get data from and send it to your database. As we mentioned earlier, the tool to do that is generally some sort of database API. Any API, however, requires a basic understanding of managing a connection, the transactions under that connection, and the processing of the data associated with those transactions.

Connections

The starting point of your database interaction is in making the connection. The details behind what exactly it is to be a connection vary from API to API. Nevertheless, making a connection is basically establishing some sort of link between your code and the database. The variance comes in the form of logical and physical connections. Under some APIs, a connection is a physical connection—a network link is established. Other APIs, however, may not establish a physical link until long after you make a connection, to ensure that no network traffic takes place until you actual need the connection.

The details about whether or not a connection is logical or physical generally should not concern a database programmer. The important thing is that once a connection is established, you can use that connection to interact with the database.

Once you are done with your connection, you need to close it and free up any resources it may have used. It stands to reason that before you actually issue a

Copyright © 2001 O'Reilly & Associates, Inc.

query, you should first connect to the database. It is not uncommon, however, for people to forget the other piece of the puzzle—cleaning up after themselves. You should always free up any database resources you grab the minute you are done with them. In a long-running application like an Internet daemon process, a badly written system can eat up database resources until it locks up the system.

Part of cleaning up after yourself involves proper error handling. Better programming languages make it harder for you to fail to handle exceptional conditions (network failure, duplicate keys on insert, SQL syntax errors, etc.); but, regardless of your language of choice, you must make sure that you know what error conditions can arise from a given API call and act appropriately for each exceptional situation.

Transactions

You talk to the database in the form of transactions.* A simple description of a database transaction is one or more database statements that must be executed together, or not at all.A bank account transfer is a very good example of a complex transaction. In short, an account transfer is actually two separate events: a debit of one account and a credit to another. Should the database crash after the debit occurs but before the credit, the application should be able to back out of the debit. A database transaction enables a programmer to mark when a transaction begins, when it ends, and what should happen should one of the pieces of the transaction fail.

Until recently, MySQL had no support for transactions. In other words, when you executed a SQL statement under old versions of MySQL, it took effect immediately. This behavior is still the default for MySQL. Newer versions of MySQL, however, support the ability to use transactions with certain tables in the database. Specifically, the table must use a transaction-safe table format. Currently, MySQL supports two transaction-safe table types: BDB (Berkeley DB) and InnoDb.

In Chapter 4, we described the MySQL syntax for managing transactions from the MySQL client command line. Managing transactions from within applications is often very different. In general, each API will provide a mechanism for beginning, committing, and rolling back transactions. If it does not, then you likely can follow the command line SQL syntax to get the desired effect.

* Even if you are using a version of MySQL without support for transactions, each statement you send to the database can, in a sense, be thought of as an individual transaction. You simple have no option to abort or package multiple statements together in a complex transaction.

Copyright © 2001 O'Reilly & Associates, Inc.

Transaction Isolation Levels

Managing transactions may seem simple, but there are many issues you need to consider when using transactions in a multi-user environment. First of all, transactions come with a heavy price in terms of performance. MySQL did not originally support transactions because MySQL's goal was to provide a fast database engine. Transactions seriously impact database performance. In order to understand how this works, you need to have a basic understanding of transaction isolation level.

A transaction isolation level basically determines what other people see when you are in the middle of a transaction. In order to understand transaction isolation levels, however, you first need to understand a few common terms:

dirty read
: A dirty read occurs when one transaction views the uncommitted changes of another transaction. If the original transaction rolls back its changes, the one that read the data is said to have "dirty" data.

repeatable read
: A repeatable read occurs when one transaction always reads the same data from the same query no matter how many times the query is made or how many changes other transactions make to the rows read by the first transaction. In other words, a transaction that mandates repeatable reads will not see the committed changes made by another transaction. An application needs to start a new transaction to see those changes.

phantom read
: A phantom read deals with changes occurring in other transactions that would result in the new rows matching your transaction's `WHERE` clause. Consider, for example, a situation in which you have a transaction that reads all accounts with a balance of less than $100. Your transaction performs two reads of that data. Between the two reads, another transaction adds a new account to the database with no balance. That account will now match your query. If your transaction isolation allows phantom reads, you will see the new "phantom" row. If it disallows phantom reads, then you will see the same set of rows each time.

MySQL supports the following transaction isolations levels:

READ UNCOMMITTED
: The transaction allows dirty reads, non-repeatable reads, and phantom reads.

READ COMMITTED
: The transaction disallows dirty reads, but it allows non-repeatable reads and phantom reads.

Copyright © 2001 O'Reilly & Associates, Inc.

REPEATABLE READ
: Committed, repeatable reads as well as phantom reads are allowed. Non-repeatable reads are not allowed.

SERIALIZABLE
: Only committed, repeatable reads are allowed. Phantom reads are specifically disallowed.

As you climb the transaction isolation chain, from no transactions to serializable transactions, you decrease the performance of your application. You therefore need to balance your data integrity needs with your performance needs. In general, READ COMMITTED is as high as an application wants to go, except in a few very exceptional cases.

Using READ UNCOMMITTED

One mechanism of getting the performance of READ UNCOMMITTED but the data integrity of READ COMMITTED is to make a row's primary key the normal primary key plus a timestamp reflecting the time in milliseconds when the row was last updated. When an application performs an update on the underlying row in the database, it updates that timestamp but uses the old one in the WHERE clause:

```
UPDATE ACCOUNT
SET BALANCE = 5.00, LAST_UPDATE_TIME = 996432238000
WHERE ACCOUNT_ID = 5 AND LAST_UPDATE_TIME = 996432191119
```

If this transaction has dirty data, the update will fail and throw an error. The application can then re-query the database for the new data.

Object/Relational Modeling

Accessing a relational database from an object-oriented environment exposes a special paradox: the relational world is entirely about the manipulation of data while the object world is about the encapsulation of data behind a set of behaviors. In an object-oriented application, the database serves as a tool for saving objects across application instances. Instead of seeing the query data as a rowset, an object-oriented application sees the data from a query as a collection of objects.

The most basic question facing the object-oriented developer using a relational database is how to map relational data into objects. Your immediate thought might be to simply map object attributes to fields in a table. Unfortunately, this approach does not create the perfect mapping for several reasons.

- Objects do not store only simple data in their attributes. They may store collections or relationships with other objects.

Copyright © 2001 O'Reilly & Associates, Inc.

- Most relational databases—including MySQL—have no way of modeling inheritance.

Rules of Thumb for Object/Relational Modeling

- Each persistent class has a corresponding database table.
- Object fields with primitive datatypes (integers, characters, strings, etc.) map to columns in the associated database table.
- Each row from a database table corresponds to an instance of its associated persistent class.
- Each many-to-many object relationship requires a join table just as database entities with many-to-many relationships require join tables.
- Inheritance is modeled through a one-to-one relationship between the two tables corresponding to the class and subclass.

Think about an address book application. You would probably have something like the `address` and `person` tables shown in Figure 9-5.

The least apparent issue facing programmers is one of mindset. The basic task of object-oriented access to relational data is to grab that data and *immediately* instantiate objects. An application should only manipulate data through the objects. Most traditional programming methods, including most C, PowerBuilder, and VisualBasic development, require the developer to pull the data from the database and then process that data. The key distinction is that in object-oriented database programming, you are dealing with objects, not data.

Figure 9-6 shows the object model that maps to the data model from Figure 9-5. Each row from the database turns into a program object. Your application therefore takes a result set and, for each row returned, instantiates a new `Address` or `Person` instance. The hardest thing to deal with here is the issue mentioned earlier: how do you capture the relationship between a person and her address in the database application? The `Person` object, of course, carries a reference to that person's `Address` object. But you cannot save the `Address` object within the `person` table of a relational database. As the data model suggests, you store object relationships through foreign keys. In this case, we carry the `address_id` in the `person table`.

Copyright © 2001 O'Reilly & Associates, Inc.

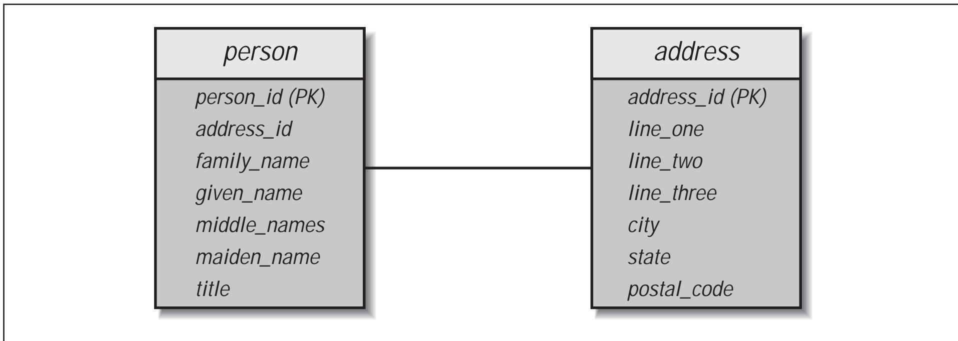


Figure 9-5. The data model for a simple address book application

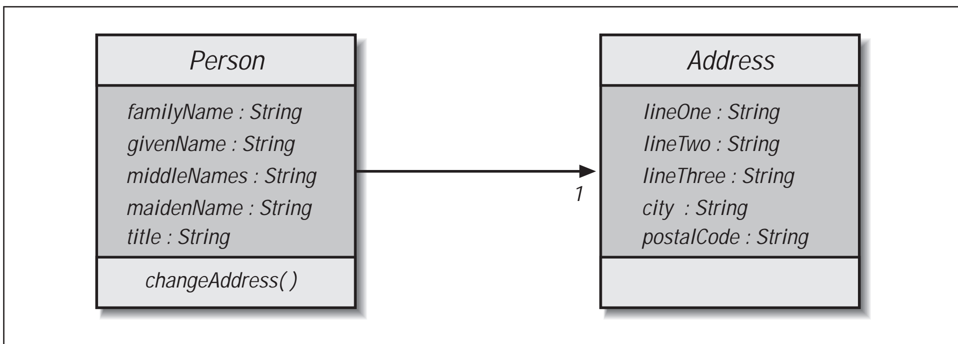


Figure 9-6. The object model supporting a simple address book application

With just a tiny amount of extra complexity to the object model, we can add a world of complexity to the challenge of mapping our objects to a data model. The extra bit of complexity could be to have `Person` inherit from `Entity` with a `Company` class also inheriting from `Entity`. How do we capture an `Entity` separate from a `Person` or a `Company`? The rule we outlined above is actually more of a guideline. In some instances, the base class may be purely abstract and subsequently have no data associated with it in the database. In that instance, you would not have an entity in the database for that class.

Copyright © 2001 O'Reilly & Associates, Inc.

Perl

The Perl programming language has gone from a tool primary used by Unix systems administrators to the most widely used development platform for the World Wide Web. Perl was not designed for the web, but its ease of use and powerful text handling abilities have made it a natural for Web application development. Similarly MySQL, with its small footprint, speed and large feature set, has been very attractive to web developments that need to serve thousands of transactions a day. Therefore, it was only a natural that a Perl interface to MySQL was developed that allowed for the best of both worlds.

Note: At the time of this writing Perl has standardized on the DBI suite of modules for all database interaction, including MySQL. However, many legacy systems still use an older interface to MySQL called MySQL.pm. This module is not compatible with the DBI standard and is no longer actively developed. All new development should certainly use the standard DBI modules, and any sites using MySQL.pm should consider upgrading to DBI for any future development.

DBI

The recommended method for accessing MySQL databases from Perl is the DBD/DBI interface. DBD/DBI stands for DataBase Driver/DataBase Interface. The name arises from the two-layer implementation of the interface. At the bottom is the database driver layer. Here, modules exist for each type of database accessible from Perl. On top of these database dependent driver modules lies a database independent interface layer. This is the interface that you use to access the database. The advantage of this scheme is that the programmer only has to learn one API, the database interface layer. Every time a new database comes along, someone needs only to write a DBD module for it and it will be accessible to all DBD/DBI programmers.

As with all Perl modules, you must use the DBI to get access:

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
```

When running and MySQL Perl programs, you should always include the 'use warnings' statement early in the script. With this present, DBI

will redirect all MySQL specific error messages to STDERR so that you can see any database errors without checking for them explicity in your program.

All interactions between Perl and MySQL are conducted through what is known as a database handle. The database handle is an object—represented as a scalar reference in Perl—that implements all of the methods used to communicate with the database. You may have as many database handles open at once as you wish. You are limited only by your system resources. The connect() method used a connection format of DBI:servertype:database:hostname:post (hostname and port are optional), with additional arguments of username and password to create a handle:

```
# We will use the variable name 'dbh' to indicate a database handle.
# This is a very common idiom among DBI users.
my $dbh = DBI->connect('DBI:mysql:mydata', undef, undef);
my $dbh = DBI->connect('DBI:mysql:mydata:myserver', undef, undef);
my $dbh = DBI->connect('DBI:mysql:mydata', 'me', 'mypass');
```

The servertype attribute is the name of the DBD database-specific module, which in our case will be 'mysql' (note capitalization). The database is the name of a database within the server, and the hostname and port determine the location of the server. If connection via a Unix socket on the local machine, the path of the socket can be used instead of a numerical port.

The first version used above creates a connection to the MySQL server on the local machine via a Unix-style socket. This is the most efficient way to communicate with the database and should be used if you are connecting to a local server. If the hostname is supplied it will connect to the server on that host using the standard port unless the port is supplied as well. If you do not provide a username and password when connecting to a MySQL server, the user executing the program must have sufficient privileges within the MySQL database.

Note: Perl 5 has two difference calling conventions for modules. With the object-oriented syntax, the arrow symbol "->" is used to reference a method in a particular class or object (as in DBI->connect). Another method is the indirect syntax, in which the method name is followed by the class name, then the arguments. The las connect method above would be written as connect DBI 'DBI:mysql:mydata', 'me', 'mypass'. Because of conventions used in early versions of the MySQL Perl modules, a lot of older Perl code that interfaces with MySQL will have lines in it like SelectDB $dbh 'test' wher a simple $dbh->selectdb('test') would do. If you haven't guess, we are partial to the object-oriented syntax, if only because the arrow makes the relationship between class/object and method clear.

Once you have connected to the MySQL server, the database handle -- $dbh in all of the examples in this section – is the gateway to the database server. For instance to prepare a SQL query:

```
$dbh->prepare($query);
```

MySQL allows clients to use any number of different databases during a session, and different databases can even be accessed simultaneously during a query. Each connection also has a default database, which can

be changed at any time. This is the database that is used if no specific database name is given. However, sometimes it is necessary to access two databases that reside on entire separate servers at the same time. To enable this, DBI allows a program to open any number of simultaneous database handles and use them side-by-side.

Chapter XX, Perl Reference, describes the full range of method and variables supplied by DBI.

As an example of the use of DBI consider the following simple programs. In example XX-1, datashow.cgi is a CGI program which accepts a hostname as a parameter--"localhost" is assumed if no parameter is present. The program then displays all of the databases available on that host.

Example 10-1. The CGI program database.cgi shows all of the databases on a MySQL server.

```
#!/usr/bin/perl

use warnings;
use strict;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my ($server, $sock, $host);

$server = param('server') || '';

# Prepare the MySQL DBD driver
my $driver = DBI->install_driver('mysql');

my @databases = $driver->func($server, '_ListDBs');

# If @databases is undefined we assume that means that
# the host does not have a running MySQL server. However, there
# could be other reasons for the failer. You can find a complete
# error message by checking $DBI::errmsg.
if (not @databases) {
         print header, start_html('title'=>"Information on $server",
                'bgcolor' => 'white' );
         print <<END_OF_HTML;
<h1>$server</h1>
$server does not appear to have a running MySQL server.
</body></html>
END_OF_HTML
        exit(0);
}

        print header, start_html('title'=>"Information on $host",
                'bgcolor'=>'white');
        print <<END_OF_HTML;
<h1>$host</h1>
<p>
$host\'s connection is on socket $sock.
<p>
```

```
Databases:<br>
<ul>
END_OF_HTML
foreach (@databases) {
   print "<li>$_\n";
}
print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
exit(0);
```

You probably noticed in this example that we never created a database handle. This is because we never required a connection to a specific database on the server. Instead, we only needed a list of the available databases. To do this, we first loaded the DBD driver for MySQL. This step is normally performed automatically when you make an explicit connection to the server. However, since we are not making a connection, we need to explicitly load the driver in order to use any of its functions.

Once we have loaded the DBD driver, we can make use of any methods it provides. Most methods require a connection to the database server, but a few do not. In our case we want to get a list of databases on a particular server, which is a function that does not require a pre-existing database connection. We call the '_ListDBs' function as parameter to the 'func' method of the driver. This is different than standard DBI methods that are called directly as methods against a database handle.

However, as complete as DBI is, there are some features it does not provide, especially if they are specific to a certain database server. On such feature is the ability to list the available databases on a server. The database servers supported by DBI do not have a common concept of a 'database'. For many of them, being able to list the available databases would not be as useful as it is for MySQL. For this reason, the DBI does not provide a standard method for listing the available databases on a database server.

However, the author of DBI anticipated that DBI would not be able to provide every piece of functionality present in every supported database server. Therefore DBI was given the ability to run database server-specific functions. This should generally be avoided, as code that uses database server specific functionality can not be directly ported to a new database server if need be. But sometime it is necessary to resort to database server specific functionality to get the job done.

Database server specific functions are accessed via the 'func' method that is present in most DBI objects. In our case, we have a driver object created from the DBD MySQL module. Through the 'func' method on this object, we are able to call the MySQL-specific 'ListDBs' function and retrieve a list of the databases on a specific database and server.

Once we have that information, we can create an HTML response page that lists the databases available on a MySQL server.

Now that we know what databases are available to use, the next step is to see what tables we can use. In Example XX-2 tableshow.cgi accepts the name of a database server

(default is "localhost") and the name of a database on that server. The program then shows all of the available tables on that server.

Example XX-2. The CGI program tableshow.cgi shows all of the tables within a database.

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

use strict;
use warnings;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my $db = param('db') or die "Database not supplied!";
my $host = param('host') || 'localhost';

# Connect to the requested server.
my $dbh = DBI->connect("DBI:mysql:$db:$host", undef, undef);

# If $dbh does not exist, the attempt to connect to the database
# server failed. The server may not be running, or the given
# database may not exist.
if (not $dbh) {
    print header, start_html('title'=>'Information on $host => $db'
            "bgcolor" => 'white');

    print <<END_OF_HTML;
<h1>$host</h1>
<H2>$db</h2>
The connection attemp failed for the followign reason:<br>
$DBI::errstr
</body></html>
END_OF_HTML
        exit(0);
}

print header, start_html('title'=>'Information on $host => $db',
               'bgcolor' => 'write' );
print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
<p>
Tables:<br>
<ul>
END_OF_HTML
# $dbh->listtable returns an array of the tables that are available
# in the current database.
my @tables = $dbh->tables;
foreach (@tables) {
     print "<li>$_\n";
}

print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
     exit(0);
```

In this example, we created an actual connection to a MySQL server for the first time. This connection was made to the server and port number given as parameters from the client browser. If no specific hostname and port number are given, the Unix socket /tmp/mysql.sock on the localhost is used by default.

Once we have created an active connection to the desired database, we can interact with that database using the standard DBI methods. In our case, we want to obtain a list of tables that are available within the database. DBI provides the 'tables' method that returns a list of tables within a database.

Notice that at the end of the script we do not explicitly close the database handle or do any other cleanup. The DBI module will automatically close and cleanup any connections at the end of script.

Now that we know the names of all of the databases and tables available to us, we can take the last step and look at the structure and data within each table. Example XX-3 shows all of the information about a specific table, including its data.

Example XX-3. The CGI program tabledump.cgi Shows Information About a Specific Table

```
use struct;
use warnings;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my ($db, $table, $host);
$host = param('host') || '';
$db = param('db') or die “Database not supplied!”;
$table = param(‘table’) or die ‘Table not supplied!’;

# Connect to the requested server.
my $dbh = DBI->connect(“DBI:mysql:$db:$host”, undef, undef);

# WE now prepare a query for the server asking for all of the
# data in the table.
my $table_data = $dbh->prepare(“select * from $table”);
# Now send the query to the server.
$table_data->execute;

# If the return value is undefined, the table must not exist.
# (Or it could be empty; we don't check for that.)
if (not $table_data) {
     print header, start_html('title'=>
       “Information on $host => $db => $table”, 'bgcolor'=>'white');

     print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
The table '$table' does not exist in $db on $host.
</body></html>
END_OF_HTML
     exit(0);
}
```

```
# At this point, we know we have datat to display. First we show
# the layout of the table.
print header, start_html('title'=>
       "Information on $host => $db => $table", 'bgcolor'=>'white');
print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
<h3>$table</h3>
<p>
<table border>
<caption>Fields</caption>
<tr>
   <th>Field</th><th>Type</th><th>Size</th><th>NOT NULL</th>
</tr>
<ul>
END_OF_HTML

# $table_data->NAME returns a reference to an array of the fields
# of the database.
my @fields = @{$table_data->NAME};
# $table_data->TYPE returns an array reference of the types of
# fields. The types returned here are in SQL standard notation,
# not MySQL specific.
my @types = @{$table_data->TYPE};
# $table_data->is_not_null returns a boolean array reference
# indicating which fields have the 'NOT NULL' flag. Notice the
# term 'NULLABLE' has the opposite context as 'NOT NULL'
my @nullable = @{$table_data->NULLABLE};
# $table_data->PRECISION returns an array reference of the lengths
# of the fields. This is defined when the table is created.
# For CHAR-type fields, this is the maximum number of characters.
# For numeric fields this is the maximum number of significant digits.
my @length = @{$table_data->PRECISION};

# All of the above arrays were returned in the same order, so that
# fields[0], $types[0], $not_null[0] and $length[0] all refer to
# the same field.

foreach my $field (0..$#fields) {
   print <<END_OF_HTML;
<tr>
    <td>$fields[$field]</td><td>$types[$field]</td><td>
END_OF_HTML
    print $length[$field] if $types[$field] eq 'SQL_CHAR';
    print '<td>';
    print 'N' if not $nullable[$field];
    print "</tr>\n";
}

print <<END_OF_HTML;
</table>
<p>
<b>Data</b><br>
<ol>
END_OF_HTML

# Now we step through the data, row by row, using
# DBI::fetchrow_array(). We save the data in an array that has
# the same order as the informational arrays @fields, @types, etc.)
# we created earlier.
```

```
While ( my @data = $table_data->fetchrow_array ) {
   print "<li>\n<ul>";
   for (0..$#data) {
         print "<li>$fields[$_] => $data[$_]</li>\n";
   }
   print "</ul></li>";
}

print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
```

This is the most complex of the scripts by far. As in the tables script, we start of by connecting to the database using the parameters passed to us from the client browser. We then use that database connection to execute a SQL query that retrieves all of the data from a table.

The first step in executing a SQL query is to prepare it. DBI provides the 'prepare' method within database handle object. The prepare method takes a SQL query and stores it (either locally or on the database server) until execution. On database servers that store the query on the database server itself, it is possible to perform operations on the query before executing it. However, MySQL does not support that ability yet, and prepared queries are simply stored within the database driver until execution.

The result of the prepare method is an object known as a statement handle. A statement handle is a Perl program's interface to a SQL query, much like a database handle is the interface to the database server itself. While the statement handle is created when the SQL query is prepared, it is not possible to do anything useful with it until the query has been executed.

A query is executed by using the 'execute' method on a statement handle. That is, one a statement handle has been created using 'prepare', calling 'execute' on that handle will cause the query to be sent to the database server and executed. The result of executing a query depends on the type of query. If the query is a non-SELECT query that returns no data (such as INSERT, UPDATE and DELETE) the execute method will return the number of rows that were affected by the query. That is, for an insert query (that inserts one row of data), the execute query will return '1' if the query was successful.

For SELECT queries, the execute method simply returns a true value if the query was successful and a false value if there was an error. The data returned from the query is then available using various methods within the statemet handle itself.

In addition to the data returned from a SELECT query, the statement handle also contains various information about the data (called meta-data). The meta-data associated with the query can be accessed via various properties in the statement handle. In our example we use several of those properties to build a table containing information about the table in question:

$statement_handle->NAME

A reference to an array of the names of the columns in the result set. Since our query is selecting all of the columns from a table, this contains the names of all of the columns in the table.

$statement_handle->TYPE

A reference to an array of the SQL types of the columns in the result set. These types are returned as ANSI SQL standard types. While these are often the same as the MySQL data types, many of the more unusual MySQL data types (such as NUMERIC, and TEXT) are represented as simpler ANSI standard types.

$statement_handle->NULLABLE

A reference to an array of boolean values indicating whether the columns in the result set can contain NULL values. Note that this has the opposite meaning as the 'NOT NULL' which is used when defining MySQL columns. Thus, a NOT NULL column will have a value of false in the NULLABLE array, and vice versa.

$statement_handle->PRECISION

A reference to an array of the maximum lengths of the columns in the result set. These maximum values are defined when the table is created. For character-based columns, this is the maximum number of characters. For numeric columns, this is the number of significant digits.

After printing a table of this meta-data, the program then displays all of the data in the table, row by row. This is done by using the fetchrow_array method on the statement handle containing the data. The fetchrow_array method reads a single row of data from the result set and then advances an internal pointer so that the next call to fetchrow_array will return the next row of data. This continues until there are no rows left, at which time the method will return a false value.

Each row of data is returned as an array, in the order defined in the query. In our case, the query simply specifies 'SELECT *', so we don't know the order in which the fields were defined. However, it is guaranteed that the order of this array is the same as the order of the arrays of meta-data generated earlier. Therefore, we can loop through the data array and use the same indices on the meta-data arrays to describe the columns.

An Example DBI Application

DBI allows for the full range of SQL queries supported by MySQL. As an example, consider a database used by a school to keep track of student records, class schedules, test scores and so on. The database would contain several tables, one for class information, one for student information, one containing a list of tests, and a table for each test. MySQL's ability to access data across tables—such as the table-joining feature—enables all of these tables to be used together as a coherent whole to form a teacher's aide application.

To begin with, we are interested in created tests for the various subjects. To do this we need a table that contains names and ID numbers for the tests. We also need a seperate table for each test. This table will contain the scores for all of the students as wll as a perfect score for comparison. The test table has the following structure:

```
CREATE TABLE test (
```

```
        id INT NOT NULL AUTO_INCREMENT,
        name CHAR(100),
        subject INT,
        num INT
)
```

The individual tests have table structures like this:

```
CREATE TABLE t7 (
     id INT NOT NULL,
     q1 INT,
     q2 INT,
     q3 INT,
     q4 INT,
     total INT
)
```

The table name is t followed by the test ID number from the test table. The user determines the number of questions when he or she creates the table. The total field is the sum of all of the questions.

The program that access and manipulates the test information is the CGI program test.cgi. This program, which follows, allows only for adding new tests. Viewing tests and changing tests is not implemented but is left as an exercise. Using the other scripts in this chapter as a reference, completing this script should be only a moderate challenge. As it stands, this script effective demonstrates the capabilities of DBI.

```
#!/usr/bin/perl

use warnings;
use strict;

use CGI qw(:standard);

# Use the DBI module.
use DBI;
# DBI::connect uses the format 'DBI:driver:database', in our case
# we are using the MySQL driver and accessing the 'teach' database.
my $dbh = DBI->connect('DBI:mysql:teach');
# The add action itself is broken up into three seperate functions.
# The first function, add, prints out the template form for the
# user to create a new test.
sub add {
     $subject = param('subject') || '';
     $subject = '' if $subject eq 'all';

print header, start_html('title'=>'Create a New Test',
     'bgcolor'=>'white');
print <<END_OF_HTML;
<h1>Create a New Test</h1>
<form action="test.cgi" method="post">
<input type="hidden" name="action" value="add2">
Subject:
END_OF_HTML

my @ids = ();
my %subjects = ();
my $out2 =
      $dbh->prepare("SELECT id, name FROM subject ORDER BY name");
$out2->execute;
```

```
# DBI::fetchrow_array retrieves a single row of the results.
while ( my($id, $subject) = $out2->fetchrow_array ) {
     push(@ids, $id);
     $subjects{$id} = $subject;
}

print popup_menu('name'=>'subjects',
      'values'=>[@ids],
      'default'=>$subject,
      'labels'=>\%subjects);
print <<END_OF_HTML;
<br>
Number of Questions: <input name="num" size="5"><br>
An identifier for the test (such as a date):
  <input name="name" size="20">
<p>
<input type="submit" value=" Next Page ">
  <input type="reset">
</form></body></html>
END_OF_HTML
}
```

This function displays a form allowing the user to choose a subject for the test along with the number of questions and a name. In order to print out a list of available subjects, the table of subjects is queried. When using a SELECT query with DBI, the query must first be prepared and then executed. The DBI::prepare function is useful with certain database servers which allow you to perform operations on prepared queries before executing them. With MySQL however, it simply stores the query until the DBI::execute function is called.

The output of this function is sent to the add2 function as shown in the following:

```
sub add2 {
     my $subject = param('subjects');
     my $num = param('num');
     my $name = param('name') if param('name');

     my $out = $dbh->prepare("select name from subject where id=$subject");
     my ($subname) = $out->fetchrow_array;

     print header, start_html('title'=>"Creating test for $subname",
           'bgcolor'=>'white');
     print <<END_OF_HTML;
<h1>Creating test for $subname</h1>
<h2>$name</h2>
<p>
<form action="test.cgi" method="post">
<input type="hidden" name="action" value="add3">
<input type="hidden" name="subjects" value="$subject">
<input type="hidden" name="num" value="$num">
<input type="hidden" name="name" value="$name">
Enter the point value for each of the questions. The points need not add up to 100.
<p>
END_OF_HTML
     for (1..$num) {
          print qq%$_: <input name="q$_" size="3"> %;
          if (not $_ % 5) { print "<br>\n"; }
     }
     print <<END_OF_HTML;
<p>
```

```
Enter the test of the test:<br>
<textarea name="test" rows="20" cols="60">
</textarea>
<p>
<input type="submit" value="Enter Test">
<input type="reset"
</form></body></html>
END_OF_HTML
}
```

In this function, a form for the test is dynamically generated based on the parameters entered in the last form. The user can enter a point value for each question on the test and the full text of the test as well. The output of the function is then sent to the final function, `add3`, as shown in the following:

```
sub add3 {
    my $subject = param('subjects');
    my $num = param('num');

    $name = param('name') if param('name');

    my $qname;
    ($qname = $name) =~ s/'/\\'/g;
    my $q1 = "insert into test (id, name, subject, num) values (
        '', '$qname', $subject, $num)";

     my $in = $dbh->prepare($q1);
     $in->execute;

     # Retrieve the ID value MySQL created for us
     my $id = $in->insertid;

     my $query = "create table t$id (
           id INT NOT NULL,
           ";

     my $def = "insert into t$id values ( 0, ";

     my $total = 0;
     my @qs = grep(/^q\d+$/, param);
     foreach (@qs) {
        $query .= $_ . " INT,\n";
        my $value = 0;
        $value = param($_) if param($_);
        $def .= "$value, ";
        $total += $value;
    }
    $query .= "total INT\n)";
    $def .= "$total)";

    my $in2 = $dbh->prepare($query);
    $in2->execute;
    my $in3 = $dbh->preapre($def);
    $in3->execute;

    # Note that we store the tests in seperate files. Another
    # method of handling this would be to stick the entire test
    # into a TEXT column in the table.
    open (TEST, ">teach/tests/$id") or die "A: $id $!";
    print TEST param('test'), "\n";
    close TEST;
```

```
    print header, start_html('title'=>'Test Created',
           'bgcolor'=>'white');

    print <<END_OF_HTML;
<h1>Test Created</h1>
<p>
The tst has been created.
<p>
<a href=”.”>Go</a> to the Teacher's Aide home page.<br>
<a href=”test.cgi”>Go</a> to the Test main page.<br>
<a href=”test.cgi?action=add”>Add</a> another test.
</body></html>
END_OF_HTML
}
```

Here we enter the information about the test into the database. In doing so we take a step beyond the usual data insertion that we have seen so far. The information about the test is so complex that each test is best kept in a table of its own. Therefore, instead of adding data to an existing table, we have to create a while new table for each test. For we crate an ID for the new test using MySQL auto increment feature and enter the name and ID of the test info a table called test. This table is simple an index of tests so that the ID number of any test can be quickly obtained. Then we simultaneously create two new queries. The first is a CREATE TABLE query that defines our new test. The second is an INSERT query that populates our new table with the maximum score for each question. These queries are then sent to the database server, completing the process (after sending a success page to the user_. Later, after the students have taken the test, each student will get an entry in the test table. Then entries can be compared to the maximum values to determine the student's score.

Object Oriented (OO) Database Programming in Perl

Perl is rarely on anyone's list of theoretically complete object-oriented languages. However, this is mostly because of mis-education and Perl does in fact have very thorough and flexible object-oriented features. However, as with all things Perl, There Is More Than One Way To Do It. That is, while you can write object-oriented Perl, you can also write non-object-oriented Perl or a mixture of OO and non-OO. This flexibility leads to possibilities not available in most other program languages. On the other hand, it also introduces the necessity of discipline on programmers who want to use a strict Object Oriented structure.

One of the best ways to ensure discipline when creating a Object Oriented system, is to use a good design methodology. A design methodology is simply a framework that helps you visualize a system in an Object Oriented manner. There are several good methodologies in existence, but for simplicities sake we'll concentrate on one: Model/View/Controller.

Model/View/Controller

Model/View/Controller (MVC) is an Object Oriented methodology used to help design a software application. The base idea behind MVC is that any application can be split into three distinct parts, or layers: The Model, the View and the Controller. Each layer is an independent unit that performs a specific function.

View
: The View is the user-interface aspect of the application. The View is responsible for presenting information to the user, and also for collecting any user feedback. In a traditional desktop application, the View is the code that draws the screens and reads the keyboard and mouse inputs. In a Web-based application, the View is the code that generates the HTML viewed by the user's browser, as well as the code that interprets any form data submitted by the user. All I/O that involves the user of the system is done in the View. Any input by the user is passed to the Controller for processing.

Controller
: The Controller is the brains of the application. Any software logic performed by the application is done within the controller. In addition, the controller is also the communication center of the application. All user input from the View is processed here, as is all data from the Model (destined for the View). The controller should not be dependant on the View in any way. That is, it should be possible to replace the View (perhaps changing from the desktop application to a Web-based application) without altering the Controller.

Model
: The Model is the body of the application. Here, all objects that represent real-world 'things' within the application are modeled. For example, in the Teacher's Aide example used previously, concepts such as 'tests', 'classes', and 'students' may be represented as objects in the Model. The model is also responsible for any persistence of these objects. Therefore, all database-interaction is performed in that layer. The Model should not be dependant on either the View or Controller in any way. This allows entirely new applications (Controllers and Views) to be built around the same concepts and databases (Models).

As mentioned above, the Model is the only layer that interfaces with external data sources, such as databases. This provides a very convenient abstraction with designing a system. Consider the Teacher's Aide example used earlier. Each script used in that example had to deal with creating HTML, processing form input, accessing the database, and performing logic to manipulate the data. In a large project, these different tasks are logic places to divide labor and it is useful to have a way to separate them.

Using the MVC framework, creating the HTML and processing form input would be the responsibility of the View. Accessing the database is the responsibility of the Model and performing logic is the responsibility of the Controller.

Since our purpose here is to examine using Perl to interface with MySQL, it is clear that the Model is the only layer that direct affects us. This frees us from having to deal with code that is extraneous to our central purpose.

Designing the Model

Designing the Model layer is one of the most important tasks of creating an MVC application. The Model contains abstractions of all concrete 'things' that are used within the application. Therefore it is necessary to have a solid model as a foundation for the rest of the application.

Luckily for us, designing a Model for a database-driven application is very straightforward. That is because the work discovering the relavent abstractions in a system was already done when the database scheme was created. In most cases, each table in the database corresponds to one Model class. The fields of the tables correspond to the attributes of the class. Relationships between tables can usually be expressed in the following manner:

One-to-One

If two tables have a one-to-one relationship, one of two things can happen. If the relationship is one of containment, the contained object should exist as an attribute of the container class. That is, a 'Person' table can be one-to-one with an 'Address' table if a Person has exactly one address. Therefore, the Person class should contain an Address object as an attribute. If the relationship is one of aggregation, the more specific class should be a subclass of the less specific class. That is, an 'Animal' table can be one-to-one with a 'Dog' table, if the Dog has all of the fields of an Animal, plus fields of it's own. Therefore, the Dog class should be a subclass of the Animal class.

One-to-Many

If two tables have a one-to-many relationship, usually the 'One' class contains an array of 'Many' objects. That is, a 'Person' table can be one-to-many with a 'Phone' table as a Person usually has multiple phone numbers. Therefore, the Person class should contain an array of Phone objects.

Many-to-Many

If two tables have a many-to-many relationship, each class can contain an array of objects from the other class. That is, a 'Person' table can be many-to-many with an 'Employer' table (with a many-to-many join table in the middle) since a Person usually has had more than one Employer and each Employer has more than one Person. Therefore, the Person class contains an array of Employer objects, and the Employer class contains an array of Person objects. This type of construct can be very challenging to implement (when you create a Person, you create all of their Employers, each of those Employers will then contain the Person, which contains the Employers, etc., etc.). Because of this, many designers avoid many-to-many relationships when possible. If they are necessary, however, it is possible to pull it off with careful implementation.

Like all classes, a Model class is comprised of attributes and methods. As mentioned above, the attributes of a Model class are simply the fields of the underlying table (as well as possibly objects from related tables). What about the methods?

In object-oriented programming, classes have two kinds of methods: instance and static. Instance methods are only called upon actual objects created from the class. Because of this, they have access to the attribute data within the object. Static methods, on the other hand, can be called on the class itself. They have no knowledge of individual objects of that class.

If you do not mind directly accessing attributes of an object, a Model class only needs three instance methods: update, delete and create. These methods parallel the SQL 'UPDATE', 'DELETE' and 'INSERT' statements that, along with 'SELECT', make up the vast majority of SQL statements.

The update method saves the current state of the object to the database. That is, when an attribute of a Model object is altered somewhere in the application, that change only happens within that object. If the application were to terminate, the object will leave memory and the change would disappear. To make a change permanent, it is necessary to save the attribute of the object to the database. The update method does that by constructing a UPDATE SQL query with all of the attributes of the object and sending it to the database. Like all methods this can be called anything, including 'update', which is simple and to the point.

The delete method removes the object data from the database. The objects running within the program are not directly tied to the database, so that if an object is destroyed (as when it is garbage collected, or when the program terminates), no change is made within the database. To delete an object's data from the database, it is necessary to send the database a 'DELETE' SQL statement to remove the row containing the data. This is the purpose of the delete method. Unfortunately, 'delete' is a keyword in Perl, so it is not possible to use that is the method name. Common alternatives include 'remove', 'destroy', 'Delete' and 'deleteObject'.

As mentioned above, the create method mirrors the SQL INSERT statement which creates a new row of data in the database. Because Model objects exist as data constructs within a running application with no direct tie to the database, it is possible to create a new object that has no corresponding data in the database. Thus, when you want to persist that data, you must create a new row in the database table for it. The create method does this by generating a SQL INSERT statement that contains the data within the object and sending it to the database. The name 'create' is used for the method here, because 'creating' an object makes more logical sense to the rest of the application than 'inserting'. Because the Model hides the details of persistence from the rest of the application, the rest of the application has no idea that there is a database behind the scenes where it is necessary to insert rows. However, this is merely semantics and 'insert', or anything else, would be just as good a name for this method.

While update and delete are the only necessary instance methods in a Model class, a common OO practice is to not directly access attributes, but rather access them through methods (called accessor methods or getter/setter methods). The advantage of this is that it allows the designer to change the attribute in some way in the future, while not changing it's appearance to the rest of the application.

If you follow this practice, then each attribute of the object should have two instance methods: a 'get' method that retrieves the value of the attribute and a 'set' method that sets the attribute to a new value. They can be named anything, but a common practice is to simply prepend 'get/set' to the name of the attribute. So an attribute called 'firstName' would have the methods 'getFirstName' and 'setFirstName'.

The instance methods described above cover three of the four basic SQL commands: 'INSERT', 'UPDATE', and 'DELETE'. This leaves 'SELECT' still untouched. For this, we turn to static methods (also known as class methods). Unlike all of the previous methods, the 'SELECT' method does not operate on already existing objects. The point of a SELECT query is to retrieve data from the database. In other words, you are creating new objects containing data from the database. Therefore, it is necessary to use static methods that do not rely on any instance data.

These methods send SELECT queries to the databases and create new Model objects from the data that is returned. Also differing from the other methods considered so far, there are often several select methods within a Model class. This is because there are usually different contexts in which to create new objects. In particular, there are two situations that are called for in almost every application: Primary Key select and Generic WHERE select

Generic WHERE select

The 'Generic WHERE' select method is the most versitile and common type of select method. In this method, a SQL WHERE query is passed in a parameter (or generated from some other parameters). A SQL SELET is then sent to the database using this WHERE. Out of the resulting data, an array of Model objects is created. Because of the flexbility of the SQL WHERE clause, this method can be leveraged by more specialized select methods, such as the Primary Key select.

Primary Key select

Almost uniformly, well designed relational tables have a primary key. This is a column, or columns, that define a unique row within the table. If you know a primary key value, you can retrieve a single row of data from the table. Having this ability within the Model class allows you to create a single object corresponding to a row of data. This is done by creating a SQL WHERE clause containing the value of the primary key and then calling the generic WHERE select to execute the query. Since we are sending in the value of the primary key, we know we will get an array containing a single object in return. This method then returns this single object.

Example

As an example of a Model class consider a table containing information about a book publisher. For simplicity we'll just use two fields in the table: 'id' and 'name'. The 'id' field is the primary key and uniquely identifies each row of the table. The 'name' field is the name of the publisher.

```
# A Model Class for the publisher table.
```

```
package CBDB::publisher;

our $VERSION = '0.1';

use strict;
use warnings;
use DBI qw(:sql_types);
use CBDB::DB;
use CBDB::Cache;

our @ISA = qw( CBDB::DB );
###########################
###### CONSTRUCTOR ########
###########################
sub new {
    my $proto = shift;

    my $class = ref($proto) || $proto;
    my $self = { };
    bless($self, $class);

    return $self;
}

##############################
########### METHODS ##########
##############################

# getId() - Return Id for this publisher
sub getId {
    my $self = shift;
    return $self->{Id};
}

# setId() - Set Id for this publisher
sub setId {
    my $self = shift;
    my $pId = shift or die "publisher.setId( Id ) requires a value.";
    $self->{Id} = $pId;
}

# getName() - Return Name for this publisher
sub getName {
    my $self = shift;
    return $self->{Name};
}

# setName() - Set Name for this publisher
sub setName {
    my $self = shift;
    my $pName = shift || undef;
    $self->{Name} = $pName;
}

# remove() - Removes an object from the database
sub remove {
    my $self = undef;
    my $where = undef;
    my $is_static = undef;
    if ( ref($_[0]) and $_[0]->isa("CBDB::publisher") ) {
            $self = shift;
```

```
            $where = "WHERE id = ?";
    } elsif (ref($_[0]) eq 'HASH') {
            $is_static = 1;
            $where = 'WHERE ' . make_where($_[0]);
    } else {
            die "CBDB::publisher::remove: Unknown parameters: " . join(' ', @_);
    }

    my $dbh = CBDB::DB::getDB();
    my $query = "DELETE FROM publisher $where";

    my $sth = $dbh->prepare($query);

    if ($is_static) {
            bind_where($sth, $_[0]);
    } else {
            $sth->bind_param(1, $self->getId(), {TYPE=>4});
    }
    $sth->execute;
    $sth->finish;
    $dbh->disconnect;
}

# update() - Updates this object in the database
sub update {
    my $self = shift;
    my $dbh = CBDB::DB::getDB();
    my $query = "UPDATE publisher SET name = ?, id = ? WHERE id = ?";
    my $sth = $dbh->prepare($query);

    $sth->bind_param(1, $self->getName(), {TYPE=>1});
    $sth->bind_param(2, $self->getId(), {TYPE=>4});
    $sth->bind_param(3, $self->getId(), {TYPE=>4});
    $sth->execute;
    $sth->finish;
    $dbh->disconnect;
    CBDB::Cache::set('publisher', $self->getId(), $self);

}

# getByPrimaryKey - Retrieves a single object
#                   from the database based on a primary key
sub getByPrimaryKey {
    my $pId = shift or die "publisher.get()";

    my $where = [ {'id' => $pId } ];
    return ( get( $where, 1 ) )[0];
}

# get - Retrieves objects from the database
sub get {
    my $wheres = undef;
    my $do_all = 1;
    if (ref($_[0]) eq 'ARRAY') { $wheres = shift; $do_all = shift if @_; }
    else { $do_all = shift; }

    my $dbh = CBDB::DB::getDB();
    #my $where = "WHERE ";
    my $where .= ' WHERE  ' . make_where( $wheres );
    my $query = "SELECT publisher.name as publisher_name, publisher.id as
publisher_id FROM publisher $where";
```

```
    my $sth = $dbh->prepare($query);
    bind_where( $sth, $wheres );
    $sth->execute;
    my @publishers;
    while (my $Ref = $sth->fetchrow_hashref) {
            my $publisher = undef;
            if (CBDB::Cache::has('publisher', $Ref->{publisher_id})) {
                    $publisher = CBDB::Cache::get('publisher', $Ref-
>{publisher_id});
            } else {
            $publisher = CBDB::publisher::populate_publisher( $Ref );

                    CBDB::Cache::set('publisher', $Ref->{publisher_id}, $publisher);
            }
            push(@publishers, $publisher);
    }
    $sth->finish;
    $dbh->disconnect;
    return @publishers;
}

# populate_publisher - Return a publisher object populated from a result set
sub populate_publisher {
    my $Ref = shift;
    my $publisher = new CBDB::publisher;
    $publisher->setName($Ref->{publisher_name});
    $publisher->setId($Ref->{publisher_id});

    return $publisher;
}

# create - Inserts the object into the database
sub create {
    my $self = shift;
    my $dbh = CBDB::DB::getDB();
    my $query = "INSERT INTO publisher ( name, id ) VALUES ( ?, ? )";
    my $sth = $dbh->prepare($query);
    my $pk_id = undef;

    $sth->bind_param(1, $self->getName(), {TYPE=>1});
    $sth->bind_param(2, undef, {TYPE=>4});
    $sth->execute;
    $sth->finish;

    $pk_id = CBDB::DB::get_pk_value($dbh, 'publisher_id');
    $self->setId( $pk_id);

    $dbh->disconnect;
    CBDB::Cache::set('publisher', $self->getId(), $self);
    return $self;
}

# make_where() - Construct a WHERE clause from a well-defined hash ref
sub make_where {
    my $where_ref = shift;
    if ( ref($where_ref) ne 'ARRAY' )
            { die "CBDB::publisher::make_where: Unknown parameters: " .
                join(' ', @_); }
    my @wheres = @$where_ref;
    my $element_counter = 0;
    my $where = "";
```

```
    foreach my $element_ref (@wheres) {
            if (ref($element_ref) eq 'ARRAY')
               { $where .= make_where($element_ref); }
            elsif (ref($element_ref) ne 'HASH')
               { die "CBDB::publisher::make_where: malformed WHERE parameter: "
                   . $element_ref; }
            my %element = %$element_ref;
            my $type = 'AND';
            if (not $element_counter and scalar keys %element == 1 and
                exists($element{'TYPE'})) {
                    $type = $element{'TYPE'};
            } else {
                    my $table = "publisher";
                    my $operator = "=";
                    if (exists($element{'table'})) { $table = $element{'table'}; }
                    if (exists($element{'operator'}))
                           { $operator = $element{'operator'}; }
                    if ($element_counter) { $where .= " $type "; } else
                            { $element_counter = 1; }
                    foreach my $term
                           ( grep !/^(table|operator)$/, keys %element ) {
                            $where .= "$table.$term $operator ?";
                    }
            }
    }
    return $where;
}

sub bind_where {
    my $sth = shift;
    my $where_ref = shift;
    my $counter_ref = shift || undef;
    my $counter = ref($counter_ref) eq 'Scalar' ?
            $$counter_ref : 1;
    if ( not $sth->isa('DBI::st') or ref($where_ref) ne 'ARRAY' )
            { die "CBDB::publisher::make_where: Unknown parameters: "
                    . join(' ', @_); }
    my @wheres = @$where_ref;
    foreach my $element_ref (@wheres) {
            if (ref($element_ref) eq 'ARRAY')
              { bind_where($sth, $element_ref, \$counter); }
            elsif (ref($element_ref) ne 'HASH')
              { die "CBDB::publisher::make_where: malformed WHERE parameter: "
                  . $_; }
            my %element = %$element_ref;
            unless (not $counter and scalar keys %element == 1 and
                  exists($element{'TYPE'})) {
                    my $table = "publisher";
                    if (exists($element{'table'})) { $table = $element{'table'}; }
                    foreach my $term
                        ( grep !/^(table|operator)$/, keys %element ) {
                            $sth->bind_param($counter, $element{$term},
                                    {TYPE=>CBDB::DB::getType($table,$term)});
                            $counter++;
                    }
            }
    }
}

1;
```

There are 13 methods in this class:

- new

  This is a generic constructor that simply creates an empty object.

- getId

  This is an accessor method that retrieves the current value of the 'id' attribute.

- setId

  This is an accessor method that sets the value of the 'id' attribute. Since the 'id' field of the table is the primary key, this method will rarely be called.

- getName

  This is an accessor method that retrieves the current value of the 'name' attribute.

- setName

  This is an accessor method that sets the value of the 'name' attribute.

- remove

  This method removes the row of data corresponding to this object in the database. After this method is called, this object should be destroyed since it's underlying data is gone.

- update

  This method updates the data in the database with the attribute data in the object. In effect, this method 'saves' the current state of the object into the database. For this method to work, the object must already have a row in the database, since it uses the SQL UPDATE statement.

- getByPrimaryKey

  This is a static method that creates a single object based on a primary key. This method calls the generic 'get' method to perform the actual query.

- get

  This is a static method that creates a SQL SELECT statement based on WHERE parameters passed to the function. For each row of the result set, a new object is created and an array of these objects is returned.

- populate_publisher

  This is a static method that creates a new object based on data from a result set. This is a utility method that is used by 'get'. The advantage of making this a seperate method is that it can be used externally by other Model classes that need to create new 'publisher' objects.

- create

This method inserts a new row into the table with the data from this object. In this case, the primary key for this table is a MySQL AUTO_INCREMENT field with automatically creates a new value. Therefore, this method only inserts the value of the 'name' field. After the new row has been created, it retrieves the value of the newly created primary key and sets the 'id' attribute accordingly.

- make_where

  SQL WHERE clauses can be very complex, especially when multiple tables are involved. The same information stored within a SQL WHERE clause can also be stored in a Perl hash tree. Using this construct, the relationships between the different WHERE elements can be made clear. This method flattens the hash tree into a regular SQL WHERE clause that can be used in a SQL query.

- bind_where

  As mentioned above, the parameters for a WHERE clause in this class are stored in a hash tree that clearly indicates the relationships between the parameters. While the 'make_where' method creates the actual SQL WHERE clause, the values of the parameters still need to be bound to the statement once the statement is prepared. This method traverses the hash tree and calls the bind_param method to insert the parameter values into the query. After this method is called, the SELECT statement can be executed.

You may have noticed in this class that there are two other classes referenced: DB and Cache. These classes help the Model class in different ways.

DB

This class is the super class of all model classes. It handles creating the connection with the database and other utility matters.

```
# this is a static database helper class
# that allows us to make connections to the database
package CBDB::DB;

use strict;
use warnings;
use BM::mysql;

our $VERSION = '0.1';
our $URL = "DBI:mysql:database=CBDB;host=localhost";
our $USER = "myuser";
our $PASSWORD = "mypass";


my %types = (
    'creator' => { 'name' => 1, 'id' => 4 },
    'book' => { 'title' => 1, 'publisher_id' => 4, 'date' => 11, 'id' => 4 },
    'book_creator' => { 'book_id' => 4, 'creator_id' => 4, 'role_id' => 4 },
    'publisher' => { 'name' => 1, 'id' => 4 },
    'role' => { 'name' => 1, 'id' => 4 },
);

sub getDB {
```

```
    my $dbh = DBI->connect($URL,$USER,$PASSWORD);
    return $dbh;
}

sub get_pk_value {
    my $dbh = shift or die "DB::get_pk_value needs a Database Handle...";

    my $dbd = new BM::mysql();
    return $dbd->get_pk_value( $dbh );
}

sub getType {
    my $table = shift;
    my $col = shift;
    return $types{$table}{$col};
}

1;
```

The class contains three methods:

- getDB

  This method creates a connection to the database and returns the database handle. This method is used by the subclass to get a database handle to execute a query.

- get_pk_value

  This method returns the most recent value of the automatically created primary key of the subclass table. Many tables have a primary key that is automatically generated (this is also known as a surrogate key). This method returns the value of that key when a new row is created. In order to provide complete database abstraction this class uses yet another class for the database specific operation (see BM::mysql, below).

- get_type

  This method returns the SQL type of a column within a table used by the application. All of the fields in all of the tables are stored within this superclass so that any subclass can access the type of any of the fields from any time.

mysql

As mentioned above, an attempt has been made to abstract out any database server specific functionality so that the model base classes can use different database servers. The database-specific functionality for MySQL is placed in a utility class called simply 'mysql'.

```
package BM::mysql;

use strict;
use warnings;

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = { };
```

```
    bless($self, $class);
    return $self;
}

sub is_pk ($$$) {
    my $self = shift;
    my $sth = shift;
    my $i = shift;

    return 1 if $$sth->{mysql_is_pri_key}->[$i];
    return 0;
}

sub is_auto_increment($$$) {
    my $self = shift;
    my $sth = shift;
    my $i = shift;

    return 1 if $$sth->{mysql_is_auto_increment}->[$i];
    return 0;
}

sub get_pk_value {
    my $self = shift;
    my $dbh = shift or die "mysql::get_pk needs a Database Handle...";
    my $mysqlPk = "Select last_insert_id() as pk";
     my $mysqlSth = $dbh->prepare($mysqlPk);
     $mysqlSth->execute();
     my $mysqlHR = $mysqlSth->fetchrow_hashref;
     my $pk = $mysqlHR->{"pk"};
     $mysqlSth->finish;
     return $pk;
}
1;
```

This class contains four methods:

- new

  This is the generic object constructor

- is_pk

  This method determines whether a field is part of the primary key of the table.

- is_auto_increment

  This method determines whether the primary key of the table is an AUTO_INCREMENT-style field.

- get_pk_value

  Ths method returns the value of the most recently inserted AUTO_INCREMENT field.

Cache

The final class used by our Model class is 'Cache'. This class contains a static hash of all of the Model objects used by the application. Besides increasing performance, it also

allows multiple objects to exist that represent the same row of data in the underlying table. By using references to the cached objects, the objects will automatically reflect changes.

```
package CBDB::Cache;
# This file keeps a copy of all active objects, with records of their primary key.
use strict;
use warnings;

my %cache = ();

sub set { $cache{$_[0]}{$_[1]} = $_[2]; }
sub get { return $cache{$_[0]}{$_[1]}; }
sub has { return exists $cache{$_[0]}{$_[1]}; }
1;
```

As you can see, this class is very simple. It contains a static hash of all the model objects used by the application. Three simple methods are present to add an object to the cache, get an object from the cache and check to see if an object already exists in the cache.

Example

Now that we have our Model class and all of it's supporting classes, let's look at how it is used in practice. The following code would be part of the Controller layer of the application. The is the layer that performs all of the logic. Note how all of the actual database calls are hidden from this layer. All that is used is calls to the Model class.

```
use CBDB::publisher;

# First let's create a new publisher object
my $pub = new CBDB::publisher();

# Now let's set some data... We're in a hurry, so we can't be
# bothered with good spelling...
$pub->setName("Joe's Boks");

# Note that we don't set the 'id'. This is an auto-increment field that is
# taken care of automatically by the database. A more abstract way of thinking
# about it is that the 'id' is not a real-world property of this object, it only
# exists because of the necessities of object-relational design. Therefore, at
# the controller level, we shouldn't have to worry about it.

print 'Our new publisher is ' . $pub->getName();

# If the program were to terminate at this point, this object's data would
# be lost. We need to save it to that database for it to be persistent.
$pub->create();
# Now the object has been created in the database and is persistent…
my $new_id = $pub->getId();
# Now that the object has been persisted, it has been assigned a primary key
# We can store that primary key in a variable for later reference.

# ... (some time later)

# Let's retrieve the object we created earlier...
# We use the getByPrimaryKey value with the PK we stored earlier.
# Notice that this is a static method; called on the class itself.
my $pub2 = CBDB::publisher::getByPrimaryKey($new_id);

# Because of the caching mechanism, $pub2 is literally the same object as
```

```
# $pub.
If ($pub2 != $pub) { print ‘Whoops! Something isn’t working right!’ }

# Let’s change some data and fix that typo from earlier.
$pub2->setName(“Joe’s Books”);

# At this point, the data has been changed in the object only, not in the
# underlying data store. However, since all active instances of this object
# are references to the same object, this change takes place everywhere in
# the application immediately.
print ‘The publisher’s name is now ‘ . $pub->getName();
# This will print Joe’s Books, not Joe’s Boks, even though we didn’t explicitly
# touch $pub.

# Now let’s save these changes to the database...
$pub2->update(); # <-- This could just as well have been $pub

# The data has been changed now...

# ... (even later)

# Okay we’re done with this data now, time to delete the row...
$pub->remove();

# The underlying data row in the database has now been removed. However, this
# object (as well as $pub2) still contains the data until the program is
# terminated or the objects are destroyed.
print “The publisher “ . $pub->getName() . “ was just erased...\n”;
```

11

Python

If you are not familiar with Python and you do a lot of Perl programming, you definitely want to take a look at it. Python is an object-oriented scripting language that combines the strengths of languages like Perl and Tcl with a clear syntax that lends itself to applications that are easy to maintain and extend. The O'Reilly & Associates, Inc. book *Learning Python, 2nd Edition* by Mark Lutz and David Asher provides an excellent introduction into Python programming. This chapter assumes a working understanding of the Python language.

In order to follow the content of this chapter, you will need to download and install the MySQLdb, the MySQL version of DB-API. You can find the module at *http://dustman.net/andy/python/MySQLdb*. Chapter 23, *The Python DB-API* in the reference section includes directions on how to install MySQLdb.

DB-API

Like Java and Perl, Python has developed a unified API for database access—DB-API. This database API was developed by a Python Special Interest Group (SIG) called the Database SIG. The Database SIG is a group of influential Python developers interested Python access to various databases. On the positive side, DB-API is a very small, simple API. On the negative side, it isn't very good. Part of its problem is that it is very small and thus does not support a lot of the more complex features database programmers expect in a database API. It is also not very good because it realistically does not enable true database independence.

The Database Connection

The entry point into DB-API is really the only part of the API tied to a particular database engine. By convention, all modules supporting DB-API are named after

Copyright © 2001 O'Reilly & Associates, Inc.

the database they support with a "db" extension. The MySQL implementation is thus called MySQLdb. Similarly, the Oracle implementation would be called oracledb and the Sybase implementation sybasedb. The module implementing DB-API should contain a `connect()` method that returns a DB-API connection object. This method returns an object that has the same name as the module:

```
import MySQLdb;
conn = MySQLdb.connect(host='carthage', user='test',
                       passwd='test', db='test');
```

The above example connects using the user name/password pair 'test'/'test' to the MySQL database 'test' hosted on the machine 'carthage'. In addition to these four arguments, you can also specify a custom port, the location of a UNIX socket to use for the connection, and finally an integer representing client connection flags. All arguments must be passed to connect() as keyword/value pairs in the example above.

The API for a connection object is very simple. You basically use it to gain access to cursor objects and manage transactions. When you are done, you should close the connection:

```
conn.close();
```

Cursors

Cursors represent SQL statements and their results. The connection object provides your application with a cursor via the `cursor()` method:

```
cursor = conn.cursor();
```

This cursor is the center of your Python database access. Through the `execute()` method, you send SQL to the database and process any results. The simplest form of database access is of course a simple insert:

```
conn = MySQLdb.connect(host='carthage', user='test', passwd='test',
db='test');
cursor = conn.cursor();
cursor.execute('INSERT INTO test (test_id, test_char) VALUES (1, 'test')");
print "Affected rows: ", cursor.rowcount;
```

In this example, the application inserts a new row into the database using the cursor generated by the MySQL connection. It then verifies the insert by printing out the number of rows affected by the insert. For inserts, this value should always be 1.

Query processing is a little more complex. Again, you use the `execute()` method to send SQL to the database. Instead of checking the affected rows, how-

Copyright © 2001 O'Reilly & Associates, Inc.

ever, you grab the results from the cursor using one of many fetch methods. Example 11-1 shows a Python program processing a simple query.

Example 11-1. A Simple Query

```
import MySQLdb;

connection = None;
try:
    connection = MySQLdb.connect(host="carthage", user="user",
                                 passwd="pass", db="test");
    cursor = connection.cursor();
    cursor.execute("SELECT test_id, test_val FROM test ORDER BY test_id");
    for row in cursor.fetchall():
        print "Key: ", row[0];
        print "Value: ", row[1];
     connection.close();
 except:
    if connection:
        connection.close();
```

The cursor object actually provides several fetch methods: `fetchone()`, `fetchmany()`, and `fetchall()`. For each of these methods, a row is represented by a Python tuple. In the above example, the `fetchall()` method fetches all of the results from the query into a list of Python tuples This method, like all of the fetch methods, will throw an exception if the SQL was not a query.

Of course, fetching all of the rows at once can be very inefficient for large result sets. You can instead fetch each row one by one using the `fetchone()` method. The `fetchone()` method returns a single row as a tuple where each element represents a column in the returned row. If you have already fetched all of the rows of the result set, `fetchone()` will return `None`.

The final fetch method, `fetchmany()`, is middle ground between `fetchone()` and `fetchall()`. It enables an application to fetch a pre-defined number of rows at once. You can either pass in the number of rows you wish to see returned or instead rely on the value of `cursor.arraysize` to provide a default value.

Parameterized SQL

DB-API includes a mechanism for executing pseudo-prepared statements using the `execute()` method as well as a more complex method called `executemany()`. Parameterized SQL is a SQL statement with placeholders to which you can pass arguments. As with a simple SQL execution, the first argument to `execute()` is a SQL string. Unlike the simple form, this SQL has place holders for parameters specified by the second argument. A simple example is:

```
cursor.execute('INSERT INTO COLORS (COLOR, ABBR) VALUES (%s, %s)',
               ('BLUE', 'BL'));
```

Copyright © 2001 O'Reilly & Associates, Inc.

In this example, %s is placed in the SQL as placeholders for values that are passed as the second argument. The first %s matches the first value in the paramter tuple and the second %s matches the second value in the tuple.

DB-API actually has several ways of marking SQL parameters. You can specify the format you wish to use by setting `MySQLdb.paramstyle`. The above example is `MySQLdb.paramstyle = "format"`. The "format" value is the default for MySQLdb when a tuple of parameters is passed to `execute()` and is basically the set of placeholders from the ANSI C `printf()` function. Another possible value for `MySQLdb.paramstyle` is "pyformat". This value is the default when you pass a Python mapping as the second argument.

DB-API actually allows several other formats, but MySQLdb does not support them. This lack of support is particularly unfortunate since it is common practice in database applications in other languages to mark placeholders with a ?.

The utility of parameterized SQL actually becomes appearant when you use the `executemany()` method. This method enables you to execute the same SQL statement with multiple sets of parameters. For example, consider this code snippet that adds three rows to the database using `execute()`:

```
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('BLUE', 'BL')");
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('PURPLE', 'PPL')");
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('ORANGE', 'ORN')");
```

That same functionality using `executemany()` looks like this:

```
cursor.executemany("INSERT INTO COLOR ( COLOR, ABBREV ) VALUES (%s, %s )",
                   (("BLUE", "BL"), ("PURPLE", "PPL"), ("ORANGE", "ORN")));
```

As you can see, this one line executes the same SQL three times using different values in place of the placeholders. This can be extremely useful if you are using Python in batch processing.

MySQLdb treats all values as string values, even when their underlying database type is BIGINT, DOUBLE, DATE, etc. Thus, all conversion parameters should be %s even though you might think they should be a %d or %f.

Other Objects

DB-API provides a host of other objects to help encapulate common SQL data types so that they may be passed as parameters to `execute()` and `executemany()` and relieve developers of the burden of formatting them for different databases. These objects include Date, Time, Timestamp, and Binary. MySQLdb

Copyright © 2001 O'Reilly & Associates, Inc.

supports these objects up to a point. Specifically, when MySQLdb binds parameters, it converts each paramter to a string (via `__str__`) and places it in the SQL. The Timestamp object, in particular, includes fractional seconds and MySQL considers this illegal input. The following code creates a Date for the current time and updates the database:

```
import time;
d = MySQLdb.DateFromTicks(time.time());
cursor.execute("UPDATE test SET test_date = %s WHERE test_id = 1", (d,));
```

It is important to note that MySQLdb does not properly implement the `Date()`, `Time()`, and `Timestamp()` constructors for their respective objects. You instead have to use the `DateFromTicks()`, `TimeFromTicks()`, and `TimestampFromTicks()` methods to get a reference to the desired object. The argument for each of these methods is the number of seconds since the epoch.

Proprietary Operations

In general, you should stick to the published DB-API specification when writing database code in Python. There will be some instances, however, where you may need access to MySQL-specific functionality.MySQLdb is actually built on top of the MySQL C API, and it exposes that API to programs that wish to use it. This ability is particularly useful for applications that want meta-data about the MySQL database.

Basically, MySQLdb exposes most C methods save those governing result set processing since cursors are a better interface for that functionality. Example 11-2 shows a trivial application that uses the `list_dbs()` and `list_tables()` methods from the C API to loop through all of the tables in all of the databases on the MySQL server and print out the first row from each table. Needless to say, do not run this application against a production machine.

Example 11-2. . A Python Application Using Proprietary Functionality

```
import MySQLdb;

conn = None;
try:
    conn = MySQLdb.connect(host="carthage", user="test",
                           passwd="test", db="test");
    for db in conn.list_dbs():
        for tbl in conn.list_tables(db[0]):
            cursor = conn.cursor();
            cursor.execute("SELECT * FROM " + tbl[0]);
            print cursor.fetchone();
            cursor.close();
except:
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 11-2. . A Python Application Using Proprietary Functionality

```
    if conn:
        conn.close();
```

Chapter 23, *The Python DB-API* lists the proprietary APIs exposed by MySQLdb.

Applied DB-API

So far, we have walked you through the DB-API and showed you its basic functionality. As a comprehensive database access API, it still leaves a lot to be desired and does not compete with more mature APIs like Perl DBI and Java JDBC. You should therefore except significant change in this API over time. Now, we will go through a practical example of a Python database application using the DB-API.

Our example is a batch routine that pulls stale orders from an order database and builds an XML file. Business partners can then download this XML file and import the order information into their database. Example 11-3 shows a sample generated XML file.

Example 11-3. . An XML File Containing Order Information for a Fictitious Manufacturer

```
<?xml version="1.0"?>

<order orderID="101" date="2000" salesRepID="102">
  <customer customerID="100">
    <name>Wibble Retail</name>
    <address>
      <lines>
        <line>
          1818 Harmonika Rd.
        </line>
      </lines>
      <city>Osseo</city>
      <state>MN</state>
      <country>US</country>
      <postalCode>55369</postalCode>
    </address>
  </customer>
  <lineItem quantity="2">
    <unitCost currency="USD">12.99</unitCost>
    <product productID="104">
      <name>Wibble Scarf</name>
    </product>
  </lineItem>
  <lineItem quantity="1">
    <unitCost currency="USD">24.95</unitCost>
    <product productID="105">
      <name>Wibble Hat</name>
    </product>
  </lineItem>
</order>
```

Copyright © 2001 O’Reilly & Associates, Inc.

The XML enables the our business partners to trade information about orders without having to know anything about our data model. Every night, a Python script runs to look for orders that have not been converted to XML in the last day. Any such orders are then read from the database in Figure 11-1 and then converted into XML.

FIGURE11-1.BMP

Figure 11-1. . The Data Model for the Manufacturing Database

The Python script, *xmlgen.py*, starts with a few simple imports:

```
import sys, os;
import traceback;
import MySQLdb;
```

Much of the script defines Python objects that encapsulate the business objects in the database. Example 11-4 contains the code for these business objects.

Example 11-4.

```
class Address:
    def __init__(self, l1, l2, cty, st, ctry, zip):
        self.line1 = l1;
        self.line2 = l2;
        self.city = cty;
        self.state = st;
        self.country = ctry;
        self.postalCode = zip;

    def toXML(self, ind):
        xml = ('%s<address>\r\n' % ind);
        xml = ('%s%s  <lines>\r\n' % (xml, ind));
        if self.line1:
            xml = ('%s%s    <line>\r\n%s      %s\r\n%s    </line>\r\n' %
                   (xml, ind, ind, self.line1, ind));
        if self.line2:
            xml = ('%s%s    <line>\r\n%s      %s\r\n%s    </line>\r\n' %
                   (xml, ind, ind, self.line2, ind));
        xml = ('%s%s  </lines>\r\n' % (xml, ind));
        if self.city:
            xml = ('%s%s  <city>%s</city>\r\n' % (xml, ind, self.city));
        if self.state:
            xml = ('%s%s  <state>%s</state>\r\n' % (xml, ind, self.state));
        if self.country:
            xml = ('%s%s  <country>%s</country>\r\n' % (xml,ind,self.country));
        if self.postalCode:
            xml = ('%s%s  <postalCode>%s</postalCode>\r\n' %
                   (xml, ind, self.postalCode));
        xml = ('%s%s</address>\r\n' % (xml, ind));
        return xml;

class Customer:
    def __init__(self, cid, nom, addr):
```

Example 11-4.

```
        self.customerID = cid;
        self.name = nom;
        self.address = addr;

    def toXML(self, ind):
        xml = ('%s<customer customerID="%s">\r\n' % (ind, self.customerID));
        if self.name:
            xml = ('%s%s  <name>%s</name>\r\n' % (xml, ind, self.name));
        if self.address:
            xml = ('%s%s' % (xml, self.address.toXML(ind + '  ')));
        xml = ('%s%s</customer>\r\n' % (xml, ind));
        return xml;

class LineItem:
    def __init__(self, prd, qty, cost):
        self.product = prd;
        self.quantity = qty;
        self.unitCost = cost;

    def toXML(self, ind):
        xml = ('%s<lineItem quantity="%s">\r\n' % (ind, self.quantity));
        xml = ('%s%s  <unitCost currency="USD">%s</unitCost>\r\n' %
               (xml, ind, self.unitCost));
        xml = ('%s%s' % (xml, self.product.toXML(ind + '  ')));
        xml = ('%s%s</lineItem>\r\n' % (xml, ind));
        return xml;

class Order:
    def __init__(self, oid, date, rep, cust):
        self.orderID = oid;
        self.orderDate = date;
        self.salesRep = rep;
        self.customer = cust;
        self.items = [];

    def toXML(self, ind):
        xml = ('%s<order orderID="%s" date="%s" salesRepID="%s">\r\n' %
               (ind, self.orderID, self.orderDate, self.salesRep));
        xml = ('%s%s' % (xml, self.customer.toXML(ind + '  ')));
        for item in self.items:
            xml = ('%s%s' % (xml, item.toXML(ind + '  ')));
        xml = ('%s%s</order>\r\n' % (xml, ind));
        return xml;

class Product:
    def __init__(self, pid, nom):
        self.productID = pid;
        self.name = nom;

    def toXML(self, ind):
        xml = ('%s<product productID="%s">\r\n' % (ind, self.productID));
        xml = ('%s%s  <name>%s</name>\r\n' % (xml, ind, self.name));
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 11-4.

```
        xml = ('%s%s</product>\r\n' % (xml, ind));
        return xml;
```

Each business object defines two basic methods. The first, the constructor, does nothing more than assign values to the object's attributes. The second method, `toXML()`, converts the business object to XML. So far, we have kept all database access separate from our business objects. This is a very critical design element of good database programming.

All of the database access comes in a module method called `executeBatch()`. The purpose of this method is to find out which orders need XML generated and load them from the database into business objects. It then takes those loaded orders and sends the return value of `toXML()` to an XML file. Example 11-5 shows the `executeBatch()` method.

Example 11-5. . Database Access for the XML Generator

```
def executeBatch(conn):
    try:
        cursor = conn.cursor();
        cursor.execute("SELECT ORDER_ID FROM ORDER_EXPORT " +
                       "WHERE LAST_EXPORT <> CURRENT_DATE()");
        orders = cursor.fetchall();
        cursor.close();
    except:
        print "Error retrieving orders.";
        traceback.print_exc();
        conn.close();
        exit(0);

    for row in orders:
        oid = row[0];
        try:
            cursor = conn.cursor();
            cursor.execute("SELECT CUST_ORDER.ORDER_DATE, " +
                           "CUST_ORDER.SALES_REP_ID, " +
                           "CUSTOMER.CUSTOMER_ID, " +
                           "CUSTOMER.NAME, " +
                           "CUSTOMER.ADDRESS1, " +
                           "CUSTOMER.ADDRESS2, " +
                           "CUSTOMER.CITY, " +
                           "CUSTOMER.STATE, " +
                           "CUSTOMER.COUNTRY, " +
                           "CUSTOMER.POSTAL_CODE " +
                           "FROM CUST_ORDER, CUSTOMER " +
                           "WHERE CUST_ORDER.ORDER_ID = %s " +
                           "AND CUST_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID",
                           ( oid ) );
            row = cursor.fetchone();
            cursor.close();
            addr = Address(row[4], row[5], row[6], row[7], row[8], row[9]);
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 11-5. . Database Access for the XML Generator

```
            cust = Customer(row[2], row[3], addr);
            order = Order(oid, row[0], row[1], cust);
            cursor = conn.cursor();
            cursor.execute("SELECT LINE_ITEM.PRODUCT_ID, " +
                           "LINE_ITEM.QUANTITY, " +
                           "LINE_ITEM.UNIT_COST, " +
                           "PRODUCT.NAME " +
                           "FROM LINE_ITEM, PRODUCT " +
                           "WHERE LINE_ITEM.ORDER_ID = %s " +
                           "AND LINE_ITEM.PRODUCT_ID = PRODUCT.PRODUCT_ID",
                           oid);
            for row in cursor.fetchall():
                prd = Product(row[0], row[3]);
                order.items.append(LineItem(prd, row[1], row[2]));
        except:
            print "Failed to load order: ", oid;
            traceback.print_exc();
            exit(0);

        try:
            cursor.close();
        except:
            print "Error closing cursor, continuing...";
            traceback.print_exc();

        try:
            fname = ('%d.xml' % oid);
            xmlfile = open(fname, "w");
            xmlfile.write('<?xml version="1.0"?>\r\n\r\n');
            xmlfile.write(order.toXML(''));
            xmlfile.close();
        except:
            print ("Failed to write XML file: %s" % fname);
            traceback.print_exc();

        try:
            cursor = conn.cursor();
            cursor.execute("UPDATE ORDER_EXPORT " +
                           "SET LAST_EXPORT = CURRENT_DATE() " +
                           "WHERE ORDER_ID = %s", ( oid ));
        except:
            print "Failed to update ORDER_EXPORT table, continuing";
            traceback.print_exc();
```

The first try/except block looks in the ORDER_EXPORT table for all orders that have not had XML generated in the last day. If that fails for any reason, the script bails completely.

Each row returned from `fetchall()` represents an order in need of exporting. The script therefore loops through each of the rows ands loads all of the data for the order represented by the row. Inside the `for` loop, the script executes SQL to

Copyright © 2001 O'Reilly & Associates, Inc.

pull order and customer data from the ORDER and CUSTOMER tables. With those columns in hand, it can construct Order, Customer, and Address objects for the order, its associated customer, and the customer's address. Because ORDER to LINE_ITEM is a one-to-many relationship, we need a separate query to load the LineItem objects. The next query looks for all of the line items associated with the current order ID and loads business objects for them.

With all of the data loaded into business objects, the script opens a file and writes out their XML conversion to that file. Once the write is successful, the script goes back to the database to note that the XML is now up-to-date with the database. The script then goes to the next order and continues processing until there are no more orders.

The last part missing from the script is the functionality to call executeBatch():

```
if __name__ == '__main__':
    try:
        conn = MySQLdb.connect(host='carthage', user='test', passwd='test',
                               db='Test');
    except:
        print "Error connecting to MySQL:";
        traceback.print_exc();
        exit(0);

    executeBatch(conn);
```

This script as well as the SQL to generate the tables and data behind it are available with the rest of the examples from this book at the O'Reilly Web site.

Copyright © 2001 O'Reilly & Associates, Inc.

Copyright © 2001 O'Reilly & Associates, Inc.

C API

In this book,, we examine several different programming languages, Python, Java, Perl, PHP and C. Of these languages. C/C++ is by far the most challenging. With the other languages, your primary concern is the formulation of SQL, the passing of that SQL to a function call, and the manipulation of the resulting data. C adds the very complex issue of memory management into the mix.

MySQL provides C libraries that enable the creation of MySQL database applications. MySQL derives its API very heavily from mSQL, and older database server still used to back-end many Internet web sites. However, due to it's extensive development, MySQL is much more feature-rich than mSQL. In this chapter, we will examine the details of the MySQL C API by building an object-oriented C++ API than can be used to interface C++ programs to a MySQL database server.

The API

Whether you are using C or C++, the MySQL API is the gateway into the database. How you use it, however, can be very different depending on whether you are using C or the object-oriented features of C++. C database programming must be attacked in a linear fashion, where you step through your application process to understand where the database calls are made and where clean up needs to occur. Object-oriented C++, on the other hand, requires an OO interface into the API of your choice. The objects of that API can then take on some of the responsibility for database resource management.

Table 13-1 shows the function calls of the MySQL C API. We will go into the details of how these functions are used later in the chapter. Right now, you should just take a minute to see what is available to you. Naturally, the reference section lists each of these methods with detailed prototype information, return values, and descriptions.

Table 13-1. The C API for MySQL

MySQL
mysql_affected_rows()
mysql_close()
mysql_connect()

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_create_db()

mysql_data_seek()

mysql_drop_db()

mysql_eof()

mysql_error()

mysql_fetch_field()

mysql_fetch_lengths()

mysql_fetch_row()

mysql_field_count()

mysql_field_seek()

mysql_free_result()

mysql_get_client_info()

mysql_get_host_info()

mysql_get_proto_into()

mysql_get_server_info()

mysql_init()

mysql_insert_id()

mysql_list_dbs()

mysql_list_fields()

mysql_list_processes()

mysql_list_tables()

mysql_num_fields()

mysql_num_rows()

mysql_query()

mysql_real_query()

mysql_reload()

mysql_select_db()

mysql_shutdown()

mysql_stat()

mysql_store_result()

mysql_use_result()

You may notice that many of the function names do not seem directly related to access database data. In many cases, MySQL is actually only providing an API interface into database administration functions. By just reading the function names, you might have gathered that any database application you write might minimally look something like this:

Connect

 Copyright © 2001 O'Reilly & Associates, Inc.

Select DB
Query
Fetch row
Fetch field
Close

Example 13-1 shows as simple select statement that retrieves data from a MySQL database using the MySQL C API.

Example 13-1. A Simple Program that Select All Data in a Test Database and Displays the Data

```
#include <sys/time.h>
#include <stdio.h>
#include <mysql.h>
int main(char **args) {
    MYSQL_RES *result;
    MYSQL_ROW row;
    MYSQL *connection, mysql;
    int state;

    /* connect to the MySQL database at my.server.com */
    mysql_init(&mysql);
    connection = mysql_real_connect(&mysql,
            "my.server.com", 0, "db_test", 0, 0);
    /* check for a connection error */
    if (connection == NULL) {
            /* print the error message */
            printf(mysql_error(&mysql));
            return 1;
    }
    state = mysql_query(connection,
            "SELECT test_id, test_val FROM test");
    if (state != 0) {
            printf(mysql_error(connection));
            return 1;
    }
    /* must call mysql_store_result() before can issue
       any other query calls */
    result = mysql_store_result(connection);
    printf("Rows: %d\n", mysql_num_rows(result));
    /* process each row in the result set */
    while ( ( row = mysql_fetch_row(result)) != NULL ) {
        printf("id: %s, val: %s\n",
             (row[0] ? row[0] : "NULL"),
             (row[1] ? Row[1] : "NULL"));
     }
     /* free the result set */
     mysql_free_result(result);
     /* close the connection */
     mysql_close(connection);
     printf("Done.\n");
}
```

Of the #include files, both mysql.h and stdio.h should be obvious to you. The mysql.h header contains the prototypes and variables required for MySQL, and the stdio.h the prototype for printf(). The sys/time.h header, on the other hand, is not actually used by this application. It is instead required by the mysql.h header as the MySQL file uses

definitions from sys/time.h without actually including it. To compile this program using the GNU C compiler, use the command line:

```
gcc -L/usr/local/mysql/lib -I/usr/local/mysql/include -o select select.c\
        -lmysql -lnsl -lsocket
```

You should of course substitute the directory where you have MySQL installed for /usr/local/mysql in the preceding code.

The main() function follows the steps we outlines earlier – it connects to the server, selects a database, issues a query, processes the result sets, and cleans up the resources it used. We will cover each of these steps in detail as the chapter progresses. For now you should just take the time to read the code and get a feel for what it is doing.

As we discussed earlier in the book, MySQL supports a complex level of user authentication with user name and password combinations. The first argument of the connection API for MySQL is peculiar at first inspection. It is basically a way to track all calls not otherwise associated with a connection. For example, if you try to connect and the attempt fails, you need to get the error message associated with that failure. The MySQL mysql_error() function, however, requires a pointer to a valid MySQL connection. The null connection you allocate early on provides that connection. You must, however, have a valid reference to that value for the lifetime of your application—an issue of great importance in more structured environment than a straight “connect, query, close” application. The C++ examples later in the chapter will shed more light on this issue.

Object-oriented Database Access in C++

The C API works great for procedural C development. It does not, however, fit into the object-oriented world of C++ all that well. In order to demonstrate how the API works in real code, we will spend some time using it to create a C++ API for object-oriented database development.

Because we are trying to illustrate MySQL database access, we will focus on issues specific to MySQL and not try to create the perfect general C++ API. In the MySQL world, there are three basic concepts: the connection, the result set, and the rows in the result set. We will use the concepts as the core of the object model on which our library will be based. Figure 13-1 shows the objects in a UML diagram.

The Database Connection

Database access in any environment starts with the connection. We will start our object-oriented library by abstracting on that concept and creating a Connection object. A Connection object should be able to establish a connection to the server, select the appropriate database, send queries, and return results. Example 13-2 is the header file that declares the interface for the Connection object.

Example 13-2. The Connection Class Header

 Copyright © 2001 O’Reilly & Associates, Inc.

```
#ifndef l_connection_h
#define l_connection_h

#include <sys/time.h>
#include <mysql.h>

#include “result.h”

class Connection {
private:
    int affected_rows;
    MYSQL mysql;
    MYSQL *connection;

public:
    Connection(char *, char *);
    Connection(char *, char *, char *, char *);
    ~Connection();
    void Close();
    void Connect(char *host, char *db, char *uid, char *pw);
    int GetAffectedRows();
    char *GetError();
    int IsConnected();
    Result *Query(char *);
};

#endif // l_connection_h
```

The methods the Connection class will expose to the world are uniform no matter which database engine you use. Underneath the covers, however, the class will have private data members specific to the library you compile it against. For making a connection, the only distinct data members are those that represent a database connection. As we noted earlier, MySQL uses a MYSQL pointer with an addition MYSQL value to handle establishing the connection.

Connecting to the database

Any applications we write against this API now need only to create a new Connection instance using one of the associated constructors in order to connect to the database. Similarly, an application can disconnect by delete the Connection instance. In can even ruse a Connection instance by making direct calls to Close() and Connect(). Example 13-3 shows the implementation for the constructors and the Connect() method.

Example 13-3. Connecting to MySQL Inside the Connection Class

```
#include “connection.h”
Connection::Connection(char *host, char *db) {
    connection = (MYSQL *)NULL;
    Connect( host, db, (char *)NULL, (char *)NULL);
}
Connection::Connection(char *host, char *db, char *uid, char *pw) {
    connection = (MYSQL *)NULL;
    Connect( host, db, uid, pw );
}

void Connection::Connect(char *host, char *db, char *uid, char *pw) {

    if (IsConnected() ) {
```

Copyright © 2001 O’Reilly & Associates, Inc.

```
            throw "Connection has already been established.";
    }

    mysql_init(&mysql);
    connection = mysql_real_connect(&mysql, host,
                            uid, pw,
                            db, 0, 0);
    if (!IsConnected() ) {
            throw GetError();
    }
}
```

The two constructors are designed to allow for different connection needs. In most circumstances, the username and password will be used and the four-argument constructor is called for. In some cases, however, it is not necessary to supply a username and password explicitly, and the two-argument constructor can be used. The actual database connectivity occurs in the Connect() method.

The Connect() method encapsulates all steps required for a connection. The mainly entails a call to mysql_real_connect(). If this fails, Connect() throws and exception.

Disconnecting from the database

A Connection's other logic function is to disconnect from the database and free up the resources it has hidden from the application. The functionality occurs in the Close() method. Example 13-4 provides all of the functionality for disconnecting from MySQL.

Example 13-4. Freeing up Database Resources

```
Connection::~Connection() {
    if (IsConnected()) {
            Close();
    }
}

void Connection::Close() {
    if ( !IsConnected() ) {
            return;
    }
    mysql_close(connection);
    connection = (MYSQL *)NULL;
}
```

The mysql_close() function frees up the resources associated with connections to MySQL.

Making Calls to the database

In between opening a connection and closing it, you generally want to send statements to the database. The Connection class accomplished this via a Query() method that takes a SQL statement as an argument. If the statement was a query, it return an instance of the Result class from the object model in Figure 13-1. If, on the other hand, the statement was an update, the method will return NULL and set the affected_rows value to the number of rows affected by the update. Example 13-5 shows how the Connection class handles queries against MySQL databases.

Copyright © 2001 O'Reilly & Associates, Inc.

Example 13-5. Querying the Databases

```
Result *Connection::Query(char *sql) {
    T_RESULT *res;
    int state;

    // if not connectioned, there is nothing we can do
    if ( !IsConnected() ) {
            throw "Not connected.";
    }
    // execute the query
    state = mysql_query(connection, sql);
    // an error occurred
    if ( state > 0 ) {
            throw getError();
    }
    // grab the result, if there was any
    res = mysql_store_result(connection);
    // if the result was null, if was an update or an error occurred
    if (res == (T_RESULT *)NULL ) {
            // field_count != 0 means an error occurred
            int field_count = mysql_num_fields(connection);

            if (field_count != 0) {
                    throw GetError();
            } else {
                    // store the affected rows
                    affected_rows = mysql_affected_rows(connection);
            }

            // return NULL for updates
            return (Result *)NULL;
    }
    // return a Result instance for queries
    return new Result(res);
```

The first part of a making-a-database call is calling mysql_query() with the SQL to be executed. Both APIs return a nonzero on error. The next step is to cal mysql_store_result() to check if results were generated and make those result usable by your application.

The mysql_store_result() function is used to place the results generated by a query into storage managed by the application. To trap errors from this call, you need to wrapper mysql_store_result() with some exception handling. Specifically, a NULL return value from mysql_store_result() can mean either the call was a non-query or an error occurred in storing the results. A call to mysql_num_fields() will tell you which is in fact the case. A field count not equal to zero means an error occurred. The number of affected rows, on the other hand, may be determined by a call to mysql_affected_rows(). *

* One particular situation behaves differently. MySQL is optimized for cases where you delete all records in a table. This optimization incorrectly causes some versions of MySQL to return 0 for a mysql_affacted_rows() call.

Copyright © 2001 O'Reilly & Associates, Inc.

Other Connection Behaviors

Throughout the Connection class are calls to two support methods, IsConnected() and GetError(). Testing for connection status is simple – you just check the value of the connection attribute. If should always be non-NULL for MySQL. Error messages, on the other hand, require some explanation.

Because MySQL is a multithreaded application, it needs to provide threadsafe access to any error messages. If manages to make error handling work in a multithreaded environment by hiding error messages behind the mysql_error() function. Example 13-6 shows MySQL error handling in the GetError() method as well as a connection testing in IsConnected().

Example 13-6. Reading Error and Other Support Tasks of the Connection Class

```
int Connection::GetAffectedRows() {
    return affected_rows;
}

char *Connection::GetError() {
    if (IsConnected() ) {
            return mysql_error(connection);
    } else {
            return mysql_error(&mysql);
    }
}

int Connection::IsConnected() {
    return !(!connection);
}
```

Error Handling Issues

While the error handling above is rather simple because we have encapsulated it into a simple API call in the Connection class, you should be aware of some potential pitfalls you can encounter.

MySQL manages the storage of error messages inside the API. Because you have no control over that storage, you may run into another issue regarding the persistence of error messages. In our C++ API, we are handling the error messages right after they occur – before the application makes any other database calls. If we wanted to move on with other processing before dealing with an error message, we would need to copy the error message into storage manage by our application.

Result Sets

The Result class is an abstraction on the MySQL result concept. Specifically, should provide access to the data in a result set as well as the meta-data surrounding the result set. According to the object model from Figure 13-1, our Result class will support looping through the rows of a result set and getting the row count of a result set. Example 13-7 is the header file for the Result class.

 Copyright © 2001 O’Reilly & Associates, Inc.

Exmaple XX-7. The interface for a Result Class in result.h

```
#ifndef l_result_h
#define l_result_h
#incude <sys/time.h>
#include <mysql.h>
#include "row.h"

class Result {
    private:
            int row_count;
            T_RESULT *result;
            Row *current_row;

    public:
            Result(T_RESULT *);
            ~Result();'

            void Close();
            Row *GetCurrentRow();
            int GetRowCount();
            int Next();
    };

#endif // l_result_h
```

Navigating results

Our Result class enables a developer to work through a result rest one row at a time. Upon getting a Result instance from a call to Query(), an application should call Next() and GetCurrentRow() in succession until Next() return 0. Example XX-8 shows how this functionality looks for MySQL

Example XX-8. Result Set Navigation

```
int Result::Next() {
    T_ROW row;

    if( result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }

    row = mysql_fetch_row(result);

    if (!row) {
            current_row = (Row *)NULL;
            return 0;
    } else {
            current_row = new Row(result, row);
            return 1;
    }
}

Row *Result::GetCurrentRow() {
    if( result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }
    return current_row;
}
```

The row.h header file in Example 13-10 defines T_ROW and T_RESULT as abstractions of the MySQL-specific ROW and RESULT structures. The functionality for moving to the next row is simple. You simply call mysql_fetch_row(). If the call returns NULL, there are no more rows left to process.

In an object-oriented environment, this is the only kind of navigation you should ever use. A database API in an OO world exists only to provide you access to the data – not as a tool for the manipulation of that data. Manipulation should be encapsulated in domain objects. Not all applications, however, are object-oriented applications. MySQL provides a function that allows you to move to specific rows in the database. That method is mysql_data_seek().

Cleaning up and row count

Database applications need to clean up after themselves. In talking about the Connection class, we mentioned how the result sets associated with a query are moved into storage managed by the application. The Close() method in the Result class frees the storage associated with that result. Example 13-9 shows how to clean up results and get a row count for a result set.

Example 13-9. Clean up and Row Count

```
void Result::Close() {
    if (result == (T_RESULT *)NULL) {
            return;
    }

    mysql_free_result(result);
    result = (T_RESULT *)NULL;
}

int Result::GetRowCount() {
    if (result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }
    if ( row_count > -1 ) {
            return row_count;
    } else {
            row_count = mysql_num_rows(result);
    }
}
```

Rows

An individual row from a result set is represented in our object model by the Row class. The Row class enables an application to get at individual fields in a row. Example 13-10 shows the declaration of a Row class.

Example 13-10. The Row Class from row.h

```
#ifndef l_row_h
#define l_row_h

#include <sys/types.h>
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
#include <mysql.h>
#define T_RESULT MYSQL_RES
#define T_ROW    MYSQL_ROW

class Row {
   private:
           T_RESULT *result;
           T_ROW fields;

   public:
           Row(T_RESULT *, T_ROW);
           ~Row();

           char *GetField(int);
           int GetFieldCount();
           int IsClosed();
           void Close();
};
```

Both APIs have macros for datatypes representing a result set and a row within that result set. In both APIs, a row is really nothing more than an array of strings containing the data from that row. Access to that data is controlled by indexing on that array based on the query order. For example, if your query was SELECT user_id, password FROM users, then index 0 would contain the user ID and index 1 the password. Our C++ API makes this indexing a little more user friendly. GetField(1) will actually return the first field, or fields[0]. Example 13-11 contains the full source listing for the Row class.

Example 13-11. The implementation of the Row Class

```
#include <malloc.h>

#include "row.h"

Row::Row(T_RESULT *res, T_ROW row) {
    fields = row;
    result = res;
}

Row::~Row() {
    if (!IsClosed()) {
           Close();
    }
}

void Row::Close() {
    if (! IsClosed() ) {
           throw "Row closed.";
    }
    fields = (T_ROW)NULL;
    result = (T_RESULT *)NULL;
}

int Row::GetFieldCount() {
    if ( IsClosed() ) {
           throw "Row closed.";
    }

           return mysql_num_fields(result);
}
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
// Caller should be prepared for a possible NULL
// return value from this method.
char *Row::GetField(int field) {
    if ( IsClosed() ) {
            throw "Row closed.";
    }

    if ( field < 1 || field > GetFieldCount() ) {
            throw "Field index out of bounds.";
    }
    return fields[field-1];
}

int Row::IsClosed() {
    return (fields == (T_ROW)NULL);
}
```

As example application using these C++ classes is packages with the examples from this book.

Copyright © 2001 O'Reilly & Associates, Inc.

14

Java

Java is one of the simplest languages in which you can write MySQL applications. Its database access API JDBC (Java DataBase Connectivity) is one of the more mature database-independent database access APIs in common use. Most of what we cover in this chapter can be applied to Oracle, Sybase, MS SQL Server, mSQL, and any other database engine as well as MySQL. In fact, nearly none of the MySQL-specific information in this chapter has anything to do with coding. Instead, the "proprietary" information relates only to downloading MySQL support for JDBC and configuring the runtime environment. Everything else is largely independent of MySQL, excepting for features not supported by MySQL like transactions.

In this chapter, we assume a basic understanding of the Java programming language and Java concepts. If you do not already have this background, we strongly recommend taking a look at *Learning Java* (O'Reilly & Associates, Inc.). For more details on how to build the sort of three-tier database applications we discussed in Chapter 6, *Database Applications*, take a look at *Database Programming with JDBC and Java, 2nd Edition* (O'Reilly & Associates, Inc.).

The JDBC API

Like all Java APIs, JDBC is a set of classes and interfaces that work together to support a specific set of functionality. In the case of JDBC, this functionality is naturally database access. The classes and interfaces that make up the JDBC API are thus abstractions from concepts common to database access for any kind of database. A `Connection`, for example, is a Java interface representing a database connection. Similarly, a `ResultSet` represents a result set of data returned from a SQL `SELECT` statement. Java puts the classes that form the JDBC API together in the `java.sql` package which Sun introduced in JDK 1.1.

Copyright © 2001 O'Reilly & Associates, Inc.

The underlying details of database access naturally differ from vendor to vendor. JDBC does not actually deal with those details. Most of the classes in the `java.sql` package are in fact interfaces—and thus no implementation details. Individual database vendors provide implementations of these interfaces in the form of something called a JDBC driver. As a database programmer, however, you need to know only a few details about the driver you are using—the rest you manage via the JDBC interfaces.

The first database-dependent thing you need to know is what drivers exist for your database. Different people provide different JDBC implementations for a variety of databases. As a database programmer, you want to select a JDBC implementation that will provide the greatest stability and performance for your application. Though it may seem counterintuitive, JDBC implementations provided by the database vendors generally sit at the bottom of the pack when it comes to stability and flexibility. As an Open Source project, however, MySQL relies on drivers provided by other developers in the community.

Sun has created four classifications that divide JDBC drivers based on their architectures. Each JDBC driver classification represents a trade-off between performance and flexibility.

Type 1
: These drivers use a bridging technology to access a database. The JDBC-ODBC bridge that comes with JDK 1.2 is the most common example of this kind of driver. It provides a gateway to the ODBC API. Implementations of the ODBC API in turn perform the actual database access. Though useful for learning JDBC and quick testing, bridging solutions are rarely appropriate for production environments.

Type 2
: Type 2 drivers are native API drivers. "Native API" means that the driver contains Java code that calls native C or C++ methods provided by the database vendor. In the context of MySQL, a Type 2 driver would be one that used MySQL's C API under the covers to talk to MySQL on behalf of your application. Type 2 drivers generally provide the best performance, but they do require the installation of native libraries on clients that need to access the database. Applications using Type 2 drivers have a limited degree of portability.

Type 3
: Type 3 drivers provide a client with a pure Java implementation of the JDBC API where the driver uses a network protocol to talk to middleware on the server. This middleware, in turn, performs the actual database access. The middleware may or may not use JDBC for its database access. The Type 3 architecture is actually more of a benefit to driver vendors than application

Copyright © 2001 O'Reilly & Associates, Inc.

architects since it enables the vendor to write a single implementation and claim support for any database that has a JDBC driver. Unfortunately, it has weak performance and unpredictable stability.

Type 4

Using network protocols built into the database engine, Type 4 drivers talk directly to the database using Java sockets. This is the most direct pure Java solution. Because these network protocols are almost never documented, most Type 4 drivers come from the database vendors. The Open Source nature of MySQL, however, has enabled several independent developers to write different Type 4 MySQL drivers.

Practically speaking, Type 2 and Type 4 drivers are the only viable choices for a production application. At an abstract level, the choice between Type 2 and Type 4 comes down to a single issue: Is platform independence critical? By platform independence, we mean that the application can be bundled up into a single jar and run on any platform. Type 2 drivers have a hard time with platform independence since you need to package platform-specific libraries with the application. If the database access API has not been ported to a client platform, then your application will not run on the platform. On the other hand, Type 2 drivers tend to perform better than Type 4 drivers.

Knowing the driver type provides only a starting point for making a decision about which JDBC driver to use in your application. The decision really comes down to knowing the drivers that exist for your database of choice and how they compare to each other. Table 11-1 lists the JDBC drivers available for MySQL. Of course, you are also able to use any sort of ODBC bridge to talk to MySQL as well—but we do not recommend it under any circumstance for MySQL developers.

Table 0-1. . JDBC Drivers for MySQL

Driver Name	OSI[a] License	JDBC Version	Home Page
mm (GNU)	LGPL	1.x and 2.x	http://mmmysql.sourceforge.net/
twz	no	1.x	http://www.voicenet.com/~zellert/tjFM/
Caucho	QPL	2.x	http://www.caucho.com/projects/jdbc-mysql/index.xtp

[a] Open Source Initiative (http://www.opensource.org). For drivers released under an OSI-approved license, the specific license is referenced.

Of the three MySQL JDBC drivers, twz sees the least amount of development and thus likely does not serve the interests of most programmers these days. The GNU driver (also known as mm MySQL), on the other hand, has seen constant develop-

Copyright © 2001 O'Reilly & Associates, Inc.

ment and is the most mature of the three JDBC drivers. Not to be outdone, Caucho claims significant performance benefits over the GNU driver.

The JDBC Architecture

We have already mentioned that JDBC is a set of interfaces implemented by different vendors. Figure 11-1 shows how database access works from an application's perspective. In short, the application simply makes method calls to the JDBC interfaces. Under the covers, the implementation being used by that application performs the actual database calls.

FIGURE14-1.BMP

Figure 0-1. . The JDBC architecture

JDBC is divided up into two Java packages:

- `java.sql`
- `javax.sql`

The `java.sql` package was the original package that contained all of the JDBC classes and interfaces. JDBC 2.0, however, introduced something called the JDBC Optional Package—the `javax.sql` package—with interfaces that a driver does not have to implement. In fact, the interfaces themselves are not even part of the J2SE as of JDK 1.3 (though it always has been part of the J2EE).

As it turns out, some of the functionality in the JDBC Optional Package is so important that it has been decided that it is no longer "optional" and should instead be part of the J2SE with the release of JDK 1.4. For backwards compatibility, the Optional Package classes remain in `javax.sql`.

Connecting to MySQL

JDBC represents a connection to a database through the `Connection` interface. Connecting to MySQL thus requires you to get an instance of the `Connection` interface from your JDBC driver. JDBC supports two ways of getting access to a database connection:

1. Through a JDBC Data Source
2. Using the JDBC Driver Manager

The first method—the data source—is the preferred method for connecting to a database. Data sources come from the Optional Package and thus support for them is still spotty. No matter what environment you are in, you can rely on driver manager connectivity.

Copyright © 2001 O'Reilly & Associates, Inc.

Data Source Connectivity

Data source connectivity is very simple. In fact, the following code makes a connection to any database—it is not specific to MySQL:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/myds");
Connection conn = ds.getConnection("userid", "password");
```

The first line in this example actually comes from the Java Naming and Directory Interface (JNDI*) API. JNDI is an API that provides access to naming and directory services. Naming and directory services are specialized data stores that enable you to associate related data under a familiar name. In a Windows environment, for example, network printers are stored in Microsoft ActiveDirectory under a name. In order to print to the networked color printer, a user does not need to know all of the technical details about the printer. Those details are stored in the directory. The user simply needs to know the name of the printer. The directory, in other words, stored all of the details about the printer in a directory where an application could access those details by name.

Though data source connectivity does not require a data source be stored in a directory, you will find that a directory is the most common place you will want to store data source configuration details. As a result, you can simply ask the directory for the data source by name. In the above example, the name of the data source is "jdbc/myds". JNDI enables your application to grab the data source from the directory by its name without worrying about all of the configuration details.

Though this sounds simple enough, you are probably wondering how the data source got in the directory in the first place. Someone has to put it there. Programmatically, putting the data source in the directory can be as simple as the following code:

```
SomeDataSourceClass ds = new SomeDataSourceClass();
Context ctx = new InitialContext();

// configure the DS by setting configuration attributes
ctx.bind("jdbc/myds", ds);
```

We have two bits of "magic" in this code. The first bit of magic is the `SomeDataSourceClass` class. In short, it is an implementation of the `javax.sql.DataSource` interface. In some cases, this implementation may come from the JDBC vendor—but not always. In fact, none of the MySQL drivers currently ship with a

* A full discussion of JNDI is way beyond the scope of this chapter. You minimally need a JNDI service provider (analogous to a JDBC driver) and to set some environment variables to support that service provider. You also need a directory service to talk to. If you do not have access to a directory service, you can always practice using the file system service provider available on the JNDI home page at http://java.sun.com/products/jndi or use the driver manager approach.

Copyright © 2001 O'Reilly & Associates, Inc.

`DataSource` implementation. If you are using some sort of application server like Orion or WebLogic, then those application servers will provide a `DataSource` implementation for you that will work with MySQL.

Configuring your data source depends on the properties demanded by the data source implementation class. In most cases, a data source implementation will want to know the JDBC URL and name of the `java.sql.Driver` interface implementation for the driver. We will cover these two things in the next section on driver manager connectivity.

Though we have been very vague about configuring a JDBC data source programmatically, you should not despair. You should never have to configure a JDBC data source programmatically. The vendor that provides your data source implementation should provide you with a configuration tool capable for publishing the configuration for a data source to a directory. All application servers come with such a tool. A tool of this sort will prompt you for the values it needs in order to enter a new data source in a directory and then allow you to save that configuration to the directory. Your application can then access the data source by name as shown earlier in the chapter.

Driver Manager Connectivity

One of the few implementation classes in the `java.sql`.package is the `DriverManager` class. It maintains a list of implementations of the JDBC `java.sql.Driver` class and provides you with database connections based on JDBC URLs you provide it. A JDBC URL comes in the form of *jdbc:protocol:subprotocol*. It tells a `DriverManager` which database engine you wish to connect to and it provides the `DriverManager` with enough information to make a connection.

JDBC uses the word "driver" in multiple contexts. In the lower-case sense, a JDBC driver is the collection of classes that together implement all of the JDBC interfaces and provide an application with access to at least one database. In the upper-case sense, the `Driver` is the class that implements `java.sql.Driver`. Finally, JDBC provides a `DriverManager` that can be used to keep track of all of the different `Driver` implementations.

The protocol part of the URL refers to a given JDBC driver. The protocol for the Caucho MySQL driver, for example, is *mysql-caucho* while the GNU driver uses *mysql*. The subprotocol provides the implementation-specific connection data. All MySQL drivers require a host name and database name in order to make a connection. Optionally, they may require a port if your database engine is not run-

Copyright © 2001 O'Reilly & Associates, Inc.

ning as root. Table 11-2 shows the configuration information for the MySQL JDBC drivers.

Table 0-2. . Configuration Information for MySQL JDBC Drivers

Driver	Implementation	URL
Caucho	`com.caucho.jdbc.mysql.Driver`	jdbc:mysql-caucho://HOST[:PORT]/DB
GNU	`org.gjt.mm.mysql.Driver`	jdbc:mysql://[HOST][:PORT]/DB[?PROP1=VAL1][&PROP2=VAL2]...
twz	`twz1.jdbc.mysql.jdbcMysqlDriver`	jdbc:z1MySQL://HOST[:PORT]/DB[?PROP1=VAL1][&PROP2=VAL2]...

As you can see, the URLs for the GNU driver and twz driver are very different from the Caucho driver. As a general rule, the format of the Caucho driver is actually the preferred format since the you can specify properties separately.

Your first task is to register the driver implementation with the JDBC `DriverManager`. There are two key ways to register a driver:

1. You can specify the name of the drivers you want to have registered on the command line of your application using the jdbc.drivers property: *java -Djdbc.drivers=com.caucho.jdbc.mysql.Driver MyAppClass*.
2. You can explicitly load the class in your program by doing a `new` or a `Class.forName()`: `Class.forName("twz1.jdbc.mysql.jdbcMysqlDriver").newInstance()`.

For portability's sake, we recommend that you put all configuration information in some sort of configuration file like a properties file and then load the configuration data from that configuration file. By taking this approach, your application will have no dependencies on MySQL or the JDBC driver you are using. You can simply change the values in the configuration file to move from the GNU driver to Caucho or from MySQL to Oracle.

Once you have registered your driver, you can then ask the DriverManager for a Connection. You do this by calling the getConnection() method in the driver with the information identifying the desired connection. This information minimally includes a JDBC URL, user ID, and password. You may optionally include a set of parameters:

```
Connection conn = DriverManager.getConnection("jdbc:mysql-caucho://carthage/Web", "someuser", "somepass");
```

Copyright © 2001 O'Reilly & Associates, Inc.

This code returns a connection associated with the database "Web" on the MySQL server on the machine carthage using the Caucho driver under the user ID "someuser" and authenticated with "somepass". Though the Caucho driver has the simplest URL, connecting with the other drivers is not much more difficult. They just ask that you specify connection properties such as the user ID and password as part of the JDBC URL. Table 11-3 lists the URL properties for the GNU driver and Table 11-4 lists them for the twz driver.

Table 0-3. . URL Properties for the GNU (mm) JDBC Driver

Name	Default	Description
autoReconnect	false	Causes the driver to attempt a reconnect when the connection dies.
characterEncoding	none	The Unicode encoding to use when Unicode is the character set.
initialTimeout	2	The initial time between reconnects in seconds when autoReconnect is set.
maxReconnects	3	The maximum number of times the driver should attempt a reconnect.
maxRows	0	The maximum number of rows to return for queries. 0 means return all rows.
password	none	The password to use in connecting to MySQL
useUnicode	false	Unicode is the character set to be used for the connection.
user	none	The user to use for the MySQL connection.

Table 0-4. . URL Properties for the twz JDBC Driver

Name	Default	Description
autoReX	true	Manages automatic reconnect for data update statements.
cacheMode	memory	Dictates where query results are cached.
cachePath	.	The directory to which result sets are cached if cacheMode is set t "disk".
connectionTimeout	120	The amount of time, in seconds, that a thread will wait on action by a connection before throwing an exception.
db	mysql	The MySQL database to which the driver is connected.
dbmdDB	<connection>	The MySQL database to use for database meta-data operations.
dbmdMaxRows	66536	The maximum number of rows returned by a database meta-data operation.
dbmdPassword	<connection>	The password to use for database meta-data operations.

Copyright © 2001 O'Reilly & Associates, Inc.

Table 0-4. . URL Properties for the twz JDBC Driver

Name	Default	Description
dbmdUser	<connection>	The user ID to use for database meta-data operations.
dbmdXcept	false	Exceptions will be thrown on unsupported database meta-data operations instead of the JDBC-compliant behavior of returning an empty result.
debugFile	none	Enables debugging to the specified file.
debugRead	false	When debugging is enabled, data read from MySQL is dumped to the debug file. This will severely degrade the performance of the driver.
debugWrite	false	When debugging is enabled, data written to MySQL is dumped to the debug file. This will severely degrade the performance of the driver.
host	localhost	The host machine on which MySQL is running.
maxField	*65535*	The maximum field size for data returned by MySQL. Any extra data is silently truncated.
maxRows	Integer.MAX_VALUE	The maximum number of rows that can be returned by a MySQL query.
moreProperties	none	Tells the driver to look for more properties in the named file.
multipleQuery	true	Will force the caching of the result set allowing multiple queries to be open at once.
password	none	The password used to connect to MySQL.
port	3306	The port on which MySQL is listening.
socketTimeout	none	The time in seconds that a socket connection will block before throwing an exception.
user	none	The user used to connect to MySQL.
RSLock	false	Enables locking of result sets for a statement for use in multiple threads.

As a result, connections for these two drivers commonly look like:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://carthage/
Web?user=someuser&password=somepass");
```

or for twz:

```
Connection conn =
 DriverManager.getConnection("jdbc:z1MySQL://carthage/Web?user=someuser&password="somepass");
```

Copyright © 2001 O'Reilly & Associates, Inc.

Instead of passing the basic connection properties of "user" and "password" as a second and third argument to getConnection(), GNU and twz instead pass them as part of the URL. In fact, you can pass any of the properties as part of the URL. JDBC, however, has a standard mechanism for passing driver-specific connection properties to getConnect():

```
Properties p = new Properties();
Connection conn;

p.put("user", "someuser");
p.put("password", "somepass");
p.put("useUnicode", "true");
p.put("characterEncoding", "UTF-8");
conn = DriverManager.getConnection(url, p);
```

Unfortunately, the way in which MySQL supports these optional properties is a bit inconsistent. It is thus best to go with the preferred manner for your driver, however unwieldy it makes the URLs.

Example 11-1 shows how to make a connection to MySQL using the GNU driver.

Example 0-1. A Complete Sample of Making a JDBC Connection

```
import java.sql.*;

public class Connect {
  public static void main(String argv[]) {
    Connection con = null;

    try {
      // here is the JDBC URL for this database
      String url = "jdbc:mysql://athens.imaginary.com/Web?user=someuser&password=somepass";
      // more on what the Statement and ResultSet classes do later
      Statement stmt;
      ResultSet rs;

      // either pass this as a property, i.e.
      // -Djdbc.drivers=org.gjt.mm.mysql.Driver
      // or load it here like we are doing in this example
      Class.forName("org.gjt.mm.mysql.Driver");
      // here is where the connection is made
      con = DriverManager.getConnection(url);
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 0-1. A Complete Sample of Making a JDBC Connection (continued)

```
  }
}
```

The line `con = DriverManager.getConnection(url)` makes the database connection in this example. In this case, the JDBC URL and `Driver` implementation class names are actually hard coded into this application. The only reason this is acceptable is because this application is an example driver. As we mentioned earlier, you want to get this information from a properties file or the command line in real applications.

Maintaining Portability Using Properties Files

Though our focus is on MySQL, it is good Java programming practice to make your applications completely portable. To most people, portability means that you do not write code that will run on only one platform. In the Java world, however, the word "portable" is a much stronger term. It means no hardware resource dependencies, and that means no database dependencies.

We discussed how the JDBC URL and `Driver` name are implementation dependent, but we did not discuss the details of how to avoid hard coding them. Because both are simple strings, you can pass them on the command line as runtime arguments or as parameters to applets. While that solution works, it is hardly elegant since it requires command line users to remember long command lines. A similar solution might be to prompt the user for this information; but again, you are requiring that the user remember a JDBC URL and a Java class name each time they run an application.

Properties Files

A more elegant solution than either of the above solutions would be to use a properties file. Properties files are supported by the `java.util.ResourceBundle` and its subclasses to enable an application to extract runtime specific information from a text file. For a JDBC application, you can stick the URL and `Driver` name in the properties file, leaving the details of the connectivity up to an application administrator. Example 11-2 shows a properties file that provides connection information.

Example 0-2. The SelectResource.properties File with Connection Details for a Connection

```
Driver=org.gjt.mm.mysql.Driver
URL=jdbc:mysql://athens.imaginary.com/Web?user=someuser&password=somepass
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 11-3 shows the portable `Connect` class.

Example 0-3. Using a Properties File to Maintain Portability

```
import java.sql.*;
import java.util.*;

public class Connect {
  public static void main(String argv[]) {
    Connection con = null;
    ResourceBundle bundle = ResourceBundle.getBundle("SelectResource");

    try {
      String url = bundle.getString("URL");
      Statement stmt;
      ResultSet rs;

       Class.forName(bundle.getString("Driver"));
      // here is where the connection is made
      con = DriverManager.getConnection(url);
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
  }
}
```

We have gotten rid of anything specific to MySQL or the GNU driver in the sample connection code. One important issue still faces portable JDBC developers. JDBC requires any driver to support SQL2 entry level. This is an ANSI standard for minimum SQL support. As long as you use SQL2 entry level SQL in your JDBC calls, your application will be 100% portable to other database engines. Fortunately, MySQL is SQL2 entry level, even though it does not support many of the advanced SQL2 features.

Data Sources Revisited

Earlier in the chapter, we fudged a bit on how data sources were configured. Specifically, we stated that you can configure a data source either using a tool or through Java code. In most cases you will do so using a tool. The way in which you go about configuring a data source is very dependent on the vendor providing the data source. Now that you have a greater appreciation of connection properties, you should have a good idea of what you will need to configure a data source to support MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

In order to better illustrate the way in which a data source can be set up for an application, it helps to look at a real world application environment. Orion is a J2EE compliant application server that is free for non-commercial use. In this application, it is serving up Java Server Pages (JSPs) that go against a MySQL database. The JSP makes the following JDBC call in order to do its database work:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/AddressBook");
Connection = ds.getConnection();
```

This looks familiar so far? Of course, it begs the question: how exactly does "jdbc/AddressBook" book get configured? In Orion, you configure the data source by editing a file called data-sources.xml. Here is the entry for "jdbc/AddressBook":

```
<data-source connection-driver="org.gjt.mm.mysql.Driver"
        class="com.evermind.sql.DriverManagerDataSource"
        name="AddressBook"
        url="jdbc:mysql://carthage/Address?user=test&amp;password=test"
        location="jdbc/AddressBook"/>
```

Simple Database Access

The `Connect` example did not do much. It simply showed you how to connect to MySQL. A database connection is useless unless you actually talk to the database. The simplest forms of database access are SELECT, INSERT, UPDATE, and DELETE statements. Under the JDBC API, you use your database `Connection` instance to create `Statement` instances. A `Statement` represents any kind of SQL statement. Example 11-4 shows how to insert a row into a database using a `Statement`.

Example 0-4. Inserting a Row into MySQL Using a JDBC Statement Object

```
import java.sql.*;
import java.util.*;

public class Insert {
   // We are inserting into a table that has two columns: TEST_ID (int)
   // and TEST_VAL (char(55))
   // args[0] is the TEST_ID and args[1] the TEST_VAL
   public static void main(String argv[]) {
      Connection con = null;
      ResourceBundle bundle = ResourceBundle.getBundle("SelectResource");

      try {
         String url = bundle.getString("URL");
         Statement stmt;

         Class.forName(bundle.getString("Driver"));
         // here is where the connection is made
         con = DriverManager.getConnection(url, "user", "pass");
         stmt = con.createStatement();
         stmt.executeUpdate("INSERT INTO TEST (TEST_ID, TEST_VAL) " +
```

Copyright © 2001 O’Reilly & Associates, Inc.

Example 0-4. Inserting a Row into MySQL Using a JDBC Statement Object

```
                "VALUES(" + args[0] + ", '" + args[1] + "')");
    }
    catch( SQLException e ) {
      e.printStackTrace();
    }
    finally {
      if( con != null ) {
        try { con.close(); }
        catch( Exception e ) { }
      }
    }
  }
}
```

If this were a real application, we would of course verified that the user entered an INT for the TEST_ID, that it was not a duplicate key, and that the TEST_VAL entry did not exceed 55 characters. This example nevertheless shows how simple performing an insert is. The `createStatement()` method does just what it says: it creates an empty SQL statement associated with the `Connection` in question. The `executeUpdate()` method then passes the specified SQL on to the database for execution. As its name implies, `executeUpdate()` expects SQL that will be modifying the database in some way. You can use it to insert new rows as shown earlier, or instead to delete rows, update rows, create new tables, or do any other sort of database modification.

Queries and Result Sets

Queries are a bit more complicated than updates because queries return information from the database in the form of a `ResultSet`. A `ResultSet` is an interface that represents zero or more rows matching a database query. A JDBC `Statement` has an `executeQuery()` method that works like the `executeUpdate()` method—except it returns a `ResultSet` from the database. Exactly one `ResultSet` is returned by `executeQuery()`, however, you should be aware that JDBC supports the retrieval of multiple result sets for databases that support multiple result sets. MySQL, however, does not support multiple result sets. It is nevertheless important for you to be aware of this issue in case you are ever looking at someone else's code written against another database engine. Example 11-5 shows a simple query. Figure 11-2 shows the data model behind the test table.

Example 0-5. A Simple Query

```
import java.sql.*;
import java.util.*;

public class Select {
  public static void main(String argv[]) {
    Connection con = null;
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 0-5. A Simple Query (continued)

```
    ResourceBundle bundle =
        ResourceBundle.getBundle("SelectResource");

    try {
        String url = bundle.getString("URL");
        Statement stmt;
        ResultSet rs;

        Class.forName(bundle.getString("Driver"));
        // here is where the connection is made
        con = DriverManager.getConnection(url, "user", "pass");
        stmt = con.createStatement();
        rs = stmt.executeQuery("SELECT * from TEST ORDER BY TEST_ID");
        System.out.println("Got results:");
        while(rs.next()) {
            int a= rs.getInt("TEST_ID");
            String str = rs.getString("TEST_VAL");

            System.out.print(" key= " + a);
            System.out.print(" str= " + str);
            System.out.print("\n");
        }
        stmt.close();
    }
    catch( SQLException e ) {
        e.printStackTrace();
    }
    finally {
        if( con != null ) {
            try { con.close(); }
            catch( Exception e ) { }
        }
    }
  }
}
```

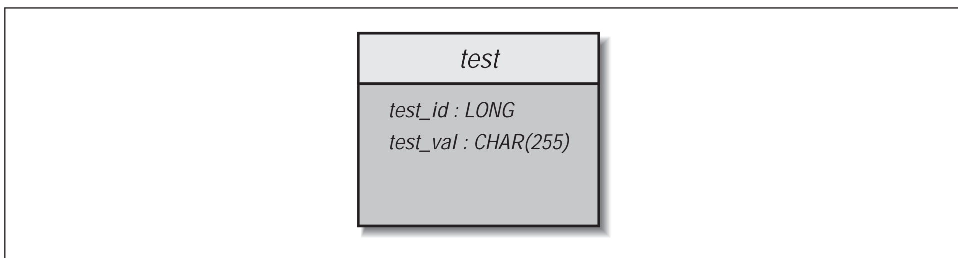


Figure 0-2. The test table from the sample database

The `Select` application executes the query and then loops through each row in the `ResultSet` using the `next()` method. Until the first call to `next()`, the `ResultSet` does not point to any row. Each call to `next()` points the `ResultSet`

Copyright © 2001 O'Reilly & Associates, Inc.

to the subsequent row. You are done processing rows when `next()` returns `false`.

You can specify that you result set is scrollable, meaning that you can move around in the result set—not just forward on a row-by-row basis. The ResultSet instances generated by a Statement are scrollable if the statement was created to support scrollable result sets. Connection enables this to happen by an alternate form of the createStatement() method:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
```

The first argument says that any result sets of the newly created statement should be scrollable. By default, a statement's result sets are not scrollable. The second argument relates to an advanced feature of JDBC, updatable result sets, that lies beyond the scope of this book.

With a scrollable result set, you can make calls to `previous()` to navigate backwards through the results and `absolute()` and `relative()` to move to arbitrary rows. Like `next()`, `previous()` moves one row through the result set, except in the opposite direction. The `previous()` method returns false when you attempt to move before the first row. Finally `absolute()` moves the result set to a specific row, whereas `relative()` moves the result set a specific number of rows before or after the current row.

Dealing with a row means getting the values for each of its columns. Whatever the value in the database, you can use the getter methods in the `ResultSet` to retrieve the column value as whatever Java datatype you like. In the `Select` application, the call to `getInt()` returned the TEST_ID column as an `int` and the call to `getString()` returned the TEST_VAL column as a `String`. These getter methods accept either the column number—starting with column 1—or the column name. You should, however, avoid retrieving values using a column name at all costs since retrieving results by column name is many, many times slower than retrieving them by column number.

One area of mismatch between Java and MySQL lies in the concept of a SQL `NULL`. SQL is specifically able to represent some data types as null that Java cannot represent as null. In particular, Java has no way of representing primitive data types as nulls. As a result, you cannot immediately determine whether a 0 returned from MySQL through `getInt()` really means a 0 is in that column or if no value is in that column. JDBC addresses this mismatch through the `wasNull()` method.

As its name implies, `wasNull()` returns `true` if the last value fetched was SQL `NULL`. For calls returning a Java object, the value will generally be `NULL` when a SQL `NULL` is read from the database. In these instances, `wasNull()` may appear somewhat redundant. For primitive datatypes, however, a valid value—like 0—

Copyright © 2001 O'Reilly & Associates, Inc.

may be returned on a fetch. The `wasNull()` method gives you a way to see if that value was `NULL` in the database.

Error Handling and Clean Up

All JDBC method calls can throw SQLException or one of its subclasses if something happens during a database call. Your code should be set up to catch this exception, deal with it, and clean up any database resources that have been allocated. Each of the JDBC classes mentioned so far has a `close()` method associated with it. Practically speaking, however, you only really need to make sure you close things whose calling process might remain open for a while. In the examples we have seen so far, you only really need to close your database connections. Closing the database connection closes any statements and result sets associated with it automatically. If you intend to leave a connection open for any period of time, however, it is a good idea to go ahead and close the statements you create using that connection when you finish with them. In the JDBC examples you have seen, this clean up happens in a finally clause. You do this since you want to make sure to close the database connection no matter what happens.

Dynamic Database Access

So far we have dealt with applications where you know exactly what needs to be done at compile time. If this were the only kind of database support that JDBC provided, no one could ever write tools like the *mysql* interactive command line tool that determines SQL calls at runtime and executes them. The JDBC `Statement` class provides the `execute()` method for executing SQL that may be either a query or an update. Additionally, `ResultSet` instances provide runtime information about themselves in the form of an interface called `ResultSetMetaData` which you can access via the `getMetaData()` call in the `ResultSet`.

Meta Data

The term meta data sounds officious, but it is really nothing other than extra data about some object that would otherwise waste resources if it were actually kept in the object. For example, simple applications do not need the name of the columns associated with a `ResultSet`—the programmer probably knew that when the code was written. Embedding this extra information in the `ResultSet` class is thus not considered by JDBC's designers to be core to the functionality of a `ResultSet`. Data such as the column names, however, is very important to some database programmers—especially those writing dynamic database access. The JDBC designers provide access to this extra information—the meta data—via the `ResultSetMetaData` interface. This class specifically provides:

Copyright © 2001 O'Reilly & Associates, Inc.

- The number of columns in a result set
- Whether NULL is a valid value for a column
- The label to use for a column header
- The name for a given column
- The source table for a given column
- The datatype of a given column

Example 11-6 shows some of the source code from a command line tool like mysql that accepts arbitrary user input and sends it to MySQL for execution. The rest of the code for this example can be found at the O'Reilly Web site with the rest of the examples from this book.

Example 0-6. An Application for Executing Dynamic SQL

```
import java.sql.*;

public class Exec {
   public static void main(String args[]) {
      Connection con = null;
      String sql = "";

      for(int i=0; i<args.length; i++) {
         sql = sql + args[i];
         if( i < args.length - 1 ) {
            sql = sql + " ";
         }
      }
      System.out.println("Executing: " + sql);
      try {
         Class.forName("com.caucho.jdbc.mysql.Driver").newInstance();
         String url = "jdbc:mysql-caucho://athens.imaginary.com/TEST";
         con = DriverManager.getConnection(url, "test", "test");
         Statement s = con.createStatement();

         if( s.execute(sql) ) {
            ResultSet r = s.getResultSet();
            ResultSetMetaData meta = r.getMetaData();
            int cols = meta.getColumnCount();
            int rownum = 0;

            while( r.next() ) {
               rownum++;
               System.out.println("Row: " + rownum);
               for(int i=0; i<cols; i++) {
                  System.out.print(meta.getColumnLabel(i+1) + ": "
                           + r.getObject(i+1) +  ", ");
               }
               System.out.println("");
            }
         }
```

Copyright © 2001 O'Reilly & Associates, Inc.

Example 0-6. An Application for Executing Dynamic SQL

```
        else {
          System.out.println(s.getUpdateCount() + " rows affected.");
        }
        s.close();
        con.close();
      }
      catch( Exception e ) {
        e.printStackTrace();
      }
      finally {
        if( con != null ) {
          try { con.close(); }
          catch( SQLException e ) { }
        }
      }
    }
}
```

Each result set provides a `ResultSetMetaData` instance via the `getMetaData()` method. In the case of dynamic database access, we need to find out the how many columns are in a result set so that we are certain to retrieve each column as well as the names of each of the columns for display to the user. The meta data for our result set provides all of this information via the `getColumnCount()` and `getColumnLabel()` methods.

Processing Dynamic SQL

The concept introduced in Example 11-6 is the dynamic SQL call. Because we do not know whether we will be processing a query or an update, we need to pass the SQL call through the `execute()` method. This method returns `true` if the statement returned a result set or `false` if none was produced. In the example, if it returns `true`, the application gets the returned `ResultSet` through a call to `getResultSet()`. The application can then go on to do normal result set processing. If, on the other hand, the statement performed some sort of database modification, you can call `getUpdateCount()` to find out how many rows were modified by the statement.

A Guest Book Servlet

You have probably heard quite a bit of talk about Java applets. We discussed in Chapter 6, however, how doing database access in the client is a really bad idea. We have packaged with the examples in this book a servlet that uses the JDBC knowledge we have discussed in this chapter to store the comments from visitors to a Web site in a database and then display the comments in the database. While servlets are not in themselves part of the three-tier solution we discussed in Chapter 6,

Copyright © 2001 O'Reilly & Associates, Inc.

this example should provide a useful example of how JDBC can be used. For this example, all you need to know about servlets is that the `doPost()` method handles HTTP POST events and `doGet()` handles HTTP GET events. The rest of the code is either simple Java code or an illustration of the database concepts from this chapter. You can see the servlet in action at *http://www.imaginary.com/~george/guestbook.shtml*.

Copyright © 2001 O'Reilly & Associates, Inc.

Copyright © 2001 O'Reilly & Associates, Inc.

16

SQL Syntax for MySQL

In this chapter, we cover the full range of SQL supported by MySQL. If you are interested in compatibility with other SQL databases, MySQL supports the ANSI SQL2 standard. In that case, you should avoid using any proprietary MySQL extensions to the SQL standard.

Basic Syntax

SQL is a kind of controlled English language consisting of verb phrases. These verb phrases begin with a SQL command followed by other SQL keywords, literals, identfiers, or punctuation. Keywords are never case sensitive. Identifiers for database names and table names are case sensitive when the underlying file system is case sensitive (all UNIX except Mac OS X) and case insensitive when the underlying file system is case insensitive (Mac OS X and Windows). You should, however, avoid referring to the same database or table name in a single SQL statement using different cases—even if the underlying operating system is case insensitive. For example, the following SQL is troublesome:

```
SELECT TBL.COL FROM tbl;
```

Table aliases are case sensitive, but column aliases are case insensitive.

If all of this case sensitivity nonsense is annoying to you, you can force MySQL to convert all table names to lower case by starting *mysqld* with the argument *-O lower_case_table_names=1*.

Literals

Literals come in the following varieties:

Copyright © 2001 O'Reilly & Associates, Inc.

String Literals

String literals may be enclosed either by single quotes or double quotes. If you wish to be ANSI compatible, you should always use single quotes. Within a string literal, you may represent special characters through escape sequences. An escape sequence is a backslash followed by another character to indicate to MySQL that the second character has a meaning other than its normal meaning. Table 16-1 shows the MySQL escape sequences. Quotes can also be escaped by doubling them up: `'This is a ''quote'''`. However, you do not need to double up on single quotes when the string is enclosed by double quotes: `"This is a 'quote'"`.

Binary Literals

Like string literals, binary literals are enclosed in single or double quotes. You must use escape sequences in binary data to escape `NUL` (ASCII 0), `"` (ASCII 34), `'` (ASCII 39), and \ (ASCII 92).

Number Literals

Numbers appear as a sequence of digits. Negative numbers are preceded by a – sign and a `.` indicates a decimal point. You may also use scientific notation: -45198.2164e+10.

Hexadecimal Literals

MySQL also supports the use of hexadecimal literals in SQL. The way in which that hexadecimal is interpreted is dependent on the context. In a numeric context, the hexadecimal literal is treated is a numeric value. Absent of a numeric context, it is treated as a binary value. This 0x1 + 1 is 2, but 0x4d7953514c by itself is 'MySQL'.

Null

The special keyword `NULL` signifies a null literal in SQL. In the context of import files, the special escape sequence `\N` signifies a null value.

Table 16-1. . MySQL Escape Sequences

Escape Sequence	Value
`\0`	NUL
`\'`	Single quote
`\"`	Double quote
`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\z`	Ctrl-z (workaround for Windows use of ctrl-z as EOF)
`\\`	Backslash

Copyright © 2001 O'Reilly & Associates, Inc.

Table 16-1. . MySQL Escape Sequences

Escape Sequence	Value
\%	Percent sign (only in contexts where a percent sign would be interpreted as a wild card)
_	Underscore (only in contexts where an underscore would be interpreted as a wild card)

Identifiers

Identifiers are names you make up to reference database objects. In MySQL, database objects consist of databases, tables, and columns. These objects fit into a hierarchical namespace whose root element is the database in question. You can reference any given object on a MySQL server—assuming you have the proper rights—in one of the following conventions:

Absolute Naming

Absolute naming is specifying the full tree of the object you are referencing. For example, the column `BALANCE` in the table `ACCOUNT` in the database `BANK` would be referenced absolutely as:

```
BANK.ACCOUNT.BALANCE
```

Relative Naming

Relative naming allows you to specify only part of the object's name with the rest of the name being assumed based on your current context. For example, if you are currently connected to the `BANK` database, you can reference the `BANK.ACCOUNT.BALANCE` column simply as `ACCOUNT.BALANCE`. In a SQL query where you have specified you are selecting from the `ACCOUNT` table, you can reference the column using only `BALANCE`. You must provide an extra layer of context whenever relative naming might result in ambiguity. An example of such ambiguity would be a `SELECT` statement pulling from two tables that both have `BALANCE` columns.

Aliasing

Aliasing enables you to reference an object using an alternate name that helps avoid both ambiguity and the need to fully qualify a long name.

In general, MySQL allows you to use any character in an identifier.* This rule is limited, however, for databases and tables since these values must be treated as files on the local file system. You can therefore use only characters valid for the underlying file system's file naming conventions in a database or table name. Spe-

* Older versions of MySQL limited identifiers to valid alphanumeric characters from the default character set as well as $ and _.

Copyright © 2001 O'Reilly & Associates, Inc.

cifically, you may not use / or . in a database or table name. You can never use NUL (ASCII 0) or ASCII 255 in an identifier.

Given these rules, it is very easy to shoot yourself in the foot when naming things. As a general rule, it is a good idea to stick to alphanumeric characters from whatever character set you are using.

When an identifier is also a SQL keyword, you must enclose the identifier in backticks:

```
CREATE TABLE `select` ( `table` INT NOT NULL PRIMARY KEY AUTO_INCREMENT);
```

Since MySQL 3.23.6, MySQL supports the quoting of identifiers using both backticks and double quotes. For ANSI compatibility, however, you should use double quotes for quoting identifiers. You must, however, be running MySQL in ANSI mode.

Comments

You can introduce comments in your SQL to specify text that should not be interpreted by MySQL. This is particularly useful in batch scripts for creating tables and loading data. MySQL specifically supports three kinds of commenting: C, shell-script, and ANSI SQL commenting.

C commenting treats anything between /* and */ as comments. Using this form of commenting, your comments can span multiple lines. For example:

```
/*
 * Creates a table for storing customer account information.
*/
DROP TABLE IF EXISTS ACCOUNT;

CREATE TABLE ACCOUNT ( ACCOUNT_ID BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                      BALANCE DECIMAL(9,2) NOT NULL );
```

Within C comments, MySQL still treats single quotes and double quotes as a start to a string literal. In addition, a semi-colon in the comment will cause MySQL to think you are done with the current statement.

Shell-script commenting treats anything from a # character to the end of a line as a comment:

```
CREATE TABLE ACCOUNT ( ACCOUNT_ID BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                      BALANCE DECIMAL(9,2) NOT NULL ); # Not null ok?
```

MySQL does not really support ANSI SQL commenting, but it comes close. ANSI SQL commenting is -- to the end of a line. MySQL supports two dashes and a space ('-- ') followed by the comment. The space is the non-ANSI part:

```
DROP TABLE IF EXISTS ACCOUNT; -- Drop the table if it already exists
```

Copyright © 2001 O'Reilly & Associates, Inc.

SQL Commands

ALTER TABLE

Syntax

```
ALTER [IGNORE] TABLE table action_list
```

Description

The `ALTER` statement covers a wide range of actions that modify the structure of a table. This statement is used to add, change, or remove columns from an existing table as well as to remove indexes. To perform modifications on the table, MySQL creates a copy of the table and changes it, meanwhile queuing all table altering queries. When the change is done, the old table is removed and the new table put it its place. At this point the queued queries are performed. As a safety precaution, if any of the queued queries create duplicate keys that should be unique, the `ALTER` statement is rolled back and cancelled. If the `IGNORE` keyword is present in the statement, duplicate unique keys are ignored and the `ALTER` statement proceeds as if normal. Be warned that using `IGNORE` on an active table with unique keys is inviting table corruption. Possible actions include:

`ADD [COLUMN]` *`create_clause`* `[FIRST | AFTER` *`column`*`]`
: Adds a new column to the table. The *create_clause* is simply the SQL that would define the column in a normal table creation. The column will be created as the first column if the `FIRST` keyword is specified. Alternately, you can use the `AFTER` keyword to specify which column it should be added after. If neither `FIRST` nor `AFTER` is specified, then the column is added at the end of the table's column list. You may add multiple columns at once by separating create clauses by commas.

`ADD INDEX [`*`name`*`] (`*`column`*`, ...)`
: Adds an index to the altered table. If the name is omitted, one will be chosen automatically by MySQL.

`ADD PRIMARY KEY (`*`column`*`, ...)`
: Adds a primary key consisting of the specified columns to the table. An error occurs if the table already has a primary key.

`ADD UNIQUE[`*`name`*`] (`*`column`*`, ...)`
: Adds a unique index to the altered table similar to the `ADD INDEX` statement.

Copyright © 2001 O'Reilly & Associates, Inc.

ALTER [COLUMN] *column* SET DEFAULT *value*

Assigns a new default value for the specified column. The COLUMN keyword is optional and has no effect.

ALTER [COLUMN] *column* DROP DEFAULT

Drops the current default value for the specified column. A new default value will be assigned to the column based on the CREATE statement used to create the table. The COLUMN keyword is optional and has no effect.

CHANGE [COLUMN] *column create_clause*
MODIFY [COLUMN] *create_clause*

Alters the definition of a column. This statement is used to change a column from one type to a different type while affecting the data as little as possible. The create clause is a full clause as specified in the CREATE statement. This includes the name of the column. The MODIFY version is the same as CHANGE if the new column has the same name as the old. The COLUMN keyword is optional and has no effect. MySQL will try its best to perform a reasonable conversion. Under no circumstance will MySQL give up and return an error when using this statement; a conversion of some sort will always be done. With this in mind you should (1) make a backup of the data before the conversion and (2) immediately check the new values to see if they are reasonable.

DROP [COLUMN] *column*

Deletes a column from a table. This statement will remove a column and all of its data from a table permanently. There is no way to recover data destroyed in this manner other than from backups. All references to this column in indices will be removed. Any indices where this was the sole column will be destroyed as well. (The COLUMN keyword is optional and has no effect.)

DROP PRIMARY KEY

Drops the primary key from the table. If no primary key is found in the table, the first unique key is deleted.

DROP INDEX *key*

Removes an index from a table. This statement will completely erase an index from a table. This statement will not delete or alter any of the table data itself, only the index data. Therefore, an index removed in this manner can be recreated using the ALTER TABLE ... ADD INDEX statement.

Copyright © 2001 O'Reilly & Associates, Inc.

RENAME [AS] *new_table*
RENAME [TO] *new_table*

Changes the name of the table. This operation does not affect any of the data or indices within the table, only the table's name. If this statement is performed alone, without any other `ALTER TABLE` clauses, MySQL will not create a temporary table as with the other clauses, but simply perform a fast Unix-level rename of the table files.

ORDER BY *column*

Forces the table to be re-ordered by sorting on the specified column name. The table will no longer be in this order when new rows are inserted. This option is useful for optimizing tables for common sorting queries.

table_options

Enables a redefinition of the tables options such as the table type.

Multiple `ALTER` statements may be combined into one using commas as in the following example:

```
ALTER TABLE mytable DROP myoldcolumn, ADD mynewcolumn INT
```

MySQL also provides support for actions to alter the `FOREIGN KEY`, but they do nothing. . The syntax is there simply for compatibility with other databases.

To perform any of the `ALTER TABLE` actions, you must have `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `CREATE`, and `DROP` privileges for the table in question.

Examples

```
# Add the field 'address2' to the table 'people' and make
# it of type 'VARCHAR' with a maximum length of 200.
ALTER TABLE people ADD COLUMN address2 VARCHAR(100)
# Add two new indexes to the 'hr' table, one regular index for the
# 'salary' field and one unique index for the 'id' field. Also, continue
# operation if duplicate values are found while creating
# the 'id_idx' index (very dangerous!).
ALTER TABLE hr ADD INDEX salary_idx ( salary )
ALTER IGNORE TABLE hr ADD UNIQUE id_idx ( id )
# Change the default value of the 'price' field in the
# 'sprockets' table to $19.95.
ALTER TABLE sprockets ALTER price SET DEFAULT '$19.95'
# Remove the default value of the 'middle_name' field in the 'names' table.
ALTER TABLE names ALTER middle_name DROP DEFAULT
# Change the type of the field 'profits' from its previous value (which was
# perhaps INTEGER) to BIGINT.
ALTER TABLE finanaces CHANGE COLUMN profits profits BIGINT
# Remove the 'secret_stuff' field from the table 'not_private_anymore'
ALTER TABLE not_private_anymore DROP secret_stuff
# Delete the named index 'id_index' as well as the primary key from the
# table 'cars'.
ALTER TABLE cars DROP INDEX id_index, DROP PRIMARY KEY
# Rename the table 'rates_current' to 'rates_1997'
ALTER TABLE rates_current RENAME AS rates_1997
```

Copyright © 2001 O'Reilly & Associates, Inc.

CREATE DATABASE

Syntax

```
CREATE DATABASE dbname
```

Description

Creates a new database with the specified name. You must have the proper privileges to create the database. Running this command is the same as running the *mysqladmin create* utility.

Example

```
CREATE DATABASE Bank;
```

CREATE FUNCTION

Syntax

```
CREATE [AGGREGATE] FUNCTION name
RETURNS return_type SONAME library
```

Description

The `CREATE FUNCTION` statement allows MySQL statements to access precompiled executable functions. These functions can perform practically any operation, since they are designed and implemented by the user. The return value of the function can be `STRING`, for character data; `REAL`, for floating point numbers; or `INTEGER` for integer numbers. MySQL will translate the return value of the C function to the indicated type. The library file that contains the function must be a standard shared library that MySQL can dynamically link into the server.

Example

```
CREATE FUNCTION multiply RETURNS REAL SONAME mymath
```

CREATE INDEX

Syntax

```
CREATE [UNIQUE] INDEX name ON table (column, ...)
```

Description

The `CREATE INDEX` statement is provided for compatibility with other implementations of SQL. In older versions of SQL this statement does nothing. As of 3.22, this statement is equivalent to the `ALTER TABLE ADD INDEX` statement. To per-

Copyright © 2001 O'Reilly & Associates, Inc.

form the CREATE INDEX statement, you must have INDEX privileges for the table in question.

Example

```
CREATE UNIQUE INDEX TransIDX ON Translation ( language, locale, code );
```

CREATE TABLE

Syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table
(create_clause, ...) [table_options]
[[IGNORE|REPLACE] select]
```

Description

The CREATE TABLE statement defines the structure of a table within the database. This statement is how all MySQL tables are created. If the TEMPORARY keyword is used, the table exists only as long as the current client connection exists, unless it is dropped first

The IF NOT EXISTS clause tells MySQL to create the table only if the table does not already exist. If the table does exist, nothing happens. If the table exists and IF NOT EXISTS and TEMPORARY are not specified, an error will occur. If TEMPORARY is specified and the table exists but IF NOT EXISTS is not specified, the existing table will simply be invisible to this client for the duration of the new temporary table's life.

This statement consists of the name of the new table followed by any number of field definitions. The syntax of a field definition is the name of the field followed by its type, followed by any modifiers (e.g., `name char(30) not null`). MySQL supports the data types described in Chapter 17. The allowed modifiers are:

DEFAULT *value*
: This attribute assigns a default value to a field. If a row is inserted into the table without a value for this field, this value will be inserted. If a default is not defined, a null value is inserted unless the field is defined as NOT NULL in which case MySQL picks a value based on the type of the field.

NOT NULL
: This attribute guarantees that every entry in the column will have some non-NULL value. Attempting to insert a NULL value into a field defined with NOT NULL will generate an error.

NULL
: This attribute specifies that the field is allowed to contain NULL values. This is the default if neither this nor the NOT NULL modifier are specified. Fields that

Copyright © 2001 O'Reilly & Associates, Inc.

are contained within an index cannot contain the NULL modifier. (It will be ignored, without warning, if it does exist in such a field.)

`PRIMARY KEY`

This attribute automatically makes the field the primary key (see later) for the table. Only one primary key may exist for a table. Any field that is a primary key must also contain the NOT NULL modifier.

`REFERENCES` *table* `[(`*column*`, . . .)] [MATCH FULL | MATCH PARTIAL] [ON DELETE` *option*`] [ON UPDATE` *option*`]`

This attribute currently has no effect. MySQL understands the full references syntax but does not implement its behavior. The modifier is included to make it easier to import SQL from different SQL sources. In addition, this functionality may be included in a future release of MySQL.

MySQL supports the concept of an index of a table, as described in Chapter 8, *Database Design*. Indexes are created by means of special "types" that are included with the table definition:

`FULLTEXT (` *column*`, ... )`

Since MySQL 3.23.23, MySQL has supported full text indexing. To create a full text index, use the `FULLTEXT` keyword:

```
CREATE TABLE Item ( itemid INT NOT NULL PRIMARY KEY,
                    name VARCHAR(25) NOT NULL,
                    description TEXT NOT NULL,
                    FULLTEXT ( name, description )
);
```

`KEY/INDEX [`*name*`] (`*column*`, ...)`

Creates a regular index of all of the named columns (`KEY` and `INDEX`, in this context, are synonyms). Optionally the index may be given a name. If no name is provided, a name is assigned based on the first column given and a trailing number, if necessary, for uniqueness. If a key contains more than one column, leftmost subsets of those columns are also included in the index. Consider the following index definition.

```
INDEX idx1 ( name, rank, serial );
```

When this index is created, the following groups of columns will be indexed:

— `name, rank, serial`

— `name, rank`

— `name`

`PRIMARY KEY`

Creates the primary key of the table. A primary key is a special key that can be defined only once in a table. The primary key is a `UNIQUE` key with the

Copyright © 2001 O'Reilly & Associates, Inc.

name "PRIMARY." Despite it's privileged status, in function it is the same as every other unique key.

`UNIQUE [name] (column, ...)`
Creates a special index where every value contained in the index (and therefore in the fields indexed) must be unique. Attempting to insert a value that already exists into a unique index will generate an error. The following would create a unique index of the "nicknames" field:

```
UNIQUE (nicknames);
```

when indexing character fields (CHAR, VARCHAR and their synonyms only), it is possible to index only a prefix of the entire field. For example, this following will create an index of the numeric field 'id' along with the first 20 characters of the character field 'address':

```
INDEX adds ( id, address(20) );
```

When performing any searches of the field 'address', only the first 20 characters will be used for comparison unless more than one match is found that contains the same first 20 characters, in which case a regular search of the data is performed. Therefore, it can be a big performance bonus to index only the number of characters in a text field that you know will make the value unique.

Fields contained in an index must be defined with the `NOT NULL` modifier. When adding an index as a separate declaration, MySQL will generate an error if `NOT NULL` is missing. However, when defining the primary key by adding the `PRIMARY KEY` modifier to the field definition, the `NOT NULL` modifier is automatically added (without a warning) if it is not explicitly defined.

In addition to the above, MySQL supports the following special "types":

- FOREIGN KEY *(name (column, [column2, . . .])*
- CHECK

These keywords do not actually perform any action. They exist so that SQL exported from other databases can be more easily read into MySQL. Also, some of this missing functionality may be added into a future version of MySQL.

As of MySQL 3.23, you can specify table options at the end of a `CREATE TABLE` statement. These options are:

`AUTO_INCREMENT = start`
Specifies the first value to be used for an `AUTO_INCREMENT` column.

`AVG_ROW_LENGTH = length`
An option for tables containing large amounts of variable-length data. The average row length is an optimization hint to help MySQL manage this data.

Copyright © 2001 O'Reilly & Associates, Inc.

`CHECKSUM = 0` *or* `1`
: When set to 1, this option forces MySQL to maintain a checksum for the table to improve data consistency. This option creates a performance penalty.

`COMMENT =` *comment*
: Provides a comment for the table. The comment may not exceed 60 characters.

`DELAY_KEY_WRITE = 0` *or* `1`
: For MyISAM tables only. When set, this option delays key table updates until the table is closed.

`MAX_ROWS =` *rowcount*
: The maximum number of rows you intend to store in the table.

`MIN_ROWS =` *rowcount*
: The minimum number of rows you intend to store in the table.

`PACK_KEYS = 0` *or* `1`
: For MyISAM and ISAM tables only. This option provides a performance booster for heavy-read tables. Set to 1, this option causes smaller keys to be created and thus slows down writes while speeding up reads.

`PASSWORD = '`*password*`'`
: Only available to MySQL customers with special commercial licenses. This option uses the specified password to encrypt the table's *.frm* file. This option has no effect on the standard version of MySQL.

`ROW_FORMAT = DYNAMIC` *or* `STATIC`
: For MyISAM tables only. Defines how the rows should be stored in a table.

`TYPE =` *rowtype*
: Specifies the table type of the database. If the selected table type is not available, then the closest table type available is used. For example, BDB is not available yet for Mac OS X. If you specified `TYPE=BDB` on a Mac OS X system, MySQL will instead create the table as a MyISAM table. Table 16-2 contains a list of supported table types and their advanatages. For a more complete discussion of MySQL tables types, see the MySQL table type reference.

Table 16-2. . MySQL Table Types

Type	Transactional	Description
BDB	yes	Transaction-safe tables with page locking.
Berkeley_db	yes	Alias for BDB
HEAP	no	Memory-based table. Not persistent.
ISAM	no	Ancient format. Replaced by MyISAM.
InnoDB	yes	Transaction-safe tables with row locking.
MERGE	no	A collection of MyISAM tables merged as a single table.

Copyright © 2001 O'Reilly & Associates, Inc.

Table 16-2. . MySQL Table Types

Type	Transactional	Description
MyISAM	no	A newer table type to replace ISAM that is portable.

You must have CREATE privileges on a database to use the CREATE TABLE statement.

Examples

```
# Create the new empty database 'employees'
CREATE DATABASE employees;
# Create a simple table
CREATE TABLE emp_data ( id INT, name CHAR(50) );
# Make the function make_coffee (which returns a string value and is stored
# in the myfuncs.so shared library) available to MySQL.
CREATE FUNCTION make_coffee RETURNS string SONAME "myfuncs.so";
```

DELETE

Syntax

```
DELETE [LOW_PRIORITY] FROM table [WHERE clause] [LIMIT n]
```

Description

Deletes rows from a table. When used without a WHERE clause, this will erase the entire table and recreate it as an empty table. With a clause, it will delete the rows that match the condition of the clause. This statement returns the number of rows deleted to the user.

As mentioned above, not including a WHERE clause will erase this entire table. This is done using an efficient method that is much faster than deleting each row individually. When using this method, MySQL returns 0 to the user because it has no way of knowing how many rows it deleted. In the current design, this method simply deletes all of the files associated with the table except for the file that contains the actual table definition. Therefore, this is a handy method of zeroing out tables with unrecoverably corrupt data files. You will lose the data, but the table structure will still be in place. If you really wish to get a full count of all deleted tables, use a WHERE clause with an expression that always evaluates to true:

```
DELETE FROM TBL WHERE 1 = 1;
```

The LOW_PRIORITY modifier causes MySQL to wait until no clients are reading from the table before executing the delete.

The LIMIT clauses establishes the maximum number of rows that will be deleted in a single shot.

You must have DELETE privileges on a database to use the DELETE statement.

Copyright © 2001 O'Reilly & Associates, Inc.

Examples

```
# Erase all of the data (but not the table itself) for the table 'olddata'.
DELETE FROM olddata
# Erase all records in the 'sales' table where the 'syear' field is '1995'.
DELETE FROM sales WHERE syear=1995
```

DESCRIBE

Syntax

DESCRIBE *table* [*column*]
DESC *table* [*column*]

Description

Gives information about a table or column. While this statement works as advertised, its functionality is available (along with much more) in the SHOW statement. This statement is included solely for compatibility with Oracle SQL. The optional column name can contain SQL wildcards, in which case information will be displayed for all matching columns.

Example

```
# Describe the layout of the table 'messy'
DESCRIBE messy
# Show the information about any columns starting
# with 'my_' in the 'big' table.
# Remember: '_' is a wildcard, too, so it must be
# escaped to be used literally.
DESC big my\_%
```

DESC

Synonym for DESCRIBE.

DROP DATABASE

Syntax

DROP DATABASE [IF EXISTS] *name*

Description

Permanently remove a database from the MySQL. Once you execute this statement, none of the tables or data that made up the database are available. All of the support files for the database are deleted from the file system. The number of files deleted will be returned to the user. Because three files represent most tables, the number returned is usually the number of tables times three. This is equivalent to running the *mysqladmin drop* utility. As with running *mysqladmin*, you must be the administrative user for MySQL (usually root or mysql) to perform this state-

Copyright © 2001 O'Reilly & Associates, Inc.

ment.You may use the IF EXISTS clause to prevent any error message that would result from an attempt to drop a non-existent table.

DROP FUNCTION

Syntax

```
DROP FUNCTION name
```

Description

Will remove a user defined function from the running MySQL server process. This does not actually delete the library file containing the function. You may add the function again at any time using the CREATE FUNCTION statement. In the current implementation DROP FUNCTION simply removes the function from the function table within the MySQL database. This table keeps track of all active functions.

DROP INDEX

Syntax

```
DROP INDEX idx_name ON tbl_name
```

Description

Provides for compatibility with other SQL implementations. In older versions of MySQL, this statement does nothing. As of 3.22, this statement is equivalent to ALTER TABLE ... DROP INDEX. To perform the DROP INDEX statement, you must have SELECT, INSERT, DELETE, UPDATE, CREATE, and DROP privileges for the table in question.

DROP TABLE

Syntax

```
DROP TABLE [IF EXISTS] name [, name2, ...]
```

Description

Will erase an entire table permanently. In the current implementation, MySQL simply deletes the files associated with the table. As of 3.22, you may specify IF EXISTS to make MySQL not return an error if you attempt to remove a table that does not exist. You must have DELETE privileges on the table to use this statement.

Copyright © 2001 O'Reilly & Associates, Inc.

`DROP` is by far the most dangerous SQL statement. If you have drop privileges, you may permanently erase a table or even an entire database. This is done without warning or confirmation. The only way to undo a `DROP` is to restore the table or database from backups. The lessons to be learned here are: (1) always keep backups; (2) don't use `DROP` unless you are really sure; and (3) always keep backups.

EXPLAIN

Syntax

EXPLAIN [*table_name* | *sql_statement*]

Description

Used with a table name, this command is an alias for `SHOW COLUMNS FROM` *`table_name`*.

Used with a SQL statement, this command displays verbose information about the order and structure of a `SELECT` statement. This can be used to see where keys are not being used efficiently. This information is returned as a result set with the following columns:

table
: The name of the table referenced by the result set row explaining the query.

type
: The type of join that will be performed. These types, in order of performance, are:

  system
  : A special case of the `const` type, this join supports a table with a single row.

  const
  : Used for tables with at most a single matching row that will be read at the start of the query. MySQL treats this value as a constant to speed up processing.

  eq_ref
  : Reads one row from the table for each combination of rows from the previous tables. It is used when all parts of the index are used by the join and the index is unique or a primary key.

Copyright © 2001 O'Reilly & Associates, Inc.

ref

Reads all rows with matching index values from the table for each combination of rows from the previous tables. This join occurs when only the leftmost part of an index or if the index is not unique or a primary key. This is a good join when the key in use matches only a few rows.

range

Reads only the rows in a given range using an index to select the rows. They key column indicates the `key` in use and the `key_len` column contains the longest part of the key. The ref column will be `NULL` for this type.

index

Reads all rows based on an index tree scan.

ALL

A full table scan is done for each combination of rows from the previous tables. This join is generally a very bad thing. If you see it, you probably need to build a different SQL query or better organize your indices.

possible_keys

Indicates which indices MySQL could use to build the join. If this column is empty, there are no relevant indices and you probably should build some to enhance performance.

key

Indicates which index MySQL decided to use.

key_len

Provides the length of the key MySQL decided to use for the join.

ref

Describes which columns or constants were used with the key to build the join.

rows

Indicates the number of rows MySQL estimates it will need to examine to perform the query.

Extra

Additional information indicating how MySQL will perform the query.

Example

```
EXPLAIN SELECT customer.name, product.name FROM customer, product, purchases
WHERE purchases.customer=customer.id AND purchases.product=product.id
```

Copyright © 2001 O’Reilly & Associates, Inc.

FLUSH

Syntax

```
FLUSH option[, option...]
```

Description

Flushes or resets various internal processes depending on the options given. You must have reload privileges to execute this statement. The option can be any of the following:

HOSTS
: Empties the cache table that stores hostname information for clients. This should be used if a client changes IP addresses, or if there are errors related to connecting to the host.

LOGS
: Closes all of the standard log files and reopens them. This can be used if a log file has changed inode number. If no specific extension has been given to the update log, a new update log will be opened with the extension incremented by one.

PRIVILEGES
: Reloads all of the internal MySQL permissions grant tables. This must be run for any changes to the tables to take effect.

STATUS
: Resets the status variables that keep track of the current state of the server.

TABLES
: Closes all currently opened tables and flushes any cached data to disk.

GRANT

Syntax

```
GRANT privilege
[ (column, ...) ] [, privilege [( column, ...) ] ...]
ON {table} TO user [IDENTIFIED BY 'password']
[, user [IDENTIFIED BY 'password'] ...]
[WITH GRANT OPTION]
```

Previous to MySQL 3.22.11, the GRANT statement was recognized but did nothing. In current versions, GRANT is functional. This statement will enable access rights to a user (or users). Access can be granted per database, table or individual column. The table can be given as a table within the current database, '*' to affect all tables within the current database, '*.*' to affect all tables within all databases or 'database.*' to effect all tables within the given database.

Copyright © 2001 O'Reilly & Associates, Inc.

The following privileges are currently supported:

`ALL PRIVILEDGES/ALL`
: Effects all privileges

`ALTER`
: Altering the structure of tables

`CREATE`
: Creating new tables

`DELETE`
: Deleting rows from tables

`DROP`
: Deleting entire tables

`FILE`
: Creating and removing entire databases as well as managing log files

`INDEX`
: Creating and deleting indices from tables

`INSERT`
: Inserting data into tables

`PROCESS`
: Killing process threads

`REFERENCES`
: Not implemented (yet)

`RELOAD`
: Refreshing various internal tables (see the `FLUSH` statement)

`SELECT`
: Reading data from tables

`SHUTDOWN`
: Shutting down the database server

`UPDATE`
: Altering rows within tables

`USAGE`
: No privileges at all

The user variable is of the form *user@hostname*. Either the user or the hostname can contain SQL wildcards. If wildcards are used, either the whole name must be quoted, or just the part(s) with the wildcards (e.g., joe@"%.com " and *"joe@%.com"* are both valid). A user without a hostname is considered to be the same as *user@"%"*.

Copyright © 2001 O'Reilly & Associates, Inc.

If you have a global GRANT privilege, you may specify an optional INDENTIFIED BY modifier. If the user in the statement does not exist, it will be created with the given password. Otherwise the existing user will have his or her password changed.

Giving the GRANT privilege to a user is done with the WITH GRANT OPTION modifier. If this is used, the user may grant any privilege they have onto another user.

Examples

```
# Give full access to joe@carthage for the Account table
GRANT ALL ON bankdb.Account TO joe@carthage;
# Give full access to jane@carthage for the
# Account table and create a user ID for her
GRANT ALL ON bankdb.Account TO jane@carthage IDENTIFIED BY 'mypass';
# Give joe on the local machine the ability
# to SELECT from any table on the webdb database
GRANT SELECT ON webdb.* TO joe;
```

INSERT

Syntax

```
INSERT [DELAYED | LOW_PRIORITY ] [IGNORE]
[INTO] table [ (column, ...) ]
VALUES ( values  [, values... ])

INSERT [LOW_PRIORITY] [IGNORE]
[INTO] table [ (column, ...) ]
SELECT ...

INSERT [LOW_PRIORITY] [IGNORE]
[INTO] table
SET column=value, column=value,...
```

Description

Inserts data into a table. The first form of this statement simply inserts the given values into the given columns. Columns in the table that are not given values are set to their default value or NULL. The second form takes the results of a SELECT query and inserts them into the table. The third form is simply an alternate version of the first form that more explicitly shows which columns correspond with which values. If the DELAYED modifier is present in the first form, all incoming SELECT statements will be given priority over the insert, which will wait until the other activity has finished before inserting the data. In a similar way, using the LOW_PRIORITY modifier with any form of INSERT will cause the insertion to be postponed until all other operations from the client have been finished.

Copyright © 2001 O'Reilly & Associates, Inc.

When using a SELECT query with the INSERT statement, you cannot use the ORDER BY modifier with the SELECT statement. Also, you cannot insert into the same table you are selecting from.

Starting with MySQL 3.22.5 it is possible to insert more than one row into a table at a time. This is done by adding additional value lists to the statement separated by commas.

You must have INSERT privileges to use this statement.

Examples

```
# Insert a record into the 'people' table.
INSERT INTO people ( name, rank, serial_number )
VALUES ( 'Bob Smith', 'Captain', 12345 );
# Copy all records from 'data' that are older than a certain date into
# 'old_data'. This would usually be followed by deleting the old data from
# 'data'.
INSERT INTO old_data ( id, date, field )
SELECT ( id, date, field)
FROM data
WHERE date < 87459300;
# Insert 3 new records into the 'people' table.
INSERT INTO people (name, rank, serial_number )
VALUES ( 'Tim O\'Reilly', 'General', 1),
       ('Andy Oram', 'Major', 4342),
       ('Randy Yarger', 'Private', 9943);
```

KILL

Syntax

```
KILL thread_id
```

Description

Terminates the specified thread. The thread ID numbers can be found using the SHOW PROCESSES statement. Killing threads owned by users other than yourself require process privilege.

Example

```
# Terminate thread 3
KILL 3
```

LOAD

Syntax

```
LOAD DATA [LOCAL] INFILE file [REPLACE|IGNORE]
INTO TABLE table [delimiters] [(columns)]
```

Copyright © 2001 O'Reilly & Associates, Inc.

Description

Reads a text file that is in a readable format and inserts the data into a database table. This method of inserting data is much quicker than using multiple INSERT statements. Although the statement may be sent from all clients just like any other SQL statement, the file referred to in the statement is assumed to be located on the server unless the LOCAL keyword is used.. If the filename does not have a fully qualified path, MySQL looks under the directory for the current database for the file.

With no delimiters specified, LOAD DATA INFILE will assume that the file is tab delimited with character fields, special characters escaped with the backslash (\), and lines terminated with a newline character.

In addition to the default behavior, you may specify your own delimiters using the following keywords:

FIELDS TERMINATED BY '*c*'
: Specifies the character used to delimit the fields. Standard C language escape codes can be used to designate special characters. This value may contain more than one character. For example, FIELDS TERMINATED BY ',' denotes a comma delimited file and FIELDS TERMINATED BY '\t' denotes tab delimited. The default value is tab delimited.

FIELDS ENCLOSED BY '*c*'
: Specifies the character used to enclose character strings. For example, FIELD ENCLOSED BY '"' would mean that a line containing "this, value", "this", "value" would be taken to have three fields: "this,value", "this", and "value". The default behavior is to assume that no quoting is used in the file.

FIELDS ESCAPED BY '*c*'
: Specifies the character used to indicate that the next character is not special, even though it would usually be a special character. For example, with FIELDS ESCAPED BY '^' a line consisting of First,Second^,Third,Fourth would be parsed as three fields: "First", "Second,Third" and "Fourth". The exceptions to this rule are the null characters. Assuming the FIELDS ESCAPED BY value is a backslash, \0 indicates an ASCII NUL (character number 0) and \N indicates a MySQL NULL value. The default value is the backslash character. Note that MySQL itself considers the backslash character to be special. Therefore to indicate backslash in that statement you must backslash the backslash like this: FIELDS ESCAPED BY '\\'.

LINES TERMINATED BY '*c*'
: Specifies the character that indicates the start of a new record. This value can contain more than one character. For example, with LINES TERMINATED BY '.', a file consisting of a,b,c.d,e,f.g,h,k. would be parsed as three sepa-

Copyright © 2001 O'Reilly & Associates, Inc.

rate records, each containing three fields. The default is the newline character. This means that by default, MySQL assumes that each line is a separate record.

The keyword FIELDS should only be used for the entire statement. For example:

```
LOAD DATA INFILE data.txt FIELDS TERMINATED BY ',' ESCAPED BY '\\'
```

By default, if a value read from the file is the same as an existing value in the table for a field that is part of a unique key, an error is given. If the REPLACE keyword is added to the statement, the value from the file will replace the one already in the table. Conversely, the IGNORE keyword will cause MySQL to ignore the new value and keep the old one.

The word NULL encountered in the data file is considered to indicate a null value unless the FIELDS ENCLOSED BY character encloses it.

Using the same character for more than one delimiter can confuse MySQL. For example, FIELDS TERMINATED BY ',' ENCLOSED BY ',' would produce unpredictable behavior.

If a list of columns is provided, the data is inserted into those particular fields in the table. If no columns are provided, the number of fields in the data must match the number of fields in the table, and they must be in the same order as the fields are defined in the table.

You must have SELECT and INSERT privileges on the table to use this statement.

Example

```
# Load in the data contained in 'mydata.txt' into the table 'mydata'. Assume
# that the file is tab delimited with no quotes surrounding the fields.
LOAD DATA INFILE 'mydata.txt' INTO TABLE mydata
# Load in the data contained in 'newdata.txt' Look for two comma delimited
# fields and insert their values into the fields 'field1' and 'field2' in
# the 'newtable' table.
LOAD DATA INFILE 'newdata.txt'
INTO TABLE newtable
FIELDS TERMINATED BY ','
( field1, field2 )
```

LOCK

Syntax

```
LOCK TABLES name
[AS alias] READ|WRITE [, name2 [AS alias] READ|WRITE, ...]
```

Copyright © 2001 O'Reilly & Associates, Inc.

Description

Locks a table for the use of a specific thread. This command is generally used to emulate transactions. If a thread creates a READ lock all other threads may read from the table but only the controlling thread can write to the table. If a thread creates a WRITE lock, no other thread may read from or write to the table.

Using locked and unlocked tables at the same time can cause the process thread to freeze. You must lock all of the tables you will be accessing during the time of the lock. Tables you access only before or after the lock do not need to be locked. The newest versions of MySQL generate an error if you attempt to access an unlocked table while you have other tables locked.

Example

```
# Lock tables 'table1' and 'table3' to prevent updates, and block all access
# to 'table2'. Also create the alias 't3' for 'table3' in the current thread.
LOCK TABLES table1 READ, table2 WRITE, table3 AS t3 READ
```

OPTIMIZE

Syntax

```
OPTIMIZE TABLE name
```

Description

Recreates a table eliminating any wasted space. This is done by creating the optimized table as a separate, temporary table and then moving over to replace the current table. While the procedure is happening, all table operations continue as normal (all writes are diverted to the temporary table).

Example

```
OPTIMIZE TABLE mytable
```

REPLACE

Syntax

```
REPLACE INTO table [(column, ...)] VALUES (value, ...)

REPLACE INTO table [(column, ...)] SELECT select_clause
```

Copyright © 2001 O'Reilly & Associates, Inc.

Description

Inserts data to a table, replacing any old data that conflicts. This statement is identical to INSERT except that if a value conflicts with an existing unique key, the new value replaces the old one. The first form of this statement simply inserts the given values into the given columns. Columns in the table that are not given values are set to their default value or NULL. The second form takes the results of a SELECT query and inserts them into the table.

Examples

```
# Insert a record into the 'people' table.
REPLACE INTO people ( name, rank, serial_number )
VALUES ( 'Bob Smith', 'Captain', 12345 )
# Copy all records from 'data' that are older than a certain date into
# 'old_data'. This would usually be followed by deleting the old data from
# 'data'.
REPLACE INTO old_data ( id, date, field )
SELECT ( id, date, field)
FROM data
WHERE date < 87459300
```

REVOKE

Syntax

```
REVOKE privilege [(column, ...)] [, privilege [(column, .
..) ...]
ON table FROM user
```

Description

Removes a privilege from a user. The values of privilege, table, and user are the same as for the GRANT statement. You must have the GRANT privilege to be able to execute this statement.

SELECT

Syntax

```
SELECT [STRAIGHT_JOIN] [DISTINCT|ALL] value[, value2...]
[INTO OUTFILE 'filename' delimiters]
FROM table[, table2...] [clause]
```

Description

Retrieve data from a database. The SELECT statement is the primary method of reading data from database tables.

Copyright © 2001 O'Reilly & Associates, Inc.

If you specify more than one table, MySQL will automatically join the tables so that you can compare values between the tables. In cases where MySQL does not perform the join in an efficient manner, you can specify `STRAIGHT_JOIN` to force MySQL to join the tables in the order you enter them in the query.

If the `DISTINCT` keyword is present, only one row of data will be output for every group of rows that is identical. The `ALL` keyword is the opposite of distinct and displays all returned data. The default behavior is `ALL`.

The returned values can be any one of the following:

Aliases
: Any complex column name or function can be simplified by creating an alias for it. The value can be referred to by its alias anywhere else in the `SELECT` statement (e.g., `SELECT DATE_FORMAT(date,"%W, %M %d %Y") as nice_date FROM calendar`).

Column names
: These can be specified as column, table.column or database.table.column. The longer forms are necessary only to disambiguate columns with the same name, but can be used at any time (e.g., `SELECT name FROM people; SELECT mydata.people.name FROM people`).

Functions
: MySQL supports a wide range of built-in functions (see later). In addition, user defined functions can be added at any time using the `CREATE FUNCTION` statement (e.g., `SELECT COS(angle) FROM triangle`).

By default, MySQL sends all output to the client that sent the query. It is possible however, to have the output redirected to a file. In this way you can dump the contents of a table (or selected parts of it) to a formatted file that can either be human readable, or formatted for easy parsing by another database system.

The `INTO OUTFILE 'filename'` modifier is the means in which output redirection is accomplished. With this the results of the `SELECT` query are put into *filename*. The format of the file is determined by the delimiters arguments, which are the same as the `LOAD DATA INFILE` statement with the following additions:

- The `OPTIONALLY` keyword may be added to the `FIELDS ENCLOSED BY` modifier. This will cause MySQL to thread enclosed data as strings and non-enclosed data as numeric.
- Removing all field delimiters (i.e., `FIELDS TERMINATED BY '' ENCLOSED BY ''`) will cause a fixed-width format to be used. Data will be exported according to the display size of each field. Many spreadsheets and desktop databases can import fixed-width format files.

The default behavior with no delimiters is to export tab delimited data using backslash (\) as the escape character and to write one record per line.

Copyright © 2001 O'Reilly & Associates, Inc.

The list of tables to join may be specified in the following ways:

Table1, Table2, Table3, . . .

This is the simplest form. The tables are joined in the manner that MySQL deems most efficient. This method can also be written as `Table1 JOIN Table2 JOIN Table3, ...`. The `CROSS` keyword can also be used, but it has no effect (e.g., `Table1 CROSS JOIN Table2`) Only rows that match the conditions for both columns are included in the joined table. For example, `SELECT * FROM people, homes WHERE people.id=homes.owner` would create a joined table containing the rows in the people table that have id fields that match the owner field in the homes table.

Like values, table names can also be aliased (e.g., `SELECT t1.name, t2.address FROM long_table_name t1, longer_table_name t2`)

`Table1` `STRAIGHT_JOIN` *`Table2`*

This is identical to the earlier method, except that the left table is always read before the right table. This should be used if MySQL performs inefficient sorts by joining the tables in the wrong order.

`Table1` `LEFT [OUTER] JOIN` *`Table2`* `ON` *`clause`*

This checks the right table against the clause. For each row that does not match, a row of `NULL`s is used to join with the left table. Using the previous example `SELECT * FROM people, homes LEFT JOIN people, homes ON people.id=homes.owner`, the joined table would contain all of the rows that match in both tables, as well as any rows in the people table that do not have matching rows in the homes table, `NULL` values would be used for the homes fields in these rows. The `OUTER` keyword is optional and has no effect.

`Table1` `LEFT [OUTER] JOIN` *`Table2`* `USING` *`(column[, column2 . . .])`*

This joins the specified columns only if they exist in both tables (e.g., `SELECT * FROM old LEFT OUTER JOIN new USING (id)`)

`Table1` `NATURAL LEFT [OUTER] JOIN` *`Table2`*

This joins only the columns that exist in both tables. This would be the same as using the previous method and specifying all of the columns in both tables (e.g., `SELECT rich_people.salary, poor_people.salary FROM rich_people NATURAL LEFT JOIN poor_people`)

`{ oj Table1` `LEFT OUTER JOIN` *`Table2`* `ON` *`clause }`*

This is identical to *`Table1`* `LEFT JOIN` *`Table2`* `ON` *`clause`* and is only included for ODBC compatibility. (The “oj” stands for “Outer Join”.)

Copyright © 2001 O’Reilly & Associates, Inc.

If no clause is provided, `SELECT` returns all of the data in the selected tables.

The search clause can contain any of the following substatements:

`WHERE` *statement*
: The `WHERE` statement construct is the most common way of searching for data in SQL. This statement is usually a comparison of some type but can also include any of the functions listed below, except for the aggregate functions. Named values, such as column names and aliases, and literal numbers and strings can be used in the statement.

`GROUP BY` *column*`[,` *column2,...*`]`
: This gathers all of the rows together that contain data from a certain column. This allows aggregate functions to be performed upon the columns (e.g., `SELECT name,MAX(age) FROM people  GROUP BY name`).

`HAVING` *clause*
: This is the same as a `WHERE` clause except that it is performed upon the data that has already been retrieved from the database. The `HAVING` statement is a good place to perform aggregate functions on relatively small sets of data that have been retrieved from large tables. This way, the function does not have to act upon the whole table, only the data that has already been selected (e.g., `SELECT name,MAX(age) FROM people GROUP BY name HAVING MAX(age)>80`).

`ORDER BY` *column* `[`*ASC*`|`*DESC*`][,` *column2* `[`*ASC*`|`*DESC*`],...]`
: Sorts the returned data using the given column(s). If `DESC` is present, the data is sorted in descending order, otherwise ascending order is used. Ascending order can also be explicitly stated with the `ASC` keyword (e.g., `SELECT name, age FROM people ORDER BY age DESC`).

`LIMIT [`*start,*`]` *rows*
: Returns Only the specified number of rows. If the start value is supplied, that many rows are skipped before the data is returned. The first row is number 0 (e.g., `SELECT url FROM links LIMIT 5,10` (returns URL's numbered 5 through 14).

`PROCEDURE` *name*
: In early versions of MySQL, this does not do anything. It was provided to make importing data from other SQL servers easier. Starting with MySQL 3.22, this substatement lets you specify a procedure that modifies the query result before returning it to the client.

SELECT supports the concept of functions. MySQL defines several built-in functions that can operate upon the data in the table, returning the computed value(s)

Copyright © 2001 O'Reilly & Associates, Inc.

to the user. With some functions, the value returned depends on whether the user wants to receive a numerical or string value. This is regarded as the "context" of the function. When selecting values to be displayed to the user, only text context is used, but when selecting data to be inserted into a field, or to be used as the argument of another function, the context depends upon what the receiver is expecting. For instance, selecting data to be inserted into a numerical field will place the function into a numerical context. MySQL functions are detailed in full in Chapter 18.

Examples

```
# Find all names in the 'people' table where the 'state' field is 'MI'.
SELECT name FROM people WHERE state='MI'
# Display all of the data in the 'mytable' table.
SELECT * FROM mytable
```

SET

Syntax

```
SET OPTION SQL_OPTION=value
```

Description

Defines an option for the current session. Values set by this statement are not in effect anywhere but the current connection, and they disappear at the end of the connection. The following options are current supported:

`CHARACTER SET` *`charsetname or`* `DEFAULT`
: Changes the character set used by MySQL. Specifying `DEFAULT` will return to the original character set.

`LAST_INSERT_ID=`*`number`*
: Determines the value returned from the `LAST_INSERT_ID()` function.

`SQL_BIG_SELECTS=0` *`or`* `1`
: Determines the behavior when a large `SELECT` query is encountered. If set to 1, MySQL will abort the query with an error if the query would probably take too long to compute. MySQL decides that a query will take too long it will have to examine more rows than the value of the `max_join_size` server variable. The default value is 0, which allows all queries.

`SQL_BIG_TABLES=0` *`or`* `1`
: Determines the behavior of temporary tables (usually generated when dealing with large data sets). If this value is 1, temporary tables are stored on disk, which is slower than primary memory but can prevent errors on systems with low memory. The default value is 0, which stores temporary tables in RAM.

Copyright © 2001 O'Reilly & Associates, Inc.

`SQL_LOG_OFF=0` *or* `1`
: When set to 1, turns off standard logging for the current session. This does not stop logging to the ISAM log or the update log. You must have `PROCESS LIST` privileges to use this option. The default is 0, which enables regular logging.

`SQL_SELECT_LIMIT=`*number*
: The maximum number of records returned by a `SELECT` query. A `LIMIT` modifier in a `SELECT` statement overrides this value. The default behavior is to return all records.

`SQL_UPDATE_LOG=0` *or* `1`
: When set to 0, turns off update logging for the current session. This does not affect standard logging or ISAM logging. You must have `PROCESS LIST` privileges to use this option. The default is 1, which enables regular logging.

`TIMESTAMP=`*value* `or DEFAULT`
: Determines the time used for the session. This time is logged to the update log and will be used if data is restored from the log. Specifying `DEFAULT` will return to the system time.

Example

```
# Turn off logging for the current connection.
SET OPTION SQL_LOG_OFF=1
```

SHOW

Syntax

```
SHOW COLUMNS FROM table [FROM database] [LIKE clause]
SHOW DATABASES [LIKE clause]
SHOW FIELDS FROM table [FROM database] [LIKE clause]
SHOW GRANTS
SHOW INDEX FROM table [FROM database]
SHOW KEYS FROM table [FROM database]
SHOW PROCESSLIST
SHOW STATUS
SHOW TABLE STATUS [FROM database [LIKE clause]]
SHOW TABLES [FROM database [LIKE expression]]
SHOW VARIABLES [LIKE clause]
```

Description

Displays various information about the MySQL system. This statement can be used to examine the status or structure of almost any part of MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

Examples

```
# Show the available databases
SHOW DATABASES
# Display information on the indexes on table 'bigdata'
SHOW KEYS FROM bigdata
# Display information on the indexes on table 'bigdata'
# in the database 'mydata'
SHOW INDEX FROM bigdata FROM mydata
# Show the tables available from the database 'mydata' that begin with the
# letter 'z'
SHOW TABLES FROM mydata LIKE 'z%'
# Display information about the columns on the table 'skates'
SHOW COLUMNS FROM stakes
# Display information about the columns on the table 'people'
# that end with '_name'
SHOW FIELDS FROM people LIKE '%\_name'
# Show server status information.
SHOW STATUS
# Display server variables
SHOW VARIABLES
```

UNLOCK

Syntax

```
UNLOCK TABLES
```

Description

Unlocks all tables that were locked using the `LOCK` statement during the current connection.

Example

```
# Unlock all tables
UNLOCK TABLES
```

UPDATE

Syntax

```
UPDATE [LOW_PRIORITY] table
SET column=value, ...
[WHERE clause]
[LIMIT n]
```

Description

Alters data within a table. This statement is used to change actual data within a table without altering the table itself. You may use the name of a column as a value when setting a new value. For example, `UPDATE health SET miles_`

Copyright © 2001 O'Reilly & Associates, Inc.

`ran=miles_ran+5` would add five to the current value of the miles_ran column. The statement returns the number of rows changed.

You must have `UPDATE` privileges to use this statement.

Example

```
# Change the name 'John Deo' to 'John Doe' everywhere in the people table.
UPDATE people SET name='John Doe' WHERE name='John Deo'
```

USE

Syntax

```
USE database
```

Description

Selects the default database. The database given in this statement is used as the default database for subsequent queries. Other databases may still be explicitly specified using the `database.table.column` notation.

Example

```
# Make db1 the default database.
USE db1
```

Copyright © 2001 O'Reilly & Associates, Inc.

17

MySQL Data Types

MySQL supports a wide variety of data types to support the storage of different kinds of data. This chapter lists the full range of these data types as well as a description of their functionality, syntax, and data storage requirements. For each data type, the syntax shown uses square brackets ([]) to indicate optional parts of the syntax. The following example shows how BIGINT is explained in this chapter:

```
BIGINT[(display_size)]
```

This indicates that you can use `BIGINT` alone or with a display size value. The italics indicate that you do not enter `display_size` literally, but instead place your own value in there. Thus possible uses of `BIGINT` include:

```
BIGINT
BIGINT(20)
```

Like the `BIGINT` type above, many MySQL data types support the specification of a display size. Unless otherwise specified, this value must be an integer between 1 and 255.

Table 17-1 lists the data types and categorizes them as numeric, string, date, or complex. You can then find the full description of the data type in the appropriate section of this chapter.

Table 17-1. . MySQL Data Types.

Data Type	Classification
`BIGINT`	Numeric
`BLOB`	String
`CHAR`	String
`CHARACTER`	String
`CHARACTER VARYING`	String

Copyright © 2001 O'Reilly & Associates, Inc.

Table 17-1. . MySQL Data Types.

Data Type	Classification
DATE	Date
DATETIME	Date
DEC	Numeric
DECIMAL	Numeric
DOUBLE	Numeric
DOUBLE PRECISION	Numeric
ENUM	Complex
FLOAT	Numeric
INT	Numeric
INTEGER	Numeric
LONGBLOB	String
LONGTEXT	String
MEDIUMBLOB	String
MEDIUMINT	Numeric
MEDIUMTEXT	String
NCHAR	String
NATIONAL CHAR	String
NATIONAL CHARACTER	String
NATIONAL VARCHAR	String
NUMERIC	Numeric
REAL	Numeric
SET	Complex
SMALLINT	Numeric
TEXT	String
TIME	Date
TIMESTAMP	Date
TINYBLOB	String
TINYINT	Numeric
TINYTEXT	String
VARCHAR	String
YEAR	Date

In some cases, MySQL silently changes the column type you specify in your table creation to something else. These cases are:

- `VARCHAR -> CHAR`: When the specified `VARCHAR` column size is less than 4 characters, it is converted to `CHAR`.

Copyright © 2001 O'Reilly & Associates, Inc.

- CHAR -> VARCHAR: When a table has at least one column of a variable length, all CHAR columns greater than three characters in length are converted to VARCHAR.
- TIMESTAMP display sizes: Display sizes for TIMESTAMP fields must be an even value between 2 and 14. A display size of 0 or greater than 14 will convert the field to a display size of 14. An odd-valued display size will be converted to the next highest even value.

Numeric Data Types

MySQL supports all ANSI SQL2 numeric data types. MySQL numeric types break down into two groups: integer and floating point types. Within each group, the types differ basically by the amount of storage required for them. The floating types allow you to optionally specify the number of digits that follow the decimal point. In such cases, the digits value should be an integer from 0 to 30 and no greater than two less than the display size. If you do make the digits value greater than two less than the display size, the display size will automatically change to be two greater than the digits value.

When you insert a value into a column that requires more storage than the data type allows, it will be clipped to the minimum value for that data type (negative values) or the maximum value for that data type (positive values). MySQL will issue a warning when such clipping occurs during ALTER TABLE, LOAD DATA INFILE, UPDATE and multi-row INSERT statements.

The AUTO_INCREMENT attribute may be supplied for at most one column of an integer type in a table. In addition to being an integer type, the column must be either a primary key or the sole column in a unique index. When you attempt an insert into a table with such an integer field and fail to specify a value for that field (or specify a NULL value), a value of one greater than the column's current maximum value will be automatically inserted.

The UNSIGNED attribute may be used with any numeric type. An unsigned column may contain only positive integers or floating point values.

The ZEROFILL attribute indicates that the column should be left padded with zeroes when displayed by MySQL. The number of zeroes padded is determined by the column's display width.

Copyright © 2001 O'Reilly & Associates, Inc.

BIGINT

Syntax

```
BIGINT[(display_size)] [AUTO_INCREMENT] [UNSIGNED]
[ZEROFILL]
```

Storage

8 bytes

Description

Largest integer type supporting the range of whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (0 to 18,446,744,073,709,551,615 unsigned). `BIGINT` has some issues when you perform arithmetic on unsigned values. MySQL performs all arithmetic using signed `BIGINT` or `DOUBLE` values. You should therefore avoid performing any arithmetic operations on unsigned `BIGINT` values greater than 9,223,372,036,854,775,807. If you do, you may end up with imprecise results.

DEC

Synonym for `DECIMAL`.

DECIMAL

Syntax

```
DECIMAL[(precision, [scale])] [ZEROFILL]
```

Storage

precision + 2 bytes

Description

Stores floating point numbers where precision is critical, such as for monetary values. `DECIMAL` types require you to specify the precision and scale. The precision is the number of significant digits in the value. The scale is the number of those digits that come after the decimal point. A `BALANCE` column declared as `DECIMAL(9, 2)` would thus store numbers with 9 significant digits, two of which are to the right of the decimal point. The range for this declaration would be -9,999,999.99 to 9,999,999.99. If you specify a number with more decimal points, it is rounded so it fits the proper scale. Values beyond the range of the `DECIMAL` are clipped to fit within the range.

MySQL actually stores `DECIMAL` values as strings, not floating point numbers. It uses one character for each digit as well as one character for the decimal points

Copyright © 2001 O'Reilly & Associates, Inc.

when the scale is greater than 0 and one character for the sign of negative numbers. When the scale is 0, the value contains no fractional part. Prior to MySQL 3.23, the precision actually had to include space for the decimal and sign. This requirement is no longer in place in accordance with the ANSI specification.

ANSI SQL supports the omission of precision and/or scale where the omission of scale creates a default scale of zero and the omission of precision defaults to an implementation-specific value. In the case of MySQL, the default precision is 10.

DOUBLE

Syntax

`DOUBLE[(display_size, digits)] [ZEROFILL]`

Storage

8 bytes

Description

A double-precision floating point number. This type is used for the storage of large floating point values. `DOUBLE` columns can store negative values between -1.7976931348623157E+308 and -2.2250738585072014E-308, 0, and positive numbers between 2.2250738585072014E-308 and 1.7976931348623157E+308.

DOUBLE PRECISION

Synonym for `DOUBLE`.

FLOAT

Syntax

`FLOAT[(display_size, digits)] [ZEROFILL]`

Storage

4 bytes

Description

A single-precision floating point number. This type is used for the storage of small floating point numbers. FLOAT columns can store negative values between -3.402823466E+38 and -1.175494351E-38, 0, and positive values between 1.175494351E-38 and 3.402823466E+38.

Copyright © 2001 O'Reilly & Associates, Inc.

INT

Syntax

```
INT[(display_size)] [AUTO_INCREMENT] [UNSIGNED]
[ZEROFILL]
```

Storage

4 bytes

Description

A basic whole number with a range of -2,147,483,648 to 2,147,483,647 (0 to 4,294,967,295 unsigned).

INTEGER

Synonym for `INT`.

MEDIUMINT

Syntax

```
MEDIUMINT[(display_size)] [AUTO_INCREMENT] [UNSIGNED]
[ZEROFILL]
```

Storage

3 bytes

Description

A basic whole number with a range of -8,388,608 to 8,388,607 (0 to 16,777,215 unsigned).

NUMERIC

Synonym for `DECIMAL`.

REAL

Synonym for `DOUBLE`.

Copyright © 2001 O'Reilly & Associates, Inc.

SMALLINT

Syntax

```
SMALLINT[(display_size)] [AUTO_INCREMENT] [UNSIGNED]
[ZEROFILL]
```

Storage

2 bytes

Description

A basic whole number with a range of -32,768 to 32,767 (0 to 65,535 unsigned).

TINYINT

Syntax

```
TINYINT[(display_size)] [AUTO_INCREMENT] [UNSIGNED]
[ZEROFILL]
```

Storage

1 byte

Description

A basic whole number with a range of -128 to 127 (0 to 255 unsigned).

String Data Types

String data types store various kinds of text data. By default, string columns are sorted and compared in a case-insensitive fashion in accordance with the sorting rules for the default character set. Each string type has a corresponding binary type. For CHAR and VARCHAR, the binary types are declared using the BINARY attribute. All of the TEXT types, however, have corresponding BLOB types as their binary counterparts.

Text fields are compared and sorted in a case-insensitive fashion in accordance with the default character set of the server. Binary fields, on the other hand, are not interpreted according to a character set. Comparisons and sorts occur on a byte-by-byte basis without interpretation. In other words, binary values are treated case-sensitive.

BLOB

Binary form of `TEXT`.

Copyright © 2001 O'Reilly & Associates, Inc.

CHAR

Syntax

```
CHAR(size) [BINARY]
```

Size

Specified by the *size* value in a range of 0 to 255 (1 to 255 prior to MySQL 3.23).

Storage

size bytes

Description

A fixed-length text field. String values with fewer characters than the column's size will be right-padded with spaces. The right-padding is removed on retrieval of the value from the database.

`CHAR(0)` fields are useful for backwards compatibility with legacy systems that no longer store values in the column or for binary values (`NULL` vs. `''`).

CHARACTER

Synonym for `CHAR`.

CHARACTER VARYING

Synonym for `VARCHAR`.

LONGBLOB

Binary form of `LONGTEXT`.

LONGTEXT

Syntax

```
LONGTEXT
```

Size

0 to 4,294,967,295

Storage

length of value + 4 bytes

Copyright © 2001 O'Reilly & Associates, Inc.

Description

Storage for large text values.

While the theoretical limit to the size of the text that can be stored in a `LONGTEXT` column exceeds 4GB, the practical limit is much less due to limitations of the MySQL communication protocol and the amount of memory available on both the client and server ends of the communication.

MEDIUMBLOB

Binary form of `MEDIUMTEXT`.

MEDIUMTEXT

Syntax

```
MEDIUMTEXT
```

Size

0 to 16,777,215

Storage

length of value + 3 bytes

Description

Storage for medium-sized text values.

NCHAR

Synonym of `CHAR`.

NATIONAL CHAR

Synonym of `CHAR`.

NATIONAL CHARACTER

Synonym of `CHAR`.

NATIONAL VARCHAR

Synonym of `VARCHAR`.

Copyright © 2001 O'Reilly & Associates, Inc.

TEXT

Syntax

```
TEXT
```

Size

0 to 65,535

Storage

length of value + 2 bytes

Description

Storage for most text values.

TINYBLOB

Binary form of `TINYTEXT`.

TINYTEXT

Syntax

```
TINYTEXT
```

Size

0 to 255

Storage

length of value + 1 byte

Description

Storage for short text values.

VARCHAR

Syntax

```
VARCHAR(size) [BINARY]
```

Size

Specified by the *size* value in a range of 0 to 255 (1 to 255 prior to MySQL 3. 23).

Storage

length of value + 1 byte

Copyright © 2001 O'Reilly & Associates, Inc.

Description

Storage for variable-length text. Trailing spaces are removed from VARCHAR values when stored in the database in conflict with the ANSI specification.

Date Data Types

MySQL date types are extremely flexible tools for storing date information. They are also extremely forgiving in the belief that it is up to the application, not the database, to validate date values. MySQL only checks that months are in the range of 0-12 and dates are in the range of 0-31. February 31, 2001 is therefore a legal MySQL date. More useful, however, is the fact that February 0, 2001 is a legal date. In other words, you can use 0 to signify dates where you do not know a particular piece of the date.

Though MySQL is somewhat forgiving on the input format, your applications should actually attempt to format all date values in MySQL's native format to avoid any confusion. MySQL always expects the year to be the left-most element of a date format. If you assign illegal values in a SQL operation, MySQL will insert zeroes for that value.

MySQL will also perform automatic conversion of date and time values to integer values when used in an integer context.

DATE

Syntax

`DATE`

Format

YYYY-MM-DD (2001-01-01)

Storage

3 bytes

Description

Stores a date in the range of January 1, 1000 (`'1000-01-01'`) to December 31, 9999 (`'9999-12-31'`) in the Gregorian calendar.

DATETIME

Syntax

`DATETIME`

Copyright © 2001 O'Reilly & Associates, Inc.

Format

YYYY-MM-DD hh:mm:ss (2001-01-01 01:00:00)

Storage

8 bytes

Description

Stores a specific time in the range of 12:00:00 AM, January 1, 1000 (`'1000-01-01 00:00:00'`) to 11:59:59 PM, December 31, 9999 (`'9999-12-31 23:59:59'`) in the Gregorian calendar.

TIME

Syntax

`TIME`

Format

hh:mm:ss (06:00:00)

Storage

3 bytes

Description

Stores a time value in the range of midnight (`'00:00:00'`) to 1 second before midnight (`'23:59:59'`).

TIMESTAMP

Syntax

`TIMESTAMP[(`*`display_size`*`)]`

Format

YYYYMMDDhhmmss (20010101060000)

Storage

4 bytes

Description

A simple representation of a point in time down to the second in the range of midnight on January 1, 1970 to one minute before midnight on December 31, 2037. Its primary utility is in keeping track of table modifications. When you insert a `NULL` value into a `TIMESTAMP` column, the current date and time will be inserted instead. When you modify any value in a row with a `TIMESTAMP` col-

Copyright © 2001 O'Reilly & Associates, Inc.

umn, the first `TIMESTAMP` column will automatically be updated with the current date and time.

YEAR

Syntax

```
YEAR
```

Format

YYYY (2001)

Storage

1 byte

Description

Stores a year of the Gregorian calendar in the range of 1900 to 2155.

Complex Data Types

MySQL's complex data types `ENUM` and `SET` are really nothing more than special string types. We break them out because they are conceptually more complex and represent a lead into the SQL3 data types that MySQL may one day support.

ENUM

Syntax

```
ENUM(value1, value2, ...)
```

Storage

1-255 members: 1 byte
256-65,535 members: 2 bytes

Description

Stores one value of a predefined list of possible strings. When you create an ENUM column, you provide a list of all possible values. Inserts and updates are then allowed to set the column to values only from that list. Any attempt to insert a value that is not part of the enumeration will cause an empty string to be stored instead.

You may reference the list of possible values by index where the index of the first possible value is 0. For example:

```
SELECT COLID FROM TBL WHERE COLENUM = 0;
```

Copyright © 2001 O'Reilly & Associates, Inc.

Assuming `COLID` is a primary key column and `COLENUM` is the column of type `ENUM`, this SQL will retrieve the primary keys of all rows with `COLENUM` value equals the first value of that list. Similarly, sorting on `ENUM` columns happens according to index, not string value.

The maximum number of elements allowed for an `ENUM` column is 65,535.

SET

Syntax

`SET(`*`value1, value2, ...`*`)`

Storage

1-8 members: 1 byte
9-16 members: 2 bytes
17-24 members: 3 bytes
25-32 members: 4 bytes
33-64 members: 8 bytes

Description

A list of values taken from a pre-defined set of values. A SET is basically an ENUM that allows multiple values. A SET, however, is not stored according to index but instead as a complex bit map. Given a SET with the members "Orange", "Apple", "Pear", and "Banana", each element is represented by an "on" bit in a byte as shown in Table 17-2

Table 17-2. . MySQL Representations of Set Elements

Member	Decimal Value	Bitwise Representation
Orange	1	0001
Apple	2	0010
Pear	4	0100
Banana	8	1000

The values "Orange" and "Pear" are therefore stored in the database as 5 (0101).

You can store a maximum of 64 values in a SET column. Though you can assign the same value multiple times in a SQL statement updating a SET column, only a single value will actually be stored.

Copyright © 2001 O'Reilly & Associates, Inc.

Copyright © 2001 O'Reilly & Associates, Inc.

18

Operators and Functions

Like any other programming language, SQL carries among its core elements operators and named procedures. This reference lists all of those operators and functions and explains how they evaluate into useful expressions.

Operators

MySQL operators may be divided into three kinds of operators: arithmetic, comparison, and logical.

Rules of Precedence

When your SQL contains complex expressions, the sub-expressions are evaluated based on MySQL's rules of precedence. Of course, you may always override MySQL's rules of precedence by enclosing an expression in parentheses.

1. BINARY
2. NOT
3. - (unary minus)
4. * / %
5. + -
6. << >>
7. &
8. |
9. < <= > >= = <=> <> IN IS LIKE REGEXP
10. BETWEEN

Copyright © 2001 O'Reilly & Associates, Inc.

11. AND
12. OR

Arithmetic Operators

Arithmetic operators perform basic arithmetic on two values.

`+` Adds two numerical values

`-` Subtracts two numerical values

`*` Multiplies two numerical values

`/` Divides two numerical values

`%` Gives the modulo of two numerical values

`|` Performs a bitwise OR on two integer values

`&` Performs a bitwise AND on two integer values

`<<` Performs a bitwise left shift on an integer value

`>>` Performs a bitwise right shift on an integer value

Comparison Operators

Comparison operators compare values and return 1 if the comparison is true, 0 otherwise. Except for the <==> operator, NULL values cause a comparison operator to evaluate to NULL.

`<>` *or* `!=`
: Match rows if the two values are not equal.

`<=`
: Match rows if the left value is less than or equal to the right value.

`<`
: Match rows if the left value is less than the right value.

`>=`
: Match rows if the left value is greater than or equal to the right value.

`>`
: Match rows if the left value is greater than the right value.

`value` `BETWEEN` *`value1`* `AND` *`value2`*
: Match rows if *value* is between *value1* and *value2*, or equal to one of them.

`value` `IN (`*`value1,value2,...`*`)`
: Match rows if *value* is among the values listed.

Copyright © 2001 O'Reilly & Associates, Inc.

`value NOT IN (value1, value2,...)`
: Match rows if *value* is not among the values listed.

`value1 LIKE value2`
: Compares *value1* to *value2* and matches the rows if they match. The right-hand value can contain the wildcard `'%'` which matches any number of characters (including 0) and `'_'` which matches exactly one character. This is probably the single most used comparison in SQL. The most common usage is to compare a field value with a literal containing a wildcard (e.g., `SELECT name FROM people WHERE name LIKE 'B%'`).

`value1 NOT LIKE value2`
: Compares *value1* to *value2* and matches the rows if they differ. This is identical to `NOT (value1 LIKE value2)`.

`value1 REGEXP/RLIKE value2`
: Compares *value1* to *value2* using the extended regular expression syntax and matches the rows if they match. The right hand value can contain full Unix regular expression wildcards and constructs (e.g., `SELECT name FROM people WHERE name RLIKE '^B.*'`).

`value1 NOT REGEXP value2`
: Compares *value1* to *value2* using the extended regular expression syntax and matches the rows if they differ. This is identical to `NOT (value1 REXEXP value2)`.

Logical Operators

Logical operators check the truth value of one or more expressions. In SQL terms, a logical operator checks whether its operands are 0, non-zero, or `NULL`. A 0 value means false, non-zero true, and `NULL` means no value.

`NOT` *or* `!`
: Performs a logical not (returns 1 if the value is 0 and returns 0 otherwise).

`OR` *or* `||`
: Performs a logical or (returns 1 if any of the arguments are not 0, otherwise returns 0)

`AND` *or* `&&`
: Performs a logical and (returns 0 if any of the arguments are 0, otherwise returns 1)

Functions

MySQL provides built-in functions that perform special operations.

Copyright © 2001 O’Reilly & Associates, Inc.

Aggregate Functions

Aggregate functions operate on a set of data. The usual method of using these is to perform some action on a complete set of returned rows. For example, `SELECT AVG(height) FROM kids` would return the average of all of the values of the `height` field in the `kids` table.

`AVG(`*`expression`*`)`
: Returns the average value of the values in *expression* (e.g., `SELECT AVG(score) FROM tests`).

`BIT_AND(`*`expression`*`)`
: Returns the bitwise `AND` aggregate of all of the values in *expression* (e.g., `SELECT BIT_AND(flags) FROM options`).

`BIT_OR(`*`expression`*`)`
: Returns the bitwise `OR` aggregate of all of the values in *expression* (e.g., `SELECT BIT_OR(flags) FROM options`).

`COUNT(`*`expression`*`)`
: Returns the number of times *expression* was not null. `COUNT(*)` will return the number of rows with some data in the entire table (e.g., `SELECT COUNT(*) FROM folders`).

`MAX(`*`expression`*`)`
: Returns the largest of the values in *expression* (e.g., `SELECT MAX (elevation) FROM mountains`).

`MIN(`*`expression`*`)`
: Returns the smallest of the values in *expression* (e.g., `SELECT MIN(level) FROM toxic_waste`).

`STD(`*`expression`*`)/STDDEV(`*`expression`*`)`
: Returns the standard deviation of the values in *expression* (e.g., `SELECT STDDEV(points) FROM data`).

`SUM(`*`expression`*`)`

Returns the sum of the values in *expression* (e.g., `SELECT SUM(calories) FROM daily_diet`).

General Functions

General functions operate on one or more discreet values.

`ABS(`*`number`*`)`
: Returns the absolute value of *number* (e.g., `ABS(-10)` returns 10).

Copyright © 2001 O'Reilly & Associates, Inc.

`ACOS(`*`number`*`)`
: Returns the inverse cosine of number in radians (e.g., `ACOS(0)` returns 1.570796).

`ASCII(`*`char`*`)`
: Returns the ASCII value of the given character (e.g., `ASCII('h')` returns 104).

`ASIN(`*`number`*`)`
: Returns the inverse sine of number in radians (e.g., `ASIN(0)` returns 0.000000).

`ATAN(`*`number`*`)`
: Returns the inverse tangent of number in radians (e.g., `ATAN(1)` returns 0.785398.)

`ATAN2(`*`X, Y`*`)`
: Returns the inverse tangent of the point (*X*,*Y*) (e.g., `ATAN(-3,3)` returns -0.785398).

`BIN(`*`decimal`*`)`
: Returns the binary value of the given decimal number. This is equivalent to the function `CONV(decimal,10,2)` (e.g., `BIN(8)` returns 1000).

`BIT_COUNT(`*`number`*`)`
: Returns the number of bits that are set to 1 in the binary representation of the number (e.g., `BIT_COUNT(17)` returns 2).

`CEILING(`*`number`*`)`
: Returns the smallest integer larger than or equal to *number* (e.g., `CEILING (5.67)` returns 6).

`CHAR(`*`num1`*`[,`*`num2`*`,. . .])`
: Returns a string made from converting each of the numbers to the character corresponding to that ASCII value (e.g., `CHAR(122)` returns 'z').

`COALESCE(`*`expr1`*`, `*`expr2`*`, ...)`
: Returns the first non-null expression in the list (e.g., `COALESCE(NULL, NULL, 'cheese', 2)` returns 3).

`CONCAT(`*`string1`*`,`*`string2`*`[,`*`string3`*`,. . .])`
: Returns the string formed by joining together all of the arguments (e.g., `CONCAT('Hi',' ','Mom','!')` returns "Hi Mom!").

`CONV(`*`number, base1, base2`*`)`
: Returns the value of *number* converted from *base1* to *base2*. *Number* must be an integer value (either as a bare number or as a string). The bases can be any integer from 2 to 36 (e.g., `CONV(8,10,2)` returns 1000 (the number 8 in decimal converted to binary)).

Copyright © 2001 O'Reilly & Associates, Inc.

`COS(`*`radians`*`)`
Returns the cosine of the given number, which is in radians (e.g., `COS(0)` returns 1.000000).

`COT(`*`radians`*`)`
Returns the cotangent of the given number, which must be in radians (e.g., `COT(1)` returns 0.642093).

`CURDATE()/CURRENT_DATE()`
Returns the current date. A number of the form YYYYMMDD is returned if this is used in a numerical context, otherwise a string of the form `'YYYY-MM-DD'` is returned (e.g., `CURDATE()` could return "1998-08-24").

`CURTIME()/CURRENT_TIME()`
Returns the current time. A number of the form HHMMSS is returned if this is used in a numerical context, otherwise a string of the form HH:MM:SS is returned (e.g., `CURRENT_TIME()` could return 13:02:43).

`DATABASE()`
Returns the name of the current database (e.g., `DATABASE()` could return "mydata").

`DATE_ADD(`*`date,`*` INTERVAL `*`amount type`*`)/ADDDATE(`*`date,`*` INTERVAL `*`amount type`*`)`
Returns a date formed by adding the given amount of time to the given date. The type of time to add can be one of the following: `SECOND`, `MINUTE`, `HOUR`, `DAY`, `MONTH`, `YEAR`, `MINUTE_SECOND` (as "minutes:seconds"), `HOUR_MINUTE` (as "hours:minutes"), `DAY_HOUR` (as "days hours"), `YEAR_MONTH` (as "years-months"), `HOUR_SECOND` (as "hours:minutes:seconds"), `DAY_MINUTE` (as "days hours:minutes") and `DAY_SECOND` (as "days hours:minutes:seconds"). Except for those types with forms specified above, the amount must be an integer value (e.g., `DATE_ADD("1998-08-24 13:00:00", INTERVAL 2 MONTH)` returns "1998-10-24 13:00:00").

`DATE_FORMAT(`*`date, format`*`)`
Returns the date formatted as specified. The format string prints as given with the following values substituted:

%a Short weekday name (Sun, Mon, etc.)

%b Short month name (Jan, Feb, etc.)

%D Day of the month with ordinal suffix (1st, 2nd, 3rd, etc.)

%d Day of the month

%H 24-hour hour (always two digits, e.g., 01)

%h/%I
12-hour hour (always two digits, e.g., 09)

Copyright © 2001 O'Reilly & Associates, Inc.

%i Minutes

%j Day of the year

%k 24-hour hour (one or two digits, e.g., 1)

%l 12-hour hour (one or two digits, e.g., 9)

%M Name of the month

%m
Number of the month (January is 1).

%p AM or PM

%r 12-hour total time (including AM/PM)

%S Seconds (always two digits, e.g., 04)

%s Seconds (one or two digits, e.g., 4)

%T 24-hour total time

%U
Week of the year (new weeks begin on Sunday)

%W
Name of the weekday

%w
Number of weekday (0 is Sunday)

%Y Four digit year

%y Two digit year

%%
A literal "%" character.

`DATE_SUB(`*`date`*`, INTERVAL `*`amount type`*`)/SUBDATE(`*`date`*`, INTERVAL `*`amount type`*`)`

Returns a date formed by subtracting the given amount of time from the given date. The same interval types are used as with `DATE_ADD` (e.g., `SUBDATE("1999-05-20 11:04:23", INTERVAL 2 DAY)` returns "1999-05-18 11:04:23").

`DAYNAME(`*`date`*`)`

Returns the name of the day of the week for the given date (e.g., `DAYNAME('1998-08-22')` returns "Saturday").

`DAYOFMONTH(`*`date`*`)`

Returns the day of the month for the given date (e.g., `DAYOFMONTH('1998-08-22')` returns 22).

Copyright © 2001 O'Reilly & Associates, Inc.

DAYOFWEEK(*date*)

Returns the number of the day of the week (1 is Sunday) for the given date (e.g., DAY_OF_WEEK('1998-08-22') returns 7).

DAYOFYEAR(*date*)

Returns the day of the year for the given date (e.g., DAYOFYEAR('1983-02-15') returns 46).

DECODE(*blob*, *passphrase*)

Decodes encrypted binary data using the specified passphrase. The encrypted binary is expected to be one encrypted with the ENCODE() function:

```
mysql> SELECT DECODE(ENCODE('open sesame', 'please'), 'please');
+---------------------------------------------------+
| DECODE(ENCODE('open sesame', 'please'), 'please') |
+---------------------------------------------------+
| open sesame                                       |
+---------------------------------------------------+
1 row in set (0.01 sec)
```

DEGREES(*radians*)

Returns the given argument converted from radians to degrees (e.g., DEGREES(2*PI()) returns 360.000000).

ELT(*number,string1,string2, . . .*)

Returns *string1* if *number* is 1, *string2* if *number* is 2, etc. A null value is returned if *number* does not correspond with a string (e.g., ELT(3, "once","twice","thrice","fourth") returns “thrice”).

ENCODE(*secret*, *passphrase*)

Creates a binary encoding of the *secret* using the *passphrase* as salt. You may later decode the secret using DECODE() and the passphrase.

ENCRYPT(*string*[, *salt*])

Password-encrypts the given string. If a salt is provided, it is used to generate the password (e.g., ENCRYPT('mypass','3a') could return “3afi4004idgv”).

EXP(*power*)

Returns the number *e* raised to the given power (e.g., EXP(1) returns 2. 718282).

EXPORT_SET(*num*, *on*, *off*, [*separator*, [*num_bits*]])

Examines a number and maps the on and off bits in that number to the strings specified by the on and off arguments. Examples:

```
mysql> SELECT EXPORT_SET(5, "y", "n", "", 8);
+--------------------------------+
| EXPORT_SET(5, "y", "n", "", 8) |
+--------------------------------+
| ynynnnnn                       |
+--------------------------------+
1 row in set (0.00 sec)
```

Copyright © 2001 O’Reilly & Associates, Inc.

```
mysql> SELECT EXPORT_SET(5, "y", "n", ",", 8);
+--------------------------------+
| EXPORT_SET(5, "y", "n", ",", 8) |
+--------------------------------+
| y,n,y,n,n,n,n,n                |
+--------------------------------+
1 row in set (0.00 sec)
```

`EXTRACT(`*`interval`*` FROM `*`datetime`*`)`

Returns the specified part of a `DATETIME` (e.g., `EXTRACT(YEAR FROM '2001-08-10 19:45:32')` returns 2001).

`FIELD(`*`string,string1,string2, . . .`*`)`

Returns the position in the argument list (starting with *string1*) of the first string that is identical to *string*. Returns 0 if no other string matches *string* (e.g., `FIELD('abe','george','john','abe','bill')` returns).

`FIND_IN_SET(`*`string,set`*`)`

Returns the position of *string* within *set*. The *set* argument is a series of strings separated by commas (e.g., `FIND_IN_SET ('abe', 'george, john, abe, bill')` returns 3).

`FLOOR(`*`number`*`)`

Returns the largest integer smaller than or equal to *number* (e.g., `FLOOR(5.67)` returns 5).

`FORMAT(`*`number, decimals`*`)`

Neatly formats the given number, using the given number of decimals (e.g., `FORMAT(4432.99134,2)` returns "4,432.99").

`FROM_DAYS(`*`days`*`)`

Returns the date that is the given number of days (where day 1 is the Jan 1 of year 1) (e.g., `FROM_DAYS(728749)` returns "1995-04-02").

`FROM_UNIXTIME(`*`seconds`*`[, `*`format`*`])`

Returns the date (in GMT) corresponding to the given number of seconds since the epoch (January 1, 1970 GMT). If a format string (using the same format as `DATE_FORMAT`) is given, the returned time is formatted accordingly (e.g., `FROM_UNIXTIME(903981584)` returns "1998-08-24 18:00:02").

`GET_LOCK(`*`name,seconds`*`)`

Creates a named user-defined lock that waits for the given number of seconds until timeout. This lock can be used for client-side application locking between programs that cooperatively use the same lock names. If the lock is successful, 1 is returned. If the lock times out while waiting, 0 is returned. All others errors return a `NULL` value. Only one named lock may be active at a time for a singe session. Running `GET_LOCK()` more than once will silently

Copyright © 2001 O'Reilly & Associates, Inc.

remove any previous locks (e.g., `GET_LOCK("mylock",10)` could return 1 within the following 10 seconds).

`GREATEST(`*`num1, num2`*`[, `*`num3, . . .`*` ])`
: Returns the numerically largest of all of the arguments (e.g., `GREATEST(5,6,68,1,4)` returns 68).

`HEX(`*`decimal`*`)`
: Returns the hexadecimal value of the given decimal number. This is equivalent to the function `CONV(decimal,10,16)` (e.g., `HEX(90)` returns "3a").

`HOUR(`*`time`*`)`
: Returns the hour of the given time (e.g., `HOUR('15:33:30')` returns 15).

`IF(`*`test, value1, value2`*`)`
: If *test* is true, returns *value1*, otherwise returns *value2*. The *test* value is considered to be an integer, therefore floating point values must be used with comparison operations to generate an integer (e.g., `IF(1>0,"true","false")` returns true).

`IFNULL(`*`value, value2`*`)`
: Returns *value* if it is not null, otherwise returns *value2* (e.g., `IFNULL(NULL, "bar")` returns "bar").

`INSERT(`*`string,position,length,new`*`)`
: Returns the string created by replacing the substring of *string* starting at *position* and going *length* characters with *new* (e.g., `INSERT('help',3,1,' can jum')` returns "he can jump").

`INSTR(`*`string,substring`*`)`
: Identical to `LOCATE` except that the arguments are reversed (e.g., `INSTR('makebelieve','lie')` returns 7).

`ISNULL(`*`expression`*`)`
: Returns 1 if the expression evaluates to `NULL`, otherwise returns 0 (e.g., `ISNULL(3)` returns 0).

`INTERVAL(`*`A,B,C,D, . . .`*` )`
: Returns 0 if *A* is the smallest value, 1 if *A* is between *B* and C, 2 if *A* is between *C* and *D*, etc. All of the values except for *A* must be in order (e.g., `INTERVAL(5,2,4,6,8)` returns 2 (because 5 is in the second interval, between 4 and 6).

`LAST_INSERT_ID()`
: Returns the last value that was automatically generated for an `AUTO_INCREMENT` field (e.g., `LAST_INSERT_ID()` could return 4).

Copyright © 2001 O'Reilly & Associates, Inc.

`LCASE(`*`string`*`)`/`LOWER(`*`string`*`)`

Returns *string* with all characters turned into lower case (e.g., `LCASE('BoB')` returns "bob").

`LEAST(`*`num1, num2`*`[, `*`num3, . . .`*`])`

Returns the numerically smallest of all of the arguments (e.g., `LEAST(5,6,68,1,4)` returns 1).

`LEFT(`*`string,length`*`)`

Returns *length* characters from the left end of *string* (e.g., `LEFT("12345",3)` returns "123").

`LENGTH(`*`string`*`)`/`OCTET_LENGTH(`*`string`*`)`/`CHAR_LENGTH(`*`string`*`)`/`CHARACTER_LENGTH(`*`string`*`)`

Returns the length of *string* (e.g., `CHAR_LENGTH('Hi Mom!')` returns 7). In character sets that use multibyte characters (such as Unicode, and several Asian character sets), one character may take up more than one byte. In these cases, MySQL's string functions should correctly count the number of characters, not bytes, in the string. However, in versions prior to 3.23, this did not work properly and the function returned the number of bytes.

`LOAD_FILE(`*`filename`*`)`

Reads the contents of the specified file as a string. This file must exist on the server and be world readable. Naturally, you must also have `FILE` privileges.

`LOCATE(`*`substring,string`*`[,`*`number`*`])`/`POSITION(`*`substring,string`*`)`

Returns the character position of the first occurrence of *substring* within *string*. If *substring* does not exist in *string*, 0 is returned. If a numerical third argument is supplied to `LOCATE`, the search for *substring* within *string* does not start until the given position within *string* (e.g., `LOCATE('SQL','MySQL')` returns 3).

`LOG(`*`number`*`)`

Returns the natural logarithm of *number* (e.g., `LOG(2)` returns 0.693147).

`LOG10(`*`number`*`)`

Returns the common logarithm of *number* (e.g., `LOG10(1000)` returns 3. 000000).

`LPAD(`*`string,length,padding`*`)`

Returns *string* with padding added to the left end until the new string is *length* characters long (e.g., `LPAD(' Merry X-Mas',18,'Ho')` returns "HoHoHo Merry X-Mas").

`LTRIM(`*`string`*`)`

Returns *string* with all leading whitespace removed (e.g., `LTRIM('    Oops')` returns "Oops").

Copyright © 2001 O'Reilly & Associates, Inc.

`MAKE_SET(bits, string1, string2, ...)`

Creates a MySQL `SET` based on the binary representation of a number by mapping the on bits in the number to string values. Example:

```
mysql> SELECT MAKE_SET(5, "a", "b", "c", "d", "e", "f");
+------------------------------------------+
| MAKE_SET(5, "a", "b", "c", "d", "e", "f") |
+------------------------------------------+
| a,c                                      |
+------------------------------------------+
1 row in set (0.01 sec)
```

`MD5(string)`

Creates an MD5 checksum for the specified string. The MD5 checksum is always a string of 32 hexadecimal numbers.

`MID(string,position,length)/SUBSTRING(string,position,length)/`
`SUBSTRING(string FROM position FOR length)`

Returns the substring formed by taking *length* characters from *string*, starting at *position* (e.g., `SUBSTRING('12345',2,3)` returns "234").

`MINUTE(time)`

Returns the minute of the given time (e.g., `MINUTE('15:33:30')` returns 33).

`MOD(num1, num2)`

Returns the modulo of *num1* divided by *num2*. This is the same as the % operator (e.g., `MOD(11,3)` returns 2).

`MONTH(date)`

Returns the number of the month (1 is January) for the given date (e.g., `MONTH('1998-08-22')` returns 8).

`MONTHNAME(date)`

Returns the name of the month for the given date (e.g., `MONTHNAME('1998-08-22')` returns "August").

`NOW()/SYSDATE()/CURRENT_TIMESTAMP()`

Returns the current date and time. A number of the form YYYYMMDDHHMMSS is returned if this is used in a numerical context, otherwise a string of the form 'YYYY-MM-DD HH:MM:SS' is returned (e.g., `SYSDATE()` could return "1998-08-24 12:55:32").

`OCT(decimal)`

Returns the octal value of the given decimal number. This is equivalent to the function `CONV(decimal,10,8)` (e.g., `OCT(8)` returns 10).

`PASSWORD(string)`

Returns a password-encrypted version of the given string (e.g., `PASSWD('mypass')` could return "3afi4004idgv").

Copyright © 2001 O'Reilly & Associates, Inc.

PERIOD_ADD(*date,months*)

Returns the date formed by adding the given number of months to *date* (which must be of the form YYMM or YYYYMM) (e.g., PERIOD_ADD(9808,14) returns 199910).

PERIOD_DIFF(*date1, date2*)

Returns the number of months between the two dates (which must be of the form YYMM or YYYYMM) (e.g., PERIOD_DIFF(199901,8901) returns 120).

PI()

Returns the value of pi: 3.141593.

POW(*num1, num2*)/POWER(*num1, num2*)

Returns the value of *num1* raised to the *num2* power (e.g., POWER(3,2) returns 9.000000).

QUARTER(*date*)

Returns the number of the quarter of the given date (1 is January-March) (e.g., QUARTER('1998-08-22') returns 3).

RADIANS(*degrees*)

Returns the given argument converted from degrees to radians (e.g., RADIANS(-90) returns -1.570796).

RAND([*seed*])

Returns a random decimal value between 0 and 1. If an argument is specified, it is used as the seed of the random number generator (e.g., RAND(3) could return 0.435434).

RELEASE_LOCK(*name*)

Removes the named locked created with the GET_LOCK function. Returns 1 if the release is successful, 0 if it failed because the current thread did not own the lock and a NULL value if the lock did not exist (e.g., RELEASE_LOCK("mylock")).

REPEAT(*string,number*)

Returns a string consisting of the original *string* repeated *number* times. Returns an empty string if *number* is less than or equal to zero (e.g., REPEAT('ma',4) returns 'mamamama').

REPLACE(*string,old,new*)

Returns a string that has all occurrences of the substring *old* replaced with *new* (e.g., REPLACE('*black jack*','*ack*','*oke*') returns "bloke joke").

REVERSE(*string*)

Returns the character reverse of *string* (e.g., REVERSE('my bologna') returns "angolob ym").

Copyright © 2001 O'Reilly & Associates, Inc.

`RIGHT(`*`string,length`*`)/SUBSTRING(string FROM length)`
: Returns *length* characters from the right end of *string* (e.g., `SUBSTRING("12345" FROM 3)` returns "345").

`ROUND(`*`number`*`[,`*`decimal`*`])`
: Returns *number*, rounded to the given number of decimals. If no *decimal* argument is supplied, *number* is rounded to an integer (e.g., `ROUND(5.67,1)` returns 5.7).

`RPAD(`*`string,length,padding`*`)`
: Returns *string* with padding added to the right end until the new string is *length* characters long (e.g., `RPAD('Yo',5,'!')` returns "Yo!!!").

`RTRIM(`*`string`*`)`
: Returns *string* with all trailing whitespace removed (e.g., `RTRIM('Oops        ')` returns "Oops").

`SECOND(`*`time`*`)`
: Returns the seconds of the given time (e.g., `SECOND('15:33:30')` returns 30).

`SEC_TO_TIME(`*`seconds`*`)`
: Returns the number of hours, minutes and seconds in the given number of seconds. A number of the form HHMMSS is returned if this is used in a numerical context, otherwise a string of the form HH:MM:SS is returned (e.g., `SEC_TO_TIME(3666)` returns "01:01:06").

`SIGN(`*`number`*`)`
: Returns -1 if *number* is negative, 0 if it's zero, or 1 if it's positive (e.g., `SIGN(4)` returns 1).

`SIN(`*`radians`*`)`
: Returns the sine of the given number, which is in radians (e.g., `SIN(2*PI())` returns 0.000000).

`SOUNDEX(`*`string`*`)`
: Returns the Soundex code associated with string (e.g., `SOUNDEX('Jello')` returns "J400").

`SPACE(`*`number`*`)`
: Returns a string that contains *number* spaces (e.g., `SPACE(5)` returns "   ").

`SQRT(`*`number`*`)`
: Returns the square root of *number* (e.g., `SQRT(16)` returns 4.000000).

`STRCMP(`*`string1, string2`*`)`
: Returns 0 if the strings are the same, -1 if *string1* would sort before than *string2*, or 1 if *string1* would sort after than *string2* (e.g., `STRCMP('bob','bobbie')` returns -1).

Copyright © 2001 O'Reilly & Associates, Inc.

`SUBSTRING(string,position)`
: Returns all of *string* starting at *position* characters (e.g., `SUBSTRING("123456",3)` returns "3456").

`SUBSTRING_INDEX(string,character,number)`
: Returns the substring formed by counting *number* of *character* within *string* and then returning everything to the right if count is positive, or everything to the left if count is negative (e.g., `SUBSTRING_INDEX('1,2,3,4,5',',',-3)` returns "1,2,3").

`TAN(radians)`
: Returns the tangent of the given number, which must be in radians (e.g., `TAN(0)` returns 0.000000).

`TIME_FORMAT(time, format)`
: Returns the given time using a format string. The format string is of the same type as `DATE_FORMAT`, as shown earlier.

`TIME_TO_SEC(time)`
: Returns the number of seconds in the *time* argument (e.g., `TIME_TO_SEC('01:01:06')` returns 3666).

`TO_DAYS(date)`
: Returns the number of days (where day 1 is the Jan 1 of year 1) to the given date. The date may be a value of type `DATE`, `DATETIME` or `TIMESTAMP`, or a number of the form YYMMDD or YYYYMMDD (e.g., `TO_DAYS(19950402)` returns 728749).

`TRIM([BOTH|LEADING|TRAILING] [remove] [FROM] string)`
: With no modifiers, returns *string* with all trailing and leading whitespace removed. You can specify whether to remove either the leading or the trailing whitespace, or both. You can also specify another character other than space to be removed (e.g., `TRIM(both '-' from '---look here---')` returns "look here").

`TRUNCATE(number, decimals)`
: Returns *number* truncated to the given number of decimals (e.g., `TRUNCATE(3.33333333,2)` returns 3.33).

`UCASE(string)/UPPER(string)`
: Returns *string* with all characters turned into uppercase (e.g., `UPPER('Scooby')` returns "SCOOBY").

`UNIX_TIMESTAMP([date])`
: Returns the number of seconds from the epoch (January 1, 1970 GMT) to the given date (in GMT). If no date is given, the number of seconds to the current date is used (e.g., `UNIX_TIMESTAMP('1998-08-24 18:00:02')` returns 903981584).

Copyright © 2001 O'Reilly & Associates, Inc.

`USER()/SYSTEM_USER()/SESSION_USER()`
Returns the name of the current user (e.g., `SYSTEM_USER()` could return "ryarger").

`VERSION()`
Returns the version of the MySQL server itself (e.g., `VERSION()` could return "3.22.5c-alpha").

`WEEK(`*`date`*`)`
Returns the week of the year for the given date (e.g., `WEEK('1998-12-29')` returns 52).

`WEEKDAY(`*`date`*`)`
Returns the numeric value of the day of the week for the specified date. Day numbers start with Monday as 0 and end with Sunday as 6.

`YEAR(`*`date`*`)`
Returns the year of the given date (e.g., `YEAR('1998-12-29')` returns 1998).

Copyright © 2001 O'Reilly & Associates, Inc.

19

MySQL System Variables

A variety of settings can be used to customize the operation of MySQL. Settings fall into two main categories:

- Operating System Environment variables
- MySQL variables, which can be set via the command line or in a configuration file.

Variables defined in multiple places follow obey the following order of precedence:

1. Items defined on the command line take precedence over configuration files and environment variables.
2. Items defined in configuration files take precedence over environment variables.

For more information on the use of configuration files to set system variables and command line options, refer to Chapter 5, "Database Administration."

Environment Variables

The following variables are specific to MySQL programs. They may be defined in the current shell or as part of a shell script. To set a variable for the MySQL daemon (*mysqld*), define the variable in the *safe_mysqld* script that is used to start the daemon or define the variables in the MySQL configuration file.

MYSQL_DEBUG
: The debug trace level for the program. This option can be used with any MySQL program. There are a wide variety of debug trace options which are documented in the MySQL documented in the MySQL documentation at *http://www.mysql.com/documentation* in the section on "Debugging a MySQL server" and "Debugging a MySQL client."

Copyright © 2001 O'Reilly & Associates, Inc.

MYSQL_HOST
: The hostname used to connect to a remote MySQL database server. This option can be used with any of the MySQL client programs (*mysql*, *mysqlshow*, *mysqladmin*, etc.). This is equivalent to the —host command line option.

MYSQL_HISTFILE
: The location of the MySQL history file, used by the mysql client. By default this is "$HOME/.mysql_history". There is no equivalent command line options.

MYSQL_PWD
: The password used to connect to the MySQL database server. This option can be used with any of the MySQL client programs, and is equivalent to the --password commnd line option.

> Be careful where you put your passwords. A common use for environment variables is to set them within scripts. Of course, setting this particular variable in a script would make your password visible to anyone who can run the script. Even setting the variable manually on the command line exposes it to the superuser and any else who has the ability to examine the system memory.

MYSQL_TCP_PORT
: When used with a client program, this is the TCP port on a remote machine used to connect to the MySQL database server. When used with *mysqld*, this is the port used to listen for incoming connections. This is equivalent to the --port command line option.

MYSQL_UNIX_PORT
: When used with a client program, this is the Unix socket file used to connect to the MySQL database server. When used with *mysqld*, this is the name of the Unix socket file created that allows local connections. This is equivalent to the --socket command line option.

In addition, the MySQL programs use the following environment variables that are routinely set as part of the Unix environment.

EDITOR
VISUAL
: The path of the default editor. The *mysql* program uses this program to edit SQL statements if a `\e` or `edit` command is encountered.

HOME
: The home directory of the current user.

LOGIN
LOGNAME
USER
: The username of the current user. On Windows, you can use the USER environment variable to indicate the default user when connecting to mysqld.

PATH
: The list of directories used to find programs.

Copyright © 2001 O'Reilly & Associates, Inc.

POSIXLY_CORRECT
: If this variable is defined, no special processing is done on command line options. Otherwise, command line options are reordered so that extended options can be used. This variable can be used with any MySQL program.

TMP
TMPDIR
: The directory in which temporary files are kept. If this variable is not defined '/tmp' is used.

TZ
: The time zone of the local machine.

UMASK
: The umask used when creating new files.

UMASK_DIR
: The umask used with creating new directories. This is ANDed with UMASK.

The following environment variables influence the configuration of your MySQL build. Use these when building MySQL from source.

CCX
: Indicates which C++ compiler to use. This is used by the configure script.

CC
: Indicates which C compiler to use. Used by the configure script.

CFLAGS
: Indicates which C compiler flags to use.

CCXXFLAGS
: Indicates which C++ compiler flags to use.

MySQL System Variables

In this section we document the system variables for the MySQL server mysqld. The client programs also have variables and command line options that control their operation. These are documented with the individual clients and utilities in Chapter 20, "MySQL Programs and Utilities". You also may to refer to the command line reference for mysqld in Chapter 20

Many of these options are supplied via the –O or –set-variable command line option to mysqld. Some are specified using specific command line options. Unless otherwise noted these are specified using the -O or -set-variable.

The current values of these options can be determined by using the show variables SQL command or with the mysqladmin variables command.

ansi_mode
: This indicates whether the ANSI mode is on or off. This is ON if mysqld is started with the --ansi option.

back_log
: The number of TCP connections that can be queued at once. When MySQL receives a connection request, it starts a new thread to handle that connection. There is a short

lag between the time the request is received and the session thread is created. If a large number of connection requests hit the MySQL server at the same time, they are queued up. This should only be increased if you expect a high-number of simultaneous connection requests. Another thing to note is that the operating system typically imposes a cap on back_log. Setting back_log higher than the operating system limit will be ineffective.

basedir
: The location of your MySQL installation. All other paths are usually resolved relative to this. Set with the --basedir command line option.

bdb_cache_size
: The size of the buffer used to cache BDB rows and indexes. If you are not using BDB tables, use the --skip-bdb option so that memory is not wasted on this cache.

bdb_log_buffer_size
: ???

bdb_home
: The location of your BDB tables. Set with the --bdb-home command line option.

bdb_max_lock
: The maximum number of locks allowed per BDB table. By default this is set to 1000. This should be increased if you get errors like "bdb: Lock table is out of available locks" or "Got error 12 from …".

bdb_logdir
: The location of your BDB log files. Set with the --bdb_logdir command line option.

bdb_shared_data
: Indicates whether you are using shared data for BDB tables. This is ON if mysqld is started with the --bdb_shared_data option.

bdb_tmpdir
: The location of your BDB temporary files. Set with the --bob-tmpdir command line option.

binlog_cache_size
: The size of the cache for storing binary log transactions. If you use large, multi-statement transactions, this can be increased to improve performance.

character_set
: The default character set. This is set with the --default-character-set command line option.

character_sets
: The supported character sets. The available character sets are determined at compile time. The --with-charset and --with-extra-charsets options to the configure script. See Chapter 3 for more information on installing character sets.

concurrent_inserts
: If this is ON, MySQ.L will allow INSERTs on MyISAM tables concurrently with SELECTs. The default is ON. This is turned off with the --safe command line option or the --skip-new command line option.

 Copyright © 2001 O'Reilly & Associates, Inc.

connect_timeout
: The number of seconds the *mysqld* server waits for a connect packet before responding with “Bad handshake”.

datadir
: The location of the database files. Set with the --datadir command line option. By default this is <basedir>/data.

delay_key_write
: If this is ON, MySQL will not flush the key buffer on every indedx update. It will only flush this buffer on table close. This only applies to tables created with the delay_key_write CREATE TABLE option. By default this is ON. You can disable this with the --skip-new or --safe-mode command line options.

  This increases the performance of key updates considerably, however we recommend automatic checking of all tables with myisamchk --fast --force.

  If you use the --delay-key-write-for-all-tables command line option, MySQL will treat all tables as if they had been created with the delay_key_write option.

delayed_insert_limit
: Causes the `INSERT DELAYED` handler to check whether there are any `SELECT` statements pending after inserting *delayed_insert_limit* rows. If so, the handler allows the statements to execute before continuing.

delayed_insert_timeout
: How long an `INSERT DELAYED` thread should wait for `INSERT` statements to finish before terminating.

delayed_queue_size
: How big a queue (in rows) should be allocated for handling `INSERT DELAYED`. If the queue becomes full, any client that does an `INSERT DELAYED` must wait until there is room in the queue again.

flush
: If this is ON, MySQL will flush all changes disk after every SQL command. Normally MySQL writes the data and lets the operating system handle flushing. This is enabled with --flush command line option.

flush_time
: If set, all tables are closed then every flush_time seconds to free resources and synchronize changes to disk. This is only recommended for use on Windows 95/98 or on systems with very little resources.

have_bdb
: If this is YES, mysqld supports Berkeley DB tables (which is determined at compile time with the --with-dbd configure option). NO mains that MySQL was not compiled with support for this option. DISABLED means the server was started with the --skip-bdb command line option.

have_innodb
: If this is YES, mysqld supports Berkeley DB tables (which is determined at compile time with the --with-dbd configure option). NO means that MySQL was not compiled with support for this option. DISABLED means the server was started with the --skip-bdb command line option.

Copyright © 2001 O’Reilly & Associates, Inc.

have_raid
: If this is YES, mysqld has support for RAID compiled in. NO means that the server was compiled without support for RAID.

have_ssl
: If this is YES, mysqld supports SSL encryption on the client/server protocol. This is enabled at compile time with the --with-ssl configure option. NO means that MySQL was compiled without support for this option.

init_file
: The name of the file specified with the --init-file command line option. nysqld will execute the SQL commands included in this file during startup.

join_buffer_size
: The size of a buffer used when performing full table joins (i.e. joings that do not use indexes). This buffer is allocated one time for each full join between two tables. Normally the best way to increase performance of full joins is to add indexes (see Chapter 6 for more details). However, if adding indexes is not possible, increasing this value will improve the performance of full joins.

key_buffer_size
: The size of a buffer shared by all threads to store index blocks. Increasing this buffer size will improve performanced of index reads and writes. Increase this to as much as you can possibly afford. For example, 64Mb on a dedicated MySQL server with 256 Mb of physical memory is common. Be warned, however, that setting this too high (i.e. greater than 50% of physical memory) can cause your operating system to start paging heavily, which will negatively impact the performance of your server down, sometimes severely.

language
: The language used for error messages. This is specified with the --language command line option.

large_file_support
: This is ON if mysqld was compiled support for big files.

locked_in_memory
: This is ON if mysqld was locked in memory with the --memlock command line option.

log
: This is ON if query logging is enabled. This is enabled with the --log command line option. See Chapter 5 for more information on the query log.

log_update
: This is ON if update logging is enabled. This is enabled with the --log-update command line option. See Chapter 5 for more information on the update log.

log_bin
: This is ON if binary logging is enabled. This is enabled with the --log-bin command line option.

log_slave_updates
: Turn this ON if updates from the slave should be logged.

Copyright © 2001 O'Reilly & Associates, Inc.

long_query_time
If set, the slow_queries counter is incremented each time a query takes more than *long_query_time* seconds. If you have enabled the slow query log with the --log-slow-queries command line option, each slow query will be logged in the slow query log. See Chapter 5 for more information about the slow query log.

lower_case_table_names
If set to 1, tables names are stored in lowercase on the file system. This will allow for case-insensitive table access on Unix.

max_allowed_packet
The maximum size of one packet. The message buffer is initialized to net_buffer_length bytes, but is allowed to grow up to max_allowed_packet bytes.

By default, this is small to catch big packets that are possibly erroneous. However, if you are using BLOB columns, you will need to increase this value to the size of the largest BLOB you use.

This is limited to 16M in the current protocol.

max_binlog_cache_size
Increase this if a multi-statement transaction returns the error “Multi-statement transaction required more than ‘max_binlog_cache_size’ bytes of storage”.

max_binlog_size
Rotate the the binary log when reaches the size specified by max_binlog_size. The value must be greater than 1024 bytes, but no more than 1Gb. The default is 1Gb.

max_connections
The number of simultaneous clients allowed by mysqld. Upping this value increases the number of file descriptors needed by mysqld. See the open_files_limit variable for more information.

max_connect_errors
If set, the server blocks further connections from a remote host when the number of interrupted connections from that host exceeds *max_connect_errors*. You can unblock a host with the command `FLUSH HOSTS`.

max_delayed_threads
Start no more than this number of threads to handle `INSERT DELAYED`. If a client tries to use `INSERT DATA` to insert new data after this limit is reached, the request is handled as if the `DELAYED` attribute was not specified.

max_heap_table_size
mysqld disallows creation of help tables larger than this size.

max_join_size
When mysqld estimates a join will read more than max_join_size records, it returns an error. Use this variable if you need to protect against ill-formed queries.

max_sort_length
The maximum number of characters to examine when sorting a `BLOB` or `VARCHAR` field. Only the first max_sort_length bytes of each value are used for the sort operation.

Copyright © 2001 O’Reilly & Associates, Inc.

max_user_connections
The maximum number of active connections for a single user. A value of 0 indicates no limit.

max_tmp_tables
Maximum number of temporary tables a client can keep open at the same time. This option doesn't do anything yet, but it is going to control the maximum number of temporary tables a client can keep open at the same time.

max_write_lock_count
After this many write locks, allow some read locks to run in between.

myisam_recover_options
This is used to control how MySQL handles MyISAM tables that are marked as crashed or weren't opened properly. This is specified with th --myisam-recover command line option. Refer to the mysqld documentation in Chapter 20 for more details on option.

myisam_sort_buffer_size
The size of a buffer that is allocated when sorting an index during a REPAIR operation or when creating indexes with CREATE INDEX or ALTER TABLE.

myisam_max_extra_sort_file_size
If the temporary file created for fast index creation would be myisam_max_extra_sort_file_size megabytes bigger that using the key cache, mysqld will use the key cache instead. This is primarily to force the use of the slower key cache method when creating long character keys in large tables.

Note that this variable is specified in megabytes.

myisam_max_sort_file_size
If the temporary file created for recreating an index is greater than myisam_max_sort_file_size megabytes, mysqld will use the key cache instead. Note that this variable is specified in megabytes.

net_buffer_length
The client/server communication buffer is reset to this size between queries. If you have very little memory, you can set this to the expected length of SQL statements from clients. If statements exceed this length, this buffer is automatically enlarged, up to max_allowed_packet length.

net_read_timeout
The number of seconds to wait for more data from a connection before aborting the read operation. This only applies when we expect data from a connection, otherwise the timeout is specified by net_write_timeout.

net_retry_count
Retry net_retry_count times if a read on a communication port is interrupted. On FreeBSD, this value should be high since internal interrupts are sent to all threads.

net_write_timeout
The number of seconds to wait for a block to be written to a connections before aborting the write.

open_files_limit
If set to a non-zero value, mysqld will reserve open_files_limit file descriptors with the setrlimit() system call.

Copyright © 2001 O'Reilly & Associates, Inc.

If this is zero, then the greater of max_connections*5 and max_connections + table_cache*2 will be reserved.

Increase this if you receive a 'Too many open files' error from mysqld.

pid_file
: The path to the pid file used by safe_mysqld. This is specified with the --pid-file command line option.

port
: The port on which mysqld is listening for TCP/IP connections. This is specified using the --port command line option.

protocol_version
: The protocol version used by the MySQL server. The --old-protocol command line option will enable the old MySQL protocol.

query_buffer_size
: The initial size of the query buffer. If most of your queries are long, like when inserting blobs, increase this value.

record_buffer
: The size of a buffer used to perform a sequential scanson tables (i.e. for non-index retrievals). Increase this if you perform lots of non-index reads from tables.

record_rnd_buffer
: When reading rows in sorted order after a sort, they are read through this buffer to avoid disk reads. If not set, the size of the record_rnd_buffer is the same as record_buffer.

safe_show_databases
: Don't show databases to a user unless they have been granted database or table privileges on that table. This can improve security, as it will prevent users from seeing databases they don't have access too.

server_id
: The value of the --server_id command line option.

skip_locking
: This is ON if system locking has been disabled. The --skip-locking command line option contols this. When this is OFF, you must shut down the server to run isamchk or myisamchk.

skip_networking
: This is ON if mysqld only allows local (socket) connections. The --skip-networking command line option controls this.

skip_show_databases
: If this is ON, mysqld will not allow users to run SHOW DATABASES if they don't have the PROCESS_PRIV privilege.

slave_read_timeout
: The number of seconds to wait for more data from a master/slave connection before aborting the read.

slow_launch_time
: If creating a thread takes longer than slow_launch_time seconds, the slow_launch_threads counter is incremented.

Copyright © 2001 O'Reilly & Associates, Inc.

socket
: The path to the UNIX socket used by mysqld. This is specified with the --socket command line option.

sort_buffer
: The size of the buffer used when performing sorts on retrieved data. Increasing this can speed up performance for queries that use `ORDER BY` or `GROUP BY` statements.

table_cache
: The maximum number of tables the database server can have open at once. Increasing this value increases the required number of file descriptors. See the max_open_files variable for more information. Make sure that your operating system can support the number of open file descriptors by the table_cache setting. If it can't MySQL may refuse connections, fail to perform queries, etc.

table_type
: The default table type.

thread_cache_size
: This specifies the number of threads to keep in a cache this is used to recycle client threads. Increasing this number can improve performance if you have a high number of connections.

thread_concurrency
: On Solaris, mysqld will invoke the thr_setconncurrency() system call with the value of this variable. This will give the operating system a hint about the desired number of threads to be run concurrently.

thread_stack
: The stack size for each thread. The default is large enough for most normal operation.

timezone
: The timezone from the server.

tmp_table_size
: The maximum size of temporary tables used by the database server. If an in-memory temporary table exceeds this size, MySQL will convert it to an on-disk MyISAM table. Increasing this value can improve your performance if you do lots of GROUP BY queries and have memory available.

tmpdir
: The directory used for temporary files and tables. This specified using the --tmpdir command line option.

version
: The version number of the mysqld server. This variable cannot be changed.

wait_timeout
: The number of seconds the server waits for activity on a connection before closing

Copyright © 2001 O'Reilly & Associates, Inc.

21

MySQL System Tables

In Chapter XXX: Security we saw how MySQL used its own internal tables to store the access information used to perform authentication. There are actually several internal tables that are created as part of every MySQL server installation. MySQL uses these tables for a variety of purposes.

Columns_Priv

Field	Type	Null	Key	Default	Extra
Host	char(60) binary		PRI		
Db	char(64) binary		PRI		
User	char(16) binary		PRI		
Table_name	char(64) binary		PRI		
Column_name	char(64) binary		PRI		
Timestamp	timestamp(14)	YES		NULL	
Column_priv	set('Select','Insert', 'Update','References')				

The columns_priv table controls column-level access to a MySQL database table. The column-level check is the final check performed in the access verification sequence describe in Chapter XXX: Security. The information in this table can be broken into four sections: location, scope, privilege and timestamp.

Location

These fields determine to whom this privilege applies. Any user who matches both of these fields in a given row will be subjected to the privilege rule for that row.

Host - The hostname or IP address of the user.

Copyright © 2001 O'Reilly & Associates, Inc.

User - The authenticated username of the user.

Scope

These fields determine exactly which column this rule is for. Any columns which match all three of these fields in a given row will have the privilege rule for that row apply.

Db - The database that contains the table with the column
Table_name - The name of the table containing the column
Column_name - The name of the column

Privledge

The singe privilege column, Column_priv, determines which privileges are allowed for location and scope defined in the other fields in this table. The value of this column is a set that can contain any of the following privileges: Select, Insert, Update, References. Chapter XXX: Security describes the meanings of the individual privilege options.

Timestamp

This field keeps a timestamp which records the last time the table was modified.

db

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The db table controls database-level access to a MySQL server. The database-level check is the second check (after user) performed in the access verification sequence describe in Chapter XXX: Security. The information in this table can be broken into three sections: location, scope and privilege.

 Copyright © 2001 O'Reilly & Associates, Inc.

Location

These fields determine to whom this privilege applies. Any user who matches both of these fields in a given row will be subjected to the privilege rule for that row.

Host - The hostname or IP address of the user.
User - The authenticated username of the user.

Scope

The sole scope field for this table, 'Db' determines exactly which database this rule is for. Any database that matches this field in a given row will have the privilege rule for that row apply.

Privledge

These fields determine which privileges are allowed for location and scope defined in the other fields in this table. The values of all of these fields is an enumeration consisting of either 'Yes' or 'No' depending on whether the privilege is allowed for a particular rule.

Select_priv - The Select privilege
Insert_priv - The Insert privilege
Update_priv - The Update privilege
Delete_priv - The Delete privilege
Create_priv - The Create privilege
Drop_priv - The Drop privilege
Grant_priv - The Grant privilege
References_priv - The References privilege
Index_priv - The Index privilege
Alter_priv - The Alter privilege

func

```
+-------+------------------------------+------+-----+----------+-------+
| Field | Type                         | Null | Key | Default  | Extra |
+-------+------------------------------+------+-----+----------+-------+
| name  | char(64) binary              |      | PRI |          |       |
| ret   | tinyint(1)                   |      |     | 0        |       |
| dl    | char(128)                    |      |     |          |       |
| type  | enum('function','aggregate') |      |     | function |       |
+-------+------------------------------+------+-----+----------+-------+
```

The func table contains a list of all currently active user defined functions. Chapter XXX: Extending MySQL describes how to create a user defined function as well as how to use the SQL commands CREATE FUNCTION and REMOVE FUNCTION to manipulate the data in this table. It should be noted that the fields in this table merely contain the location of the user defined function, not the function code itself; that is stored in a system-specific dynamic library.

Copyright © 2001 O’Reilly & Associates, Inc.

name - The name of the user defined function. This name must match the name of the C or C++ function within the dynamic library for this function.
ret - The return value of this function. This is stored as an integer that is keyed to an enumeration defined in the standard MySQL C header file. A String return value is 0, a real (floating point) numeric value is 1 and an integer numeric value is 2.
dl - The name of the dynamic library containing the function. This library must be accessible to MySQL through the dynamic loading mechanism defined for the MySQL server's system. For example, on most Unix systems, the library must be in a directory within the LD_LIBRARY_PATH environment variable.
type - This field defines whether the function is a standard function (takes one or more single-value arguments) or an aggregate function (is given a set of values on which to perform an operation, as with the built-in MySQL functions SUM(), COUNT() and AVG()). The value of this field is an enumeration which must be either 'function' or 'aggregate'.

host

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
```

The host table controls remote host-level access to a MySQL server. The host-level check is the third check (after user and db) performed in the access verification sequence describe in Chapter XXX: Security. The information in this table can be broken into three sections: location, scope and privilege.

Location

The sole location field, Host, determines to whom this privilege applies. Any user connection from a host that matches this field in a given row will be subjected to the privilege rule for that row.

Scope

The sole scope field for this table, 'Db' determines exactly which database this rule is for. Any database that matches this field in a given row will have the privilege rule for that row apply.

 Copyright © 2001 O'Reilly & Associates, Inc.

Privledge

These fields determine which privileges are allowed for location and scope defined in the other fields in this table. The values of all of these fields is an enumeration consisting of either 'Yes' or 'No' depending on whether the privilege is allowed for a particular rule.

Select_priv - The Select privilege
Insert_priv - The Insert privilege
Update_priv - The Update privilege
Delete_priv - The Delete privilege
Create_priv - The Create privilege
Drop_priv - The Drop privilege
Grant_priv - The Grant privilege
References_priv - The References privilege
Index_priv - The Index privilege
Alter_priv - The Alter privilege

tables_priv

```
+-------------+------------------------------------+------+-----+---------+-------+
| Field       | Type                               | Null | Key | Default | Extra |
+-------------+------------------------------------+------+-----+---------+-------+
| Host        | char(60) binary                    |      | PRI |         |       |
| Db          | char(64) binary                    |      | PRI |         |       |
| User        | char(16) binary                    |      | PRI |         |       |
| Table_name  | char(60) binary                    |      | PRI |         |       |
| Grantor     | char(77)                           |      | MUL |         |       |
| Timestamp   | timestamp(14)                      | YES  |     | NULL    |       |
| Table_priv  | set('Select','Insert','Update',    |      |     |         |       |
|             |     'Delete','Create','Drop',      |      |     |         |       |
|             |     'Grant','References',          |      |     |         |       |
|             |     'Index','Alter')               |      |     |         |       |
| Column_priv | set('Select','Insert',             |      |     |         |       |
|             |     'Update','References')         |      |     |         |       |
+-------------+------------------------------------+------+-----+---------+-------+
```

The tables_priv table controls table-level access to a MySQL database. The table-level check is the second to last check performed (before column-level) in the access verification sequence describe in Chapter XXX: Security. The information in this table can be broken into four sections: location, scope, privilege and other.

Location

These fields determine to whom this privilege applies. Any user who matches both of these fields in a given row will be subjected to the privilege rule for that row.

Host - The hostname or IP address of the user.
User - The authenticated username of the user.

Copyright © 2001 O'Reilly & Associates, Inc.

Scope

These fields determine exactly which table this rule is for. Any table that matches both of these fields in a given row will have the privilege rule for that row apply.

Db - The database that contains the table with the column
Table_name - The name of the table containing the column

Privledge

These fields determine which privileges are allowed for location and scope defined in the other fields in this table.

Table_priv - The value of this field is a set that determines the privileges allowed for the table(s) matching this rule. The values of this set can be any of the following: Select, Insert, Update, Delete, Create, Drop, Grant, References, Index, and Alter.
Column_priv - The value of this field is a set that determines the privileges allowed for all columns in the matching table(s). The values of this set can be any of the following: Select, Insert, Update, and References. Chapter XXX: Security describes the meanings of the individual privilege options.

Other

These fields store meta-data related to the access rule.

Timestamp – This field keeps a timestamp which records the last time the table was modified.
Grantor – The user name of the user which created this rule. This field will only be automatically populated if the rule was created using the SQL GRANT statement. If a rule is created by manually adding a row to this table, this field must also be filled in manually.

user

```
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Password        | char(16) binary |      |     |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Reload_priv     | enum('N','Y')   |      |     | N       |       |
| Shutdown_priv   | enum('N','Y')   |      |     | N       |       |
| Process_priv    | enum('N','Y')   |      |     | N       |       |
| File_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
| Index_priv      | enum('N','Y')   |      |     | N        |       |
| Alter_priv      | enum('N','Y')   |      |     | N        |       |
+-----------------+-----------------+------+-----+----------+-------+
```

The user table controls user-level access to a MySQL server. The user-level check is the first check performed in the access verification sequence describe in Chapter XXX: Security. The information in this table can be broken into two sections: location/identification and privilege. Note the absence of the scope-related fields that are present in all of the other security tables. As might be inferred from this, privileges set in this table apply to every database, table and column in the server.

Location/Identification

These fields determine to whom this privilege applies. Any user who matches all three of these fields in a given row will be subjected to the privilege rule for that row.

Host - The hostname or IP address of the user.
User - The username of the user.
Password - The password of the user.

Privledge

These fields determine which privileges are allowed for location and scope defined in the other fields in this table. The values of all of these fields is an enumeration consisting of either 'Yes' or 'No' depending on whether the privilege is allowed for a particular rule.

Select_priv - The Select privilege
Insert_priv - The Insert privilege
Update_priv - The Update privilege
Delete_priv - The Delete privilege
Create_priv - The Create privilege
Drop_priv - The Drop privilege
Grant_priv - The Grant privilege
References_priv - The References privilege
Index_priv - The Index privilege
Alter_priv - The Alter privilege
References_priv - The References privilege
Reload_priv - The Reload privilege
Shutdown_priv - The Shutdown privilege
Process_priv - The Process privilege
File_priv - The file privilege

Copyright © 2001 O'Reilly & Associates, Inc.

22

MySQL and Programs and Utilities

MySQL comes with a wealth of programs and utilities to make interacting with the database server easier. Some of these programs are used by the end user to read and write from the database, while others are meant for the database administrator to maintain and repair the database as a whole.

In this chapter, we'll provide detailed documentation for all the programs and utilities that are available. First, here's an inventory of them all with a brief description of each.

mysql
: The MySQL SQL command shell. This is presented first because it is used so frequently..

mysqld
: The MySQL server. In this section we cover all of the command line options for the mysql server. In addition, we cover safe_mysqld, the mysqld wrapper script, and mysql.server, a script for starting and stopping mysqld on System-V style Unix systems.

myisamchk/isamchk
: Maintenance utilities for MyISAM/ISAM files. Among other things, these are used check tables for errors and repair them.

myisampack/pack_isam
: Compressed, read-only table generators.

mysqlaccess
: A script to check the access privileges for a host, user, and database combination.

mysqladmin
: A utility for performing administrative operations.

mysqlbug
: A utility for submitting MySQL bugs.

Copyright © 2001 O'Reilly & Associates, Inc.

mysqlcheck
: Maintenance utility for MyISAM files. This is similar to myisamchk but can be used while the server is running.

mysqldump
: A utility to dump the contents a one or more databases as a set of SQL commands. Can be used to back up databases or create a copy of a database.

mysqlhotcopy
: A perl script to make a binary copy of a database.

mysqlimport
: A utility to load data into a database. This is essentially a command line version of the LOAD DATA INFILE SQL command.

mysqlshow
: A utility to show information about databases, tables and columns.

Configuration Files

Many of the programs and utilities allow you to specify options in a configuration file as well as on the command line. The utilities that support this are:

- mysql
- mysqladmin
- mysqld
- safe_mysqld
- mysqldump
- mysqlimport,
- myisamchk
- myisampack.

Each of the programs that support configuration files support the following options:

--no-defaults
: Don't read any option files.

--print-defaults
: Print the program name and all options that it will get.

--defaults-file=full-path-to-default-file
: Only use the given configuration file.

--defaults-extra-file=full-path-to-default-file
: Read this configuration file after the global configuration file but before the user configuration file.

See Chapter 5 for an in-depth discussion of how to set up and use configuration files.

 Copyright © 2001 O'Reilly & Associates, Inc.

mysql, the MySQL Command Shell

mysql

mysql [options] [database]

Since mysql is the utility you'll likely be using the most, we'll cover that first.

mysql is a simple SQL command shell. It is a general purpose client which will allow you to execute arbitrary SQL statements against your database. This is the utility you use when you want to run ad-hoc queries against your database, create tables or indexes, etc.

It supports interactive and non-interactive (i.e. as a filter) use. In interactive mode, GNU readline capabilities are provided and data is is displayed in an ASCII-table format. In non-interactive mode, the data is presented in a tab-separated format.

Using mysql interactively is simple. Simply type

```
% mysql <dbname>
```

or

```
% mysql --user=<username> --password=<password> <dbname>
```

where <dbname> is the name of the database you wish to connect to, and <username>/<password> are for the user you wish to connect as. If you don't specify --user, the $USER environment variable will be used. If you don't supply a password, mysql will prompt for it.

Once started, `mysql` presents a prompt. Here you can type SQL commands. Commands can span multiple lines and must be terminated with ';' or '\g'. So, for example, typing

```
mysql> SELECT *
mysql> FROM FOOBAR ;
```

would execute the SELECT statement. That's all there is too it.

mysql has command line editing (like a bash shell), because it uses the same GNU readline library that bash uses. For example, you can complete a word by using the tab key, press Ctrl-a to jump to the start of the current line or Ctrl-e to jump to the end, press Ctrl-r to perform a reverse search, and use the up arrow to retrieve the previous command.

`mysql` also provides history. By hitting the up/down arrow, you can scroll through your history of SQL commands. This works similar to `bash` history.

To run, mysql non-interactively, you redirect your SQL commands into mysql. You can also redirect the output to a file. For example,

```
% mysql --user=<user> --password=<password> <dbname> < script.sql > script.out
```

Copyright © 2001 O'Reilly & Associates, Inc.

In this way, the mysql command can be combined in shell pipelines just like any other UNIX filter. This is tremendously useful for constructing scripts to access your database.

mysql has a number of built-in commands. Each command has a long format and a short format. These are listed below. The short format is listed in parentheses.

When using full word commands (go, print, etc.) the command must be entered on a line by itself. Escape character commands (\g, \p, etc.) can be used at the end of any line. In addition, a semicolon can be used to end an SQL statement just like \g.

help (\h)
: Display the help for mysql.

? (\h)
: Synonym for `help'.

clear (\c)
: Clear command. Clear the query buffer.

connect (\r)
: Reconnect to the server. Optional arguments are db and host.

edit (\e)
: Edit command buffer with the text editor specified in the environment variable $EDITOR.

ego (\G)
: Send command to mysql server, display result vertically.

exit (\q)
: Exit mysql. Same as quit.

go (\g)
: Send command to mysql server.

nopager (\n)
: Disable pager, print to stdout.

notee (\t)
: Disable tee. Don't write into outfile.

pager (\P)
: Print the query results via the command specified in the PAGER environment variable or in the --pager command line option..

print (\p)
: Print current command.

quit (\q)
: Quit mysql.

rehash (\#)
: Rebuild completion hash.

source (\.)
: Execute a SQL script file. Takes a file name as an argument.

 Copyright © 2001 O'Reilly & Associates, Inc.

status (\s)
: Get status information from the server.

tee (\T)
: Append all output into given outfile.

use (\u)
: Use another database. Takes database name as argument.

mysql supports the following command line options:

-?, --help
: Display the help for mysql and exit.

-A, --no-auto-rehash
: Disable automatic rehashing. Normally, when mysql starts up, it reads the table and column names for a database to provide tab completion. When this is disabled, mysql will start up faster, but you will have to use the 'rehash' mysql command to get table and field completion.

-B, --batch
: Print results with a tab as separator, each row on a new line.

--character-sets-dir=...
: Directory where character sets are located.

-C, --compress
: Use compression in server/client protocol.

-#, --debug[=...]
: Enable the debug log. Default is 'd:t:o,/tmp/mysql.trace'.

-D, --database=...
: Database to use. This is mainly useful in the my.cnf file to set your default database.

--default-character-set=...
: Set the default character set.

-e, --execute=...
: Execute command and quit. This option allows you to supply an SQL command on the mysql command line. It will execute the command and return the results as if it were it --batch mode.

-E, --vertical
: Print the output of a query (rows) vertically. Without this option you can also force this output by ending your statements with \G.

-f, --force
: Continue even if we get a SQL error.

-g, --no-named-commands
: Long format built-in commands are disabled. Use * form only, or use named commands only in the beginning of a line ending with a semicolon (;).

-G, --enable-named-commands
: Long format built-in commands are enabled, as well as shortened * commands.

-i, --ignore-space
: Ignore space after function names.

Copyright © 2001 O'Reilly & Associates, Inc.

-h, --host=...
: Connect to the MySQL server on the specified host.

-H, --html
: Produce HTML output.

-L, --skip-line-numbers
: Don't write line number for errors. This is useful when one wants to compare result files that includes error messages

--no-pager
: Disable pager and print to stdout.

--no-tee
: Disable outfile.

-n, --unbuffered
: Flush buffer after each query.

-N, --skip-column-names
: Don't write column names in results.

-O, --set-variable var=option
: Set a variable. The mysql variables are described below. Use --help to list variables.

-o, --one-database
: Only update the database specified on the command line. This is useful for playing back a set of updates from the update log and/or binary log. All updates to databases other than the database on the command line will be ignored.

--pager[=...]
: Set the pager to use for displaying output in interactive mode. If this is unspecified, it defaults to the pager defined by your PAGER environment variable. Valid pagers are less, more, cat [> filename], etc. This option does not work in batch mode. --pager works only on UNIX.

-p[password], --password[=...]
: Password to use when connecting to server. If a password is not given on the command line, you will be prompted for it. Note that if you use the short form -p you can't have a space between the option and the password.

-P --port=...
: TCP/IP port number of the server you wish to connect to.

-q, --quick
: Don't cache results, instead print them row-by-row. This may slow down the server if the output is suspended.

-r, --raw
: Write column values without escape conversion. Typically used with --batch

-s, --silent
: Be more silent.

-S --socket=...
: Socket file of the server you wish to connect to.

-t --table
: Output in ASCII table format. This is the default in interactive mode.

 Copyright © 2001 O'Reilly & Associates, Inc.

-T, --debug-info
Print some debug information at exit.

--tee=outfile
Append everything into outfile. Does not work in batch mode.

-u, --user=#
User for login if not current user.

-U, --safe-updates[=#], --i-am-a-dummy[=#]
Only allow UPDATE and DELETE statements that are constrained in the WHERE clause by a indexed column. This is useful for preventing accidental deletion of all rows from a table, for example. In addition, the select_limit and max_join_size variables are consulted. SELECT statements are limited to select_limit rows, and all queries with joins that need to examine more than max_join_size rows are aborted. You can reset this option if you have it in your my.cnf file by using --safe-updates=0.

-v, --verbose
Enables more verbose output (-v -v -v enables the table output format).

-V, --version
Output version information and exit.

-w, --wait
Wait and retry if connection is down instead of aborting.

`mysql` also provides a small set of variables that can be set with the -O or --set-variable command:

connect_timeout
Number of seconds before connection is timed out. 0 indicates not timeout. The default is 0.

max_allowed_packet
Maximum packet length to send/receive from server.

net_buffer_length
Size of the buffer for TCP/IP and socket communication.

select_limit
Row limit for SELECT with --safe-updates/--i-am-a-dummy enabled.

max_join_size
Maximum number of rows to be examined to satisfy a join with --safe-updates/--i-am-a-dummy enabled.

mysqld, the MySQL Server

mysqld, mysqld-max

mysqld [*options*]

mysqld-max [*options*]

Copyright © 2001 O'Reilly & Associates, Inc.

mysqld is the MySQL server. The recommended way to invoke mysqld is via the safe_mysqld script (described below). mysqld-max is a version of the MySQL server with support for BDB and InnoDB tables compiled in.

mysqld supports the following options:

--ansi
Use ANSI SQL syntax instead of MySQL syntax. This has the following effects:

- || is acts the string concatenation operator instead of OR.
- Any number of spaces are allowed between a function name and the `('. This forces all function names to be treated as reserved words.
- `"' acts as an identifier quote character (like the MySQL ``' quote character) and not a string quote character.
- REAL is a synonym for FLOAT instead of a synonym of DOUBLE.
- The default transaction isolation level is SERIALIZABLE

--ansi is equivalent to --sql= REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,SERIALIZE,ONLY_FULL_GROUP_BY

-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 the file system. It solves most 'table full' errors, but also slows down the queries where in-memory tables would suffice. Since Version 3.23.2, MySQL is able to handle this automatically by using memory for small temporary tables and switching to disk tables where necessary.

--bind-address=IP
IP address to bind to.

--character-sets-dir=path
Directory where character sets are.

--chroot=path
Chroot mysqld daemon during startup. Recommended security measure. It will somewhat limit LOAD DATA INFILE and SELECT ... INTO OUTFILE though.

--core-file
Write a core file if mysqld dies. For some systems you must also specify --core-file-size to safe_mysqld. See the following section about safe_mysqld.

-h, --datadir=path
Path to the database root.

--default-character-set=charset
Set the default character set.

 Copyright © 2001 O'Reilly & Associates, Inc.

--default-table-type=type
Set the default table type for creating tables.

--debug[...]=
If MySQL is configured with --with-debug, you can use this option to get a trace file of what mysqld is doing.

--delay-key-write-for-all-tables
Don't flush key buffers between writes for any MyISAM table.

--enable-locking
Enable system locking. Note that if you use this option on a system which a not fully working lockd() (as on Linux) mysqld will deadlock.

-T, --exit-info
This is a bit mask of different flags one can use for debugging the mysqld server; One should not use this option if one doesn't know exactly what it does!

--flush
Flush all changes to disk after each SQL command. Normally MySQL only does a write of all changes to disk after each SQL command and lets the operating system handle the syncing to disk.

-?, --help
Display short help and exit.

--init-file=file
Read SQL commands from this file at startup.

-L, --language=...
Language to use for client error messages. May be given as a full path.

-l, --log[=file]
Enable the query log. See Chapter 5 for more details.

--log-isam[=file]
Enable the ISAM/MyISAM log (only used when debugging ISAM/MyISAM).

--log-slow-queries[=file]
Enable the slow query log. See Chapter 5 for more details.

--log-update[=file]
Enable the update log. See Chapter 5 for more details.

--log-long-format
Log some extra information to update log. If you are using --log-slow-queries then queries that are not using indexes are logged to the slow query log.

--low-priority-updates
All table-modifying operations (INSERT/DELETE/UPDATE) will have lower priority than selects. It can also be done via {INSERT | REPLACE | UPDATE | DELETE} LOW_PRIORITY ... to lower the priority of only one query, or by SET OPTION SQL_LOW_PRIORITY_UPDATES=1 to change the priority in one thread.

--memlock
Lock the mysqld process in memory. This works only if your system supports the mlockall() system call (like Solaris). This may help if you have a problem where the operating system is causing the mysqld processs to swap.

Copyright © 2001 O'Reilly & Associates, Inc.

--myisam-recover [=option[,option...]]]

Set the MyISAM recovery options. option is any combination of DEFAULT, BACKUP, FORCE or QUICK. You can also set option explicitly to "" if you want to disable this option. If this option is used, mysqld examine each MyISAM file on open. If the table is marked as crashed or if the table wasn't closed properly, mysqld will run check on the table. If the table was corrupted, mysqld will attempt to repair it. The following options affects how the repair works.

- DEFAULT The same as not giving any option to --myisam-recover.
- BACKUP If the data table was changed during recover, save a backup of the `table_name.MYD' data file as `table_name-datetime.BAK'.
- FORCE Run recover even if more than one row will be lost from the .MYD file.
- QUICK Don't check the rows in the table if there aren't any delete blocks.

Before a table is automatically repaired, MySQL will add a note about this in the error log. If you want to be able to recover from most things without user intervention, you should use the options BACKUP,FORCE. This will force a repair of a table even if some rows would be deleted, but it will keep the old data file as a backup so that you can later examine what happened.

--pid-file=path

Path to pid file used by safe_mysqld.

-P, --port=...

Port number to listen for TCP/IP connections.

-o, --old-protocol

Use the version 3.20 protocol for compatibility with some very old clients.

--one-thread

Only use one thread (for debugging under Linux).

-O, --set-variable var=option

Give a variable a value. --help lists all variables. See Chapter 18 for more information about variables.

--safe-mode

Skip some optimize stages. Implies --skip-delay-key-write.

--safe-show-database

Don't show databases for which the user doesn't have any privileges.

--safe-user-create

If this is enabled, a user can't create new users with the GRANT command, if the user doesn't have INSERT privilege to the mysql.user table or any column in this table.

--skip-concurrent-insert

Turn off the ability to select and insert at the same time on MyISAM tables. This should only to be used if you think you have found a bug in this feature.

Copyright © 2001 O'Reilly & Associates, Inc.

--skip-delay-key-write
Ignore the delay_key_write option for all tables.

--skip-grant-tables
This option causes the server not to use the privilege system at all. This gives everyone full access to all databases! You can tell a running server to start using the grant tables again by executing mysqladmin flush-privileges or mysqladmin reload.

--skip-host-cache
Don't' use host name cache for faster IP address resolution. This causes mysqld to query DNS server on every connect.

--skip-locking
Disable system locking. When this is disabled, you must shut down the server to use isamchk or myisamchk

--skip-name-resolve
Hostnames are not resolved. All Host column values in the grant tables must be IP numbers or localhost.

--skip-networking
Don't listen for TCP/IP connections at all. All interaction with mysqld must be made via Unix sockets. This option is highly recommended for systems where only local requests are allowed.

--skip-new
Don't use new, possibly wrong ISAM routines. Implies --skip-delay-key-write. This will also set default table type to ISAM.

--skip-symlink
Don't delete or rename any files that a symbolically linked file in the data directory points to.

--skip-safemalloc
If MySQL is configured with --with-debug=full, all programs will check the memory for overruns for every memory allocation and memory freeing. As this checking is very slow, you can avoid this, when you don't need memory checking, by using this option.

--skip-show-database
Don't allow 'SHOW DATABASE' commands, unless the user has process privilege.

--skip-stack-trace
Don't write stack traces. This option is useful when you are running mysqld under a debugger.

--skip-thread-priority
Disable using thread priorities for faster response time.

--socket=path
Socket file to use for local connections instead of default /tmp/mysql.sock.

--sql-mode=option[,option[,option...]]
Option can be any combination of: REAL_AS_FLOAT, PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, SERIALIZE, ONLY_FULL_GROUP_BY. It can also be empty ("") if you want to reset this. By specifying all of the above options is same as using --ansi.

Copyright © 2001 O'Reilly & Associates, Inc.

transaction-isolation=level
: Sets the default transaction isolation level. Possible level values are READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, or SERIALIZABLE.

-t, --tmpdir=path
: Path for temporary files. Use this if your default /tmp directory resides on a partition too small to hold temporary tables.

-u, --user=user_name
: Run mysqld daemon as user user_name. This option is mandatory when starting mysqld as root.

-V, --version
: Output version information and exit.

-W, --warnings
: Print out warnings like Aborted connection... to the .err file.

---

safe_mysqld

safe_mysqld [`options`]

safe_mysqld is a wrapper script for mysqld and is the recommended way to start a MySQL server. safe_mysqld is will attempt to determine location of the installation, so it is able to start a server that was installed from a source or binary distribution. safe_mysqld will restart the server when an error occurs and it redirects run-time output to a log file.

When starting up, safe_mysqld first looks relative to the current working directory for the ‘bin’ and ‘data’ directories (from a binary distribution), or for ’libexec’ and ‘var’ directories (from a source distribution). So if you start safe_mysqld from the your installation directory, it ought to be able to find all the relevant files.

If the server binaries and databases cannot be found relative to the current working directory, safe_mysqld looks in the standard locations. These locations are depend on the type of distribution you have installed. If you installed your distribution in a standard location, it should be able to find everything.

If the options --mysqld and --mysqld-version are not specified safe_mysqld will start the mysqld-max if it can be found. If mysqld-max is not found, mysqld will be started.

safe_mysqld supports a few options over and above those supported by mysqld. But all options supplied on the command line to safe_mysqld are passed directly to mysqld. So, if you want to specify any safe_mysqld options, they must be supplied in an option file. safe_mysqld will read all options from the [mysqld], [server] and [safe_mysqld] sections of option files. See Chapter 5 for more information on setting up option files for your server.

The options supported by safe_mysqld are:

Copyright © 2001 O’Reilly & Associates, Inc.

--basedir=path
: Same as for mysqld.

--core-file-size=#
: Limit the size of the core file mysqld can create. Passed to ulimit -c.

--datadir=path
: Same as for mysqld.

--err-log=path
: Location of the error log.

--ledir=path
: Path to mysqld

--log=path
: Location of the query log.

--mysqld=mysqld-version
: The full name of the mysqld binary version in the ledir directory to start.

--mysqld-version=version
: Similar to --mysqld= but here you only give the suffix for mysqld. For example if you use --mysqld-version=max, safe_mysqld will start the ledir/mysqld-max version. If the argument to --mysqld-version is empty, ledir/mysqld will be used.

--open-files-limit=#
: Number of files mysqld should be able to open. Passed to ulimit -n. Note that you need to start safe_mysqld as root for this to work properly!

--pid-file=path
: Path to the pid file used by safe_mysqld.

--port=#
: Same as for mysqld.

--socket=path
: Same as for mysqld.

--timezone=#
: Set the timezone (the TZ) variable to the value of this parameter.

--user=#
: Same as for mysqld.

---

mysql.server

mysql.server start

mysql.server stop

mysql.server is a script that can, on a System-V like UNIX sytem, be used to automatically start and stop mysqld at system boot and shutdown. See chapter 5 for more information on automatically starting and stopping your server.

Copyright © 2001 O'Reilly & Associates, Inc.

Other programs and utilities

myisamchk/isamchk

myisamchk [*options*] *table_file* [*table_file...*]

isamchk [*options*] *table_file* [*table_file...*]

myisamchk and isamchk are identical except that they operate on different file type. myisamchk is meant to work with MyISAM files -- they have an extension of .MYI. isamchk is meant to work with ISAM files -- those with an extension of .ISM. For the remainder of this dicussion, we will refer only to myisamchk, but all of the concepts also apply to isamchk.

This utility is used to check and repair the files, as well as report information about them. You must provide the correct path to the ISAM file you wish to examine. For example,

```
% myisamchk /usr/local/data/foobar/*.MYI
```

will execute against all MyISAM files in the database 'foobar'.

myisamchk/isamchk should only be used when the MySQL is not running. When the server is running, you can use the mysqlcheck command (see below).

-# or --debug=debug_options
: Output debug log. The debug_options string often is 'd:t:o,filename'.

-? or --help
: Display a help message and exit.

-O var=option, --set-variable var=option
: Set the value of a variable. myisamchk --help will report all variables and values. Two important variables are:

key_buffer_size
: key_buffer_size is used when you are checking the table with --extended-check or when the keys are repaired by inserting key row by row in to the table (like when doing normal inserts). Repairing through the key buffer is used in the following cases:

- If you use --safe-recover.
- If you are using a FULLTEXT index.
- If the temporary files needed to sort the keys would be more than twice as big as when creating the key file directly. This is often the case when you have big CHAR, VARCHAR or TEXT keys as the sort needs to store the whole keys during sorting. If you have lots of temporary space and you can force myisamchk to repair by sorting you can use the --sort-recover option.

Copyright © 2001 O'Reilly & Associates, Inc.

Reparing through the key buffer takes much less disk space than using sorting, but is also much slower.

sort_buffer_size

sort_buffer_size is used when the keys are repaired by sorting keys, which is the normal case when you use --recover.

-s or --silent

Silent mode. Write output only when errors occur. You can use -s twice (-ss) to make myisamchk very silent.

-v or --verbose

Verbose mode. Print more information. This can be used with -d and -e. Use -v multiple times (-vv, -vvv) for more verbosity!

-V or --version

Print the myisamchk version and exit.

-w or, --wait

Instead of giving an error if the table is locked, wait until the table is unlocked before continuing. Note that if you are running mysqld on the table with --skip-locking, the table can only be locked by another myisamchk command.

Check Options

-c or --check

Check table for errors. This is the default operation if you are not giving myisamchk any options that override this.

-e or --extend-check

Check the table VERY thoroughly (which is quite slow if you have many indexes). This option should only be used in extreme cases. myisamchk or myisamchk --medium-check should, in most cases, be able to discover any errors in the table. If you are using --extended-check and have much memory, you should increase the value of the key_buffer_size variable considerably.

-F or --fast

Check only tables that haven't been closed properly.

-C or --check-only-changed

Check only tables that have changed since the last check.

-f or --force

If myisamchk finds any errors in the table, restart myisamchk with -r (repair) on the table,.

-i or --information

Print informational statistics about the table that is checked.

-m or --medium-check

Faster than extended-check, but only finds 99.99% of all errors. Should, however, be good enough for most cases.

-U or --update-state

Store in the MyISAM file when the table was checked and if the table crashed. This should be used to get full benefit of the --check-only-changed option. Don’t use this

Copyright © 2001 O’Reilly & Associates, Inc.

option if the mysqld server is using the table and you are running mysqld with --skip-locking.

-T or --read-only
: Don't mark table as checked. This is useful if you use myisamchk to check a table that is in use by some other application that doesn't use locking (like mysqld --skip-locking).

Repair Options

The following options are used if you start myisamchk with -r or -o:

-D # or --data-file-length=#
: Max length of data file (when re-creating data file when it's 'full').

-e or --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 except as a last resort.

-f or --force
: Overwrite old temporary files (table_name.TMD) instead of aborting.

-k # or keys-used=#
: If you are using ISAM, tells the ISAM table handler to update only the first # indexes. If you are using MyISAM, tells which keys to use, where each binary bit stands for one key (first key is bit 0). This can be used to get faster inserts.

-l or --no-symlinks
: Do not follow symbolic links. Normally myisamchk repairs the table a symlink points at.

-r or --recover

Can fix almost anything except unique keys that aren't unique (which is an extremely unlikely error with ISAM/MyISAM tables). If you want to recover a table, this is the option to try first. Only if myisamchk reports that the table can't be recovered by -r, you should then try -o. If you have lots of memory, you can increase the sort_buffer_size variable to make this run faster.

-o or --safe-recover
: Uses an old recovery method (reads through all rows in order and updates all index trees based on the found rows); this is a magnitude slower than -r, but can handle a couple of very unlikely cases that -r cannot handle. This recovery method also uses much less disk space than -r. Normally one should always first repair with -r, and only if this fails use -o. If you have lots of memory, you can increase the size of the key_buffer_size variable to make this run faster.

-n or --sort-recover
: Force myisamchk to use sorting to resolve the keys even if the temporary files should be very big. This will not have any effect if you have fulltext keys in the table.

--character-sets-dir=...
: Directory where character sets are stored.

--set-character-set=name
: Change the character set used by the index

 Copyright © 2001 O'Reilly & Associates, Inc.

.t or --tmpdir=path
Path for storing temporary files. If this is not set, myisamchk will use the environment variable TMPDIR for this.

-q or --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 or --unpack
Unpack file packed with myisampack.

Other Options

-a or --analyze
Analyze the distribution of keys. This improves join performance by enabling the join optimizer to better choose in which order it should join the tables and which keys it should use.

-d or --description
Prints some information about table.

-A or --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 or --sort-index
Sort the index tree blocks in high-low order. This will optimize seeks and will make table scanning by key faster.

-R or --sort-records=#
Sorts records according to an index. This makes your data much more localized and may speed up ranged SELECT and ORDER BY operations on this index. To find out a table's index numbers, use SHOW INDEX, which shows a table's indexes in the same order that myisamchk sees them. Indexes are numbered beginning with 1.

---

myisampack/pack_isam

myisampack [*options*] table_name

isam_pack [*options*] table_name

These utilities generate compresses, read-only MyISAM and ISAM files. myisampack is used to compress MyISAM tables, and pack_isam is used to compress ISAM tables. Table compression reduces datafile size from 40 to 70% while maintaining speedy access. myisampack works with all column types. pack_isam will not work with tables that have BLOB or TEXT columns.

myisampack and pack_isam only modify the specific datafiles. To update the indexes, run myisamchk -rq/isamchk -rq after running myisampack/pack_isam.

Options

-b, --backup
Make a backup of the table as tbl_name.OLD.

Copyright © 2001 O'Reilly & Associates, Inc.

-#, --debug=debug_options
Output debug log. The debug_options string often is 'd:t:o,filename'.

-f, --force
Force packing of the table even if it becomes bigger or if the temporary file exists. myisampack creates a temporary file named `tbl_name.TMD' while it compresses the table. If you kill myisampack, the `.TMD' file may not be deleted. Normally, myisampack exits with an error if it finds that `tbl_name.TMD' exists. With --force, myisampack packs the table anyway.

-?, --help
Display a help message and exit.

-j big_tbl_name, --join=big_tbl_name
Join all tables named on the command line into a single table big_tbl_name. All tables that are to be combined MUST be identical (same column names and types, same indexes, etc.).

-p #, --packlength=#
Specify the record length storage size, in bytes. The value should be 1, 2, or 3. myisampack stores all rows with length pointers of 1, 2, or 3 bytes. In most cases, myisampack can determine the right pack length value before it begins packing the file. Sometimes it may notice during the packing process that it could have used a shorter length. In this case, myisampack will print a note that the next time you pack the same file, you could use a shorter record length.

-s, --silent
Silent mode. Write output only when errors occur.

-t, --test
Don't actually pack table, just test packing it.

-T dir_name, --tmp_dir=dir_name
Use the named directory as the location in which to write the temporary table.

-v, --verbose
Verbose mode. Write information about progress and packing result.

-V, --version
Display version information and exit.

-w, --wait
Wait and retry if table is in use. If the mysqld server was invoked with the --skip-locking option and the table has a chance of being updated during while pack is being performed, it is not a good idea to invoke myisampack.

---

mysqlaccess

mysqlaccess [*host* [`user` [*database*]]] options

This script is used to test the result of adding privileges to a database. It functions by creating temporary copies of the `user`, `db`, and `host` tables from the `mysql` database. IF the privileges work out, you can commit them back to the `mysql` database. This is very useful for testing a set of changes before applying them to the system.

 Copyright © 2001 O’Reilly & Associates, Inc.

Options

-b, --brief
: Display results in an abbreviated, single-line format.

--commit
: Copies the temporary grant tables into the `mysql` database. As always, run `mysqladmin flush-privileges` after perfoming the commit so the server reloads the privileges.

--copy
: Loads the grant tables into the temporary tables.

-d database, --db=database
: Specifies the database name.

--debug=n
: Sets the debug level. n can be an integer from 0 to 3.

--howto
: Displays some examples of how to use mysqlaccess.

--old_server
: Specifies that the server is older than MySQL 3.21. mysqlaccess needs to use different queries in this case.

--plan
: Displays a list of enhancements planned for future releases of mysqlaccess.

--preview
: Display the privilege differences between the actual and temporary grant tables.

-H hostname, --rhost=hostname
: The remote server to connect to.

--rollback
: Undoes the changes made to the temporary tables.

-P password, -spassword=password
: The password of the MySQL superuser, or any user with sufficient privileges to modify the grant tables.

-U username, --superuser=username
: The user name of the MySQL superuser.

-t, --table
: Display results in tabular format.

---

mysqlaccess

mysqladmin [*options*] command [*command-options*] …

Commands

create databasename
: Create a new database.

drop databasename
: Delete a database and all its tables.

extended-status
  Gives an extended status message from the server.

flush-hosts
  Flush all cached hosts.

flush-logs
  Flush all logs.

flush-tables
  Flush all tables.

flush-privileges
  Reload grant tables (same as reload).

kill id,id,...
  Kill mysql threads. Use mysqladmin processlist or the SQL command SHOW PROCESSLIST to see the active server threads.

password
  Set a new password. Change old password to new-password.

ping
  Check if mysqld is alive.

processlist
  Show list of active threads in server.

reload
  Reload grant tables (same as flush-privileges).

refresh
  Flush all tables and close and open logfiles.

shutdown
  Shut the server down.

slave-start
  Start slave replication thread.

slave-stop
  Stop slave replication thread.

status
  Gives a short status message from the server.

variables
  Prints variables available. See chapter 18 for more information on server variables.

version
  Get version info from server.

Options

-?, --help
  Display the help for mysql and exit.

-C, --compress
  Use compression in server/client protocol.

Copyright © 2001 O'Reilly & Associates, Inc.

-#, --debug[=...]
Enable the debug log. Default is 'd:t:o,/tmp/mysql.trace'.

-f, --force
If multiple commands are specified on the command line, mysqlaccess will normally halt when an error occurs. When --force is enabled, mysqlaccess will continue to the next command on error.

Also, --force will force a "drop database" operation without confirmation.

-h, --host=...
Connect to the MySQL server on the specified host.

-p[password], --password[=...]
Password to use when connecting to server. If a password is not given on the command line, you will be prompted for it. Note that if you use the short form -p you can't have a space between the option and the password.

-P --port=...
TCP/IP port number of the server you wish to connect to.

--relative
When used with the extended-status command with the --sleep option, this displays the difference between the current and previous values.

-s, --silent
Be more silent.

-i n, --sleep=n
Executes the command(s) on the command line every n seconds.

-S, --socket=...
Socket file of the server you wish to connect to.

-t n, --timeout=n
Timeout after n seconds when attempting to connect to the server.

-u, --user=#
User for login if not current user.

-V, --version
Output version information and exit.

- w [n], --wait[=n]
Retry n times when a attempts to connect to the server fail. By default, n is 1.

-?, --help
Display usage information.

-b, --brief
Display results as a brief single line table.

--commit
Move changes from temporary table to the actual grant tables. You must run mysqladmin reload before the changes will take effect.

--copy
Renew the temporary table from the actual grant tables.

Copyright © 2001 O'Reilly & Associates, Inc.

-d database, --db=database
    The database to which to connect.

--debug=debuglevel
    Set the debugging level (0 through 3).

-h host, --host=host
    The host whose access rights are examined.

--howto
    Usage examples for the program.

-H host, --rhost=host
    Connect to a database server on a remote host.

--old-server
    Connect to a pre-3.21 MySQL server.

-p password, --password=password
    Check the password of the user being examined.

--plan
    Display suggestions for future releases.

--preview
    Show difference between temporary table and actual grant tables.

-P password, --spassword=password
    Administrative password used to access the grant tables.

--relnotes
    Display the release notes for the program.

--rollback
    Undo the changes made to the temporary table.

-t, --table
    Display results in full table format.

-u username, --user=username
    User to be examined.

-U username. –superuser=username
    Administrative username used to access the grant tables.

-v, --version
    Display version information.

mysqlbug

mysqlbug

Report a bug in a MySQL program or utility. This program collects information about your MySQL installation and sends a detailed problem report to the MySQL team.

mysqlcheck

mysqlcheck [*options*] *database* [*tables*]

Copyright © 2001 O'Reilly & Associates, Inc.

mysqlcheck [*options*] --databases *dbname* [*dbname* …]

mysqlcheck [*options*] --all-databases

mysqlcheck provides a convenient way to execute the MySQL CHECK, REPAIR, ANALYZE and OPTIMIZE commands from the command line. This utility can be used while the server is running in contrast with the myisamcheck/isamcheck which should only be used when the server is down.

The mysqlcheck binary can be renamed to change it's default behavior. The possible names for the binary are

mysqlrepair
: The default option is --repair.

mysqlanalyze
: The default option is --analyze.

mysqloptimize
: The default option is --optimize.

mysqlcheck
: The default option is --check.

Options

-A, --all-databases
: Check all the databases. This is same as --databases with all databases listed.

-1, --all-in-1
: Instead of making one query for each table, execute one query separately for each database. Table names will be in a comma separated list.

-a, --analyze
: Enable analyze mode. This is the default when the binary is named mysqlanalyze.

-#, --debug=...
: Output debug log. Often this is 'd:t:o,filename'

--character-sets-dir=...
: Directory where character sets are

-c, --check
: Enable check mode. This is the default when the binary is named mysqlcheck.

--compress
: Use compression in server/client protocol.

-?, --help
: Display a help message and exit.

-B, --databases
: To operate against several databases. All arguments are regarded as database names.

--default-character-set=...
: Set the default character set

-f, --force
: Continue even if an sql error is encountered.

Copyright © 2001 O'Reilly & Associates, Inc.

-e, --extended
: If you are using this option with "check" mode, it will ensure that the table is 100 percent consistent, but will take a long time. If you are using this option with "repair" mode, it will run an extended repair on the table, which may not only take a long time to execute, but may produce a lot of garbage rows also!

-h, --host=...
: Connect to host.

-o, --optimize
: Enable optimize mode. This is the default behavior with the binary is named mysqloptimize.

-p, --password[=...]
: Password to use when connecting to server.

-P, --port=...
: Port number to use for connection.

-q, --quick
: If you are using this option with "check" mode, it prevents the check from scanning the rows to check for wrong links. This is the fastest check. If you are using this option with "repair" mode, it will try to repair only the index tree. This is the fastest repair method for a table.

-r, --repair
: Enable repair mode. Can fix almost anything except unique keys that aren't unique.

-s, --silent
: Print only error messages.

-S, --socket=...
: Socket file to use for connection.

--tables
: Specify a the set of tables to operate on. Overrides option --databases (-B).

-u, --user=#
: User to connect as.

-v, --verbose
: Print info about the various stages.

-V, --version
: Output version information and exit.

Check Options

--auto-repair
: If a checked table is corrupted, automatically fix it. Repairing will be done after all tables have been checked, if corrupted ones were found.

-C, --check-only-changed
: Check only tables that have changed since last check or haven't been closed properly.

-F, --fast
: Check only tables that have not been closed properly

Copyright © 2001 O'Reilly & Associates, Inc.

-m, --medium-check
: Faster than extended-check, but only finds 99.99 percent of all errors. Should be good enough for most cases.

---

mysqldump

mysqldump [*options*] *database* [*tables*]

mysqldump [*options*] --databases *dbname* [*dbname* …]

mysqldump [*options*] --all-databases

Outputs the contents of the given database (or table within a database) as a series of ANSI SQL commands. This can be used to back up a database, or copy the contents of a database to another SQL database, MySQL or otherwise. The dump files are ASCII files so they are portable across different machine architectures. This command is also handy for breaking up a database; use the -1 and -opt options.

Options

--add-locks
: Add lock table and unlock table commands for each table. This will enable you to get faster inserts into MySQL when the dump is read back in.

--add-drop-table
: Add a drop table before each create statement.

-A, --all-databases
: Dump all the databases. This will be same as --databases with all databases listed.

-a, --all
: Include all MySQL-specific create options.

--allow-keywords
: Allow creation of column names that are keywords. This is done by prefixing each column name with the table name.

-c, --complete-insert
: Use complete insert statements.

-C, --compress
: Compress all client/server communication.

-B, --databases
: To dump several databases. All name arguments are regarded as database names. use db_name commands will be included in the output for each database.

--delayed
: Insert rows with the insert delayed command.

-e, --extended-insert
: Use the new multiline insert syntax.

-#, --debug[=option_string]
Trace usage of the program (for debugging).

--help
Display a help message and exit.

--fields-terminated-by=...
--fields-enclosed-by=...
--fields-optionally-enclosed-by=...
--fields-escaped-by=...
--lines-terminated-by=...
These options are used with the -T option and have the same meaning as the corresponding clauses for the LOAD DATA INFILE SQL command.

-F, --flush-logs
Flush log file in the MySQL server before starting the dump.

-f, --force,
Continue even if we get a SQL error during a table dump.

-h, --host=..
Dump data from the MySQL server on the named host.

-l, --lock-tables.
Lock all tables before starting the dump. The tables are locked with READ LOCAL to allow concurrent inserts in the case of MyISAM tables.

-n, --no-create-db
create database statements will not be put in the output file. If this is not specified, create database statements will be added if --databases or --all-databases option are given.

-t, --no-create-info
Don't write create table statements.

-d, --no-data
Don't write any row information for the table. This is very useful if you just want to get a dump of the structure for a database or a table.

--opt
Same as --quick --add-drop-table --add-locks --extended-insert --lock-tables. Should give you the fastest possible dump for reading into a MySQL server.

-p*password*, --password[=*password*]
The password to use when connecting to the server.

-P port_num, --port=port_num
The TCP/IP port number to use for connecting to a host.

-q, --quick
Dump queries directly to stdout without buffering.

-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' (new line + carriage return).

 Copyright © 2001 O'Reilly & Associates, Inc.

-S --socket=...
    Socket file of the server you wish to connect to.

--tables
    Specifies tables to copy. Overrides option --databases (-B).

-T, --tab=path-to-some-directory
    Creates a table_name.sql file, that contains the SQL commands, and a table_name.txt file, that contains the data, for each give table. This only works if mysqldump is run on the same machine as the mysqld daemon. The format of the .txt file is made according to the --fields-xxx and --lines--xxx options.

-u, --user=#
    User for login if not current user.

-O var=option, --set-variable var=option
    Set the value of a variable. The variables are listed below.

    net_buffer_length=#
        When creating multi-row-insert statements (as with option --extended-insert or --opt), mysqldump will create rows up to net_buffer_length length. If you increase this variable, you should also ensure that the max_allowed_packet variable in the MySQL server is bigger than the net_buffer_length.   net_buffer_length must be less than 16M.

-v, --verbose
    Verbose mode. Print out more information on what the program does.

-V, --version
    Print version information and exit.

-w, --where='where-condition'
    Dump only selected records. The quotes are mandatory:

mysqlhotcopy

mysqlhotcopy *database* [*backup_dir*]

mysqlhotcopy *database* [*database*] *backup_dir*

mysqlhotcopy is a perl script that uses LOCK TABLES, FLUSH TABLES and cp or scp to quickly make a binary backup of a database. It's the fastest way to make a backup of the database. Backups make with mysqlhotcopy are not machine architecture independent.

Options

-?, --help
    Display a help message and exit

-u, --user=#
    User for database login

-p, --password=#
    Password to use when connecting to server

Copyright © 2001 O'Reilly & Associates, Inc.

-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 to make the backup smaller and faster The indexes can later be reconstructed with myisamchk -rq..

--method=#
: Method for copy: cp or scp.

-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.

--tmpdir=#
: Directory to use for temporary files.

---

mysqlimport

mysqlimport [*options*] *database* [*file*]

Reads a file of data in a variety of common formats (such as comma delimited or fixed width) and inserts the data into a database. A table with the same name as the file must exist in the database with enough columns of the appropriate type to store the data. mysqlimport is essentially a command line version of the LOAD DATA INFILE SQL statement.

Options

-c, --columns=...
: A comma-separated list of field names. The field list is used to create a proper LOAD DATA INFILE command, which is then passed to MySQL.

 Copyright © 2001 O'Reilly & Associates, Inc.

-C, --compress
 Compress all client/server communication.

-#, --debug[=option_string]
 Trace usage of the program (for debugging).

-d, --delete
 Empty the table before importing the text file.

--fields-terminated-by=...

--fields-enclosed-by=...

--fields-optionally-enclosed-by=...

--fields-escaped-by=...

--lines-terminated-by=...

These options have the same meaning as the corresponding clauses for LOAD DATA INFILE.

-f, --force
 Ignore errors. If a table for a text file doesn't exist, continue processing any remaining files. Without --force, mysqlimport exits if a table doesn't exist.

--help
 Display a help message and exit.

-h host_name, --host=host_name
 Import data to the MySQL server on the named host.

-i, --ignore
 The --replace and --ignore options control handling of input records that duplicate existing records on unique key values. If you specify --replace, new rows replace existing rows that have the same unique key value. If you specify --ignore, input rows that duplicate an existing row on a unique key value are skipped. If you don't specify either option, an error occurs when a duplicate key value is found, and the rest of the text file is ignored.

-l, --lock-tables
 Lock ALL tables for writing before processing any text files. This ensures that all tables are synchronized on the server.

-L, --local
 Read input files from the client. By default, text files are assumed to be on the server if you connect to localhost

-pyour_pass, --password[=your_pass]
 The password to use when connecting to the server.

-P port_num, --port=port_num
 The TCP/IP port number to use for connecting to a host.

-r, --replace
 The --replace and --ignore options control handling of input records that duplicate existing records on unique key values. If you specify --replace, new rows replace existing rows that have the same unique key value. If you specify --ignore, input rows that duplicate an existing row on a unique key value are skipped. If you don't specify either option, an error occurs when a duplicate key value is found, and the rest of the text file is ignored.

Copyright © 2001 O'Reilly & Associates, Inc.

-s, --silent
: Silent mode. Write output only when errors occur.

-S /path/to/socket, --socket=/path/to/socket
: The socket file to use when connecting to localhost.

-u user_name, --user=user_name
: The MySQL user name to use when connecting to the server. The default value is your Unix login name.

-v, --verbose
: Verbose mode. Print out more information what the program does.

-V, --version
: Print version information and exit.

---

mysqlshow

mysqlshow [*options*] [*database*] [*table*] [*field*]

Displays the layout of the requested database, table or field. If no argument is given, a list of all of the databases is given. With one argument the layout of the given database is show. With two arguments, a table within the database is displayed. If all three arguments are present, the information about a specific field within a table is presented.

Options

-?, --help
: Display usage information.

-# debuglevel, --debug=debuglevel
: Set the debugging level.

-h hostname, --host=hostname
: Connect to a remote database server.

-k, --keys
: Display the keys of a table.

-p [password], --password]=password]
: Password used to connect to the database server. If no argument is given, the password is asked from the command line.

-P port, --port=port
: Port used to connect to a remote database server.

-S file, --socket=file
: The Unix socket used to connect to the local database server.

-u username, --user=username
: Username used to connect to the database server.

-V, --version
: Display version information.

 Copyright © 2001 O’Reilly & Associates, Inc.

23

PHP Reference

PHP provides a wide range of functions that are useful when creating database-driven applications. So many, in fact, that it would be unwieldy to list all of them in a book about MySQL. Therefore this reference chapter concentrates on the functions that PHP provides to interface directly with MySQL. This includes the new PHP database abstraction layer which promises to someday unify the various PHP database APIs into a single set of functions.

---

mysql_affected_rows

- Returns the number of rows affected by the last non-SELECT statement

$num_rows = mysql_affected_rows([$mysql])

mysql_affected_rows returns the number of rows altered in any way by the last statement. Since this only reports on rows that have been changed in some way, it has no meaning when used after a SELECT statement. Also there are a couple of cases where MySQL performs optimizations that affect the result of this function:

UPDATE - It is import to note, as mentioned above, this function returns the number of rows that are changed in some way by the query. This means that an UPDATE query that matches a row, but does not change its value, is not counted. For example, the query UPDATE mytable SET column = ‘fnord’ will return 0 if every row in the table already has ‘fnord’ for the value of ‘column’.

DELETE - MySQL performs an optimization with deleting the entire contents of a table that makes it impossible to tell the number of rows that were in that table. Therefore, if you delete all of a table using ‘DELETE from tablename’ with no WHERE clause, this function will return 0.

A specific connection can be specified by passing the connection identifier variable as a parameter to this function. Otherwise, the most recently opened connection is used.

Copyright © 2001 O’Reilly & Associates, Inc.

This function should be called immediately after the query you are interested in. This holds true even when using tables that use transactions; this function should be called after the query, not the commit.

Example

```
<? mysql_query("DELETE from people where firstname like 'P%'"); ?>
You have deleted <?= mysql_affected_rows() ?> people.

<? // $mysql is a seperate server connection that was established earlier.
    mysql_query("UPDATE people SET lastname = 'Smith' where
UPPER(lastname)='SMITH'",
         $mysql);  ?>
You have corrected the case in <?= mysql_affected_rows($mysql) ?> instances of
'Smith'.
<? // Note that this will accurately report the number of 'Smith's that were
changed. Any records
     // that were already 'Smith' were not counted because they were not changed,
even though
     // they matched the WHERE clause. ?>
```

mysql_change_user

- Changes the currently authenticated user for a MySQL session

$success = mysql_change_user($username, $password [, $database [, $mysql]])

mysql_change_user re-authenticates a MySQL server connection with the given username and password. If a third argument is give, it is used as the default database if the re-authentication is successful. By default, this function uses the most recently opened MySQL connection. A specific connection can be specified as the forth argument.

This function returns a true value on success and a false value if the re-authentication fails. In the case of failure the authentication information in effect before the function was called stays active (and the default database does not change).

Example

```
<? // Switch users to 'newuser', 'newpass'
     mysql_change_user( 'newuser', 'newpass' );

     // Switch users to 'newuser', 'newpass' and change the default database to
'newdb'
     // If the change is unsuccessful, print a warning.
    if (! mysql_change_user('newuser', 'newpass', 'newdb') { ?>
Warning! Database re-authentication failed!.
    <? }

    // Switch users to 'newuser', 'newpass', using the existing MySQL connection
$mysql.
    mysql_change_user('newuser', 'newpass', '', $mysql); ?>
```

mysql_close

- Closes MySQL connection

$success = mysql_close([$mysql])

mysql_close closes the most recently opened MySQL server connection. A specific connection can be specified as the first parameter. The function returns true if the

 Copyright © 2001 O'Reilly & Associates, Inc.

connection was successfully closed and false if there was an error.

It is generally not necessary to use this function as non-persistent connections are automatically closed at the end of the script where they are used. This function only has effect on non-persistent connections. Persistent connections, opened with *mysql_pconnect*, will be unaffected.

Example

```
<? // Close the most recent connection.
     mysql_close();

     // Attempt to close a persistent connection referenced by the variable $mysql.
     if (! mysql_close( $mysql ) ) ?>
The connection didn't close! This is probably because it was a persistent
connection.
     <? } ?>
```

mysql_connect

- Open a connection to a MySQL Server

$mysql = mysql_connect([$host[, $user [, $password]]])

mysql_connect attempts to open a connection with a MySQL server. If successful this function returns a MySQL connection variable that can be used with most of the MySQL functions to specify this connection. In the case of failure, a false value is returned.

If no arguments are given, PHP attempts to connect to the MySQL server on the local host at port 3306 using the username of the user that owns the PHP process and a blank password. The hostname can be specified as the first parameter. If a TCP connection is desired on any port other than 3306, it is specified as part of the hostname, using a ':' separator. If a local Unix socket connection is needed, the pathname of the socket should be specified in the same manner. The second argument specifies the username, and the third specifies the password.

Note: If PHP is running in "safe mode," only the default hostname, port, username and password are allowed.

Note: Care should be taken when specifying password information within a PHP script. Other users of the same machine will likely be able to view the script and see the password. Visitors over the web will not be able to view the script, however, so this is only an issue if you share the server machine with other people.

If more than one call is made to *mysql_connect* using identical arguments, all of the calls after the first will return the connection variable created with the first call (if that connection is still open).

Example

```
<? // Connect to the mysql server on the local host at port 3306 using the username
of the
     // owner of the PHP process and a blank password:
    $mysql = mysql_connect();
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    // Connect to the mysql server on the localhost using the Unix socket
/tmp/mysql.sock and
    // the default username and password
    $mysql = mysql_connect( 'localhost:/tmp/mysql.sock' );

    // Connect to the mysql server at my.server.com, port 3333 using the username
'me' and a
    // blank password
    $mysql = mysql_connect('my.server.com:3333', 'me');

   // Connect to the mysql server on the localhost, port 3306, with username 'me'
and the
   // password 'mypass'
   $mysql = mysql_connect('', 'me', 'mypass');

   // Use the same connection parameters as above:
   $mysql2 = mysql_connect('', 'me', 'mypass');
   // $mysql2 is not a new connection, but rather another reference to the same
connection
   // as $mysql
?>
```

mysql_create_db

- creates a new database

$success = mysql_create_db ($database [, $mysql]

mysql_create_db attempts to create a new database using the given database name. The database is created using the most recently mysql connection. A specific connection can be specified as the second argument.

Note: *mysql_createdb* is an alias to *mysql_create_db* provided for backwards compatibility. It should not be used in new scripts.

The function returns true in the case of success and false on failure.

Example

```
<? // Create a new database called 'newdb'
     mysql_createdb("newdb");

     // Attempt to create a database called 'newdb2' using the connection given by
the
     // $mysql variable

     if (! mysql_createdb("newdb2", $mysql) ) { ?>
Attempt to create database newdb2 failed!
      <? } ?>
```

mysql_data_seek

- Move internal result pointer

$success = mysql_data_seek ($result, $row_number)

mysql_data_seek moves the internal pointer within the given result set variable to a specific row. The next attempt to read a row from the result set (such as via *mysql_fetch_row*) will return the row specified here. The first row of a result set is always 0.

 Copyright © 2001 O'Reilly & Associates, Inc.

This function returns true on success and a false value in the case of failure. This function will fail if the given row number does not exist in the result set. This commonly occurs when using this function to move back to the beginning of a result set by seeking to row 0. This will always work, unless the result set is empty (i.e. the query did not return any rows), in which case there is no row 0 and this function will fail. Therefore, it may be wise to check the number of rows in the result set, using mysql_num_rows to make sure the row you are seeking to exists.

Example

```
<? // Reset the result pointer (given by the variable $result) to the third row of
the result set
      mysql_data_seek( $result, 2 );

     // Reset the result pointer back to the beginning of the result set only if
this is possible.
     if (mysql_num_rows( $result ) > 0 ) {
            mysql_data_seek( $result, 0 );
     }
?>
```

mysql_db_name

- Read a database name from a result set

$dbname = mysql_db_name ($result, $row_num [, $unused])

mysql_db_name returns the name of a database from a result variable created from a call to *mysql_list_dbs*. The second argument to this function indicates which database name out of all of those in the result set to return. The first database in the result set is always 0. The number of database names in the result set can be obtained from *mysql_num_rows*.

This function returns a false value in the case of an error. Supplying a row number that does not exist in the result set will result in an error.

> This function is implemented as an alias to mysql_result. Because of this, it is possible to supply a third argument to this function, which represents the name of a field in the result set. However, this function is not useful in the context of mysql_db_name.
>
> mysql_dbname is available as an alias to this function for backwards compatibility but should not be used for new scripts.

Example

```
<? // $result is the result of a call to mysql_list_dbs. It contains the names of
all of the databases
    // available to the user.
    for ( $i = 0; $i < mysql_num_rows( $result ); $i++ ) {
           echo mysql_db_name( $result, $i );
    }
?>
```

mysql_db_query

- Executes a query with a specific default database

$result = mysql_db_query($database, $query [, $mysql])

mysql_db_query executes a SQL query given as the second argument, using the first argument as the default database for the query. The query is executed using the most recently opened database connection. A specific server connection can be specified using a third argument. The database specified as the first argument becomes the new default database for the connection.

If the query is a SELECT query and is successful, the function returns a result set variable that can be used with functions like *mysql_fetch_row* to retrieves the contents of the results. If the query is a non-SELECT query (such as INSERT, UPDATE or DELETE) and is successful, the function returns a true value. If the query fails, a false value is returned.

The function *mysql* is provided as an alias for backwards compatibility, but should not be used with new scripts.

Example

```
<? // Select all of the rows from the 'people' table in the 'mydb' database
     $result = mysql_db_query( "mydb", "select * from people" );
     // $result now contains a result set that can be used with myql_fetch_row() to
read the values.

     // Perform a query against the 'mydb' database using the connection specified
with the
     // $mysql connection variable and check to make sure it's a valid result set.
     if (! $result = mysql_db_query( "mydb",
                     "select firstname from people where firstname like 'P%'",
$mysql ) ) { ?>
               The query was unsuccessful!
       <? } ?>

     // Insert a new row into the table 'people' in the database 'mydb' and check
to make sure
     // the insert worked.
     if (! $result = mysql_db_query("mydb",
                     "insert into people values ('John', 'Doe')" ) { ?>
              The insert failed!
       <? } ?>
```

mysql_drop_db

- Deletes a database

$success = mysql_drop_db($database [, $mysql])

mysql_drop_db attempts to delete the given database. This is an irrevocable operation which will permanently delete all of the data within the database. This function uses the most recently opened server connection. A specific server connection can be specified with the second argument.

The function returns true if the database is successfully dropped and false in the case of an error.

 Copyright © 2001 O'Reilly & Associates, Inc.

The function mysql_dropdb is provided as an alias for backwards compatability but should not be used with new scripts.

Example

```
<? // Drop the database 'olddb'
     if (! mysql_drop_db( "olddb" ) ) { ?>
Error deleting the database!
      <? }

     // Drop the database 'otherdb' using the connection given by the $mysql
variable
    mysql_drop_db( "otherdb", $mysql );
?>
```

mysql_errno

- Return the last error code

$error_code = mysql_errno([$mysql])

mysql_errno returns the MySQL-specific error code for the last MySQL error that occurred during the current connection. Any successful MySQL-related function call resets the value of this function to 0. Because of this, you should always call this function immediately after the function you are interested in checking for errors.

Example

```
The MySQL-related function returned an error-code of <?= mysql_errno() ?>.
```

mysql_error

- Returns the text description of the last error

$error = mysql_error([$mysql])

mysql_error returns the human-readable description of the last MySQL error that occurred during the current connection. Any successful MySQL-related function call resets the value of this function to an empty string. Because of this, you should always call this function immediately after the function you are interested in checking for errors.

Example

```
<? // The variable $mysql is a MySQL connection variable
The last MySQL-related error was <?= mysql_error($mysql) ?>.
```

mysql_escape_string

- Escapes a string for use in a mysql_query.

$escaped_string = mysql_escape_string ($string)

mysql_escape_string takes a string as an argument and returns a copy of that string that has any special characters escaped so that is is safe to use in a MySQL-query. Specifically, it escapes any single quotes "'" as a double-single quote ("''").

Example

```
<? // $value contains some data, which may contain special characters.
$query = "INSERT into table values ('" + mysql_escape_string($value) + "'";
// $query now contains a SQL query that is safe to execute. ?>
```

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_fetch_array

- Retrieve a row of a result set as an array

$row_values = mysql_fetch_array ($result [, $type_of_array])

mysql_fetch_array returns an array of values from the result set pointed to by the first argument. The function returns all of the fields in the next row of data in the result set. It also advances the internal “pointer” of the result set, so that the next call to *mysql_fetch_array* (or any similar function) will return the next row in the result set.

By default, the array returned by this function is both a numerically indexed array and an associative array. The fields in the query are stored using numeric indices with 0 being the first field in the SQL statement. The fields are also stored as an associative array with the names of the fields being the keys. If you want to use just one type of array, passing a second argument to the function can set that behavior. If the argument is MYSQL_NUM a numerically indexed array is used; if the argument is MYSQL_ASSOC an associative array is used and if the argument is MYSQL_BOTH both types of arrays are used (this is the default).

When using an associative array some care should be taken to make sure the names of the fields are unique. If two or more fields in the query have the same name, only the last field is available via the associative array. The other fields must be accessed via their numeric index.

> Prior to PHP 4.05, a field that has a null value within a row would not show up within the associative array. This could create problems when checking the array for field names that should exist. As of PHP version 4.05, this problem has been fixed.

The function returns a false value if there are no more rows of data in the result set.

Example

```
<? // $result is a result set variable from the query “SELECT firstname, lastname
from People”
$firstrow = mysql_fetch_array( $result );
?> The first person in the result set is <?= $firstrow[0] ?> <?= $firstrow[1]
?>.<br>
$secondrow = mysql_fetch_array( $result  ); ?>
The second person in the result set is
    <?= $secondrow[‘firstname’] ?> <?= secondrow[‘lasrtname’‘]  ?>
// Fetch the third row as only associatve
$thirdrow = mysql_fetch_array( $result, MYSQL_ASSOC ); ?>
The third row of data has <?= $thirdrow[‘firstname’] ?> <?= $thirdrow[‘lastname’]
?>
```

mysql_fetch_assoc

- Retrieve a row of a result set as an associative array

$assoc_array = mysql_fetch_assoc ($result)

mysql_fetch_assoc returns an associative array of values from the result set pointed to by the first argument, with the names of the fields in the SQL query as the keys of the array.

 Copyright © 2001 O’Reilly & Associates, Inc.

The function returns all of the fields in the next row of data in the result set. It also advances the internal "pointer" of the result set, so that the next call to *mysql_fetch_assoc* (or any similar function) will return the next row in the result set.

Some care should be taken to make sure the names of the fields are unique. If two or more fields in the query have the same name, only the last field is available via the associative array.

The function returns a false value if there are no more rows of data in the result set.

Example

```
<? // $result is a result variable from the query "SELECT * from People"
     $firstrow = mysql_fetch_assoc( $result ) ?>
The first row of data has is <?= $firstrow['firstname'] ?> <?=
$firstrow['lastname'] ?>
```

mysql_fetch_field

- Retrieve meta-information about a field from a result set

$field_info = mysql_fetch_field($result [, $field_number])

mysql_fetch_fields returns an object containing information about a field contained in a result set. The first argument is a result set variable and the second indicates which field in the result set to retrieve information about. If no second argument is provided, the first field that has not yet been retrieved is used. Thus, successive calls to *mysql_fetch_fields* can be used to retrieve information about all of the fields in a result set.

The object returned by the function contains the following properties:

name - The name of the field as specified in the SQL query
table - The name of the table to which the field belongs, if any
max_length - The maximum length of the column.
not_null - Indicates if the field can be set to a null value (true if it cannot)
primary_key - Inticates if the field is part of a primary key (true if it is)
unique_key - Indicates if the field is part of a unique key (true if it is)
multiple_key - Indicates if the field is part of a non-unique key (true if it is)
numeric - Indicates if the field is a numeric type (true if it is)
blob - Indicates if the field is a BLOB type (true if it is)
type - The type of the field
unsigned - Indicates if the field is an unsigned numeric (true if it is)
zerofill - Indicates if the field is automatically filled with null-characters (0's)

Example

```
<? // $result is a result set created from the query "select firstname, lastname
from People"
      $fname_info = mysql_fetch_field( $result, 0 );
?>
Info about the '<?= $fname_info->table ?>.<?= $fname_info->name ?>' field:<br>
Maximum length: <?= $fname_info->max_length<br>
Type: <?= $fname_info->type ?><br>

<? $lname_info = mysql_fetch_field( $result ); // Returns the next field that
hasn't been
```

```
                                                                    // returned yet;
which is 'lastname' in this case.
       if ( $lname_info->multiple_key ) {
?>
<?= $lname_info->table ?>.<?= $lname_info->name ?> is an indexed field.
<? } ?>
```

mysql_fetch_lengths

- Retrieve the field-lengths for the last row in a result set

$lengths = mysql_fetch_lengths($result)

mysql_fetch_lengths returns an array of field lengths for the last row of data returned from a result set. The first argument to the function is a result set variable pointing to a result set that has had at least one row read from it. The values of this array correspond to the values of the fields as returned by *mysql_fetch_row* or a similar function. The values are the actual length of data that was returned.

The function returns a false value in the case of an error.

Example

```
<? // $result is a result set variable created earlier. It must be accessed at
least once using a
     // function such as mysql_fetch_row
     $lengths = mysql_fetch_lengths( $result );
?>
The first field in the most recent row fetched was <?= $lenghts[0] ?> characters
long.
```

mysql_fetch_object

- Retrieve a row of a result set as an object

$object = mysql_fetch_object($result [, $data_type])

mysql_fetch_object returns an object created from the result set pointed to by the first argument. This contains all of the fields in the next row of data in the result set. It also advances the internal "pointer" of the result set, so that the next call to *mysql_fetch_object* (or any similar function) will return the next row in the result set.

By default, the object's fields are the names of the fields in the query and also the numeric indices of the query. If you want the fields to have just the names or just the numeric indices, that behavior can be set by passing a second argument to the function. If the argument is MYSQL_NUM the numerically indexed fields are used; if the argument is MYSQL_ASSOC the field names are used and if the argument is MYSQL_BOTH both sets of fields are used (this is the default).

The function returns a false value if there are no more rows of data in the result set.

Example

```
<? // $result is a result set variable from the query "SELECT firstname, lastname
from People"
$firstrow = mysql_fetch_object( $result );
?> The first person in the result set is <?= $firstrow->0 ?> <?= $firstrow->1
?>.<br>
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
$secondrow = mysql_fetch_object( $result  ); ?>
The second person in the result set is
    <?= $secondrow->firstname ?> <?= secondrow->lasrtname  ?>
// Fetch the third row as only associatve
$thirdrow = mysql_fetch_array( $result, MYSQL_ASSOC ); ?>
The third row of data has <?= $thirdrow->firstname ?> <?= $thirdrow->lastname ?>
```

mysql_fetch_row

- Retrieve a row of a result set as an array

$array = mysql_fetch_row($result)

mysql_fetch_row returns an numerically-indexed array of values from the result set pointed to by the first argument, with the indexes in the same order as in the SQL query that created the result set. The function returns all of the fields in the next row of data in the result set. It also advances the internal "pointer" of the result set, so that the next call to *mysql_fetch_row* (or any similar function) will return the next row in the result set.

Unlike in the associative array functions, like *mysql_fetch_assoc*, the array returned by this function will contain all of the fields of the result set, even if some fields had the same name (or no name at all). This function is also slightly faster than the other fetch functions, but it is a minimal difference and the other functions should be used when appropriate.

The function returns a false value if there are no more rows of data in the result set.

Example

```
<? // $result is a result variable from the query "SELECT count(*) from People"
     $firstrow = mysql_fetch_row( $result ) ?>
There are <?= $firstrow[0] ?> people in the People table.
```

mysql_field_flags

- Retrieve the flags associated with a field in a result set

$flags = mysql_field_flags($result, $which_field)

mysql_field_flags returns a string containing any special flags associated with a field in the result set. The first argument is the result set variable and the second argument is the number of the field as given in the SQL query which created the result set. The first field is 0.

The flags are returned as a string with the flag names separated by a space. The following flags are currently possible:

not_null - The field cannot contain a NULL value
primary_key - The field is a primary key
unique_key - The field is a unieque key (there can be no duplicates)
multiple_key - The field is a multiple key (they can be duplicates)
blob - The field is a BLOB type, such as BLOB or TEXT
unsigned - The field is an unsigned numeric-type
zerofill - The field will fill any unused space with zero's up to the maximum field length
binary - The field may contain binary data and will use binary-safe comparisons

enum - The field is an enumeration, which can contain one of a number of pre-defined values
auto_increment - The field is an auto-increment field
timestamp - The field is an automatic timestamp field

> The function mysql_fieldflags is provided as an alias to this function for backwards compatability. It should not be used for new scripts.

Example

```
<? // $result is a result set variable created by the SQL query "Select firstname
from People"
     $flags = mysql_field_flags( $result, 0 );
?>
The first 'firstname' field in the People table has these flags: <?= $flags ?>
```

mysql_field_name

- Retrieves the name of a field in a result set

$field_name = mysql_field_name($result, $which_field)

mysql_field_name returns the name of a field within a result set given by the first argument. The second argument is the number of the field within the SQL query that created the result set. The first field of the query is always 0.

Example

```
<? // $result is a result set variable created by a SQL query
     $field_name = mysql_field_name( $result, 0 );
?>
The first field of the SQL query was '<?= $field_name ?>'
```

mysql_field_len

- Returns the length of a field within a result set

$length = mysql_field($result, $which_field)

mysql_field_len returns the maximum length of a field within a result set given by the first argument. The second argument is the number of the field within the SQL query that created the result set. The first field of the query is always 0.

This function does not return the length of the data within a result set, but the length of the field as defined in the database table.

> The function *mysql_fieldlen* is provided as an alias for backwards compatibility. It should not be used in new scripts.

Example

```
<? // $result is a result set variable created by a SQL query
      $len = mysql_field_name( $result, 0 );
?>
The first field of the SQL query has a length of <?= $len ?>
```

mysql_field_seek

- Set the internal field pointer in a result set to a specific field

Copyright © 2001 O'Reilly & Associates, Inc.

$success = mysql_field_seek($result, $which_field)

mysql_field_seek sets the internal field pointer within a result set given by the first argument. The second argument is the number of the field within the SQL query that created the result set. The first field of the query is always 0.

Once the field pointer has been set using this function, the next call to *mysql_fetch_field* will return this field.

The function return a true value of the field pointer is successfully set and a false value in the case of an error.

Example

```
<? // $ result is a result set variable created by a SQL query
         mysql_field_seek( $result, 0 );
         // The next call to mysql_fetch_field will return the first field in the
result set.
?>
```

mysql_field_table

- Retrieve the table name for a field within a result set

$table_name = mysql_field_table($result, $which_field)

mysql_field_table returns the name of the table that contains the given field within the result set given by the first argument. The second argument is the number of the field within the SQL query that created the result set. The first field of the query is always 0.

Example

```
<? // $result is a result set variable created by a SQL query
     $table = mysql_field_table( $result, 0 );
?>
The first field of the query belongs to table <?= $table ?>
```

mysql_field_type

- Retrieve the type of a field in a result set

$type = mysql_field_type($result, $which_field)

mysql_field_type returns the PHP type of a field within the result given by the first argument. The second argument is the number of the field within the SQL query that created the result set. The first field of the query is always 0.

The type of the field is returned as a string, such as ‘int’, ‘string’, ‘real’ and others.

Example

```
<? // $result is a result set variable created by a SQL query
     $type = mysql_field_type( $result, 0);
?>
The first field of the query is a <?= $type ?>
```

mysql_free_result

- Frees the memory used by a result set

Copyright © 2001 O’Reilly & Associates, Inc.

$success = mysql_free_result($result)

mysql_free_result frees any memory used by a result set created by a SQL query. This is done automatically at the end of any script that uses a result set, therefore it is never necessary to use this function. However, when dealing with large result sets it may be beneficial to call this function as soon as you are done with the result set, in order to free of the memory used.

The function returns true on success and false in the event of an error.

> The function *mysql_freeresult* is an alias to this function provided for backwards compatibility. It should not be used in new scripts.

Example

```
<? // $result is the result set created by a SQL query that returned a large number
of rows
     mysql_free_result( $result );
     // The memory used by $result has been freed and $result can no longer be used
as a result set.
?>
```

mysql_insert_id

- Get the id generated from the previous INSERT operation

$id = mysql_insert_id([$mysql])

mysql_insert_id returns the last id that was automatically generated from an auto_increment column for a certain connection. Each connection remembers it's own most recently generated id, so this function works on a per-connection bases. By default it uses the most recently opened MySQL connection. A specific can be specified as the sole parameter to this function.

The function returns a false value if there have been no auto-generated ids for the given connection.

> This function performs the same operation as the MySQL-specific LAST_INSERT_ID() SQL statement.

Example

```
<? // An INSERT statement has been executed on an auto-increment table
     $id = mysql_insert_id();
?> The ID of the row that was just created is <?= $id ?>
```

mysql_list_dbs

- Retrieve a list of databases on a MySQL server

$result = mysql_list_dbs([$mysql])

mysql_list_dbs retrieves a list of databases that are available on a MySQL server. By default, the most currently opened server connection is used. A specific server connection can be specified as the sole parameter.

 Copyright © 2001 O'Reilly & Associates, Inc.

This function returns a result set variable that contains the names of the databases. The data within the result set can be viewed using the *mysql_db_name* function.

> The function *mysql_listdbs* is provides as an alias to this function for backwards compatability. It should not be used for new scripts.

Example

```
<? $result = mysql_list_dbs();
     // $result is a now a result set variable containing a list of the databases
on the server
     // most recently opened. mysql_db_name() can be used to retrieve the actual
names

     // $mysql is a connection variable that was created earlier in the script
     $another_result = mysql_list_dbs( $mysql );
      // $another_result is a reset set variable containing a list of the database
on the server used by
      // the $mysql connection.
?>
```

mysql_list_fields

- Retrieve a list of fields for a database table

$result = mysql_list_fields($database, $table[, $mysql])

mysql_list_fields retrieves a list of fields for the given table within the given database. By default it uses the most recently opened server connection. A specific server connection can by given as the third parameter.

The function returns a result set variable that contains information about the fields. This information can be retrieved through functions like *mysql_field_name*, *mysql_field_type*, *mysql_field_len* and *mysql_field_flags*.

> The function *mysql_listfields* is provided as an alias to this function for backwards compatibility. It should not be used in new scripts.

Example

```
<? // Find out about the fields in the 'People' table of 'mydb'
      $result = mysql_list_fields( 'mydb', 'People' );
     // $result is a result set variable that can be used with mysql_field_name and
other similar
     // functions to examine the fields in this result set.
?>
```

mysql_list_tables

- Retrieve a list of tables in a MySQL database

$result = mysql_list_tables($database[, $mysql])

mysql_list_tables retrieves a list of tables for the given database. By default it uses the most recently opened server connection. A specific server connection can by given as the second parameter.

Copyright © 2001 O'Reilly & Associates, Inc.

The function returns a result set variable that contains the name of the tables in the database. These names can be retrieved through the *mysql_tablename* function.

> The function *mysql_listtables* is provided as an alias to this function for backwards compatibility. It should not be used in new scripts.

Example

```
<? // Find what tables are available in the 'mydb' database...
     $result = mysql_list_tables( 'mydb' );
    // $result is a result set variable that can be used with mysql_tablename to
get the names
?>
```

mysql_num_fields

- Find the number of fields in a result set

$num_fields = mysql_num_fields($result)

mysql_num_fields returns the number of fields contained in each row of the given result set. The result set variable is one that was returned from a function such as *mysql_query*, or any other function that returns a result set.

Note: The function *mysql_numfields* is provided as an alias for backwards compatibility. It should not be used in new scripts.

Example

```
<? // $result is a result set variable returned from a SQL query ?>
The query contains <?= mysql_num_fields( $result ) ?> fields.
```

mysql_num_rows

- Find the number of rows in a result set

$num_rows = mysql_num_rows($result)

mysql_num_rows returns the number of rows contained in the given result set. The result set variable is one that was returned from a function such as *mysql_query*, or any other function that returns a result set.

Example

```
<? // Result is a result set variable that was created from a SELECT SQL query ?>
The query returned <?= mysql_num_rows( $result ) ?> rows...
```

mysql_pconnect

- Open a persistent connection to a MySQL Server

$mysql = mysql_pconnect([$host[, $user [, $password]]])

mysql_pconnect attempts to create (or used an existing) persistent connection with a MySQL server. If successful this function returns a MySQL connection variable that can be used with most of the MySQL functions to specify this connection. In the case of failure, a false value is returned.

 Copyright © 2001 O'Reilly & Associates, Inc.

Before creating a new connection, this function looks for an existing persistent connection that was created with the same arguments. If one is found, it is used instead. Persistent connections stay available even between PHP scripts, as long as everything is running within the same PHP process.

That is, if PHP is running within a web server (such as via Apache's mod_php), persistent connections stay available to every page (see Chapter XX: PHP for a few caveats). Unless *mysql_connect*, when the script ends, the connection stays alive and can be used by another PHP script. On the other hand, if PHP is being run as a CGI, a new process is created every time and therefore the persistent connections are closed at the end of each script and this function is identical to *mysql_connect*.

If no arguments are given, PHP attempts to connect to the MySQL server on the local host at port 3306 using the username of the user that owns the PHP process and a blank password. The hostname can be specified as the first parameter. If a TCP connection is desired on any port other than 3306, it is specified as part of the hostname, using a ':' separator. If a local Unix socket connection is needed, the pathname of the socket should be specified in the same manner. The second argument specifies the username, and the third specifies the password.

> If PHP is running in "safe mode," only the default hostname, port, username and password are allowed.
>
> Care should be taken when specifying password information within a PHP script. Other users of the same machine will likely be able to view the script and see the password. Visitors over the web will not be able to view the script, however, so this is only an issue if you share the server machine with other people.

Example

```
<? // Get a persistent connection to the mysql server on the local host at port
3306 using the
     // username of the owner of the PHP process and a blank password:
    $mysql = mysql_pconnect();

    // Get a persistent connection to the mysql server on the localhost using the
Unix socket
    // /tmp/mysql.sock and the default username and password
    $mysql = mysql_pconnect( 'localhost:/tmp/mysql.sock' );

    // Get a persistent connection to the mysql server at my.server.com, port 3333
using the
    // username 'me' and a blank password
    $mysql = mysql_pconnect('my.server.com:3333', 'me');

   // Get a persistent connection to the mysql server on the localhost, port 3306,
with username
   // 'me' and the password 'mypass'
   $mysql = mysql_pconnect('', 'me', 'mypass');
?>
```

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_query

- Executes a SQL query

$result = mysql_query($query[, $mysql])

mysql_query executes the given SQL query. The query is executed using the most recently opened database connection. A specific server connection can be specified as the second argument.

If the query is a SELECT query and is successful, the function returns a result set variable that can be used with functions like *mysql_fetch_row* to retrieves the contents of the results. If the query is a non-SELECT query (such as INSERT, UPDATE or DELETE) and is successful, the function returns a true value. If the query fails, a false value is returned.

Example

```
<? // Select all of the rows from the 'people' table
     $result = mysql_query( "select * from people" );
     // $result now contains a result set that can be used with myql_fetch_row() to
read the values.

     // Perform a query using the connection specified with the $mysql connection
variable
     // and check to make sure it's a valid result set.
     if (! $result = mysql_query(
               "select firstname from people where firstname like 'P%'", $mysql ) )
{ ?>
               The query was unsuccessful!
       <? } ?>

     // Insert a new row into the table 'people' and check to make sure
     // the insert worked.
     if (! $result = mysql_db_query("insert into people values ('John', 'Doe')" ) {
?>
              The insert failed!
       <? } ?>
```

mysql_unbuffered_query

- Execute a SQL query, without immediately fetching the result rows

$result = mysql_unbuffered_query($query[, $mysql])

mysql_unbuffered_query executes the given SQL query without immediately storing the result. The query is executed using the most recently opened database connection. A specific server connection can be specified as the second argument.

If the query is a SELECT query and is successful, the function returns a result set variable that can be used with functions like *mysql_fetch_row* to retrieves the contents of the results. Unlike *mysql_query*, the results are fetched as requested one row at a time. For large queries this provides faster performance and lower memory usage. However, this also means that the *mysql_num_rows* function cannot be used with result sets created with *mysql_unbuffered_query* because there is no way to know how many rows are in the result set.

Copyright © 2001 O'Reilly & Associates, Inc.

> Because of the internal MySQL workings of this function, it is important that you fetch all of the rows of a result set created with *mysql_unbuffered_query* before creating a new query. Failure to do so may create unpredictable results.

Example

```
<? // Select all of the rows from the 'people' table
     $result = mysql_ubuffered_query( "select * from people" );
     // $result now contains a result set that can be used with myql_fetch_row() to
read the values.
     // Because this is an unbuffered query, no system resources are being used for
PHP right now
     // for this result set. The rows will be fetched from the MySQL server as you
request them
     // But remember that you have to request them all before creating a new
query...
```

mysql_result

- Retrieve a single field of data from a result set

$field = mysql_result($result, $row[, $column])

mysql_result retrieves a single field, or "cell", of information from the given result set (which was created from a call to *mysql_query* or some similar function that returns a result set). The row that contains the field is given as the second parameter. The first row of the result set is always 0.

By default, the first field of the row is returned. A specific field can be specified as a third parameter. This field can be specified by using the fields position (the first field is always 0), name as given in the SQL query, or the qualified SQL name of the field (that is the name of the table and the name of the field separated by a dot).

> *mysql_result* deals with the position inside of a result set differently than functions that return an entire row, such as *mysql_fetch_row*. Therefore you should not mix *mysql_result* with any other function that reads data from a result set.

Example

```
<? // $result is a result set created by the SQL query:
     //  SELECT firstName as fname, lastName from People
The first first name in the 'People' table is <?= mysql_result( $result, 0 ) ?><br>
The first last name in the 'People' table is <?= mysql_result( $result, 0 , 1 )
?><br>
The second first name in the 'People' table is <?= mysql_result( $result, 1,
'fname' ) ?><br>
The second last name in the 'People' table is <?= mysql_result( $result, 1,
'lastName' ) ?><br>
The third last name in the 'People' table is <?= mysql_result( $result, 2,
'People.lastName' )?><br>
// Since the name of 'firstName' is specified as the alias 'fname' we cannot use
the
// table.field notation, but must use the alias instead
```

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_select_db

- Select a new default database

$success = mysql_select_db($database[, $mysql])

mysql_select_db selects a new default database. Any SQL query that uses tables within specifying a database will use this database.

By default, this function sets the database on the most recently opened connection. A specific connection can be given as the second argument. If no specific connection is given, and no current connection exists, PHP will attempt to open a new connection as if *mysql_connect* was called with no parameters.

> The function *mysql_selectdb* is provided as an alias to this function for backwards compatibility. It should not be used in new scripts.

This function returns true on success and false in the case of failure.

Example

```
<? mysql_select_db( 'mydb');
     // Now the query 'SELECT * from People' will use the People table in 'mydb'
     mysql_selectdb('myotherdb');
     // Now the query 'SELECT * from People' will use the People table in
'myotherdb'
?>
```

mysql_tablename

- Get table name of field

$table = mysql_tablename($result, $which_table[, $unused])

mysql_tablename returns the name of a table from a result variable created from a call to *mysql_list_tables*. The second argument to this function indicates which table name out of all of those in the result set to return. The first table in the result set is always 0. The number of table names in the result set can be obtained from *mysql_num_rows*.

This function returns a false value in the case of an error. Supplying a row number that does not exist in the result set will result in an error.

> This function is implemented as an alias to *mysql_result*. Because of this, it is possible to supply a third argument to this function, which represents the name of a field in the result set. However, this function is not useful in the context of *mysql_tablename*.

Example

```
<? // $result is the result of a call to mysql_list_tables contains the names of
all of the tables
    // available to the user.
    for ( $i = 0; $i < mysql_num_rows( $result ); $i++ ) {
           echo mysql_tablename( $result, $i );
    }
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
?>
```

mysql_get_client_info

- Retrieve information about the MySQL client library

$info = mysql_get_client_info()

mysql_get_cliet_info returns a string of information about the MySQL client library that PHP is using. Currently it returns the version number of the library.

Example

```
PHP is using MySQL client library version <?= mysql_get_client_info() ?>.
```

mysql_get_host_info

- Retrieve information about a MySQL server connection

$info = mysql_get_host_info([$mysql])

mysql_get_host_info returns a string of information about a MySQL server connection. The information indicates the type of connection (TCP or Unix socket) and the port used. By default this function returns information about the most recently opened server connection. A specific connection can be given as the sole parameter.

Example

```
You are currently connect to MySQL on <?= mysql_get_host_info() ?>
```

mysql_get_proto_info

- Retrieves information about the protocol used in a MySQL connection

$info = mysql_get_proto_info([$mysql])

mysql_get_proto_info returns a string of information about the protocol used in a MySQL server connection. This information currently contains the protocol version number. By default this function returns information about the most recently opened server connection. A specific connection can be given as the sole parameter.

Example

```
You are connected to MySQL using protocol <?= mysql_get_proto_info() ?>
```

mysql_get_server_info

- Retrieves information about a MySQL server

$info = mysql_get_server_info([$mysql])

mysql_get_server_info returns a string of information about a MySQL server that the script is current connected to. This information is currently the version of MySQL that the server is running. By default the function uses the most recently opened server connection. A specific connection can be given as the sole parameter.

Example

```
The server you are connected to is using MySQL version <?= mysql_get_server_info()
?>
```

Copyright © 2001 O'Reilly & Associates, Inc.

> The following functions are part of the experimental database abstraction layer dbx. Care must be taken when mixing these functions with the mysql functions above. See Chapter XX: PHP for information on the things to watch out for while using these functions.

dbx_close

- Close a database connection

dbx_close($db_connection)

dbx_close closes an open database connection. The given connection variable must be an active connection variable created by *dbx_connect*. PHP will automatically close any open connections at the end of the script. Therefore it is usually not necessary to call this function.

The function returns true on success and false in the case of an error.

Example

```
<?   // $dbconn is an active connection variable created with dbx_connect
       dbx_close( $dbconn ); // The connection is now closed
?>
```

dbx_connect

- Connect to a database

$dbconn = dbx_connect ($db_type, $host, $database, $username, $password [, $make_persnistent])

dbx_connect attempts to create a connection to a database. The first argument is the type of database that you are connecting to. For MySQL this will be “mysql”.

The second argument is the hostname of the MySQL server. If a TCP connection is desired on any port other than 3306, it is specified as part of the hostname, using a ‘:’ separator. If a local Unix socket connection is needed, the pathname of the socket should be specified in the same manner (with the hostname given as ‘localhost’). The third argument is the default database to be used for the connection., The fourth argument is the username and the fifth argument is the password.

It is possible to create this connection as a persistent connection by passing the constant DBX_PERSISTENT as a sixth argument. If this is done, PHP will first check to see if there is an existing persistent connection created with the same parameters. If so, that connection will be used instead of creating a new one. In addition, the connection will not be closed automatically at the end of the script but will be kept open to be re-used by another script.

If the connection is successful a dbx connection variable is returned that can be used with the other dbx functions. If there is an error, the function returns false.

Besides being used with the other dbx functions, the connection variable returned by this function is also an object with three available properties. The first property is ‘database’

 Copyright © 2001 O’Reilly & Associates, Inc.

which is the name of the default database selected in the connection. The second property is 'module' which returns a constant associated with the type of database chosen ("mysql" in our case). The third property is 'handle' which is an active connection variable specific to the type of database chosen. In our case, this is an active MySQL connection variable that can be used with any of the mysql_* functions..

Example

```
<? // Connect to a MySQL server running on my.server.com with username 'me' and
     // password 'mypass' using 'mydb' as the default database.
     $dbconn = dbx_connect( 'mysql', 'my.server.com', 'mydb', 'me', 'mypass' );
     if ( ! $dbconn ) { echo "There has been an error..."; }
     .else { // The connection was successful! You have an active connection
variable that
                // can be used with the other dbx functions. This variable also has
the property
                // $dbconn->handle which can be used with any of the mysql_*
functions.
      }

      // Connect to a MySQL server running on the localhost with the socket
      //  /tmp/mysql.sock using the default database 'mydb', username 'me' and
password 'mypass'
      // Also make this a persistent connection that will remain available to other
scripts once
      // this script is finished.
      $dbconn = dbx_connect('mysql', 'localhost:/tmp/mysql.sock',
                            'mydb', 'me', 'mypass', DBX_PERSISTENT );
?>
```

dbx_error

- Return the last dbx-related error message

$error = dbx_error($dbconn)

dbx_error returns the most recent error involving the database-specific module used by the given dbx connection variable. This is not necessarily that last error that happened in this connection, but the last error that happened in any connection using that database-specific module. Since we are using the MySQL module, this function will return the last error encountered by any MySQL connection.

Example

```
<? // $dbconn is an active dbx connection variable created with dbx_connect ?>
The last MySQL-related error is <?= dbx_error( $dbconn ) ?>
```

dbx_query

- Execute a SQL query

$dbresult = dbx_query($dbconn, $query[, $flags])

dbx_query executes the given SQL query using the given database connection. If the query fails the function returns false. If the query succeeds and has data (e.g., a SELECT query) the function returns a dbx result object (see below). If the query succeeds and has no data (e.g., an INSERT, UPDATE or DELETE query), the function returns a true value.

For queries with data, the returned dbx result object is an object with several properties:

Copyright © 2001 O'Reilly & Associates, Inc.

handle

This is a module-specific result set. Since we are using MySQL, this is a MySQL result set variable that can be used with any of the mysql_* functions above that take a result set.

cols

This is the number of fields per row in the result set.

rows

This is the number of rows of data in the result set.

data

This is a two-dimensional array containing all of the data in the result set. The first dimension of this array is the row of the data. The first row is always 0. The second dimension of the array specified the field you are interested in. By default, this dimension contains both numeric and associative indices. The numeric index is the position of the field as given in the SQL query (the first field is always 0). The associative index is the name of the field as given in the SQL query (fields with no name have no associative index). The numeric and associative indices access the exact same data, so that if you modify a value using the numeric index, the corresponding associative element will change as well.

info

This is a two-dimensional array which provides access to information about the fields in the query. The first dimension of the array is a string given the type of information request. Currently 'name' and 'type' are the only valid values here. The second dimension is the position of the field desired as given in the SQL query. The first field is always 0.

The default behavior of this function can be modified by providing one or flags as the third argument. Multiple flags can be used by combining them with Boolean-or.

DBX_RESULT_INFO - This flag causes the info property to be available in the result object.
DBX_RESULT_INDEX - The flag causes the second dimension of the data property to contain numeric indices.
DBX_RESULT_ASSOC - This flag causes the second dimension of the data property to contain associative indices. This also causes the info property to be available.

Note that the behavior given by all three flags is the default behavior. Therefore, the only purpose to specify the flags is to *remove* behavior. For example, if you didn't want the info property, you would have to specify the DBX_RESULT_INDEX flag by itself, which would eliminate the DVX_RESULT_INFO flag and the DBX_RESULT_ASSOC flag (which also causes the info property to appear). Or if you did not want associative indices in the data property you could specify both DBX_RESULT_INFO and DBX_RESULT_INDEX but not DBX_RESULT_ASSOC.

Example

```
<? // $dbconn is a dbx connection variable created with dbx_connect
     $dbresult = dbx_query( $dbconn, "SELECT firstName, lastName from People" );
     if ( ! $dbresult ) { echo "There was an error!"; }
     else { ?>
The first first name in the People table is <?= $dbresult->data[0][0] ?><br>
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
The first last name in the People table is <?= $dbresult->data[0][‘lastName’]
?><br>
This should say ‘firstName’: <?= $dbresult->info[‘name’][0] ?>
This is the SQL type of the lastName field: <?= $dbresult->info[‘type’][1] ?>
There are <?= $dbresult->rows ?> people in the People table.
This should say '2' since we specified two columns: <?= $dbresult->cols ?>
<? // We can use the $dbresult->handle property with any mysql_* function
     //  that takes a result set variable.
    }

// Make a new query but don’t make the info property or the associative indices
because we’re
    expecting a large result set and want to be as effecient as possible:
    $dbresult = dbx_query( $dbconn, “SELECT * from People”, DBX_RESULT_INDEX );
?>
```

dbx_sort

- Re-sorts a result set

$success = dbx_sort($dbresult, $sort_function)

dbx_sort re-sorts the given result set, which was returned from *dbx_query*. The second argument is the name of a function that is used for sorting. The function can be any function, as long as it takes two parameters and returns an integer (-1 if the first parameter is less than the second, 0 if they are equal and 1 if the first is greater than the second).

The function resorts the ‘data’ property of the result set object. In particular, it resorts the order if the rows (the first dimension) of that property. The *dbx_sort* function passes single rows of the ‘data’ property (as an array) as the parameters to the sorting function. Therefore the sorting function should expect arrays as its parameters.

The function returns a true value on success and false in the case of failure.

Example

```
<? // $dbresult is a dbx result set returned by dbx_query

        function sort_lastname( $a, $b ) {
                   if ( $a[‘lastName’] > $b[‘lastName’] ) return 1;
                   else if ($a[‘lastName’] < $b[‘lastName’] ) return -1;
                   else return 0;
        }

         dbx_sort( $dbresult, “sort_lastname” );
          // The $dbresult->data array is now sorted by the last name of the
entries...
?>
```

dbx_compare

- Compare two rows of a dbx result set

$greater_or_less = dbx_compare($row_array_a, $row_array_b, $which_column[, $flags])

dbx_compare compares a single field of two arrays to determine which is greater. This is used with *dbx_sort* to sort a result set. The first two arguments are arrays (usually from the ‘data’ property of a dbx result set). The third argument is the name, or position (first is

always 0) of a field in the result set. The function then compares the given field of the two arrays and returns 1 if the first field is greater, -1 if the second field is greater and 0 of the two fields are equal.

Passing in one or more constants as a fourth parameter can modify this default behavior. Multiple constants should be separated by a Boolean-or. There are two different sets of constants; one of each can be specified, if desired:

DBX_CMP_ASC and DBX_CMP_DESC
: These flags determine which direction the sorting is performed. Default is DBX_CMP_ASC that performs an ascending sort. The DBX_CMP_DESC flag provides a descending sort.

DBX_CMP_NATIVE, DBX_CMP_TEXT and DBX_CMP_NUMBER
: These flags determine how the sorting is performed. The default DBX_CMP_NATIVE performs a straight character-value comparison no matter what the contents of the data are. The DBX_CMP_TEXT flag causes the fields to be compared as alphabetical text using the local character set of the PHP server. The DBX_CMP_NUMBER flag causes the fields to be compared as numbers, with any non-numerical characters ignored.

Example

```
<? //  $dbresult is a dbx result set returned by dbx_query

     // Sort the results by the last name, in descending order, making sure the
values are compared as
     // as text in the local character-set.
     function sort_lastname( $a, $b ) {
                 return dbx_compare( $a, $b, "lastName", DBX_DESC|DBX_TEXT );
     }

       dbx_sort( $dbresult, "sort_lastname" );
          // The $dbresult->data array is now sorted by the last name of the
entries...
?>
```


Copyright © 2001 O'Reilly & Associates, Inc.

24

C Reference

MySQL C API

The MySQL C API uses several defined datatypes beyond the standard C types. These types are defined in the 'mysql.h' header file that must be included when compiling any program that uses the MySQL library.

Datatypes

`MYSQL`
: *A structure representing a connection to the database server. The elements of the structure contain the name of the current database and information about the client connection among other things.*

`MYSQL_FIELD`
: A structure containing all of the information concerning a specific field in the table. Of all of the types created for MySQL, this is the only one whose member variables are directly accessed from client programs. Therefore it is necessary to know the layout of the structure:

*char *name*
: The name of the field.

*char *table*
: The name of the table containing this field. For result sets that do not correspond to real tables, this value is null.

*char *def*
: The default value of this field, if one exists. This value will always be null unless `mysql_list_fields` is called, after which this will have the correct value for fields that have defaults.

enum enum_field_types type

The type of the field. The type is one of the MySQL SQL datatypes. The following field types (along with their corresponding MySQL SQL data type) are currently defined:

FIELD_TYPE_TINY (TINYINT)
FIELD_TYPE_SHORT (SMALLINT)
FIELD_TYPE_LONG (INTEGER)
FIELD_TYPE_INT24 (MEDIUMINT)
FIELD_TYPE_LONGLONG (BIGINT)
FIELD_TYPE_DECIMAL (DECIMAL or NUMERIC)
FIELD_TYPE_FLOAT (FLOAT)
FIELD_TYPE_DOUBLE (DOUBLE or REAL)
FIELD_TYPE_TIMESTAMP (TIMESTAMP)
FIELD_TYPE_DATE (DATE)
FIELD_TYPE_TIME (TIME)
FIELD_TYPE_DATETIME (DATETIME)
FIELD_TYPE_YEAR (YEAR)
FIELD_TYPE_STRING (CHAR or VARCHAR)
FIELD_TYPE_BLOB (BLOB or TEXT)
FIELD_TYPE_SET (SET)
FIELD_TYPE_ENUM (ENUM)
FIELD_TYPE_NULL (NULL)
FIELD_TYPE_CHAR (TINYINT) (Deprecated, replaced by FIELD_TYPE_TINY)

unsigned int length

The size of the field based on the field's type.

unsigned int max_length

If accessed after calling `mysql_list_fields`, this contains the length of the maximum value contained in the current result set. If the field is a BLOB-style field (e.g. BLOB, TEXT, LONGBLOB, MEDIUMTEXT, etc.) this value will always be 8000 (8kB) if called before the actual data is retrieved from the result set (by using `mysql_store_result()`, for example). Once the data has been retrieved this field will contain the actual maximum length.

unsigned int flags

Zero or more option flags. The following flags are currently defined:

NOT_NULL_FLAG
If defined, the field cannot contain a NULL value.

PRI_KEY_FLAG
If defined, the field is a primary key.

UNIQUE_KEY_FLAG
If defined, the field is part of a unique key.

MULTIPLE_KEY_FLAG
If defined, the field is part of a key.

BLOB_FLAG
If defined, the field is of type `BLOB` or `TEXT`.

UNSIGNED_FLAG
If defined, the field is a numeric type with an unsigned value.

ZEROFILL_FLAG
If defined, the field was created with the `ZEROFILL` flag.

 Copyright © 2001 O'Reilly & Associates, Inc.

BINARY_FLAG
If defined, the field is of type CHAR or VARCHAR with the BINARY flag.

ENUM_FLAG
If defined, the field is of type ENUM.

AUTO_INCREMENT_FLAG
If defined, the field has the AUTO_INCREMENT attribute.

TIMESTAMP_FLAG
If defined, the field is of type TIMESTAMP.

SET_FLAG
If defined, the field is of type SET.

NUM_FLAG
If defined, the field is a numeric type (e.g. INT, DOUBLE, etc.).

PART_KEY_FLAG
If defined, the field is of part of a key. This flag is not meant to be used by clients, and it's behavior may change in the future.

GROUP_FLAG
This flag is not meant to be used by clients, and it's behavior may change in the future.

UNIQUE_FLAG
This flag is not meant to be used by clients, and it's behavior may change in the future.

unsigned int decimals
When used with a numeric field, it lists the number of decimals used in the field.

The following macros are provided to help examine the MYSQL_FIELD data:

IS_PRI_KEY(flags)
Returns true if the field is a primary key. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_NOT_NULL(flags)
Returns true if the field is defined as NOT NULL. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_BLOB(flags)
Returns true if the field is of type BLOB or TEXT. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_NUM(type)
Returns true if the field type is numeric. This macro takes the 'type' attribute of a MYSQL_FIELD structure as its argument.

IS_NUM_FIELD(field)
Returns true if the field is numeric. This macro takes a MYSQL_FIELD structure as its argument.

MYSQL_FIELD_OFFSET
A numerical type indicating the position of the "cursor" within a row.

MYSQL_RES
A structure containing the results of a SELECT (or SHOW) statement. The actual output of the query must be accessed through MYSQL_ROW elements of this structure.

Copyright © 2001 O'Reilly & Associates, Inc.

`MYSQL_ROW`
: *A single row of data returned from a `SELECT` query. Output of all MySQL data types are stored in this type (as an array of character strings).*

`my_ulonglong`
: A numerical type used for MySQL return values. The value ranges from 0 to 1.8E19, with -1 used to indicate errors.

mysql_affected_rows

my_ulonglong mysql_affected_rows(MYSQL *mysql)

Returns the number of rows affected by the most recent query. When used with a non-`SELECT` query, it can be used after the `mysql_query` call that sent the query. With `SELECT`, this function is identical to `mysql_num_rows`. This function returns 0, as expected, for queries that affect or return no rows. In the case of an error, the function returns –1.

When an UPDATE query causes no change in the value of a row, it is not usually considered to be 'affected'. However, if the CLIENT_FOUND_ROWS flag is used when connecting to the MySQL server, any rows that match the 'where' clause of the UPDATE query will be considered affected.

Example

```
/* Insert a row into the people table */
mysql_query(&mysql, "INSERT INTO people VALUES ('', 'Illyana Rasputin', 16)";
num = mysql_affected_rows(&mysql);
/* num should be 1 if the INSERT (of a single row) was successful, and -1 if
   there was an error */

/* Make any of 'HR', 'hr', 'Hr', or 'hR' into 'HR'. This is an easy way to
   force a consistent capitalization in a field.
mysql_query(&mysql, "UPDATE people SET dept = 'HR' WHERE dept LIKE 'HR'");
affected = mysql_affected_rows(&mysql);
/* By default, 'affected' will contain the number of rows that were changed.
   That is, the number of rows that had a dept value of 'hr', 'Hr' or 'hR'.
   If the CLIENT_FOUND_ROWS flag was used, 'affected' will contain the number
   of rows that matched the where (same as above plus 'HR'). */
```

mysql_change_user

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

Changes the currently authenticated user and database. This function, re-authenticates the current connection using the given username and password. It also changes the default database to the given database (which can be NULL if no default is desired). If the password is incorrect for the given username or if the new user does not have rights to access the given database, a false value is returned and no action is taken. Otherwise, the rights of the new user take effect, the default database is selected and a true value is returned.

 Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
if (! mysql_change_user( &mysql, new_user, new_pass, new_db ) ) {
      printf("Change of User unsuccessful!");
      exit(1);
}
/* At this point, the connection is operating under the access rights of the
   new username, and the new database is the default. */
```

mysql_character_set_name

char* mysql_character_set_name(MYSQL *mysql)

Returns the name of the default character set used by the MySQL server. A generic installation of the MySQL source uses the ISO-8859-1 character set by default.

Example

```
printf("This server uses the %s character set by default\n",
         mysql_character_set_name(&mysql));
```

mysql_close

void mysql_close(MYSQL *mysql)

Ends a connection to the database server. If there is a problem when the connection is broken, the error can be retrieved from the `mysql_err` function.

Example

```
mysql_close(&mysql);
/* The connection should now be terminated */
```

mysql_connect

MYSQL *mysql_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd)

Creates a connection to a MySQL database server. The first parameter must be a predeclared `MYSQL` structure. The second parameter is the hostname or IP address of the MySQL server. If the host is an empty string or `localhost`, a connection will be made to the MySQL server on the same machine. The final two parameters are the username and password used to make the connection. The password should be entered as plain text, not encrypted in any way. The return value is the MYSQL structure passed as the first argument, or NULL if the connection failed. (Because the structure is contained as an argument, the only use for the return value is to check if the connection succeeded.)

> This function has been deprecated in the newer releases of MySQL and the `mysql_real_connect` function should be used instead.

Example

```
/* Create a connection to the local MySQL server using the name "bob" and
   password "mypass" */
MYSQL mysql;
if(!mysql_connect(&mysql, "", "bob", "mypass")) {
```

```
                    printf("Connection error!\n");
                    exit(0);
}
/* If we've reached this point we have successfully connected to the database
   server. */
```

mysql_create_db

int mysql_create_db(MYSQL *mysql, const char *db)

Creates an entirely new database with the given name. The return value is zero if the operation was successful and nonzero if there was an error.

> This function has been deprecated in the newer releases of MySQL. MySQL now supports the `CREATE DATABASE` SQL statement. This should be used, via the *mysql_query* function, instead.

Example

```
/* Create the database 'new_database' */
result = mysql_create_db(&mysql, "new_database");
```

mysql_data_seek

void mysql_data_seek(MYSQL_RES *res, unsigned int offset)

Moves to a specific row in a group a results. The first argument is the `MYSQL_RES` structure that contains the results. The second argument is the row number you wish to seek to. The first row is 0. This function only works if the data was retrieved using `mysql_store_result` (datasets retrieved with `mysql_use_result` are not guaranteed to be complete).

Example

```
/* Jump to the last row of the results */
mysql_data_seek(results, mysql_num_rows(results)-1);
```

mysql_debug

mysql_debug(char *debug)

Manipulates the debugging functions if the client has been compiled with debugging enabled. MySQL uses the Fred Fish debugging library, which has far too many features and options to detail here.

Example

```
/* This is a common use of the debugging library. It keeps a trace of the
   client program's activity in the file "debug.out" */
mysql_debug("d:t:O,debug.out");
```

mysql_drop_db

int mysql_drop_db(MYSQL *mysql, const char *db)

Copyright © 2001 O'Reilly & Associates, Inc.

Destroys the database with the given name. The return value is zero if the operation was successful and nonzero if there was an error.

> This function has been deprecated in the newer releases of MySQL. MySQL now supports the `DROP DATABASE` SQL statement. This should be used, via the `mysql_query` function, instead.

Example

```
/* Destroy the database 'old_database' */
result = mysql_drop_db(&mysql, "old_database");
```

mysql_dump_debug_info

int mysql_dump_debug_info(MYSQL *mysql)

This function causes the database server to enter debugging information about the current connection into its logs. You must have Process privilege in the current connection to use this function. The return value is zero if the operation succeeded and nonzero in the case of an error.

Example

```
result = mysql_dump_debug_info(&mysql);
/* The server's logs should now contain information about this connection
   If something went wrong so that this is not the case, 'result' will have
   a false value.*/
```

mysql_eof

my_bool mysql_eof(MYSQL_RES *result)

Returns a nonzero value if there is no more data in the group of results being examined. If there is an error in the result set, zero is returned. This function only works of the result set was retrieved with the `mysql_use_result` function (`mysql_store_result` retrieves the entire result set, making this function unnecessary).

> This function has been deprecated in the newer releases of MySQL. The `mysql_errno` and `mysql_error` functions return more information about any errors that occur and they are more reliable.

Example

```
/* Read through the results until no more data comes out */
while((row = mysql_fetch_row(results))) {
       /* Do work */
}

if(!mysql_eof(results))
   printf("Error. End of results not reached.\n");
```

mysql_errno

unsigned int mysql_errno(MYSQL *mysql)

Copyright © 2001 O'Reilly & Associates, Inc.

Returns the error number of the last error associated with the current connection. If there have been no errors in the connection, the function returns zero. The actual text of the error can be retrieved using the `mysql_error` function. The defined names for the client errors can be found in the errmsg.h header file. The defined names for the server error can be found in the mysqld_error.h header file.

Example

```
error = mysql_errno(&mysql);
printf("The last error was number %d\n", error);
```

mysql_error

char *mysql_error(MYSQL *mysql)

Returns the error message of the last error associated with the current connection. If there have been no errors in the connection, the function returns an empty string. Error messages originating on the server will always be in the language used by the server (chosen at startup time with the `--language` option). The language of the client error messages can be chosen when compiling the client library. At the time of this writing MySQL supported the following languages: Czech, Danish, Dutch, English, Estonian, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian (standard and 'ny'), Polish, Portuguese, Romanian, Russian, Slovak, Spanish, and Swedish.

Example

```
printf("The last error was '%s'\n", mysql_error(&mysql));
```

mysql_escape_string

unsigned int mysql_escape_string(char *to, const char *from, unsigned int length)

Encodes a string so that it is safe to insert it into a MySQL table. The first argument is the receiving string, which must be at least one character greater than twice the length of the second argument, the original string. (That is, to >= from*2+1.) The third argument indicates that only that many bytes are copied from the originating string before encoding it. The function returns the number of bytes in the encoded string, not including the terminating null character.

> While not officially deprecated, this function is generally inferior to the `mysql_real_escape_string` function which does everything this function does, but also takes into account the character set of the current connection, which may affect certain escape sequences.

Example

```
char name[15] = "Bob Marley's";
char enc_name[31];
mysql_escape_string(enc_name, name);
/* enc_name will now contain "Bob Marley\'s" (the single quote is escaped).
```

mysql_fetch_field

MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)

Copyright © 2001 O'Reilly & Associates, Inc.

Returns a `MYSQL_FIELD` structure describing the current field of the given result set. Repeated calls to this function will return information about each field in the result set until there are no more fields left, and then it will return a null value.

Example

```
MYSQL_FIELD *field;

while((field = mysql_fetch_field(results)))
{
    /* You can examine the field information here */
}
```

mysql_fetch_field_direct

MYSQL_FIELD * mysql_fetch_field_direct(MYSQL_RES * result, unsigned int fieldno)

This function is the same as `mysql_fetch_field`, except that you specify which field you wish to examine, instead of cycling through them. The first field in a result set is 0.

Example

```
MYSQL_FIELD *field;

/* Retrieve the third field in the result set for examination */
field = mysql_fetch_field_direct(results, 2);
```

mysql_fetch_fields

MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES * result)

The function is the same as `mysql_fetch_field`, except that it returns an array of `MYSQL_FIELD` structures containing the information for every field in the result set.

Example

```
MYSQL_FIELD *field; /* A pointer to a single field */
MYSQL_FIELD *fields; /* A pointer to an array of fields */

/* Retrieve all the field information for the results */
fields = mysql_fetch_fields(results);
/* Assign the third field to 'field' */
field = fields[2];
```

mysql_fetch_lengths

unsigned long *mysql_fetch_lengths(MYSQL_RES *result)

Returns an array of the lengths of each field in the current row. A null value is returned in the case of an error. You must have fetched at least one row (with `mysql_fetch_row`) before you can call this function. This function is the only way to determine the lengths of variable length fields, such as `BLOB` and `VARCHAR`, before you use the data.

> This function is especially useful when reading binary data from a BLOB. Since all MySQL data is retrieved as strings (char *), it is common to use the `strlen()` function to determine the length of a

data value. However, for binary data `strlen()` returns inaccurate results because it stops at the first null character. In these cases use can use `mysql_fetch_lengths` to retrieve the accurate length for a data value.

Example

```
unsigned long *lengths;

row = mysql_fetch_row(results);
lengths = mysql_fetch_lengths(results);
printf("The third field is %d bytes long\n", lengths[2]);
```

mysql_fetch_row

MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)

Retrieves the next row of the result and returns it as a `MYSQL_ROW` structure. A null value is returned if there are no more rows or there is an error. In the current implementation, the `MYSQL_ROW` structure is an array of character strings that can be used to represent any data. If a data element is NULL within the database, the MYSQL_ROW array element for that data element will be a null pointer. This is necessary to distinguish between a value that is NULL and a value that is simply an empty string (which will be a non-null pointer to a null value).

Example

```
MYSQL_ROW row;

row = mysql_fetch_row(results);
printf("The data in the third field of this row is: %s\n", row[2]);
```

mysql_field_count

unsigned int mysql_field_count(MYSQL *mysql)

Returns the number of columns contained in a result set. This function is most useful to check the type of query last executed. If a call to `mysql_store_result` returns a null pointer for a result set, either the query was a non-SELECT query (such as an UPDATE, INSERT, etc.) or there was an error. By calling `mysql_field_count` you can determine which was the case, since a non-SELECT query will always have zero fields returned and a SELECT query will always have at least one.

Example

```
MYSQL_FIELD field;
MYSQL_RES *result;

// A query has been executed and returned success
result = mysql_store_result();
if (! result ) {
     // Ooops, the result pointer is null, either the query was a non-SELECT
     // query or something bad happened!
     if ( mysql_field_count(&mysql) ) {
          // The number of columns queried is greater than zero, it must have
          // been a SELECT query and an error must have occurred.
     } else {
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
            // Since the number of columns queried is zero, it must have been
            // a non-SELECT query, so all is well...
     }
}
```

mysql_field_seek

MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset)

Seeks a result set to the given field of the current row. The position set by this function is used when `mysql_fetch_field` is called. The `MYSQL_FIELD_OFFSET` value passed should be the return value of a `mysql_field_tell` call (or another `mysql_field_seek`). Using the value 0 will seek to the beginning of the row. The return value is the position of the cursor before the function was called.

Example

```
MYSQL_FIELD field;

/* result is a MYSQL_RES structure containing a result set */
/* ... do some stuff */
/* Seek back to the beginning of the row */
old_pos = mysql_field_seek(results, 0);
/* Fetch the first field of the row */
field = mysql_fetch_field(results);
/* Go back to where you where */
mysql_field_seek(results, old_pos);
```

mysql_field_tell

MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)

Returns the value of the current field position within the current row of the result set. This value is used with `mysql_field_seek`.

Example

```
MYSQL_FIELD field1, field2, field3;
/* results is a MYSQL_RES structure containing a result set */

/* Record my current position */
old_pos = mysql_field_tell(results);
/* Fetch three more fields */
field1 = mysql_field_field(results);
field2 = mysql_field_field(results);
field3 = mysql_field_field(results);
/* Go back to where you where */
mysql_field_seek(results, old_pos);
```

mysql_free_result

void mysql_free_result(MYSQL_RES *result)

Frees the memory associated with a `MYSQL_RES` structure. This must be called whenever you are finished using this type of structure or else memory problems will occur. This should only be used on a pointer to an actual MYSQL_RES structure. For example,

if a call to `mysql_store_result` returned a null pointer, this function should be used.

Example

```
MYSQL_RES *results;
/* Do work with results */
/* free results... we know it's not null since we just did work with
   it, but we'll check just to be safe. */
if (results)
    mysql_free_result(results);
```

mysql_get_client_info

char *mysql_get_client_info(void)

Returns a string with the MySQL library version used by the client program.

Example

```
printf("This program uses MySQL client library version %s\n",
       mysql_get_client_info()));
```

mysql_get_host_info

char *mysql_get_host_info(MYSQL *mysql)

Returns a string with the hostname of the MySQL database server and the type of connection used (e.g., Unix socket or TCP).

Example

```
printf("Connection info: %s", mysql_get_host_info(&mysql));
```

mysql_get_proto_info

unsigned int mysql_get_proto_info(MYSQL *mysql)

Returns the MySQL protocol version used in the current connection as an integer. As a general rule, the MySQL network protocol will only change between minor releases of MySQL. That is, all releases of MySQL 3.23.x should have the same protocol version number.

Example

```
printf("This connection is using MySQL connection protocol ver. %d\n",
        mysql_get_proto_info());
```

mysql_get_server_info

char *mysql_get_server_info(MYSQL *mysql)

Returns a string with the version number of the MySQL database server used by the current connection.

Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
printf("You are currently connected to MySQL server version %s\n",
        mysql_get_server_info(&mysql);
```

mysql_info

char *mysql_info(MYSQL *mysql)

Returns a string containing information about the most recent query, if the query was of a certain type. Currently, the following SQL queries supply extra information via this function: `INSERT INTO` (when used with a `SELECT` clause or a VALUES clause with more than one record); `LOAD DATA INFILE`; `ALTER TABLE`; and UPDATE. If the last query had no additional information (e.g., it was not one of the above queries), this function returns a null value.

The format of the returned string depends on which of the above queries is used:

INSERT INTO or ALTER TABLE: Records: *n* Duplicates: *n* Warnings: *n*

LOAD DATA INFILE: Records: *n* Deleted: *n* Skipped: *n* Warnings: *n*

UPDATE: Rows matched: *n* Changed: *n* Warnings: *n*

Example

```
/* We just sent LOAD DATA INFILE query reading a set of record from a file into
   an existing table */
printf("Results of data load: %s\n", mysql_info(&mysql));
/* The printed string looks like this:
Records: 30 Deleted: 0 Skipped: 0 Warnings: 0
*/
```

mysql_init

MYSQL *mysql_init(MYSQL *mysql)

Initializes a `MYSQL` structure used to create a connection to a MySQL database server. This, along with `mysql_real_connect`, is currently the approved way to initialize a server connection. You pass this function a `MYSQL` structure that you declared, or a null pointer, in which case a `MYSQL` structure will be created and returned. Structures created by this function will be properly freed when `mysql_close` is called. Conversly, if you passed your own pointer, you are responsible for freeing it when the time comes. A null value is returned if there is not enough memory to initialize the structure.

> As of the current release of MySQL, MySQL clients will crash on certain platforms (such as SCO Unix) when you pass in a pointer to a MYSQL structure that you allocated yourself. If this is happening to you, just pass in NULL and use the pointer created by the MySQL library. As an added bonus, you don't have to worry about freeing it if you do that.

Example

```
MYSQL mysql;
```

```
if (!mysql_init(&mysql)) {
                  printf("Error initializing MySQL client\n");
                  exit(1);
}
/* Now you can call mysql_real_connect() to connect to a server... */

/* Alternative method: */
MYSQL *mysql;

mysql = mysql_init(NULL);
if (!mysql) {
     printf("Error initializing MySQL client\n");
     exit(1);
}
```

mysql_insert_id

my_ulonglong mysql_insert_id(MYSQL *mysql)

Returns the generated for an `AUTO_INCREMENT` field if the last query created a new row. This function is usually used immediately after a value is inserted into an `AUTO_INCREMENT` field, to determine the value that was inserted. This value is reset to 0 after any query that does not insert a new auto-increment row.

> The MySQL-specific SQL function LAST_INSERT_ID() also returns the value of the most recent auto-increment. In addition, it is not reset after each query, and so can be called at any time to retrieve that value of the last auto-increment INSERT executed during the current session.

Example

```
/* We just inserted an employee record with automatically generated ID into
   a table */
id = mysql_insert_id(&mysql);
printf("The new employee has ID %d\n", id);
/* As soon as we run another query, mysql_insert_id will return 0 */
```

mysql_kill

int mysql_kill(MYSQL *mysql, unsigned long pid)

Attempts to kill the MySQL server thread with the specified Process ID. This function returns zero if the operation was successful and nonzero on failure. You must have Process privileges in the current connection to use this function.

The process IDs are part of the process information returned by the `mysql_list_processes` function.

Example

```
/* Kill thread 4 */
result = mysql_kill(&mysql, 4);
```


Copyright © 2001 O'Reilly & Associates, Inc.

mysql_list_dbs

MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild)

Returns a `MYSQL_RES` structure containing the names of all existing databases that match the pattern given by the second argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all databases are listed. Like all `MYSQL_RES` structures, the return value of this function must be freed with `mysql_free_result`. This function returns a null value in the case of an error.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW databases'.

Example

```
MYSQL_RES databases;
databases = mysql_list_dbs(&mysql, (char *)NULL);
/* 'databases' now contains the names of all of the databases in the
   MySQL server */
/* ... */
mysql_free_result( databases );
/* Find all databases that start with 'projectName' */
databases = mysql_list_dbs(&mysql, "projectName%");
```

mysql_list_fields

MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild)

Returns a `MYSQL_RES` structure containing the names of all existing fields in the given table that match the pattern given by the third argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all fields are listed. Like all `MYSQL_RES` structures, the return value of this function must be freed with `mysql_free_result`. This function returns a null value in the case of an error.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW COLUMNS FROM table'.

Example

```
MYSQL_RES fields;
fields = mysql_list_fields(&mysql, "people", "address%");
/* 'fields' now contains the names of all fields in the 'people' table
    that start with 'address' */
/* ... */
mysql_free_result( fields );
```

mysql_list_processes

MYSQL_RES *mysql_list_processes(MYSQL *mysql)

Returns a `MYSQL_RES` structure containing the information on all of the threads currently running on the MySQL database server. This information contained here can be used with

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_kill to remove faulty threads. Like all MYSQL_RES structures, the return value of this function must be freed with mysql_free_result. This function returns a null value in the case of an error.

The returned result set contains the information in the following order:

Process ID – The MySQL process ID. This is the ID used with mysql_kill to kill a thread.
Username – The MySQL username of the user executing a thread.
Hostname – The location of the client running a thread.
Database – The current database for the client running a thread.
Action – The type of action last run in a thread. All SQL queries of any type show up as 'Query', so this will be the most common value here.
Time – The amount of time taken (in seconds) to execute the last action in a thread.
State – The state of the current thread. This indicates whether the thread is active (currently executing a command) or idle.
Info – Any extra information about the thread. For SQL queries, this will contain the text of the query.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW PROCESSLIST'

Example

```
MYSQL_RES *threads;
MYSQL_ROW row
threads = mysql_list_processes(&mysql);

row = mysql_fetch_row( threads );
printf("The ID of the first active thread is %d\n", row[0]);
```

mysql_list_tables

MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)

Returns a MYSQL_RES structure containing the names of all existing tables in the current database that match the pattern given by the second argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all tables are listed. Like all MYSQL_RES structures, the return value of this function must be freed with mysql_free_result. This function returns a null value in the case of an error.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW TABLES'

Example

```
MYSQL_RES tables;
tables = mysql_list_tables(&mysql, "p%");
/* 'tables' now contains the names of all tables in the current database
    that start with 'p' */
```

mysql_num_fields

unsigned int mysql_num_fields(MYSQL_RES *result)

Copyright © 2001 O'Reilly & Associates, Inc.

Returns the number of fields contained in each row of the given result set. This is different from the `mysql_field_count` function, in that it operates on an actual result set, which is known to contain data where `mysql_field_count` checks the last executed query (usually to determine if an error occurred).

Example

```
/* 'results' is a MYSQL_RES result set structure */
num_fields = mysql_num_fields(results);
printf("There are %d fields in each row\n", num_fields);
```

mysql_num_rows

int mysql_num_rows(MYSQL_RES *result)

Returns the number of rows of data in the result set. This function is only accurate if the result set was retrieved with `mysql_store_result`. If `mysql_use_result` was used, the value returned by this function will be the number of rows accessed so far.

Example

```
/* 'results' is a MYSQL_RES result set structure */
num_rows = mysql_num_rows(results);
printf("There were %d rows returned, that I know about\n", num_rows);
```

mysql_odbc_escape_string

char *mysql_odbc_escape_string(MYSQL *mysql, char *result_string, unsigned long result_string_length, char *original_string, unsigned long original_string_length, void *parameters, char *(*extend_buffer))

Creates a properly escaped SQL query string from a given string. It is intended for use with ODBC clients, and `mysql_real_escape_string` provides the same functionality with a simpler interface. This function takes the string given in the fourth argument (with the length given in the fifth argument, not including the terminating null character), and escapes it so that the resulting string (which is put into the address given in the second argument, with a maximum length given in the third argument) is safe to use as a MySQL SQL statement. This function a copy of the result string (the second argument). The seventh argument must be a pointer to a function that can be used to allocate memory for result string. The function must take three arguments: A pointer to a set of parameters that control how the memory is allocated (these parameters are passed in as the sixth argument to the original function), a pointer to the result string and a pointer to the maximum length of the result string.

```
char *data = "\000\002\001";
int data_length = 3;
char *result;
int result_length = 5; /* We don't want the final string to be longer than 5.
 extend_buffer() is a function that meets the criteria given above. */
mysql_odbc_escape_string( &mysql, result, result_length, data, data_length,
   NULL, extend_buffer );
/* 'result' now contains the string '\\\000\002\001'
 (that is, a backslash, followed by ASCII 0, then ASCII 2 then ASCII 1. */
```

Copyright © 2001 O'Reilly & Associates, Inc.

mysql_odbc_remove_escape

void mysql_odbc_remove_escape(MYSQL *mysql, char *string)

Removes escape characters from a string. This function is intended for use by internal ODBC drivers, and not for general use. Given a string, this function will remove the escape character ('\') from in front of any escape characters. This will modify (and this shorten) the original string that is passed in.

```
char *escaped = "\\'an escaped quoted string.\\'";
/* escaped contains the string: \' and escaped quoted string.\' */
mysql_odbc_remove_escape(&mysql, escaped);
/* escaped now contains the string: 'an escaped quoted string.' */
```

mysql_options

int mysql_options(MYSQL *mysql, enum mysql_option option, void *value)

Sets a connect option for an upcoming MySQL connection. This function must be called after a MYSQL structure has been initialized using `mysql_init` and before a connection has actually been established using `mysql_real_connect`. The options affect the upcoming connection. This function can be called multiple times to set more than one option. The value of the third argument depends on the type of option. Some options require a character string as an argument while others take a point to an integer, or nothing at all. The options are as follows (the type of the third argument is given in parenthesis after the option name:

MYSQL_INIT_COMMAND (char *)

A SQL query to execute as soon as the connection is established. This query is re-executed if the connection is lost and automatically re-connected.

MYSQL_OPT_COMPRESS (none)

Causes the connection to use a compressed protocol with the server, to increase speed.

MYSQL_OPT_CONNECT_TIMEOUT (unsigned int *)

The number of seconds waited before giving up on connecting to the server.

MYSQL_OPT_NAMED_PIPE (none)

Causes the connection to use named pipes, as opposed to TCP, to connect to a local MySQL server running on Windows NT.

MYSQL_READ_DEFAULT_FILE (char *)

The name of the file to read for default options, in place of the default 'my.cnf'.

MYSQL_READ_DEFAULT_GROUP (char *)

The name of a group within the configuration file to read for the connection options, in place of the default 'client'. See Chapter XX: Configuration for information about the configuration file options.

Example

```
MYSQL mysql;
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
mysql_init( &mysql );
/* Prepare this connection to use the compressed protocol, execute the
   query "SHOW tables" upon connection, and read addition options from the
   'startup' stanze in the file .mysqlrc */
mysql_options(&mysql, MYSQL_OPT_COMPRESS, 0 );
mysql_options(&mysql, MYSQL_INIT_COMMAND, "SHOW tables" );
mysql_options(&mysql, MYSQL_READ_DEFAULT_FILE, ".mysqlrc" );
mysql_options(&mysql, MYSQL_READ_DEFAULT_GROUP, "startup" );
/* Now it is time to call mysql_real_connect() to make the connection using
   these options */
```

mysql_ping

int mysql_ping(MYSQL *mysql)

Checks to see if the connection to the MySQL server is still alive. If it is not, the client will attempt to reconnect automatically. This function returns zero if the connection is alive and nonzero if it cannot successful contact the server.

Example

```
while(mysql_ping(&mysql)) printf("Error, attempting reconnection...\n");
```

mysql_query

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

Executes the SQL query given in the second argument. If the query contains any binary data (particularly the null character), this function cannot be used and `mysql_real_query` should be used instead. The function returns zero if the query was successful and nonzero in the case of an error.

Once a query has been executed using this function, the result set can be retrieved using the `mysql_store_result` or `mysql_use_result` functions.

Example

```
error = mysql_query(&mysql, "SELECT * FROM people WHERE name like 'Bill%'");
if (error) {
     printf("Error with query!\n");
     exit(1);
}
```

mysql_read_query_result

int mysql_read_query_result(MYSQL *mysql)

Processes the result of a query execute with the `mysql_send_query` command. Any data processed this way is not returned. Therefore, this function is only useful when running non-SELECT queries, or for debugging (since the return value still accurately reports if there was an error). The function return zero on success, and a non-zero number if an error occurred.

Example

```
mysql_send_query(&mysql, "SELECT * INTO OUTFILE results.out FROM mytable");
```

```
/* This executes the query, but does not process the results, which is necessary
   in order to write the values into the outfile */
mysql_read_query_result(&mysql);
/* Now the results have been processed and the data written to the outfile */
```

mysql_real_connect

MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user,

const char *passwd, const char *db, uint port, const char *unix_socket,

uint client_flag)

Creates a connection with a MySQL database server. There are eight arguments to this function:

- An initialized `MYSQL` structure, created with `mysql_init`.
- The hostname or IP address of the MySQL database server (use an empty string or `localhost` to connect to the local MySQL server over a Unix socket).
- The username used to connect to the database server (an empty string may be used assuming the Unix login name of the person running the client).
- The password used to authenticate the given user. If an empty string is used, only users with no passwords are checked for authentication.
- The initial database selected when you connect (an empty string may be used to not initially choose a database).
- The port used to remotely connect to a MySQL database server over TCP (0 may be used to accept the default port).
- The filename of the Unix socket used to connect to a MySQL server on the local machine (an empty string may be used to accept the default socket).
- Zero or more of a set of flags used under special circumstances:

  `CLIENT_FOUND_ROWS`
  : When using queries that change tables, returns the number of rows found in the table, not the number of rows affected.

  CLIENT_IGNORE_SPACE
  : Allows spaces after built-in MySQL functions in SQL queries. Traditionally, functions must be followed immediately by their arguments in parenthesis. If this option is used, the function names become reserved words and cannot be used for names of tables, columns or databases.

  CLIENT_INTERACTIVE
  : Causes the server to wait for a longer amount of time (the value of the interactive_timeout server variable) before automatically disconnecting a connection. This is useful for interactive clients where the user may not enter any data for significant periods of time.

  `CLIENT_NO_SCHEMA`
  : Prevent the client from using the full `database.table.column` form to specify a column from any database.

 Copyright © 2001 O'Reilly & Associates, Inc.

CLIENT_COMPRESS
: Use compression when communicating with the server.

CLIENT_ODBC
: Tell the server the client is an ODBC connection.

CLIENT_SSL
: Use SSL encryption to secure the connection. The server must be compiled to support SSL.

Example

```
MYSQL *mysql;
mysql = mysql_init( NULL );
/* Connect to the server on the local host with standard options. */
if (! mysql_real_connect(&mysql, "localhost", "bob", "mypass", "", 0, "", 0))
{ print "Error connecting!\n";
  exit(1);
}

/* or... */
/* Connect to the server at my.server.com using a compressed, secure protocol */
if (! mysql_real_connect(&mysql, "my.server.com", "bob", "mypass",
                          "", 0, "", CLIENT_COMPRESS|CLIENT_SSL)) {
          print "Error connecting!\n";
          exit(1);
}
```

mysql_real_escape_string

unsigned long mysql_real_escape_string(MYSQL *mysql, char *result_string, char *original_string, unsigned long orginal_string_length)

Creates a properly escaped SQL query string from a given string. This function takes the string given in the third argument (with the length given in the fourth argument, not including the terminating null character), and escapes it so that the resulting string (which is put into the address given in the second argument) is safe to use as a MySQL SQL statement. This function returns the new length of the resulting string (not including the terminating null character. To be completely safe, the allocated space for the result string should be at least twice as big as the original string (in case each character has to be escaped) plus one (for the terminating null character).

> This function is safe to use with binary data. The string can contain null characters or any other binary data. This is why it is necessary to include the length of the string. Otherwise, the MySQL library would not be able to determine how long the string was if any null characters were present.

Example

```
# Properly escape a query that contains binary data.
char *data = "\002\001\000";
int original_length = 4 # 3 characters plus one for the null.
char real_data[7]; # Twice as big as the original string (3)
                   # plus one for the null.
int new_length;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
new_length = mysql_real_escape_string(&mysql, data, real_data, original_length);
/* real_query can now be safely used in as a SQL query. */
/* The returned length is '4' since the only character that needed escaping
   was \000 (the null character) */
```

mysql_real_query

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

Executes the SQL query given in the second argument. The length of the query (not including any terminating null character) must be given in the third argument. By supplying the length, you can use binary data, including null characters, in the query. This function is also faster than `mysql_query`. The function returns zero if the query was successful and nonzero in the case of an error.

Once a query has been executed using this function, the result set can be retrieved using the `mysql_store_result` or `mysql_use_result` functions.

Example

```
error = mysql_real_query(&mysql, "SELECT * FROM people WHERE name like 'Bill%'",
        44);
if (error) {
     printf("Error with query!\n");
     exit(1);
}
```

mysql_refresh

int mysql_refresh(MYSQL *mysql, unsigned int options)

Instructs the server to refresh various system operations. What exactly is refreshed depends upon the second argument which is a bitwise combination of any of the following options:

REFRESH_GRANT – Reload the permissions tables (same as mysql_reload() or the 'FLUSH PRIVILEGES' SQL command).
REFRESH_LOG – Flushes the logs files to disk and then closes and re-opens them.
REFRESH_TABLES – Flushes any open tables to disk.
REFRESH_READ_LOCK – Re-instates the read-lock on the database files stored on disk.
REFRESH_HOSTS – Reloads the internal cache MySQL keeps of known hostnames to limit DNS lookup calls.
REFRESH_STATUS – Refreshes the status of the server by checking the status of various internal operations.
REFRESH_THREADS – Clears out any dead or inactive threads from the internal cache of threads.
REFRESH_MASTER – Flushes, closes and reopens the binary log that tracks all changes to MySQL tables. This log is sent to slave server for replication.
REFRESH_SLAVE – Resets the connection with any slave servers that are replicating the master MySQL server.

Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
/* Flush the log files and the table data to disk */
if (!mysql_refresh( &mysql, REFRESH_LOG|REFRESH_TABLES )) {
     printf("Error sending refresh command...\n");
}
```

mysql_reload

int mysql_reload(MYSQL *mysql)

Reloads the permission tables on the MySQL database server. You must have Reload permissions on the current connection to use this function. If the operation is successful, zero is returned otherwise a nonzero value is returned.

> This function is deprecated and will be removed in a future version of the API. The exact same functionality can be obtained by using the SQL query 'FLUSH PRIVILEGES'.

Example

```
/* Make some changes to the grant tables... */
result = mysql_reload(&mysql);
/* The changes now take effect... */
```

mysql_row_seek

MYSQL_ROW_OFFSET mysql_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset)

Moves the pointer for a result set (MYSQL_RES structure) to a specific row. This function requires that the offset be an actual MYSQL_ROW_OFFSET structure, not a simple row number. If you only have a row number, use the `mysql_data_seek` function. A MYSQL_ROW_OFFSET can be obtained from a call to either `mysql_row_tell` or this function (which returns the row the result set was at before it was changed).

This function is only useful if the result set contains all of the data from the query. Therefore it should be used in conjunction with `mysql_store_result` and not used with `mysql_use_result`.

Example

```
/* result is a result set pointer created with mysql_store_result() */
MYSQL_ROW_OFFSET where, other_place;
where = mysql_row_tell( result );
/* Do some more work with the result set... */
/* Go back to where you were before, but remember where you are now: */
other_place = mysql_row_seek( result, where );
/* Do some more work... */
/* Go back to the second marker: */
mysql_row_seek( result, other_place );
```

mysql_row_tell

MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)

Returns the value of the cursor used as `mysql_fetch_row` reads the rows of a result set. The return value of this function can used with `mysql_row_seek` to jump to a specific row in the result set. Values from the function are only useful if the result set was created with `mysql_store_result` (which contains all of the data) and not `mysql_use_result` (which does not).

Example

```
/* results is a result set pointer created with mysql_store_result() */
MYSQL_ROW_OFFSET saved_pos = mysql_row_tell(results);
/* I can now jump back to this row at any time using mysql_row_seek() */
```

mysql_select_db

int mysql_select_db(MYSQL *mysql, const char *db)

Changes the current database. The user must have permission to access the new database. The function returns zero if the operation was successful and nonzero in the case of an error.

Example

```
if ( mysql_select_db(&mysql, "newdb") ) {
     printf("Error changing database, you probably don't have permission.\n");
}
```

mysql_send_query

int mysql_send_query(MYSQL *mysql, char *query, unsigned int query_length)

Executes a single MySQL query, without providing any results to the user. This function is useful for quickly executing a non-SELECT statement that does not return any data to the user. This function is also useful for debugging, since the return value still accurately reports whether an error occurred. The function return zero on success, and a non-zero number if an error occurred.

Example

```
/* Quickly insert a row into a table */
mysql_send_query(&mysql, "INSERT INTO mytable VALUES ('blah', 'fnor')");
```

mysql_shutdown

int mysql_shutdown(MYSQL *mysql)

Shutdown the MySQL database server. The user must have Shutdown privileges on the current connection to use this function. The function returns zero if the operation was successful and nonzero in the case of an error.

 Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
if ( mysql_shutdown(&mysql) ) {
       printf("Server not shut down... Check your permissions...\n");
} else {
       printf("Server successfully shut down!\n");
}
```

mysql_ssl_cipher

char *mysql_ssl_cipher(MYSQL *mysql)

Returns the name of the SSL cipher that is used (or going to be used) with the current connection. This could be RSA, blowfish, or any other cipher supported by the server's SSL library (MySQL uses OpenSSL by default, if SSL is enabled). The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
printf("This connection is using the %s cipher for security.\n",
             Mysql_ssl_cipher(&mysql));
```

mysql_ssl_clear

int mysql_ssl_clear(MYSQL *mysql)

Clears any SSL information associated with the current connection. If called before the connection is made, this will cause the connection to made without SSL, even if SSL options had been selected previously. The function returns zero on success and a non-zero number if an error occurs. It must be called before `mysql_real_connect` to have any effect. The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
/* init a MYSQL structure and set SSL options...*/
/* Changed my mind, I don't want this connection to use SSL: */
mysql_ssl_clear(&mysql);
```

mysql_ssl_set

int mysql_ssl_set(MYSQL *mysql, char *key, char *certificate, char *authority, char *authority_path)

Sets SSL information for the current connection and causes the connection to be made using SSL for encryption. This function must be called before `mysql_real_connect` to have any effect. The arguments are (beyond the pointer to the MYSQL structure) the text of the SSL public key being used for the connection, the filename of the certificate being used, the name of the authority that issued the certificate, and the directory that contains the authority's certificates.

Copyright © 2001 O'Reilly & Associates, Inc.

The function returns zero on success and a non-zero number if an error occurs. It must be called before mysql_real_connect to have any effect. The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
/* 'key' contains an SSL public key.
   'cert' contains the filename of a certificate
   'ca' contains the name of the certificate authority
   'capath' contains the directory containing the certificate
*/
/* Create an initialized MYSQL structure using mysql_init */
mysql_ssl_set(&mysql, key, cert, ca, capath);
/* Now, when mysql_real_connect is called, the connection will use SSL for
   encryption. */
```

mysql_stat

char *mysql_stat(MYSQL *mysql)

Returns information about the current operating status of the database server. This includes the uptime, the number of running threads, and the number of queries being processed, among other information.

Example

```
printf("Server info\n-----------\n%s\n", mysql_stat(&mysql));
/* Output may look like this:
Server info
-----------
Uptime: 259044  Threads: 1  Questions: 24  Slow queries: 0  Opens: 6
Flush tables: 1 Open tables: 0 Queries per second avg: 0.000
^^^^^ This is all on one line
*/
```

mysql_store_result

MYSQL_RES *mysql_store_result(MYSQL *mysql)

Reads the entire result of a query and stores in a MYSQL_RES structure. Either this function or mysql_use_result must be called to access return information from a query. You must call mysql_free_result to free the MYSQL_RES structure when you are done with it.

The function returns a null value in the case of an error. The function will also return a null value if the query was not of a type that returns data (such as an INSERT or UPDATE query). If receive a null pointer and are not sure if the query was supposed to return data or not, you can call mysql_field_count to find out the number of fields the query was supposed to return. If zero, then it was a non-SELECT statement and the pointer is supposed to be null. Otherwise, an error has occurred.

If the query was a SELECT-type statement, but happens to contain no data, this function will still return a valid (but empty) MYSQL_RES structure (it will not be a null-pointer).

 Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
MYSQL_RES results;
mysql_query(&mysql, "SELECT * FROM people");
results = mysql_store_result(&mysql);
/* 'results' should now contain all of the information from the 'people' table */
if (!results) { printf("An error has occurred!\n"); }

/* 'query' is some query string we obtained elsewhere,
    we're not sure what it is... */
mysql_query(&mysql, query);
results = mysql_store_result(&mysql);
if (!results) { /* An error might have occurred,
                   or maybe this is just a non-SELECT statement */
     if (! mysql_field_count(&mysql) ) { /* Aha! This is zero so it was
                                            a non-SELECT statement */
               printf("No error here, just a non-SELECT statement...\n");
     } else {
               printf("An error has occurred!\n");
     }
}
```

mysql_thread_id

unsigned long mysql_thread_id(MYSQL * mysql)

Returns the thread ID of the current connection. This value can be used with *mysql_kill* to terminate the thread in case of an error. This value will be different if you disconnect from the server and then reconnect (which may happen automatically, without warning, if you use `mysql_ping`). Therefore, you should call this function immediately before you are going to use the value.

Example

```
thread_id = mysql_thread_id(&mysql);
/* This number can be used with mysql_kill() to terminate the current thread. */
```

mysql_thread_safe

unsigned int mysql_thread_safe(void)

Indicates whether the MySQL client library is safe to use in a threaded invironment. The function returns a true value if the library is thread safe, and zero (false) if it is not.

Example

```
if (mysql_thread_safe()) {
     printf("This library is thread safe... thread away!\n");
} else {
     printf("This library is *not* thread safe, be careful!\n");
}
```

mysql_use_result

MYSQL_RES *mysql_use_result(MYSQL *mysql)

Reads the result of a query row by row and allows access to the data through a `MYSQL_RES` structure. Either this function or `mysql_store_result` must be called to access

return information from a query. Because this function does not read the entire data set at once, it is faster and more memory efficient than `mysql_store_result`. However, when using this function you must read all of the rows of the dataset from the server or else the next query will receive the left over data. Also, you cannot run any other queries until you are done with the data in this query. Even worse, no other threads running on the server can access the tables used by the query until it is finished. For this reason, you should only use this function when you are certain you can read the data in a timely manner and release it. You must call `mysql_free_result` to free the `MYSQL_RES` structure when you are done with it.

The function returns a null value in the case of an error. The function will also return a null value if the query was not of a type that returns data (such as an INSERT or UPDATE query). If receive a null pointer and are not sure if the query was supposed to return data or not, you can call `mysql_field_count` to find out the number of fields the query was supposed to return. If zero, then it was a non-SELECT statement and the pointer is supposed to be null. Otherwise, an error has occurred.

If the query was a SELECT-type statement, but happens to contain no data, this function will still return a valid (but empty) MYSQL_RES structure (it will not be a null-pointer).

Example

```
MYSQL_RES results;
mysql_query(&mysql, "SELECT * FROM people");
results = mysql_store_result(&mysql);
/* 'results' will now allow access (using mysql_fetch_row) to the table
   data, one row at a time */
```


Copyright © 2001 O'Reilly & Associates, Inc.

25

The Python DB-API

DB-API is a single API for supporting database-independent database access. The MySQL implementation of this API, MySQLdb, can be downloaded from http://dustman.net/andy/python/MySQLdb. It comes with a Redhat RPM Linux installer, a Win32 installer, and a Python script for other platforms. For the "other platforms":

1. Uncompress the .tar.gz file that contains MySQLdb using the commands *gunzip FILENAME.tar.gz* and *tar xf FILENAME.tar*.
2. Change directories into the newly generated MySQLdb directory.
3. Issue the command: *python setup.py install*

The MySQLdb module contains both the standard DB-API methods and attributes as well as several proprietary methods and attributes. Proprietary APIs are marked with asterisks.

Module: MySQLdb

The entry point into the MySQL module is via the `MySQLdb.connect()` method. The return value from this method represents a connection to a MySQL database that you can use for all of your MySQL operations.

Module Attributes

Attribute: apilevel

Synopsis

A string constant storing the version of the DB-API that MySQLdb supports.

Copyright © 2001 O'Reilly & Associates, Inc.

Attribute: paramstyle

Synopsis

Defines the type of parameter placeholder in parameterized queries. DB-API supports many valid values for this attribute, but MySQLdb actually supports only "format" and "pyformat". This attribute is largely meaningless to MySQL developers.

*Attribute: quote_conv**

Synopsis

Maps Python types to MySQL literals via a dictionary mapping.

Attribute: threadsafety

Synopsis

Specifies the level of thread-safety supported by MySQLdb. Possible values are:

0 - Threads may no share the module

1 - Threads may share the module, but not the connections

2 - Threads may share the module and connections

3 - Threads may share the module, connections, and cursors

*Attribute: type_conv**

Synopsis

Maps MySQL types from strings to the desired mapping type using. This value is initialized with the following values:

```
{ FIELD_TYPE.TINY : int,
FIELD_TYPE.SHORT: int,
FIELD_TYPE.LONG: long,
FIELD_TYPE.FLOAT: float,
FIELD_TYPE.DOUBLE: float,
FIELD_TYPE.LONGLONG: long,
FIELD_TYPE.INT24: int,
FIELD_TYPE.YEAR: int }
```

Copyright © 2001 O'Reilly & Associates, Inc.

Module Methods

Method: MySQL.connect( )

Signature

```
connection = MySQL.connect(params)
```

Synopsis

Connects to the MySQL database engine represented by the various connection keyword/value parameters. These parameters include:

host
: The name of the server on which the MySQL database is running

user
: The user ID to use for connecting to MySQL. This user should be allowed by MySQL to make the connection.

passwd
: The password to authenticate the user ID for the connection.

db
: The MySQL database to which the application is attempting to connect.

port
: Directs MySQLdb to connect to a MySQL installation on a custom part. When left unspecified, the method will use the default MySQL port of 3306.

unix_socket
: Identifies the location of a socket or named pipe to use if the value of the host allows it.

client_flags
: An integer specifying the client connection flags to use. These client connection flags are the same ones enumerated in Chapter 22, *C API* for the `mysql_real_connect()` method.

This method returns a Python object representing a connection to a MySQL database.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
```

Copyright © 2001 O'Reilly & Associates, Inc.

Connection Attributes

Attribute: db*

Synopsis

A window into the MySQL C API. MySQLdb uses this attribute to make calls to the underlying C API.

Connection Methods

Method: close( )

Signature

```
close()
```

Synopsis

Closes the current connection to the database and releases any associated resources.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
connection.close();
```

Method: commit( )

Signature

```
commit()
```

Synopsis

Commits the current transaction by sending a COMMIT to MySQL.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
connection._transactional = 1;
cursor = connection.cursor();
cursor.execute("UPDATE TNAME SET COL = 1 WHERE PK = 2045");
cursor.execute("UPDATE TNAME SET COL = 1 WHERE PK = 3200");
connection.commit();
connection.close();
```

Copyright © 2001 O'Reilly & Associates, Inc.

Method: cursor()

Signature

```
cursor = cursor()
```

Synopsis

Creates a cursor associated with this connection. Transactions involving any statements executed by the newly created cursor are governed by this connection.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute("UPDATE TNAME SET COL = 1 WHERE PK = 2045");
connection.close();
```

Method: rollback()

Signature

```
rollback()
```

Synopsis

Rollsback any uncommitted statements. This only works if MySQL is setup for transactional processing in this context.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                              passwd='test', db='test');
connection._transactional = 1;
cursor = connection.cursor();
cursor.execute("UPDATE TNAME SET COL = 1 WHERE PK = 2045");
try:
    cursor.execute("UPDATE TNAME SET COL = 1 WHERE PK = 3200");
    connection.commit();
except:
    connection.rollback();
connection.close();
```

Cursor Attributes

Attribute: arraysize

Synopsis

Specifies the number of rows to fetch at a time with the `fetchmany()` method call. By default, this value is set to 1. In other words, `fetchmany()` fetches one row at a time by default.

Copyright © 2001 O'Reilly & Associates, Inc.

Attribute: description

Synopsis

Describes a result column as a read-only sequence of seven-item sequences. Each sequence contains the following values: `name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale`, `null_ok`.

Attribute: rowcount

Synopsis

Provides the number of rows returned through the last `executeXXX()` call. This attribute is naturally read-only and has a value of -1 when no `executeXXX()` call has been made or the last operation does not provide a row count.

Cursor Methods

Method: callproc()

Signature

```
callproc(procname [,parameters])
```

Synopsis

This method is not supported by MySQL.

Method: close()

Signature

```
close()
```

Synopsis

Closes the cursor explicitly. Once closed, a cursor will throw an `ProgrammingError` will be thrown if any operation is attempted on the cursor.

Example

```
cursor = connection.cursor();
cursor.close();
```

Method: execute()

Signature

```
cursor = execute(sql [,parameters])
```

Copyright © 2001 O'Reilly & Associates, Inc.

Synopsis

Sends arbitrary SQL to MySQL for execution. If the SQL specified is parameterized, then the optional second argument is a sequence or mapping containing parameter values for the SQL. Any results or other information generated by the SQL can then be accessed through the cursor.

The parameters of this method may also be lists of tuples to enable you to perform multiple operations at once. This usage is considered depricated as of the DB-API 2.0 specification. You should instead use the `executemany()` method.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute('SELECT * FROM TNAME');
```

Method: executemany()

Signature

```
cursor.executemany(sql,parameters)
```

Synopsis

Prepares a SQL statement and sends it to MySQL for execution against all parameter sequences or mappings in the `parameters` sequence.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.executemany("INSERT INTO COLOR ( COLOR, ABBREV ) VALUES (%s, %s )",
                   (("BLUE", "BL"), ("PURPLE", "PPL"), ("ORANGE", "ORN")));
```

Method: fetchall()

Signature

```
rows = cursor.fetchmany()
```

Synopsis

Fetches all remaining rows of a query result as a sequence of sequences.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute("SELECT * FROM TNAME");
for row in cursor.fetchall():
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
        # process row
```

Method: fetchmany()

Signature

```
rows = cursor.fetchmany([size])
```

Synopsis

Fetches the next set of rows of a result set as a sequence of sequences. If no more rows are available, this method returns an empty sequence.

If specified, the `size` parameter dictates how many rows should be fetched. The default value for this parameter is the cursor's `arraysize` value. If the `size` parameter is larger than the number of rows left, then the resulting sequence will contain all remaining rows.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute("SELECT * FROM TNAME");
rows = cursor.fetchmany(5);
```

Method: fetchone()

Signature

```
row = cursor.fetchone()
```

Synopsis

Fetches the next row of a result set returned by a query as a single sequence. This method will return `None` when no more results exist. It will throw an error should the SQL executed not be a query.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute("SELECT * FROM TNAME");
row = cursor.fetchone();
print "Key: ", row[0];
print "Value: ", row[1];
```

Method: insert_id()*

Signature

```
id = cursor.insert_id()
```

Copyright © 2001 O'Reilly & Associates, Inc.

Synopsis

Returns the last inserted ID from the most recent `INSERT` on an `AUTO_INCREMENT` field.

Example

```
connection = MySQLdb.connect(host='carthage', user='test',
                             passwd='test', db='test');
cursor = connection.cursor();
cursor.execute("INSERT INTO TNAME (COL) VALUES (1)");
id = cursor.insert_id();
```

Method: nextset()

Signature

```
cursor.nextset()
```

Synopsis

This method always returns `None` for MySQL.

Method: setinputsizes()

Signature

```
cursor.setinputsizes(sizes)
```

Synopsis

This method does nothing in MySQL.

Method: setoutputsize()

Signature

```
cursor.setoutputsize(size [,column])
```

Synopsis

This method does nothing in MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

26

Perl Reference

Installation

To use the interfaces to DataBase Interface/DataBase Driver (DBI/DBD) you must have the following:

Perl 5
: You must have a working copy of Perl 5 on your system. At the time of this writing, the newest release of Perl was 5.6.1. You should have at least Perl 5.004 since earlier versions of Perl contained security related bugs. For more information about Perl, including download sites, see *http://www.perl.com.*

DBI
: The DataBase Independent portion of the DBI/DBD module can be downloaded from the Comprehensive Perl Archive Network (CPAN). At the time of this writing, the most recent version is DBI-1.15. You can find it at *http://www.perl.com/CPAN/modules/by-authors/id/Tim_Bunce/*.

Data::ShowTable
: Data::ShowTable is a module that simplifies the act of displaying large amounts of data. The Msql-Mysql modules require this. The most recent version is Data-ShowTable-3.3 and it can be found at *http://www.perl.com/CPAN/authors/id/AKSTE/Data-ShowTable-3.3.tar.gz*.

MySQL
: Chapter 3, *Installation*, contains information about how to obtain and install the MySQL database servers.

C compiler and related tools
: The DBD::mysql module requires an ANSI compliant C compiler as well some common related tools (such as *make*, *ld*, etc.). The tools that built the copy of Perl you are using should be sufficient. If you have no such tools, the GNU C compiler

Copyright © 2001 O'Reilly & Associates, Inc.

(along with all necessary supporting programs) is available free at *ftp://ftp.gnu.org/pub/gnu/*.

The current maintainer of the Msql-Mysql modules is Jochen Wiedmann, who has the CPAN author ID of JWIED. Therefore, the current release of the Msql-Mysql modules can always be found at *http://www.perl.com/authors/id/JWIED*. At the time of this writing, the current version *is Msql-Mysql-modules-1.2216..tar.gz.*

> At the time of this writing Jochen Wiedmann, the maintainer of the DBD::mysql module, was preparing to separate DBD::mysql from the rest of the Msql-Mysql-modules package. Development of DBD::mysql will continue while the rest of Msql-Mysql-modules will be discontinued.
>
> Therefore, if you are installing DBD::mysql from source, check the release notes of the DBD-mysql package to see if it's stable before downloading Msql-Mysql-modules. If DBD-mysql is stable, use it instead of Msql-Mysql-modules.

After you have downloaded the package, uncompress and untar it into a directory.

```
tar xvzf Msql-Mysql-modules-1.2216.tar.gz
cd Msql-Mysql-modules-1.2216
```

Inside the distribution directory is the file *INSTALL*, which gives several installation hints. The first step is to execute the *Makefile.PL* file:

```
perl Makefile.PL
```

This command starts by asking whether you want to install the modules for mSQL, MySQL or both. Choose MySQL.

After some system checking, the program then asks for the location of MySQL. This is the directory that contains the appropriate *lib* and *include* subdirectories. By default it is */usr/local*. This is the correct location for most installations, but you should double check in case it is located elsewhere. It is common on many systems for the MySQL headers and libraries to live in */usr/local/mysql*, separate from the system headers and libraries.

At this point, the installation script creates the appropriate makefiles and exits. The next step is to run *make* to compile the files.

```
make
```

If your Perl and MySQL are all installed correctly, the make should run without errors. When it is finished, all of the modules have been created and all that is left is to test and install them.

```
make test
```

While this is running, a series of test names will scroll down your screen. All of them should end with '. . . ok'. Finally, you need to install the modules.

```
make install
```

 Copyright © 2001 O'Reilly & Associates, Inc.

You need to have permission to write to the Perl installation directory to install the modules. In addition, you need to have permission to write to your system binary directory (usually */usr/local/bin* or */usr/bin*) to install the supporting programs that come with the module (older versions of Msql-Mysql modules include two similar command line interfaced called *pmysql* and *dbimon;* newer versions just contain *dbimon*; it is currently unclear whether the new DBI-mysql package will include either).

DBI.pm API

The DBI API is the standard database API in Perl.

use

use DBI;

This must be declared in every Perl program that uses the DBI module. By default, DBI does not import anything into the local namespace. All interaction must be done through objects or static calls to the DBI package itself. DBI does, however, provide one import tag, ':sql_types'. This tag imports the names of standard SQL types. These are useful in methods such as 'bind_param' which may need to know the SQL type of a column. These names are imported as methods, which means they can be used without punctuation.

Examples

```
use DBI;  # Load the DBI into a program

use DBI qw(:sql_types); # Load the DBI into the program, importing the names
          # of the standard SQL types. They can now be used in the program.
if ($type == SQL_CHAR) { print "This is a character type..."; }
```

DBI::available_drivers

@available_drivers = DBI->available_drivers;

@available_drivers = DBI->available_drivers($quiet);

`DBI::available_drivers` returns a list of the available DBD drivers. The function does this by searching the Perl distribution for DBD modules. Unless a true value is passed as the argument, the function will print a warning if two DBD modules of the same name are found in the distribution. In the current Msql-Mysql modules distribution the driver for MySQL is named 'mysql'.

Example

```
use DBI;

my @drivers = DBI->available_drivers;
print "All of these drivers are available:\n" . join("\n",@drivers) .
                          "\nBut we're only interested in mysql. :)\n";
```

DBI::bind_col

$result = $statement_handle->bind_col($col_num, \$col_variable);

Copyright © 2001 O'Reilly & Associates, Inc.

DBI::bind_col binds a column of a SELECT statement with a Perl variable. Every time that column is accessed or modified, the value of the corresponding variable changes to match. The first argument is the number of the column in the statement, where the first column is number 1. The second argument is a reference to the Perl variable to bind to the column. The function returns an undefined value undef if the binding fails for some reason.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);

my ($name, $date);
$myothertable_output->bind_col(1,\$name,undef);
$myothertable_output->bind_col(2,\$date,undef);
# $name and $date are now bound to their corresponding fields in the outout.

$myothertable_output->execute;
while ($myothertable_output->fetch) {
       # $name and $date are automatically changed each time.
       print "Name: $name Date: $date\n";
}
```

DBI::bind_columns

$result = $statement_handle->bind_columns(@list_of_refs_to_vars);

DBI::bind_columns binds an entire list of scalar references to the corresponding field values in the output. Each argument must be a reference to a scalar. Optionally, the scalars can be grouped into a \($var1, $var2) structure which has the same effect. There must be exactly as many scalar references as there are fields in the output or the program will die.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);

my ($name, $date);
$myothertable_output->bind_columns(\($name, $date));
# $name and $date are now bound to their corresponding fields in the outout.

$myothertable_output->execute;
while ($myothertable_output->fetch) {
       # $name and $date are automatically changed each time.
                        print "Name: $name Date: $date\n";
}
```

DBI::bind_param

$result = $statement_handle->bind_param($param_number, $bind_value);

$result = $statement_handle->bind_param($param_number, $bind_value, $bind_ type);

 Copyright © 2001 O'Reilly & Associates, Inc.

```
$result = $statement_handle->bind_param($param_number, $bind_value, \ %bind_type);
```

`DBI::bind_param` substitutes real values for the '?' placeholders in statements (see `DBI::prepare`). The first argument is the number of the placeholder in the statement. The first placeholder (from left to right) is 1. The second argument is the value with which to replace the placeholder. An optional third parameter can be supplied which determines the type of the value to be substituted. This can be supplied as a scalar or as a reference to a hash of the form `{ TYPE => &DBI::SQL_TYPE }` where '`SQL_TYPE`' is the type of the parameter. It is not documented how the DBI standard SQL types correspond to the actual types used by DBD::mysql. However, Table 21-1 contains a list of the corresponding types as of the time of this writing. The function returns `undef` if the substitution is unsuccessful.

MySQL SQL Type	**DBI**	**DBD::mysql**
CHAR	SQL_CHAR	FIELD_TYPE_CHAR
		FIELD_TYPE_STRING
DECIMAL	SQL_NUMERIC	FIELD_TYPE_DECIMAL
	SQL_DECIMAL	
INTEGER	SQL_INTEGER	FIELD_TYPE_LONG
INTEGER UNSIGNED		
INT		
INT UNSIGNED		
MIDDLEINT	SQL_INTEGER	FIELD_TYPE_INT24
MIDDLEINT UNSIGNED		
SMALLINT	SQL_SMALLINT	FIELD_TYPE_SHORT
SMALLINT UNSIGNED		
YEAR	SQL_SMALLINT	FIELD_TYPE_YEAR
FLOAT	SQL_FLOAT	FIELD_TYPE_FLOAT
	SQL_REAL	
DOUBLE	SQL_DOUBLE	FIELD_TYPE_DOUBLE
VARCHAR	SQL_VARCHAR	FIELD_TYPE_VAR_STRING
ENUM	SQL_VARCHAR	FIELD_TYPE_ENUM
SET	SQL_VARCHAR	FIELD_TYPE_SET
TIME	SQL_TIME	FIELD_TYPE_TIME
DATE	SQL_DATE	FIELD_TYPE_DATE
		FIELD_TYPE_NEWDATE
TIMESTAMP	SQL_TIMESTAMP	FIELD_TYPE_TIMESTAMP
DATETIME	SQL_TIMESTAMP	FIELD_TYPE_DATETIME
BLOB	SQL_LONGVARCHAR	FIELD_TYPE_BLOB

Copyright © 2001 O'Reilly & Associates, Inc.

TEXT		
TINYBLOB	SQL_LONGVARCHAR	FIELD_TYPE_TINY_BLOB
MEDIUMBLOB MEDIUMTEXT	SQL_LONGVARCHAR	FIELD_TYPE_MEDIUM_BLO
LONGBLOB	SQL_LONGVARCHAR	FIELD_TYPE_LONG_BLOB
BIGINT BIGINT UNSIGNED	SQL_BIGINT	FIELD_TYPE_LONGLONG
TINYINT TINYINT UNSIGNED	SQL_TINYINT	FIELD_TYPE_TINY
(currently unsupported)	SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR SQL_BIT	(currently unsupported)

Example

```
use DBI qw(:sql_types);
my $db = DBI->connect('DBD:mysql:mydata','me','mypass');
my $statement = $db->prepare(
"SELECT name, date FROM myothertable WHERE name like ? OR name like ?");

$statement->bind_param(1,'J%','SQL_CHAR');
$statement->bind_param(2,'%oe%', { TYPE => &DBI::SQL_CHAR });
# The statement will now be:
# SELECT name, date FROM myothertable WHERE name like 'J%' or name like '%oe%'

# Binding parameters also performs quoting for you!
$name1 = "%Joe's%";
$name2 = "%quote's%";
$statement->bind_param(1, $name1, 'SQL_CHAR');
$statement->bind_param(1, $name2, { TYPE => SQL_CHAR }); # I don't need the
                                  # &DBI:: before 'SQL_CHAR' because I used the
                                  # ':sql_types' tag in the use DBI line to import
                                  # the SQL types into my namespace.
# The select statement will now be:
# SELECT name, date FROM myothertable
#     WHERE name like '%Joe''s%' or name like '%quote''s%'

# Once a statement is prepared, it can be re-run with new bindings multiple times.
my $query = "INSERT INTO myothertable (name, date) VALUES (?, ?)";
$statement = $db->prepare($query);
# Let's say %dates is a hash with names as the keys and dates as the values:
foreach my $name (keys %dates) {
    my $date = $dates{$name};
    $statement->bind_param(1, $name, { TYPE => SQL_CHAR });
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
    $statement->bind_param(2, $date, { TYPE => SQL_CHAR });
    $statement->execute;
}
```

DBI::bind_param_inout

Unimplemented

DBI::bind_param_inout is used by certain DBD drivers to support stored procedures. Since MySQL currently does not have a stored procedures mechanism, this method does not work with DBD::mysql.

DBI::commit

$result = $db->commit;

`DBI::commit` instructs MySQL to irrevocably commit everything that has been done during this session since the last commit. It is only effective on tables that support transactions (such as Berkeley DB tables). If the DBI attribute AutoCommit is set to a true value, an implicit commit is performed with every action, and this method does nothing.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');
$db->{AutoCommit} = undef; # Turn off AutoCommit...

# Do some stuff...
if (not $error) { $db->commit; } # Commit the changes...
```

DBI::connect

$db = DBI->connect($data_source, $username, $password);

$db = DBI->connect($data_source, $username, $password, \%attributes);

`DBI::connect` requires at least three arguments, with an optional fourth, and returns a handle to the requested database. It is through this handle that you perform all of the transactions with the database server. The first argument is a data source. A list of available data sources can be obtained using `DBI::data_sources`. For MySQL the format of the data source is `'dbi:mysql:$database:$hostname:$port'`. You may leave the ':$port' extension off to connect to the standard port. Also, you may leave the ':$hostname:$port' extension off to connect to a server on the local host using a Unix-style socket. A database name must be supplied.

The second and third arguments are the username and password of the user connecting to the database. If they are '`undef`' the user running the program must have permission to access the requested databases.

The final argument is optional and is a reference to an associative array. Using this hash you may preset certain attributes for the connection. DBI currently defines a set of four attributes which may be set with any driver: PrintError, RaiseError, AutoCommit and

Copyright © 2001 O'Reilly & Associates, Inc.

dbi_connect_method. The first three can be set to 0 for off and some true value for on. The defaults for PrintError and AutoCommit are on and the default for RaiseError is off. The dbi_connect_method attribute defines the method used to connect to the database. It is usually either 'connect' or 'connect_cached', but can be set to special values in certain circumstances.

In addition to the above attributes, DBD::mysql defines a set of attributes which affect the communication between the application and the MySQL server:

mysql_client_found_rows (default: 0 (false))
: Generally, in MySQL, update queries that will not really change data (such as UPDATE table SET col = 1 WHERE col = 1) are optimized away and return '0 rows affected', even if there are rows that match the criteria. If this attribute is set to a true value (and MySQL is compiled to support it), the actual number of matching rows will be returned as affected for this types of queries.

mysql_compression (default: 0 (false))
: If this attribute is set to a true value, the communication between your application and the MySQL server will be compressed. This only works with MySQL version 2.22.5 or higher.

mysql_connect_timeout (default: undefined)
: If this attribute is set to a valid integer, the driver will wait only that many seconds before giving up when attempting to connect to the MySQL server. If this value is undefined, the driver will wait forever (or until the underlying connect mechanism times out) to get a response from the server.

mysql_read_default_file (default: undefined)
: Setting this attribute to a valid file name causes the driver to read that file as a MySQL configuration file. This can be used for setting things like usernames and passwords for multiple applications.

mysql_read_default_group (default: undefined)
: If 'mysql_read_default_file' has been set, this option causes the driver to use a specific stanza of options within the configuration file. This can be useful if the configuration file contains options for both the MySQL server and client applications. In general, a DBI-based Perl application should only need the client options. If no 'mysql_read_default_file' is set, the driver will look at the standard MySQL configuration files for the given stanza.

As an alternate form of syntax, all of the above attributes can also be included within the data source parameter like this: 'dbi:mysql:database;attribute=value;attribute=value'.

If the connection fails, an undefined value `undef` is returned and the error is placed in `$DBI::errstr`.

Environment Variables

When the connect method is evoked, DBI checks for the existence of several environment variables. These environment variables can be used instead of their corresponding parameters, allowing certain database options to be set on a per-user basis.

Copyright © 2001 O'Reilly & Associates, Inc.

DBI_DSN: The value of this environment variable will be used in place of the entire first parameter, if that parameter is undefined or empty.

DBI_DRIVER: The value of this environment variable will be used for the name of the DBD driver if there is no speficied driver in the first parameter (that is, if the parameter looks like 'dbi::').

DBI_AUTOPROXY: If this enviroment variable is set, DBI will use the DBD::Proxy module to create a proxy connection to the database.

DBI_USER: The value of this environment variable is used in place of the 'username' parameter if that parameter is empty or undefined.

DBI_PASS: This value of this environment variable is used in place of the 'password' parameter if that parameter is empty or undefined.

Example

```
use DBI;

my $db1 = DBI->connect('dbi:mysql:mydata',undef,undef);
# $db1 is now connected to the local MySQL server using the database 'mydata'.

my $db2 = DBI->connect('dbi:mysql:mydata:host=myserver.com','me','mypassword');
# $db2 is now connected to the MySQL server on the default port of
# 'myserver.com' using the database 'mydata'. The connection was made with
# the username 'me' and the password 'mypassword'.

my $db3 = DBI->connect('dbi:mysql:mydata',undef,undef, {
                          RaiseError => 1
});
# $db3 is now connected the same way as $db1 except the 'RaiseError'
# attribute has been set to true.

my $db4 = DBI-
>connect('dbi:mysql:mydata;host=myserver.com;port=3333;mysql_read_default_file=/hom
e/me/.my.cnf;mysql_real_default_group=perl_clients', undef, undef, { AutoCommit =>
0 });
# $db4 is now connected to the database 'mydata' on 'myserver.com' at port 3333.
# In addition, the file '/home/me/.my.cnf' is used as a MySQL configuration file
# (which could contain the username and password used to connect). Also, the
# 'AutoCommit' flag is set to '0', requiring any changes to the data be explicitly
# committed.
```

DBI::connect_cached

$db = DBI->connect_cached($data_source, $username, $password);

$db = DBI->connect_cached($data_source, $username, $password, \%attributes);

`DBI::connect_cached` creates a connection to a database server, storing it for future use in a persistant (for the life of the Perl process) hash table. This method takes the same arguments as `DBI::connect`. The difference is that `DBI::connect_cached` saves the connections once they are opened. Then if any other calls to `DBI::connect_cached` use the same parameters, the already opened database

connection is used (if valid). Before handing out any previously created connection, the driver checks to make sure the connection to the database is still active and usable.

See the attribute CachedKids (below) for information on how to manually inspect and clear the saved connection hash table.

Examples

```
use DBI;

my $db1 = DBI->connect_cached('dbi:mysql:mydata',undef,undef);
# $db1 is now connected to the local MySQL server using the database 'mydata'.

my $db2 = DBI->connect_cached('dbi:mysql:myotherdata',undef,undef);
# $db2 is a separate connection to the local MySQL server using the database\
# 'myotherdata'.

my $db3 = DBI->connect_cached('dbi:mysql:mydata', undef, undef);
# $db3 is the exact same connection as $db1 (if it is still a valid connection).
```

DBI::data_sources

@data_sources = DBI->data_sources($dbd_driver);

`DBI::data_sources` takes the name of a DBD module as its argument and returns all of the available databases for that driver in a format suitable for use as a data source in the `DBI::connect` function. The program will die with an error message if an invalid DBD driver name is supplied. In the current Msql-Mysql modules distribution, the driver for MySQL is named 'mysql'.

> Environment variables:
>
> If the name of the drive is empty or undefined, DBI will look at the value of the environment variable DBI_DRIVER.

Example

```
use DBI;

my @mysql_data_sources = DBI->data_sources('mysql');
# DBD::mysql had better be installed or the program will die.

print "MySQL databases:\n" . join("\n",@mysql_data_sources) . "\n\n";
```

DBI::do

$rows_affected = $db->do($statement);

$rows_affected = $db->do($statement, \%unused);

$rows_affected = $db->do($statement, \%unused, @bind_values);

`DBI::do` directly performs a non-`SELECT` SQL statement and returns the number of rows affected by the statement. This is faster than a `DBI::prepare/DBI::execute` pair which requires two function calls. The first argument is the SQL statement itself. The

 Copyright © 2001 O'Reilly & Associates, Inc.

second argument is unused in DBD::mysql, but can hold a reference to a hash of attributes for other DBD modules. The final argument is an array of values used to replace ‘placeholders,’ which are indicated with a ‘?’ in the statement. The values of the array are substituted for the placeholders from left to right. As an additional bonus, `DBI::do` will automatically quote string values before substitution.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $rows_affected = $db->do("UPDATE mytable SET name='Joe' WHERE name='Bob'");
print "$rows_affected Joe's were changed to Bob's\n";

my $rows_affected2 = $db->do("INSERT INTO mytable (name) VALUES (?)",
                          {}, ("Sheldon's Cycle"));
# After quoting and substitution, the statement:
# INSERT INTO mytable (name) VALUES ('Sheldon's Cycle')
# was sent to the database server.
```

DBI::disconnect

$result = $db->disconnect;

`DBI::disconnect` disconnects the database handle from the database server. With MySQL tables that do not support transactions, this is largely unnecessary because an unexpected disconnect will do no harm. However, when using MySQL’s transaction support (such as with Berkeley DB tables), database connections need to be explicitly disconnected. To be safe (and portable) you should always call disconnect before exiting the program. If there is an error while attempting to disconnect, a nonzero value will be returned and the error will be set in `$DBI::errstr`.

Example

```
use DBI;
my $db1 = DBI->connect('DBI:mysql:mydata',undef,undef);
my $db2 = DBI->connect('DBI:mysql:mydata2',undef,undef);
...
$db1->disconnect;
# The connection to 'mydata' is now severed. The connection to 'mydata2'
# is still alive.
```

DBI::dump_results

$neat_rows = $statement_handle->dump_results();

$neat_rows = $statement_handle->dump_results($maxlen);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep, $field_sep);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep, $field_sep,

$file_handle);

Copyright © 2001 O’Reilly & Associates, Inc.

`DBI::dump_results` prints the contents of a statement handle in a neat and orderly fashion by calling `DBI::neat_string` on each row of data. This is useful for quickly checking the results of queries while you write your code. All of the parameters are optional. If the first argument is present, it is used as the maximum length of each field in the table. The default is 35. A second argument is the string used to separate each line of data. The default is `\n`. The third argument is the string used to join the fields in a row. The default is a comma. The final argument is a reference to a filehandle glob. The results are printed to this filehandle. The default is `STDOUT`. If the statement handle cannot be read, an undefined value `undef` is returned.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

print $myothertable_output->dump_results();
# Print the output in a neat table.

open(MYOTHERTABLE,">>myothertable");
print $myothertable_output->dump_results(undef, undef, undef, \*MYOTHERTABLE);
# Print the output again into the file 'myothertable'.
```

DBI::execute

$rows_affected = $statement_handle->execute;

$rows_affected = $statement_handle->execute(@bind_values);

`DBI::execute` executes the SQL statement held in the statement handle. After preparing a query with DBI::prepare, this method must be called to actually run the query. For a non- `SELECT` query, the function returns the number of rows affected. The function returns '- 1' if the number of rows is not known. For a `SELECT` query, some true value is returned upon success. If arguments are provided, they are used to fill in any placeholders in the statement (see `DBI::prepare`).

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $statement_handle = $db->prepare("SELECT * FROM mytable");
my $statement_handle2 = $db->prepare("SELECT name, date FROM myothertable
   WHERE name like ?");

$statement_handle->execute;
# The first statement has now been performed. The values can now be accessed
# through the statement handle.

$statement_handle->execute("J%");
# The second statement has now been executed as the following:
# SELECT name, date FROM myothertable WHERE name like 'J%'
```

DBI::fetchall_arrayref

$ref_of_array_of_arrays = $statement_handle->fetchall_arrayref;

 Copyright © 2001 O'Reilly & Associates, Inc.

$ref_of_array_of_arrays = $statement_handle->fetchall_arrayref($ref_of_array);

$ref_of_array_of_hashes = $statement_handle->fetchall_arrayref($ref_of_hash);

`DBI::fetchall_arrayref` returns all of the remaining data in the statement handle as a reference to an array. Each row of the array is a reference to another array that contains the data in that row. The function returns an undefined value `undef` if there is no data in the statement handle. If any previous `DBI::fetchrow_*` functions were called on this statement handle, `DBI::fetchall_arrayref` returns all of the data after the last `DBI::fetchrow_*` call.

If a reference to an array is passed as a parameter, the referenced array is used to determine which columns are returned. If the referenced array is empty, the method behaves normally, otherwise, the values of the referenced array are taken as the column indices to put in the returned arrays. The index of the first column is '0'. Negative numbers can be used to choose columns starting from the last column, backwards ('-1' is the last column).

If a reference to a hash is passed as a parameter, the referenced hash is used to determine which columns are returned. If the referenced hash is empty, the method behaves normally, except that the returned array reference is a reference to an array of hashes (each element containing a single row as a hash, ala DBI::fectchrow_hashref). Otherwise, the keys are taken as the names of the columns to include in the returned hashes. The key names should be in lower case (the values can be any true value).

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date, serial_number, age FROM myothertable";
my $output = $db->prepare($query);
$output->execute;

my $data = $output->fetchall_arrayref;
# $data is now a reference to an array of arrays. Each element of the
# 'master' array is itself an array that contains a row of data.

print "The fourth date in the table is: " . $data->[3][1] . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
# Element 1 of that array is the date.

my $data = $output->fetchall_arrayref([1]);
# $data is now a reference to an array of arrays. Each element of the 'master'
# array contains an array with one element: the values of the 'date' column
# (row #1).
print "The fourth date in the table is: " . $data->[3][0] . "\n";

my $data = $output->fetchall_arrayref({});
# $data is now a reference to an array of hashes. Each element of the array
# is a hash containing a row of data, with the column names as the keys.
print "The fourth date in the table is: " . $data->[3]{date} . "\n";

my $data = $output->fetchall_arrayref({ 'date' => 1, 'age' => 1 });
# $data is now a reference to an array of hashes. Each element of the array
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
# is a hash containing a row of data with only  the columns 'date' and 'age'.
print "The fouth date in the table is: " . $data->[3]{date} . "\n";
```

DBI::fetchall_hashref

$ref_of_array_of_hashes = $statement_handle->fetchall_hashref();

DBI::fetchall_hashref returns all of the remaining data in the statement handle as a reference to an array. Each row of the array is a reference to a hash that contains the data in that row. The keys of each hash are the names of the columns of the row.

The function returns an undefined value undef if there is no data in the statement handle. If any previous DBI::fetchrow_* functions were called on this statement handle, DBI::fetchall_hashref returns all of the data after the last DBI::fetchrow_* call.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $output = $db->prepare($query);
$output->execute;

my $data = $output->fetchall_hashref;
# $data is now a reference to an array of hashes. Each element of the array
# is a hash containing a row of data, with the column names as the keys.
print "The fourth date in the table is: " . $data->[3]{date} . "\n";
```

DBI::fetchrow_array

@row_of_data = $statement_handle->fetchrow;

DBI::fetchrow returns the next row of data from a statement handle generated by DBI::execute. Each successive call to DBI::fetchrow returns the next row of data. When there is no more data, the function returns an undefined value undef. The elements in the resultant array are in the order specified in the original query. If the query was of the form SELECT * FROM . . ., the elements are ordered in the same sequence as the fields were defined in the table.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name LIKE 'Bob%'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my ($name, $date);

# This is the first row of data from $myothertable_output.
($name, $date) = $myothertable_output->fetchrow_array;
# This is the next row…
($name, $date) = $myothertable_output->fetchrow_array;
# And the next…
my @name_and_date = $myothertable_output->fetchrow_array;
```


Copyright © 2001 O'Reilly & Associates, Inc.

```
# etc…
```

DBI::fetchrow_arrayref, DBI::fetch

$array_reference = $statement_handle->fetchrow_arrayref;

$array_reference = $statement_handle->fetch;

`DBI::fetchrow_arrayref` and its alias, `DBI::fetch`, work exactly like `DBI::fetchrow_array` except that they return a reference to an array instead of an actual array.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name LIKE 'Bob%'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my $name1 = $myothertable_output->fetch->[0]
# This is the 'name' field from the first row of data.
my $date2 = $myothertable_output->fetchrow_arrayref->[1]
# This is the 'date' from from the *second* row of data.
my ($name3, $date3) = @{$myothertable_output->fetch};
# This is the entire third row of data. $myothertable_output->fetch returns a
# reference to an array. We can 'cast' this into a real array with the @{}
# construct.
```

DBI::fetchrow_hashref

$hash_reference = $statement_handle->fetchrow_hashref;

$hash_reference = $statement_handle->fetchrow_hashref($name);

`DBI::fetchrow_hashref` works like `DBI::fetchrow_arrayref` except that it returns a reference to an associative array instead of a regular array. The keys of the hash are the names of the fields and the values are the values of that row of data.

If an argument is present, it is used as the attribute used to get the names of the column (to use as the keys of the hash). By default this is 'NAME', but can also be 'NAME_lc' or 'NAME_uc'.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT * FROM mytable";
my $mytable_output = $db->prepare($query);
$mytable_output->execute;

my %row1 = $mytable_ouput->fetchrow_hashref;
my @field_names = keys %row1;
# @field_names now contains the names of all of the fields in the query.
# This needs to be set only once. All future rows will have the same fields.
my @row1 = values %row1;

while (my %row = $mytable_output->fetchrow_hashref('NAME_lc')) {
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    # %row contains a single row of the output, with the keys being the column
    # names. Because we specified 'NAME_lc' we are guaranteed that the column
    # names are all lower case.
}
```

DBI::finish

$result = $statement_handle->finish;

`DBI::finish` releases all data in the statement handle so that the handle may be destroyed or prepared again. Some database servers require this in order to free the appropriate resources. DBD::mysql does not need this function, but for portable code, you should use it after you are done with a statement handle. The function returns an undefined value `undef` if the handle cannot be freed.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');
my $query = "SELECT * FROM mytable";
my $mytable_output = $db->prepare($query);
$mytable_output->execute;
...
$mytable_output->finish;
# You can now reassign $mytable_output or prepare another statement for it.
```

DBI::func

$handle->func(@func_arguments, $func_name);

@dbs = $db->func('_ListDBs');

@dbs = $db->func("$hostname", '_ListDBs');

@dbs = $db->func("$hostname:$port", '_ListDBs');

@tables = $db->func('_ListTables');

$result = $db->func('createdb', $database, $host, $user, $password, 'admin');

$result = $db->func('createdb', $database, 'admin');

$result = $db->func('dropdb', $database, $host, $user, $password, 'admin');

$result = $db->func('dropdb', $database, 'admin');

$result = $db->func('shutdown', $host, $user, $password, 'admin');

$result = $db->func('shutdown', 'admin');

$result = $db->func('reload', $host, $user, $password, 'admin');

$result = $db->func('reload', 'admin');

`DBI::func` calls specialized nonportable functions included with the various DBD drivers. It can be used with either a database or a statement handle depending on the purpose of the specialized function. If possible, you should use a portable DBI equivalent

 Copyright © 2001 O'Reilly & Associates, Inc.

function. When using a specialized function, the function arguments are passed as a scalar first followed by the function name. DBD::mysql implements the following functions:

`_ListDBs`
: The `_ListDBs` function takes a hostname and optional port number and returns a list of the databases available on that server. It is better to use the portable function `DBI::data_sources` if possible (`DBI::data_sources` does not provide for listing remote databases).

`_ListTables`
: The `_ListTables` function returns a list of the tables present in the current database. This operation can be performed using the portable `DBI::table_info` method. Therefore, the `_ListTables` function will be removed in a future version of Msql_Mysql_Modules.

`createdb`
: The `createdb` function takes the name of a database as its argument and attempts to create that database on the server. You must have permission to create databases for this function to work. The function returns –1 on failure and 0 on success.

`dropdb`
: The `dropdb` function takes the name of a database as its argument and attempts to delete that database from the server. This function does not prompt the user in any way, and if successful, the database will be irrevocably gone forever. You must have permission to drop databases for this function to work. The function returns –1 on failure and 0 on success.

`shutdown`
: The `shutdown` function causes the MySQL server to shut down. All running MySQL processes will be terminated and the connection closed. You must have shutdown privileges to perform this operation. The function returns –1 on failure and 0 on success.

`reload`
: The `reload` function causes the MySQL server to refresh it's internal configuration, including it's access control tables. This is needed if any of the MySQL internal tables are modified manually. You must have reload privileges to perform this operation. The function return –1 on failure and 0 on success.

Example

```
use DBI;
my $db = DBI->install_driver('mysql');

my @dbs = $db->func('myserver.com', '_ListDBs');
# @dbs now has a list of the databases available on the server 'myserver.com'.
```

DBI::looks_like_number

@is_nums = DBI::looks_like_number(@numbers);

DBI::looks_like_number takes an array of unknown elements as its argument. It returns an array of equal size. For each element in the orginal array, the corresponding element in the return array is true if the element is numeric, false if it is not and undefined if it is undefined or empty.

Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
my @array = ( '43.22', 'xxx', '22', undef, '99e' );
my @results = DBI::looks_like_number( @array );
# @results contains the values: true, false, true, undef and true
```

DBI::neat

$neat_string = DBI::neat($string);

$neat_string = DBI::neat($string, $maxlen);

`DBI::neat` takes as its arguments a string and an optional length. The string is then formatted to print out neatly. The entire string is enclosed in single quotes. All unprintable characters are replaced with periods. If the length argument is present, are characters after the maximum length (minus four) are removed and the string is terminated with three periods and a single quote (`...'`). If no length is supplied, 400 is used as the default length.

Example

```
use DBI;

my $string = "This is a very, very, very long string with lots of stuff in it.";
my $neat_string = DBI::neat($string,14);
# $neat_string is now: 'This is a...'
```

DBI::neat_list

$neat_string = DBI::neat_list(\@listref, $maxlen);

$neat_string = DBI::neat_list(\@listref, $maxlen, $field_seperator);

`DBI::neat_list` takes three arguments and returns a neatly formatted string suitable for printing. The first argument is a reference to a list of values to print. The second argument is the maximum length of each field. The final argument is a string used to join the fields. `DBI::neat` is called for each member of the list using the maximum length given. The resulting strings are then joined using the last argument. If the final argument is not present, a comma is used as the separator.

Example

```
use DBI;

my @list = ('Bob', 'Joe', 'Frank');
my $neat_string = DBI::neat_list(\@list, 8);
# $neat_string is now: 'Bob', 'Joe', 'Fra...'
my $neat_string2 = DBI::neat_list(\@list, 8, '<=>');
# $neat_string2 is now: 'Bob' <=> 'Joe' <=> 'Fra...'
```

DBI::ping

$result = $db->ping;

`DBD::ping` attempts to verify if the database server is running. It returns true if the MySQL server is still responding and false otherwise.

 Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
Use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

# Later...
die "MySQL Went Away!" if not $db->ping;
```

DBI::prepare

$statement_handle = $db->prepare($statement);

$statement_handle = $db->prepare($statement, \%unused);

`DBI::prepare` takes as its argument an SQL statement, which some database modules put into an internal compiled form so that it runs faster when `DBI::execute` is called. These DBD modules (DBD::mysql is not one of them) also accept a reference to a hash of optional attributes. The MySQL server does not currently implement the concept of "preparing," so `DBI::prepare` merely stores the statement. You may optionally insert any number of '?' symbols into your statement in place of data values. These symbols are known as "placeholders." The `DBI::bind_param` function is used to substitute the actual values for the placeholders.

> Placeholders can only be used in place of data values. That is, places within the SQL query where free-form data would otherwise go. You can not use placeholders anywhere else within a query. For example "SELECT name FROM mytable WHERE age = ?" is a good use of a placeholder while "SELECT ? FROM mytable WHERE age = 3" is not a valid placeholder (the column name is not free-form data).

The function returns `undef` if the statement cannot be prepared for some reason.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

my $statement_handle = $db->prepare('SELECT * FROM mytable');
# This statement is now ready for execution.

My $statement_handle = $db->prepare(
      'SELECT name, date FROM myothertable WHERE name like ?');
# This statement will be ready for exececuting once the placeholder is filled
# in using the DBI::bind_param function.
```

DBI::prepare_cached

$statement_handle = $db->prepare_cached($statement);

$statement_handle = $db->prepare($statement, \%unused);

$statement_handle = $db->prepare($statement, \%unused, $allow_active_statements);

`DBI::prepare_cached` works identically to `DBI::prepare` except it saves the prepared query in a persistant (for the life of the Perl process) hash table. For database

engines that pre-process prepared queries, this can save the time involved with re-processing complex queries. Since MySQL does not support preparing queries, this feature provides no benefit (it can still be used, though).

If an already prepared query is still active (after having been executed) when it is re-created, DBI will call `DBI::finish` to terminate the query before releasing the prepared statement again. This behavior can be bypassed by passing a true value as the third argument. This will return the prepared statement even if it is currently being executed. If this is done, it is up to the holder of the new instance of the prepared statement to wait until the old one is finished before executing again.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

my $statement_handle = $db->prepare_cached('SELECT * FROM mytable');

# later...
my $new_statement_handle = $db->prepare_cached('SELECT * FROM mytable');
# $new_statement_handle is the exact same handle as $statement_handle
```

DBI::quote

$quoted_string = $db->quote($string);

$quoted_string = $db->quote($string, $data_type);

`DBI::quote` takes a string intended for use in an SQL query and returns a copy that is properly quoted for insertion in the query. This includes placing the proper outer quotes around the string. If the value looks like a number, it is returned as is, without any quotes inserted.

If DBI SQL type constant is provided as the second argument, the value will be quoted properly for that type. This is useful if there are special quoting rules for types other than strings or numbers. For MySQL, the default behavior is generally sufficient.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');

my $string = "Sheldon's Cycle";

my $qs = $db->quote($string);
# $qs is: 'Sheldon''s Cycle' (including the outer quotes)
# The string $qs is now suitable for use in a MySQL SQL statement
```

DBI::rollback

$result = $db->rollback;

`DBI::rollback` instructs MySQL to undo everything that has been done during this session since the last commit. It is only effective on tables that support transactions (such

 Copyright © 2001 O'Reilly & Associates, Inc.

as Berkeley DB tables). In addition, rollbacks are only possible if the DBI attribute AutoCommit is set to false.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');
$db->{AutoCommit} = undef; # Turn off AutoCommit...

# Do some stuff...
if ($error) { $db->rollback; } # Undo any changes we've made...
```

DBI::rows

$number_of_rows = $statement_handle->rows;

`DBI::rows` returns the number of rows of data contained in the statement handle. With DBD::mysql, this function is accurate for all statements, including `SELECT` statements. For many other drivers that do not hold of the results in memory at once, this function is only reliable for non-`SELECT` statements. This should be taken into account when writing portable code. The function returns '-1' if the number of rows is unknown for some reason. The variable `$DBI::rows` provides the same functionality.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name='Bob'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my $rows = $myothertable_output->rows;
print "There are $rows 'Bob's in 'myothertable'.\n";
```

DBI::selectall_arrayref

$arrayref = $dbh->selectall_arrayref($sql_statement);

$arrayref = $dbh->selectall_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectall_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectall_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array of arrays.

Each row of the resulting array is a reference to another array that contains the data in that row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

Copyright © 2001 O'Reilly & Associates, Inc.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectall_arrayref("select name, date from mytable");
# $data is now a reference to an array of arrays. Each element of the
# 'master' array is itself an array that contains a row of data.

print "The fourth date in the table is: " . $data->[3][1] . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
# Element 1 of that array is the date.
```

DBI::selectall_hashref

$hashref = $dbh->selectall_hashref($sql_statement);

$arrayref = $dbh->selectall_hashref($sql_statement, \%unused);

$arrayref = $dbh->selectall_hashref($sql_statement, \%unused, @bind_columns);

`DBI::selectall_hashref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_hashref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array of hashes, with the names of the columns as the keys of the hash.

Each row of the resulting array is a reference to a hash that contains the data in that row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectall_hashref("select name, date from mytable");
# $data is now a reference to an array of hashes. Each element of the
# 'master' array is itself an hash that contains a row of data, with the
# column names as the keys.
```

 Copyright © 2001 O'Reilly & Associates, Inc.

```
print "The fourth date in the table is: " . $data->[3]{DATE} . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
```

DBI::selectcol_arrayref

$arrayref = $dbh->selectcol_arrayref($sql_statement);

$arrayref = $dbh->selectcol_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectcol_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectcol_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array containing the value of the first column of each row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectcol_arrayref("select date, name, age from mytable");
# $data is now a reference to an array. Each element of the
# 'master' array is a 'date' from the table.

print "The fourth date in the table is: " . $data->[3] . "\n";
# Element 3 of the 'master' array is an array containing the fourth date.
```

DBI::selectrow_array

@array = $dbh->selectrow_array($sql_statement);

@array = $dbh->selectrow_array($sql_statement, \%unused);

@array = $dbh->selectrow_array($sql_statement, \%unused, @bind_columns);

`DBI::selectrow_array` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchrow_array` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves the first resulting row and puts it into an array. The function returns an undefined value `undef` if there is no data returned from the query.

Copyright © 2001 O'Reilly & Associates, Inc.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my @data = $db->selectrow_array("select name, date, age from mytable");
# @data is now a an array containing the first row of data from the query.

print "The *first* date in the table is: " . $data[1] . "\n";
# Element 1 of the array is the date.
```

DBI::selectrow_arrayref

$arrayref = $dbh->selectrow_arrayref($sql_statement);

$arrayref = $dbh->selectrow_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectrow_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectrow_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchrow_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves the first resulting row and puts it into a reference to an array. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectrow_array("select name, date, age from mytable");
# $data is now a reference to an array containing the first row of
# data from the query.

print "The *first* date in the table is: " . $data->[1] . "\n";
# Element 1 of the array is the date.
```

 Copyright © 2001 O'Reilly & Associates, Inc.

DBI::state

$sql_error = $handle->state;

`DBI::state` returns the `SQLSTATE` SQL error code for the last error DBI encountered. Currently DBD::mysql reports 'S1000' for all errors. This function is available from both database and statement handles. The variable `$DBI::state` performs the same function.

Example

```
Use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_squirrel');
...
my $sql_error = $db->state;
warn("This is your most recent DBI SQL error: $sql_error");
```

DBI::table_info

$statement_handle = $db->table_info;

$statement_handle = $db->table_info(\%unused);

`DBI::table_info` returns a statement handle (similar to DBI::prepare) that contains information about the tables in the current database. The standard attributes available to a statement handle can be used to get information about the tables in general. However, more useful are the pseodo 'fields' that are available within each row of fetched data from the statement handle:

TABLE_CAT
: This field contains the catalog name of the table. Since MySQL does not support catalogs, this field is always undef.

TABLE_SCHEMA
: This field contains the schema name of the table. Since MySQL does not support the concept of schemas, this field is always undef.

TABLE_NAME
: The name of the table

TYPE_TYPE
: The type of the table. In MySQL this is currently always 'TABLE'. Other database engines support types such as 'VIEW', 'ALIAS', 'SYNONYM' and others.

REMARKS
: A description of the table. Since MySQL currently does not store meta-data about tables, this field is always undef.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_squirrel');

# Print the list of tables in a database.
my $tinfo = $db->table_info;
while (my $table = $tinfo->fetchrow_hashref) {
     print "Table name: " . $table->{TABLE_NAME} . "\n";
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
}
```

DBI::tables

@tables = $db->tables;

`DBI::tables` returns an array of table names for the current database. In MySQL, this will always return all of the tables in the database.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata', 'webuser', 'super_secret_squirrel');

my @tables = $db->tables;
foreach (@tables) { print "Table name: " . $_ . "\n"; }
```

DBI::trace

DBI->trace($trace_level)

DBI->trace($trace_level, $trace_file)

$handle->trace($trace_level);

$handle->trace($trace_level, $trace_file);

`DBI::trace` is useful mostly for debugging purposes. If the trace level is set to 2, general purpose debugging information will be displayed. Setting the trace level to 0 disables the trace. Other values currently enable specialized logging. If `DBI->trace` is used, tracing is enabled for all handles. If `$handle->trace` is used, tracing is enabled for that handle only. This works for both database and statement handles. If a second argument is present for either `DBI->trace` or `$handle->trace`, the debugging information for all handles is appended to that file. You can turn on tracing also by setting the environment variable `DBI_TRACE`. If the environment variable is defined as a number (0 through 5, currently) tracing for all handles is enabled at that level. With any other definition, the trace level is set to 2 and the value of the environment variable is used as the filename for outputting the trace information.

Example

```
use DBI;
my $db1 = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');
DBI->trace(2);
# Tracing is now enabled for all handles at level 2.
$db2->trace(0);
# Tracing is now disabled for $db2, but it is still enabled for $db1
$db1->trace(2,'DBI.trace');
# Tracing is now enabled for all handles at level 2, with the output being
# sent to the file 'DBI.trace'.
```

DBI::trace_msg

DBI->trace_msg($message);

DBI->trace_msg($message, $trace_level);

Copyright © 2001 O'Reilly & Associates, Inc.

DBI::trace_msg writes a message to the trace file (as opened with DBI::trace) associated with the handle used. If a number is given as a second parameter, the message is only written if the trace level of the handle is equal to or higher than the given level.

Example

```
DBI->trace_msg("This will appear in the trace logs...");
DBI->trace_msg("This will be in the trace logs if the level > 3", 3);
```

DBI::type_info

@type_infos = $handle->type_info($dbi_type);

`DBI::type_info` returns a list of hash references containing information about the given DBI SQL type constant. Each element of the list contains information about a possible variant of the given SQL type for the current driver. Because drivers do not always use the same DBI SQL types for similar data types, you may also pass a reference to a list of DBI SQL type constants to this method. In that case, the array of references will contain information about the first constant that matches for the current driver.

If the given DBI SQL type is undefined, empty or 'SQL_ALL_TYPES', the resulting array will contain hashes for all of the data types supported by the current driver.

The hash of information about a given type contains the following keys:

TYPE_NAME
: The name of the data type as used by the SQL database associated with the current driver. In the case of DBD::mysql, this would return the MySQL specific data type name.

DATA_TYPE
: The DBI SQL data type constant. This is the value that was passed into this method (or the value that matched, if an array reference was passed). Internally, these values are stored as integers. However, if the ':sql_types' tag used when DBI is loaded, constant functions are important into the current namespace that return the appropriate constant values. That is, instead of typing '1' for the standard SQL CHAR type, you can type SQL_CHAR.

COLUMN_SIZE
: The maximum size of the column. This has different meanings depending on the type of the column. If it is a character type, the number is the maximum number of characters allowed in the column. If it is a date type, the number is the maximum number of characters needed to represent the date. If it is a decimal numeric type (the attribute NUM_PREC_RADIX will have the value 10), it is the maximum number of digits in the column. If it is a binary numeric type (the attribute NUM_PREC_RADIX will have the value 2), it is the maximum number of bits in the column.

LITERAL_PREFIX
: The string used to prefix a literal value of this type. Most drivers return a single quote (') for character types and not much else. An undefined value is returned if there is no defined prefix for the given data type.

LITERAL_SUFFIX
: The string used to suffix a literal value of this type. Most drivers return a single quote (') for characters types and not much else. An undefined value is returned if there is no defined prefix for the given data type.

CREATE_PARAMS
: A descriptions of the parameters required when defining a column of this type. For example, the CREATE_PARAMS value for the MySQL CHAR type is 'max length', indicating that a CHAR column is defined as CHAR(max length). The value for the MySQL DECIMAL type is 'precision,scale', indicating that a DECIMAL value is defined as DECIMAL(precision, scale).

NULLABLE
: Indicates if this type can ever support NULL values. A false value for this attribute means that NULL values are not possible with this data type. The value '1' indicates that NULL values are possible and the value '2' indicates that it is unknown if the data type can support NULL values.

CASE_SENSITIVE
: This attribute has a true value if the data type is case sensitive with regards to sorting and searching. If the type is not case sensitive, the attribute has a false value.

SEARCHABLE
: This attribute is an integer that indicates how data of this type can be used within a WHERE clause. It can have the following values:

0 – This data type is not allowed within WHERE clauses
1 – This data type is only allowed within LIKE searches
2 – This data type is allowed with all WHERE searches except for LIKE
3 – This data type is allowed within any WHERE clause

UNSIGNED_ATTRIBUTE
: This attribute has a true value if the given data type is unsigned. If it is not unsigned (but could be) a false value is used. If this data type can never be unsigned, an undefined value is used.

FIXED_PREC_SCALE
: This attribute has a true value if the data type always has the same precision and scale. MySQL currently does not have any fixed precision data types. Other database servers sometime use the MONEY data type as a fixed precision data type.

AUTO_UNIQUE_VALUE
: This attribute has a true value if the data type always inserts a new unique value for any row where the value is not given. Since MySQL supports this concept as a modifier to existing types (AUTO_INCREMENT), this attribute always returns a false value.

LOCAL_TYPE_NAME
: This attribute contains the value of the attribute TYPE_NAME localized for the current locale. If no localized version of TYPE_NAME Is available, an undefined value is used.

MINIMUM_SCALE
: The smallest scale possible with this data type. This attribute is undefined for data types that do not support scales.

Copyright © 2001 O’Reilly & Associates, Inc.

MAXIMUM_SCALE
: The largest scale possible with this data type. This attribute is undefined for data types that do not support scales.

SQL_DATA_TYPE
: This attribute is the same as DATA_TYPE except for Date/Time-related data types. For these data types the code SQL_DATETIME or SQL_INTERVAL will be set and the attribute SQL_DATETIME_SUB will contain the exact interval of time represented by the data type.

SQL_DATETIME_SUB
: For Date/Time-related data types, this attribute contains the sub-code of the specific interval represented by the data type.

NUM_PREC_RADIX
: This attribute indicates whether a numeric data type is binary (value '2'), which classifies the decimal types, or decimal (value '10') which classifies the integer types. For non-numeric data types, this attribute is undefined.

mysql_native_type
: This attribute is a DBD::mysql-specific attribute which contains the DBD::mysql code for this data type.

mysql_is_num
: This attribute is true if MySQL considers this data type to be numeric. This is a DBD::mysql-specific attribute.

See the `DBI::bind_param` method for a table correlating the DBI SQL constants, the DBD::mysql constants and the MySQL SQL data types.

Example

```
use DBI qw(:sql_types); # We need the :sql_types tag here to get access to the
                        # DBI SQL constants
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');
my @int_types = $db->type_info( SQL_INTEGER ); # SQL_INTEGER is one of the DBI SQL
                                               # constants.
print "These are the different MySQL SQL types " .
            "corresponding the DBI's SQL_INTEGER: ";
foreach (@int_types) { print $_->{TYPE_NAME}, ','; }
print "\n";

# Not sure which DBI SQL constant corresponds to MySQL BLOB field? Try
# a few of them...
my @blob_types = $db->type_info( [SQL_LONGVARBINARY, SQL_LONGVARCHAR,
          SQL_WLONGVARCHAR ]);
# @blob_types contains information about MySQL SQL types that correspond to the
# first matching DBI SQL type.
print "MySQL Type => DBI SQL Constant\n";
foreach (@blob_types) { print $_->{TYPE_NAME} . ' => ' . $_->{DATA_TYPE}; }
# This prints:
# blob => -1
# text => -1
# tinyblob => -1
# mediumblob => -1
# mediumtext => -1
# longblob => =1
# (-1 is the DBI SQL constant code for SQL_LONGVARCHAR)
```

DBI::type_info_all

$ref_of_array_of_arrays = $db->table_info_all;

`DBI::type_info_all` a reference to an array of arrays containing information about all of the SQL types supported by the current driver. The first row of this array is a hash that defines the names and positions of the information on the following rows. For DBD::mysql at the time of this writing, the contents of this hash is as follows:

TYPE_NAME => 0
DATA_TYPE => 1
COLUMN_SIZE => 2
LITERAL_PREFIX => 3
LITERAL_SUFFIX => 4
CREATE_PARAMS => 5
NULLABLE => 6
CASE_SENSITIVE => 7
SEARCHABLE => 8
UNSIGNED_ATTRIBUTE => 9
FIXED_PREC_SCALE => 10
AUTO_UNIQUE_VALUE => 11
LOCAL_TYPE_NAME => 12
MINIMUM_SCALE => 13
MAXIMUM_SCALE => 14
NUM_PREC_RADIX => 15
mysql_native_type => 16
mysql_is_num => 17

The definitions of these specific fields can be found in the description of `DBI::type_info`, above.

After the initial hash, each subsequent row is a reference to an array that contains elements corresponding to the hash entries in the first row. Using the DBD::mysql fields given above as an example, $row->[7] of any row of the array will return a true or false value indicating if this data type is case sensitive when used in searches and sorts.

Example

```
# Display all of the data types supported by DBD::mysql
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');

my $type_info = $db->type_info_all; # Get the reference to the information.
my @tinfo= @{$type_info}; # Turn the reference into an array.
%index = %{ shift @tinfo }; # The first row is a hash describing the fields in the
                            # following rows.
print join(' ', sort { $index{$a} <=> $index{$b} } keys %index), "\n";
foreach (@tinfo) {
   @row = @$_; # Turn a specific row, which is an array ref, into an array.
   print join(' ', @row);
}
```

 Copyright © 2001 O'Reilly & Associates, Inc.

Attributes

$DBI::err

$DBI::errstr

$DBI::state

$DBI::rows

$DBI::lasth

$db->{AutoCommit}

$db->{Driver}

$db->{info}

$db ->{mysql_insertid}

$db->{Name}

$db->{RowCacheSize}

$db->{Statement}

$db->{thread_id}

$handle->{Active}

$handle->{CachedKids}

$handle->{ChopBlanks}

$handle->{CompatMode}

$handle->{InactiveDestroy}

$handle->{Kids}

$handle->{LongReadLen}

$handle->{LongTruncOk}

$handle->{PrintError}

$handle->{private_*}

$handle->{RaiseError}

$handle->{ShowErrorStatement}

$handle->{Taint}

$handle->{Warn}

$statement_handle->{CursorName}

Copyright © 2001 O'Reilly & Associates, Inc.

$statement_handle->{mysql_insertid}

$statement_handle->{mysql_is_blob}

$statement_handle->{mysql_is_key}

$statement_handle->{mysql_is_num}

$statement_handle->{mysql_is_pri_key}

$statement_handle->{mysql_length}

$statement_handle->{mysql_max_length}

$statement_handle->{mysql_table}

$statement_handle->{mysql_type}

$statement_handle->{mysql_type_name}

$statement_handle->{NAME}

$statement_handle->{NAME_lc}

$statement_handle->{NAME_uc}

$statement_handle->{NULLABLE}

$statement_handle->{NUM_OF_FIELDS}

$statement_handle->{NUM_OF_PARAMS}

$statement_handle->{PRECISION}

$statement_handle->{RowsInCache}

$statement_handle->{SCALE}

$statement_handle->{Statement}

$statement_handle->{TYPE}

The DBI.pm API defines several attributes that may be set or read at any time. Assigning a value to an attribute that can be set changes the behavior of the current connection in some way. Assigning any true value to an attribute will set that attribute on. Assigning 0 to an attribute sets it off. Some values are defined only for particular databases and are not portable. The following are attributes that are present for both database and statement handles.

`$DBI::err`
: `$DBI::err` contains the error code for the last DBI error encountered. This error number corresponds to the error message returned from `$DBI::errstr`.

`$DBI::errstr`
: `$DBI::errstr` contains the error message for the last DBI error encountered. The value remains until the next error occurs, at which time it is replaced. If no error has occurred during your session, the attribute is undefined (`undef`).

 Copyright © 2001 O'Reilly & Associates, Inc.

`$DBI::state`

`$DBI::state` contains the `SQLSTATE` SQL error code for the last error DBI encountered. Currently DBD::mysql reports 'S1000' for all errors.

`$DBI::rows`

`$DBI::rows` contains the number of rows of data contained in the statement handle. With DBD::mysql, this attribute is accurate for all statements, including `SELECT` statements. For many other drivers that do not hold of the results in memory at once, this attribute is only reliable for non-`SELECT` statements. This should be taken into account when writing portable code. The attribute is set to '-1' if the number of rows is unknown for some reason.

`$DBI::lasth`

`$DBI::lasth` contains the handle used in the last DBI method call. This could be a database handle or a statement handle. If the last handle does not exist (that is, if it was destroyed), its parent is used (if one exists).

`$db->{AutoCommit}`

This attribute affects the behavior of database servers that support transactions. For MySQL, this value only has effect when dealing with tables that support transactions (such as Berkeley DB), otherwise the value is always true (on). When using tables that support transactions, the value of this attribute defaults to true. If set to false, any changes made to the table (such as from 'INSERT', 'UPDATE' and 'DELETE' SQL statements) will not take effect until an explicit commit is performed.

`$db->{Driver}`

This attribute is a reference to the DBD driver currently in use. Currently this can only be safely used to retrieve the name of the driver via $db->{Driver}->{Name}.

`$db->{info}`

This attribute is a DBD::mysql-specific attribute that is currently not used. In the future it may contain information about the current database.

`$db->{mysql_last_insertid}`

This is a nonportable attribute that is defined only for DBD::mysql. The attribute returns the last value used in a MySQL AUTO_INCREMENT column. This value is specific to the current database session.

`$db->{Name}`

This is the name of the database. It is usually the exact same name passed as part of the data source to DBI when connecting to the database.

`$db->{RowCacheSize}`

For same drivers, this attribute sets the number of rows held in memory by DBI at any one time. DBD::mysql always holds the entire result set in memory, up to the limitations of the host machine. Therefore, setting this value does nothing, it is always set to 'undef'.

`$db->{Statement}`

Contains the most recent SQL statement passed to DBI::prepare. This contains the last statement used, even if that statement was invalid for some reason. Note that this attribute contains the actual SQL statement as a string, not a statement handle.

Copyright © 2001 O'Reilly & Associates, Inc.

`$db->{thread_id}`
This is a DBD::mysql-specific attribute that is currently not used. In the future it may be used to contain the MySQL-specific thread ID of the current connection.

`$handle->{Active}`
This attribute contains the status of a handle. If used with a database handle, this attribute contains a true value if the database handle is currently connected to a database. If used with a statement handle, this attribute contains a true value if the statement has been executed and there is unread data within the statement handle.

`$handle->{CachedKids}`
This attribute contains a hash of stored handles. If used with a driver handle, it contains a hash of database handles created with `DBI::connect_cached`. If used with a database handle, it contains a hash of statement handles created with `DBI::prepare_cached`. Setting this attribute to an undefined value will clear the cache associated with this handle.

`$handle->{ChopBlanks}`
If this attribute is on, any data returned from a query (such as `DBI::fetchrow` call) will have any leading or trailing spaces chopped off. Any handles deriving from the current handle inherit this attribute. The default for this attribute is ‘off.’

`$handle->{CompatMode}`
This attribute is not meant to be used by application code. It is used to instruct the driver that some type of emulation (probably of an older driver) is taking place.

`$handle->{InactiveDestroy}`
This attribute is designed to enable handles to survive a ‘fork’ so that a child can make use of a parent’s handle. You should enable this attribute in either the parent or the child but not both. The default for this attribute is ‘off.’

`$handle->{Kids}`
This attribute contains the number of 'children' of the given handle. For a driver handle, it contains the number of database handles that were created from this handle. For a database handle, it contains the number of statement handles that were prepared from this handle.

`$handle->{LongReadLen}`
This attribute defines the maximum length of data to retrieve from 'long' columns. These are columns that can hold large amounts of data. In MySQL they are all of the *BLOB and *TEXT columns. By default, no data is read from these columns (this attribute is set to 0). Be aware that setting this attribute to very large values could cause your application to use a lot of memory when reading long columns.

`$handle->{LongTruncOk}`
If this attribute is true, DBI will not complain if the application attempts to read data from a long column that is longer than the value of the 'LongReadLen' attribute. It will simply truncate the data to fix that maximum. If the attribute is false (the default), the application will die if an attempt is made to read data that exceed LongReadLen.

`$handle->{PrintError}`
If this attribute is on, all warning messages will be displayed to the user. If this attribute is off, the errors are available only through `$DBI::errstr`. Any handles

Copyright © 2001 O’Reilly & Associates, Inc.

deriving from the current handle inherit this attribute. The default for this attribute is 'on.'

`$handle->{private_*}`
DBI will store any given data within an attribute that begin with the string 'private_'. This can be used to store arbitrary information that is associated with a particular database handle. If multiple applications are sharing instances of DBI (such as within a mod_perl environment), it is probably wise to make sure these attributes are all named differently. Also, since any data can be associated within this attribute, it can be useful to use a hash as the value of the attribute. This hash can then be used to store any other type of data needed.

`$handle->{RaiseError}`
If this attribute is on, any errors will raise an exception in the program, killing the program if no '`__DIE__`' handler is defined. Any handles deriving from the current handle inherit this attribute. The default for this attribute is 'off.'

`$handle->{ShowErrorStatement}`
If this attribute is set to a true value, any associated SQL statement will be included with the automatically generated DBI error messages. This can be used within any statement handle, as well as with database handles in conjunction with methods like prepare and do, which involve SQL statements.

`$handle->{Taint}`
This attribute enables taint checking if Perl is running in taint mode. If Perl is not in taint mode, this attribute does nothing. When enabled, this attribute causes DBI to check the arguments of it's methods for tainted data (and kill the program if any is found). In addition, all fetched data from SQL statements are marked as tainted. The application is then responsible to check the data to make sure it is safe. In the future, the return values of other DBI methods may be returned as tainted as well, along with fetched data.

`$handle->{Warn}`

If this attribute is on, warning messages for certain bad programming practices (most notably holdovers from Perl 4) will be displayed. Turning this attribute off disables DBI warnings and should be used only if you are really confident in your programming skills. Any handles deriving from the current handle (such as a statement handle resulting from a database handle query) inherit this attribute. The default for this attribute is 'on.'

`$statement_handle->{CursorName}`
This attribute contains the name of the current cursor when used with drivers that support cursors. MySQL does not currently support cursors, so this attribute is always set to 'undef'.

`$statement_handle->{mysql_insertid}`
This is a nonportable attribute that is defined only for DBD::mysql. The attribute returns the current value of the `auto_increment` field (if there is one) in the table. If no `auto_increment` field exists, the attribute returns `undef`.

`$statement_handle->{mysql_is_blob}`
This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle is of a `BLOB` type. For a statement handle that was not

Copyright © 2001 O'Reilly & Associates, Inc.

returned by a SELECT statement, $statement_handle->{mysql_is_blob} returns undef.

$statement_handle->{mysql_is_key}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle were defined as a KEY. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_key} returns undef.

$statement_handle->{mysql_is_num}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle is a number type. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_num} returns undef.

$statement_handle->{mysql_is_pri_key}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of boolean values indicating if each of the fields contained in the statement handle is a primary key. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_pri_key} returns undef.

$statement_handle->{mysql_length}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the maximum possible length of each field contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_length} returns undef.

$statement_handle->{mysql_max_length}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the actual maximum length of each field contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_max_length} returns undef.

$statement_handle->{mysql_table}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the names of the tables accessed in the query. This is particularly useful in conjunction with a JOINed SELECT that uses multiple tables.

$statement_handle->{mysql_type}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute contains a reference to a list of the types of the fields contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_type} returns undef. The values of this list are integers that correspond to an enumeration in the mysql_com.h C header file found in the MySQL distribution.

$statement_handle->{mysql_type_name}
: This is a nonportable attribute which is defined only for DBD::mysql. This attribute contains a reference to a list of the names of the types of the fields contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_type_name} returns undef.

Copyright © 2001 O'Reilly & Associates, Inc.

Whenever MySQL has more than one possible name for a field type, this attribute contains the ANSI standard SQL name, if possible.

`$statement_handle->{NAME}`
This attribute returns a reference to a list of the names of the fields contained in the statement handle. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NAME}` returns `undef`.

`$statement_handle->{NAME_lc}`
This attribute is the same as `$statement_handle->{NAME}` except that the values are guaranteed to always be lower case.

`$statement_handle->{NAME_uc}`
This attribute is the same as `$statement_handle->{NAME}` except that the values are guaranteed to always be upper case.

`$statement_handle->{NULLABLE}`
This attribute returns a reference to a list of boolean values indicating if each of the fields contained in the statement handle can have a `NULL` value. A field defined with '`NOT NULL`' will have a value of 0 in the list. All other fields will have a value of 1. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NULLABLE}` returns `undef`.

`$statement_handle->{NUM_OF_FIELDS}`
This attribute returns the number of columns of data contained in the statement handle. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NUM_OF_FIELDS}` returns 0.

`$statement_handle->{NUM_OF_PARAMS}`
This attribute returns the number of "placeholders" in the statement handle. Placeholders are indicated with a '?' in the statement. The `DBI::bind_values` function is used to replace the placeholders with the proper values.

`$statement_handle->{PRECISION}`
This attribute contains a reference to a list of integer values indicating the 'length' of each column. For numeric columns, this contains the number of significant digits in the column.

`$statement_handle->{RowsInCache}`
For drivers that cache partial results sets in memory, this attribute contains the number of rows currently in the cache. DBD::mysql does not use this feature and this attribute is always set to 'undef'.

`$statement_handle->{SCALE}`
This attribute contains a reference to a list of integer values indicating the 'scale' of each decimal column. For non-decimal columns, 'undef' is used.

`$statement_handle->{Statement}`
This attribute contains the SQL statement string used to prepare this statement handle.

`$statement_handle->{TYPE}`
This attribute contains a reference to a list of DBI SQL type constants for the columns in the result set. The values of this list can be used as arguments to DBI::type_info to retrieve information about a particular type.

Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
use DBI;
my $db = DBI->connect('mysql:mydata','me','mypassword');

$db->{RAISE_ERROR} = 1;
# Now, any DBI/DBD errors will kill the program.

my $statement_handle = $db->prepare('SELECT * FROM mytable');
$statement_handle->execute;

my @fields = @{$statement_handle->{NAME}};
# @fields now contains an array of all of the field names in 'mytable'.
my @types = @{$statement_handle->{TYPE}};
# @types now contains an array of all of the types of the fields in 'mytable'.
```


Copyright © 2001 O'Reilly & Associates, Inc.

27

The JDBC API

The java.sql package contains the entire JDBC API. It first became part of the core Java libraries with the 1.1 release. Classes new as of JDK 1.2 are indicated by the "Availability" header. Deprecated methods are preceded by a hash (#) mark. New JDK 1.2 methods in old JDK 1.1 classes are shown in bold. Figure 27-1 shows the entire java.sql package.

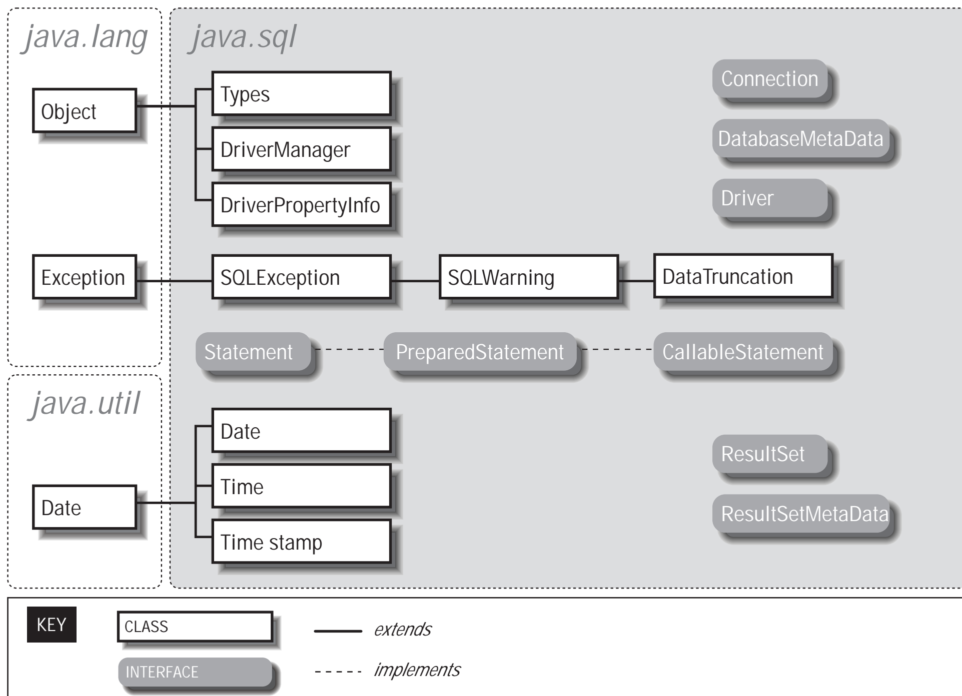


Figure 27-1. The classes and interfaces of the java.sql package

Copyright © 2001 O'Reilly & Associates, Inc.

Array

Synopsis

Class Name:java.sql.Array
Superclass:None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: New as of JDK 1.2

Description

The Array interface is a new addition to JDBC that supports SQL3 array objects. The default duration of a reference to a SQL array is for the life of the transaction in which it was created.

Class Summary

```
public interface Array {
  Object getArray() throws SQLException;
  Object getArray(Map map) throws SQLException;
  Object getArray(long index, int count)
    throws SQLException;
  Object getArray(long index, int count, Map map)
    throws SQLException;
  int getBaseType() throws SQLException;
  String getBaseTypeName() throws SQLException;
  ResultSet getResultSet() throws SQLException;
  ResultSet getResultSet(Map map) throws SQLException;
  ResultSet getResultSet(long index, int count)
    throws SQLException;
  ResultSet getResultSet(long index, int count,
                Map map) throws SQLException
}
```

Object Methods

getArray()

```
public Object getArray() throws SQLException
public Object getArray(Map map) throws SQLException
public Object getArray(long index, int count)
  throws SQLException
public Object getArray(long index, int count, Map map)
  throws SQLException
```

Description: Place the contents of this SQL array into a Java language array or, instead, into the Java type specified by a provided Map. If a map is specified but no match is found in there, then the default mapping to a Java array is used. The two versions that accept an array index and element count enable you to place a subset of the elements in the array.

Copyright © 2001 O'Reilly & Associates, Inc.

getBaseType()

```
public int getBaseType() throws SQLException
```

Description: Provides the JDBC type of the elements of this array.

getBaseTypeName()

```
public String getBaseTypeName() throws SQLException
```

Description: Provides the SQL type name for the elements of this array.

getResultSet()

```
public ResultSet getResultSet() throws SQLException
public ResultSet getResultSet(Map map)
  throws SQLException
public ResultSet getResultSet(long index, int count)
  throws SQLException
public ResultSet getResultSet(long index, int count,
                              Map map)
  throws SQLException
```

Description: Provides a result set that contains the array's elements as rows. If appropriate, the elements are mapped using the type map for the connection, or the specified type map if you pass one. Each row contains two columns: the first column is the index number (starting with 1), and the second column is the actual value.

Blob

Synopsis

Class Name: java.sql.Blob
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: New as of JDK 1.2

Description

The JDBC Blob interface represents a SQL BLOB. BLOB stands for "binary large object" and is a relational database representation of a large piece of binary data. The value of using a BLOB is that you can manipulate the BLOB as a Java object without retrieving all of the data behind the BLOB from the database. A BLOB object is only valid for the duration of the transaction in which it was created.

Class Summary

```
public interface Blob {
  InputStream getBinaryStream() throws SQLException;
  byte[] getBytes(long pos, int count)
    throws SQLException;
  long length() throws SQLException;
  long position(byte[] pattern, long start)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
        throws SQLException;
    long position(Blob pattern, long start)
        throws SQLException;
}
```

Object Methods

getBinaryStream()

```
public InputStream getBinaryStream() throws SQLException
```

Description: Retrieves the data that makes up the binary object as a stream from the database.

getBytes()

```
public byte[] getBytes(long pos, int count)
    throws SQLException
```

Description: Returns the data that makes up the underlying binary object in part or in whole as an array of bytes. You can get a subset of the binary data by specifying a nonzero starting index or by specifying a number of bytes less than the object's length.

length()

```
public long length() throws SQLException
```

Description: Provides the number of bytes that make up the BLOB.

position()

```
public long position(byte[] pattern, long start)
    throws SQLException
public long position(Blob pattern, long start)
    throws SQLException
```

Description: Searches this Blob for the specified pattern and returns the byte at which the specified pattern occurs within this Blob. If the pattern does not occur, then this method will return -1.

CallableStatement

Synopsis

Class Name: java.sql.CallableStatement
Superclass: java.sql.PreparedStatement
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

The CallableStatement is an extension of the PreparedStatement interface that provides support for SQL stored procedures. It specifies methods that handle the bind-

Copyright © 2001 O'Reilly & Associates, Inc.

ing of output parameters. JDBC prescribes a standard form in which stored procedures should appear independent of the DBMS being used. The format is:

```
{? = call ...}
{call ...}
```

Each question mark is a place holder for an input or output parameter. The first syntax provides for a single result parameter. The second syntax has no result parameters. The parameters are referred to sequentially with the first question mark holding the place for parameter 1.

Before executing a stored procedure, all output parameters should be registered using the registerOutParameter() method. You then bind the input parameters using the various set methods, and then execute the stored procedure.

Class Summary

```
public interface CallableStatement extends PreparedStatement {
  Array getArray(int index) throws SQLException;
  BigDecimal getBigDecimal(int index)
    throws SQLException;
  #BigDecimal getBigDecimal(int index, int scale)
    throws SQLException;
  Blob getBlob(int index) throws SQLException;
  boolean getBoolean(int index) throws SQLException;
  byte getByte(int index) throws SQLException;
  byte[] getBytes(int index) throws SQLException;
  Clob getClob(int index) throws SQLException;
  java.sql.Date getDate(int index, Calendar cal)
    throws SQLException;
  java.sql.Date getDate(int index) throws SQLException;
  double getDouble(int index) throws SQLException;
  float getFloat(int index) throws SQLException;
  int getInt(int index) throws SQLException;
  long getLong(int index) throws SQLException;
  Object getObject(int index) throws SQLException;
  Object getObject(int index, Map map)
    throws SQLException;
  Ref getRef(int index) throws SQLException;
  short getShort(int index) throws SQLException;
  String getString(int index) throws SQLException;
  java.sql.Time getTime(int index) throws SQLException;
  java.sql.Time getTime(int index, Calendar cal)
    throws SQLException;
  java.sql.Timestamp getTimestamp(int index)
    throws SQLException;
  java.sql.Timestamp getTimestamp(int index,
                        Calendar cal)
    throws SQLException;
  void registerOutParameter(int index, int type)
    throws SQLException;
  void registerOutParameter(int index, int type,
                    int scale)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    throws SQLException;
  void registerOutParameter(int index, int type,
                 String typename)
    throws SQLException;
  boolean wasNull() throws SQLException;
}
```

Object Methods

getBigDecimal()

```
public BigDecimal getBigDecimal(int index)
  throws SQLException
#public BigDecimal getBigDecimal(int index, int scale)
  throws SQLException
```

Description: Returns the value of the parameter specified by the index parameter as a Java BigDecimal with a scale specified by the scale argument. The scale is a nonnegative number representing the number of digits to the right of the decimal. Parameter indices start at 1; parameter 1 is thus index 1.

getArray(), getBlob(), getBoolean(), getByte(), getBytes(), getClob(), getDouble(), getFloat(), getInt(), getLong(), getRef(), getShort(), and getString()

```
public Array getArray(int index)
  throws SQLException
public Blob getBlob(int index) throws SQLException
public boolean getBoolean(int index) throws SQLException
public byte getByte(int index) throws SQLException
public byte[] getBytes(int index) throws SQLException
public Clob getClob(int index) throws SQLException
public double getDouble(int index) throws SQLException
public float getFloat(int index) throws SQLException
public int getInt(int index) throws SQLException
public long getLong(int index) throws SQLException
public Ref getRef(int index) throws SQLException
public short getShort(int index) throws SQLException
public String getString(int index) throws SQLException
```

Description: Returns the value of the parameter specified by the index argument as the Java datatype indicated by the method name.

getDate(), getTime(), and getTimestamp()

```
public Date getDate(int index) throws SQLException
public Date getDate(int index, Calendar cal)
  throws SQLException
public Time getTime(int index) throws SQLException
public Time getTime(int index, Calendar cal)
  throws SQLException
public Timestamp getTimestamp(int index)
  throws SQLException
public Timestamp getTimestamp(int index, Calendar cal)
  throws SQLException
```

Description: JDBC provides refinements on the basic java.util.Date object more suitable to database programming. These methods provide ways to access

Copyright © 2001 O'Reilly & Associates, Inc.

return values from a CallableStatement as a Date, Time, or Timestamp object. The new JDK 1.2 variants allow you to specify a Calendar.

getObject()

```
public Object getObject(int index) throws SQLException
public Object getObject(int index, Map map)
  throws SQLException
```

Description: Like the other getXXX() methods, this method returns the value of the specified output parameter. In the case of getObject(), however, the JDBC driver chooses the Java class that corresponds to the SQL type registered for this parameter using registerOutParameter() or according to the specified type map.

registerOutParameter()

```
public void registerOutParameter(int index, int type)
  throws SQLException
public void registerOutParameter(int index, int type,
                    int scale)
  throws SQLException
public void registerOutParameter(int index, int type,
                    String typename)
  throws SQLException
```

Description: Before executing any stored procedure using a CallableStatement, you must register each of the output parameters. This method registers the java.sql.Type of an output parameter for a stored procedure. The first parameter specifies the output parameter being registered and the second the java.sql.Type to register. The three-argument version of this method is for BigDecimal types that require a scale. You later read the output parameters using the corresponding getXXX() method or getObject(). The third version of this method is new to JDK 1.2 and provides a way to map REF SQL types or custom SQL types.

wasNull()

```
public boolean wasNull() throws SQLException
```

Description: If the last value you read using a getXXX() call was SQL NULL, this method will return true.

Clob

Synopsis

Class Name: java.sql.Clob
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: New as of JDK 1.2

Copyright © 2001 O'Reilly & Associates, Inc.

Description

CLOB is a SQL3 type that stands for "character large object." Like a BLOB, a CLOB represents a very large chunk of data in the database. Unlike a BLOB, a CLOB represents text stored using some sort of character encoding. The point of a CLOB type as opposed to a CHAR or VARCHAR type is that CLOB data, like BLOB data, can be retrieved as a stream instead of all at once.

Class Summary

```
public interface Clob {
  InputStream getAsciiStream() throws SQLException;
  Reader getCharacterStream() throws SQLException;
  String getSubString(long pos, int count)
    throws SQLException;
  long length() throws SQLException;
  long position(String pattern, long start)
    throws SQLException;
  long position(Clob pattern, long start)
    throws SQLException;
}
```

Object Methods

getAsciiStream()

```
public InputStream getAsciiStream() throws SQLException
```

Description: Provides access to the data that makes up this Clob via an ASCII stream.

getCharacterStream()

```
public Reader getCharacterStream() throws SQLException
```

Description: Provides access to the data that makes up this Clob via a Unicode stream.

getSubString()

```
public String getSubString(long pos, int count)
  throws SQLException
```

Description: Returns a substring of the Clob starting at the named position up to the number of character specified by the count value.

length()

```
public long length() throws SQLException
```

Description: Provides the number of characters that make up the Clob.

position()

```
public long position(String pattern, long start)
  throws SQLException;
public long position(Clob pattern, long start)
  throws SQLException;
```

Description: Searches the Clob for the specified pattern starting at the specified start point. If the pattern is found within the Clob, the index at which the pat-

Copyright © 2001 O'Reilly & Associates, Inc.

tern first occurs is returned. If it does not exist within the Clob, then this method returns -1.

Connection

Synopsis

Class Name: java.sql.Connection
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

The Connection class is the JDBC representation of a database session. It provides an application with Statement objects (and its subclasses) for that session. It also handles the transaction management for those statements. By default, each statement is committed immediately upon execution. You can use the Connection object to turn off this Autocommit feature for the session. In that event, you must expressly send commits, or any statements executed will be lost.

Class Summary

```
public interface Connection {
  static public final int TRANSACTION_NONE;
  static public final int TRANSACTION_READ_UNCOMMITTED;
  static public final int TRANSACTION_READ_COMMITTED;
  static public final int TRANSACTION_REPEATABLE_READ;
  static public final int TRANSACTION_SERIALIZABLE;

  void clearWarnings() throws SQLException;
  void close() throws SQLException;
  void commit() throws SQLException;
  Statement createStatement() throws SQLException;
  Statement createStatement(int type, int concur)
    throws SQLException;
  boolean getAutoCommit() throws SQLException;
  String getCatalog() throws SQLException;
  Map getTypeMap() throws SQLException;
  DatabaseMetaData getMetaData() throws SQLException;
  int getTransactionIsolation() throws SQLException;
  SQLWarning getWarnings() throws SQLException;
  boolean isClosed() throws SQLException;
  boolean isReadOnly() throws SQLException;
  String nativeSQL(String sql) throws SQLException;
  CallableStatement prepareCall(String sql)
    throws SQLException;
  CallableStatement prepareCall(String sql, int type,
                  int concur)
    throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    PreparedStatement prepareStatement(String sql)
      throws SQLException;
    PreparedStatement prepareStatement(String sql,
                                       int type,
                                       int concur)
      throws SQLException;
    void rollback() throws SQLException;
    void setAutoCommit(boolean ac) throws SQLException;
    void setCatalog(String catalog) throws SQLException;
    void setReadOnly(boolean ro) throws SQLException;
    void setTransactionIsolation(int level)
      throws SQLException;
    void setTypeMap(Map map) throws SQLException;
}
```

Class Attributes

TRANSACTION_NONE

```
static public final int TRANSACTION_NONE
```

Description: Transactions are not supported.

TRANSACTION_READ_UNCOMMITTED

```
static public final int TRANSACTION_READ_UNCOMMITTED
```

Description: This transaction isolation level allows uncommitted changes by one transaction to be readable by other transactions.

TRANSACTION_READ_COMMITTED

```
static public final int TRANSACTION_READ_COMMITTED
```

Description: This transaction isolation level prevents dirty reads from occurring. In other words, changes by a TRANSACTION_READ_COMMITTED transaction are invisible to other transactions until the transaction making the change commits those changes.

TRANSACTION_REPEATABLE_READ

```
static public final int TRANSACTION_REPEATABLE_READ
```

Description: This transaction isolation level prevents dirty reads and nonrepeatable reads. A nonrepeatable read is one where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time.

Object Methods

clearWarnings()

```
public void clearWarnings() throws SQLException
```

Description: Clears out all the warnings associated with this Connection so that getWarnings() will return null until a new warning is reported.

close()

```
public void close() throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

Description: This method manually releases all resources (such as network connections and database locks) associated with a given JDBC Connection. This method is automatically called when garbage collection occurs; however, it is best to manually close a Connection once you are done with it.

commit()

```
public void commit() throws SQLException
```

Description: This method makes permanent the changes created by all statements associated with this Connection since the last commit or rollback was issued. It should only be used when Autocommit is off. It does not commit changes made by statements associated with other Connection objects.

createStatement()

```
public Statement createStatement() throws SQLException
public Statement createStatement(int type, int concur)
  throws SQLException
```

Description: This method creates a Statement object associated with this Connection session. The no argument version of this method creates a Statement whose ResultSet instances are type forward-only and read-only concurrency.

getAutoCommit() and setAutoCommit()

```
public boolean getAutoCommit() throws SQLException
public void setAutoCommit(boolean ac)
  throws SQLException
```

Description: By default, all Connection objects are in Autocommit mode. With Autocommit mode on, each statement is committed as it is executed. An application may instead choose to manually commit a series of statements together as a single transaction. In this case, you use the setAutoCommit() method to turn Autocommit off. You then follow your statements with a call to commit() or rollback() depending on the success or failure of the transaction.

When in Autocommit mode, a statement is committed either when the statement completes or when the next statement is executed, whichever is first. For statements returning a ResultSet, the statement completes when the last row has been retrieved or the ResultSet has been closed. If a statement returns multiple result sets, the commit occurs when the last row of the last ResultSet object has been retrieved.

getCatalog() and setCatalog()

```
public String getCatalog() throws SQLException
public void setCatalog(String catalog) throws SQLException
```

Description: If a driver supports catalogs, then you use setCatalog() to select a subspace of the database with the specified catalog name. If the driver does not support catalogs, it will ignore this request.

getMetaData()

```
public DatabaseMetaData getMetaData() throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

Description: The DatabaseMetaData class provides methods that describe a database's tables, SQL support, stored procedures, and other information relating to the database and this Connection, which are not directly related to executing statements and retrieving result sets. This method provides an instance of the DatabaseMetaData class for this Connection.

getTransactionIsolation() and setTransactionIsolation()

```
public int getTransactionIsolation() throws SQLException
public void setTransactionIsolation(int level)
  throws SQLException
```

Description: Sets the Connection object's current transaction isolation level using one of the class attributes for the Connection interface. Those levels are called TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, and TRANSACTION_REPEATABLE_READ.

getTypeMap() and setTypeMap()

```
public Map getTypeMap() throws SQLException
public void setTypeMap(Map map) throws SQLException
```

Description: You can use these methods to define or retrieve a custom mapping for SQL structured types and distinct types for all statements associated with this connection.

getWarnings()

```
public SQLWarning getWarnings() throws SQLException
```

Description: Returns the first warning in the chain of warnings associated with this Connection object.

isClosed()

```
public boolean isClosed() throws SQLException
```

Description: Returns true if the Connection has been closed.

isReadOnly() and setReadOnly()

```
public boolean isReadOnly() throws SQLException
public void setReadOnly(boolean ro) throws SQLException
```

Description: Some databases can optimize for read-only database access. The setReadOnly() method provides you with a way to put a Connection into read-only mode so that those optimizations occur. You cannot call setReadOnly() while in the middle of a transaction.

nativeSQL()

```
public String nativeSQL(String sql) throws SQLException
```

Description: Many databases may not actually support the same SQL required by JDBC. This method allows an application to see the native SQL for a given JDBC SQL string.

prepareCall()

```
public CallableStatement prepareCall(String sql)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
  throws SQLException
public CallableStatement prepareCall(String sql,
                    int type,
                    int concur)
  throws SQLException
```

Description: Given a particular SQL string, this method creates a CallableStatement object associated with this Connection session. This is the preferred way of handling stored procedures. The default (no argument) version of this method provides a CallableStatement whose ResultSet instances are type forward-only and read-only concurrency.

prepareStatement()

```
public PreparedStatement prepareStatement(String sql)
  throws SQLException
public PreparedStatement prepareStatement(String sql,
                      int type,
                      int concur)
  throws SQLException
```

Description: Provides a PreparedStatement object to be associated with this Connection session. This is the preferred way of handling precompiled SQL statements. The default (no argument) version of this method provides a PreparedStatement whose ResultSet instances are type forward-only and read-only concurrency.

rollback()

```
public void rollback() throws SQLException
```

Description: Aborts all changes made by statements associated with this Connection since the last time a commit or rollback was issued. If you want to make those changes at a later time, your application will have to reexecute the statements that made those changes. This should be used only when auto-commit is off.

DatabaseMetaData

Synopsis

Class Name: java.sql.DatabaseMetaData
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: New as of JDK 1.1

Description

This class provides a lot of information about the database to which a Connection object is connected. In many cases, it returns this information in the form of JDBC

Copyright © 2001 O'Reilly & Associates, Inc.

ResultSet objects. For databases that do not support a particular kind of metadata, DatabaseMetaData will throw an SQLException.

DatabaseMetaData methods take string patterns as arguments where specific tokens within the String are interpreted to have a certain meaning. % matches any substring of 0 or more characters and _ matches any one character. You can pass null to methods in place of string pattern arguments; this means that the argument's criteria should be dropped from the search.

Class Summary

```
public interface DatabaseMetaData {
  static public final int bestRowTemporary;
  static public final int bestRowTransaction;
  static public final int bestRowSession;
  static public final int bestRowUnknown;
  static public final int bestRowNotPseudo;
  static public final int bestRowPseudo;
  static public final int columnNoNulls;
  static public final int columnNullable;
  static public final int columnNullableUnknown;
  static public final int importedKeyCascade;
  static public final int importedKeyRestrict;
  static public final int importedKeySetNull;
  static public final int importedKeyNoAction;
  static public final int importedKeySetDefault;
  static public final int importedKeyInitiallyDeferred;
  static public final int importedKeyInitiallyImmediate;
  static public final int importedKeyNotDeferrable;
  static public final int procedureResultUnknown;
  static public final int procedureNoResult;
  static public final int procedureReturnsResult;
  static public final int procedureColumnUnknown;
  static public final int procedureColumnIn;
  static public final int procedureColumnOut;
  static public final int procedureColumnReturn;
  static public final int procedureColumnResult;
  static public final int procedureNoNulls;
  static public final int procedureNullable;
  static public final int procedureNullableUnknown;
  static public final short tableIndexStatistic;
  static public final short tableIndexClustered;
  static public final short tableIndexHashed;
  static public final short tableIndexOther;
  static public final int typeNoNulls;
  static public final int typeNullable;
  static public final int typeNullableUnknown;
  static public final int typePredNone;
  static public final int typePredChar;
  static public final int typePredBasic;
  static public final int typeSearchable;
  static public final int versionColumnUnknown;
  static public final int versionColumnNotPseudo;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
static public final int versionColumnPseudo;

boolean allProceduresAreCallable()
   throws SQLException;
boolean allTablesAreSelectable() throws SQLException;
boolean dataDefinitionCausesTransactionCommit()
   throws SQLException;
boolean dataDefinitionIgnoredInTransactions()
   throws SQLException;
ResultSet getBestRowIdentifier(String catalog,
   String schema, String table, int scope,
   boolean nullable)
   throws SQLException;
ResultSet getCatalogs() throws SQLException;
String getCatalogSeparator() throws SQLException;
String getCatalogTerm() throws SQLException;
ResultSet getColumnPriveleges(String catalog,
      String spat, String table,
      String cpat) throws SQLException;
ResultSet getColumns(String catalog,
      String spat, String tpat,
      String cpat) throws SQLException;
ResultSet getCrossReference(String primaryCatalog,
      String primarySchema, String primaryTable,
      String foreignCatalog, String foreignSchema,
      String foreignTable) throws SQLException;
String getDatabaseProductName() throws SQLException;
String getDatabaseProductVersion()
   throws SQLException;
int getDefaultTransactionIsolation()
   throws SQLException;
int getDriverMajorVersion();
int getDriverMinorVersion();
String getDriverName() throws SQLException;
String getDriverVersion() throws SQLException;
ResultSet getExportedKeys(String catalog,
   String schema, String table)
   throws SQLException;
String getExtraNameCharacters() throws SQLException;
String getIdentifierQuoteString() throws SQLException;
ResultSet getImportedKeys(String catalog,
   String schema, String table) throws SQLException;
ResultSet getIndexInfo(String catalog,
   String schema, String table, boolean unique,
   boolean approximate) throws SQLException;
int getMaxBinaryLiteralLength() throws SQLException;
int getMaxCatalogNameLength() throws SQLException;
int getMaxCharLiteralLength() throws SQLException;
int getMaxcnameLength() throws SQLException;
int getMaxColumnsInGroupBy() throws SQLException;
int getMaxColumnsInIndex() throws SQLException;
int getMaxColumnsInOrderBy() throws SQLException;
int getMaxColumnsInSelect() throws SQLException;
int getMaxColumnsInTable() throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
int getMaxConnections() throws SQLException;
int getMaxIndexLength() throws SQLException;
int getMaxProcedureNameLength()
   throws SQLException;
int getMaxRowSize() throws SQLException;
int getMaxRowSizeIncludeBlobs()
   throws SQLException;
int getMaxSchemaNameLength() throws SQLException;
int getMaxStatementLength() throws SQLException;
int getMaxStatements() throws SQLException;
int getMaxTableNameLength() throws SQLException;
int getMaxTablesInSelect() throws SQLException;
int getMaxUserNameLength() throws SQLException;
String getNumericFunctions() throws SQLException;
ResultSet getPrimaryKeys(String catalog,
   String schema, String table) throws SQLException;
ResultSet getProcedureColumns(String catalog,
   String schemePattern, String procedureNamePattern,
  String cnamePattern) throws SQLException;
String getProcedureTerm() throws SQLException;
ResultSet getProcedures(String catalog,
   String schemaPattern, String procedureNamePattern)
   throws SQLException;
public abstract ResultSet getSchemas() throws SQLException;
public abstract String getSchemaTerm() throws SQLException;
String getSearchStringEscape() throws SQLException;
String getSQLKeywords() throws SQLException;
String getStringFunctions() throws  SQLException;
String getSystemFunctions() throws SQLException;
ResultSet getTablePriveleges(String catalog,
   String schemaPattern, String tableNamePattern)
   throws SQLException;
ResultSet getTableTypes() throws SQLException;
ResultSet getTables(String catalog,
   String schemaPattern, String tableNamePattern,
   String types[]) throws SQLException;
String getTimeDateFunctions() throws SQLException;
ResultSet getTypeInfo() throws SQLException;
String getURL() throws SQLException;
String getUserName() throws SQLException;
ResultSet getVersionColumns(String catalog,
   String schema, String table) throws SQLException;
boolean isCatalogAtStart() throws SQLException;
boolean isReadOnly() throws SQLException;
boolean nullPlusNonNullIsNull() throws SQLException;
boolean nullsAreSortedHigh() throws SQLException;
boolean nullsAreSortedLow() throws SQLException;
boolean nullsAreSortedAtStart() throws SQLException;
boolean nullsAreSortedAtEnd() throws SQLException;
boolean storesLowerCaseIdentifiers()
   throws SQLException;
boolean storesLowerCaseQuotedIdentifiers()
   throws SQLException;
boolean storesMixedCaseIdentifiers()
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    throws SQLException;
boolean storesMixedCaseQuotedIdentifiers()
    throws SQLException;
boolean storesUpperCaseIdentifiers()
    throws SQLException;
boolean storesUpperCaseQuotedIdentifiers()
    throws SQLException;
boolean supportsAlterTableWithAddColumn()
    throws SQLException;
boolean supportsAlterTableWithDropColumn()
    throws SQLException;
boolean supportsANSI92FullSQL() throws SQLException;
boolean supportsANSI92IntermediateSQL()
    throws SQLException;
boolean supportsCatalogsInDataManipulation()
    throws SQLException;
boolean suppportsCatalogsInIndexDefinitions()
    throws SQLException;
boolean supportsCatalogsInPrivelegeDefinitions()
    throws SQLException;
boolean supportsCatalogsInProcedureCalls()
    throws SQLException;
boolean supportsCatalogsInTableDefinitions()
    throws SQLException;
boolean supportsColumnAliasing() throws SQLException;
boolean supportsConvert() throws SQLException;
boolean supportsConvert(int fromType, int toType)
    throws SQLException;
boolean supportsCoreSQLGrammar() throws SQLException;
boolean supportsCorrelatedSubqueries()
    throws SQLException;
boolean
supportsDataDefinitionAndDataManipulationTransactions()
    throws SQLException;
boolean supportsDataManipulationTransactionsOnly()
    throws SQLException;
boolean supportsDifferentTableCorrelationNames()
    throws SQLException;
boolean supportsExpressionsInOrderBy()
    throws SQLException;
boolean supportsExtendedSQLGrammar()
    throws SQLException;
boolean supportsFullOuterJoins() throws SQLException;
boolean supportsGroupBy() throws SQLException;
boolean supportsGroupByBeyondSelect()
    throws SQLException;
boolean supportsGroupByUnrelated()
    throws SQLException;
boolean supportsIntegrityEnhancementFacility()
    throws SQLException;
boolean supportsLikeEscapeClause()
    throws SQLException;
boolean supportsLimitedOuterJoins()
    throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
boolean supportsMinimumSQLGrammar()
  throws SQLException;
boolean supportsMixedCaseIdentifiers()
  throws SQLException;
boolean supportsMixedCaseQuotedIdenfitiers()
  throws SQLException;
boolean supportsMultipleResultSets()
  throws SQLException;
boolean supportsMultipleTransactions()
  throws SQLException;
boolean supportsNonNullableColumns()
  throws SQLException;
boolean supportsOpenCursorsAcrossCommit()
  throws SQLException;
boolean supportsOpenCursorsAcrossRollback()
  throws SQLException;
boolean supportsOpenStatementsAcrossCommit()
  throws SQLException;
boolean supportsOpenStatementsAcrossRollback()
  throws SQLException;
boolean supportsOrderByUnrelated()
  throws SQLException;
boolean supportsOuterJoins() throws SQLException;
boolean supportsPositionedDelete()
  throws SQLException;
boolean supportsPositionedUpdate()
  throws SQLException;
boolean supportsSchemasInDataManipulation()
  throws SQLException;
boolean supportsSchemasInIndexDefinitions()
  throws SQLException;
boolean supportsSchemasInPrivelegeDefinitions()
  throws SQLException;
boolean supportsSchemasInProcedureCalls()
  throws SQLException;
boolean supportsSchemasInTableDefinitions()
  throws SQLException;
boolean supportsSelectForUpdate()
  throws SQLException;
boolean supportsStoredProcedures()
  throws SQLException;
boolean supportsSubqueriesInComparisons()
  throws SQLException;
boolean supportsSubqueriesInExists()
  throws SQLException;
boolean supportsSubqueriesInIns()
  throws SQLException;
boolean supportsSubqueriesInQuantifieds()
  throws SQLException;
boolean supportsTableCorrelationNames()
  throws SQLException;
boolean supportsTransactionIsolationLevel(int level)
  throws SQLException;
boolean supportsTransactions() throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    boolean supportsUnion() throws SQLException;
    boolean supportsUnionAll() throws SQLException;
    boolean usesLocalFilePerTable()
      throws SQLException;
    boolean usesLocalFiles() throws SQLException;
}
```

Date

Synopsis

Class Name: java.sql.Date
Superclass: java.util.Date
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

This class deals with a subset of functionality found in the java.util.Date class. It specifically worries only about days and ignores hours, minutes, and seconds.

Class Summary

```
public class Date extends java.util.Date {
    static public Date valueOf(String s);
    #public Date(int year, int month, int day);
    public Date(long date);
    public void setTime(long date);
    public String toString();
}
```

Class Methods

valueOf()

```
static public Date valueOf(String s)
```

Description: Given a String in the form of yyyy-mm-dd, this will return a corresponding instance of the Date class representing that date.

Object Constructors

Date()

```
public Date(long date)
#public Date(int year, int month, int day)
```

Description: Constructs a new Date instance. The proper way to construct a Date is to use the new JDK 1.2 Date(long) constructor. The date argument specifies the number of milliseconds since 1 January 1970 00:00:00 GMT. A negative number represents the milliseconds before that date. The second, deprecated constructor naturally should never be used since it is ambiguous with respect to calendar and time zone.

Copyright © 2001 O'Reilly & Associates, Inc.

Object Methods

setTime()

```
public void setTime(long date)
```

Description: Sets the time represented by this Date object to the specified number of milliseconds since 1 January 1970 00:00:00 GMT. A negative number represents the milliseconds before that date.

toString()

```
public String toString()
```

Description: Provides a String representing this Date in the form yyyy-mm-dd.

Driver

Synopsis

Class Name: java.sql.Driver
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

This class represents a specific JDBC implementation. When a Driver is loaded, it should create an instance of itself and register that instance with the DriverManager class. This allows applications to create instances of it using the Class.forName() call to load a driver.

The Driver object then provides the ability for an application to connect to one or more databases. When a request for a specific database comes through, the DriverManager will pass the data source request to each Driver registered as a URL. The first Driver to connect to the data source using that URL will be used.

Class Summary

```
public interface Driver {
  boolean acceptsURL(String url) throws SQLException;
  Connection connect(String url, Properties info)
    throws SQLException;
  int getMajorVersion();
  int getMinorVersion();
  DriverPropertyInfo[] getPropertyInfo(String url,
                                       Properties info)
    throws SQLException;
  boolean jdbcCompliant();
}
```

Copyright © 2001 O'Reilly & Associates, Inc.

Object Methods

acceptsURL()

```
public boolean acceptsURL(String url) throws SQLException
```

Description: Returns true if the specified URL matches the URL subprotocol used by this driver.

connect()

```
public Connection connect(String url, Properties info)
  throws SQLException
```

Description: This method attempts a connect using the specified URL and Property information (usually containing the user name and password). If the URL is not right for this driver, connect() simply returns null. If it is the right URL, but an error occurs during the connection process, an SQLException should be thrown.

getMajorVersion()

```
public int getMajorVersion()
```

Description: Returns the major version number for the driver.

getMinorVersion()

```
public int getMinorVersion()
```

Description: Returns the minor version number for the driver.

getPropertyInfo()

```
public DriverPropertyInfo[] getPropertyInfo(String url,
                                           Properties info)
  throws SQLException;
```

Description: This method allows GUI-based RAD environments to find out which properties the driver needs on connect so that it can prompt a user to enter values for those properties.

jdbcCompliant()

```
public boolean jdbcCompliant()
```

Description: A Driver can return true here only if it passes the JDBC compliance tests. This means that the driver implementation supports the full JDBC API and full SQL 92 Entry Level.

DriverManager

Synopsis

Class Name: java.sql.DriverManager
Superclass: java.lang.Object
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Copyright © 2001 O'Reilly & Associates, Inc.

Description

The DriverManager holds the master list of registered JDBC drivers for the system. Upon initialization, it loads all classes specified in the jdbc.drivers property. You can thus specify any runtime information about the database being used by an application on the command line.

During program execution, other drivers may register themselves with the DriverManager by calling the registerDriver() method. The DriverManager uses a JDBC URL to find an application's desired driver choice when requests are made through getConnection().

The DriverManager class is likely to disappear one day as the new JDBC 2.0 Standard Extension provides a much more application-friendly way of getting a database connection.

Class Summary

```
public class DriverManager {
  static void deregisterDriver(Driver driver)
    throws SQLException;
  static public synchronized Connection getConnection(String url,
    Properties info) throws SQLException;
  static public synchronized Connection getConnection(String url,
    String user, String password) throws SQLException;
  static public synchronized Connection getConnection(String url)
    throws SQLException;
  static public Driver getDriver(String url) throws SQLException;
  static public Enumeration getDrivers();
  static public int getLoginTimeout();
  #static public PrintStream getLogStream();
  static public PrintWriter getLogWriter();

  static public void println(String message);
  static public synchronized void registerDriver(Driver driver)
      throws SQLException;
  #static public void setLogStream(PrintStream out);
  static public void setLogWriter(PrintWriter out);
  static public void setLoginTimeout(int seconds);
}
```

Class Methods

deregisterDriver()

```
static public void deregisterDriver(Driver driver) throws SQLException
```

Description: Removes a Driver from the list of registered drivers.

getConnection()

```
static public synchronized Connection getConnection(String url,
    Properties info) throws SQLException
static public synchronized Connection getConnection(String url,
    String user, String password) throws SQLException
static public synchronized Connection getConnection(String url)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    throws SQLException
```

Description: Establishes a connection to the data store represented by the URL given. The DriverManager then looks through its list of registered Driver instances for one that will handle the specified URL. If none is found, it throws an SQLException. Otherwise it returns the Connection instance from the connect() method in the Driver class.

getDriver()

```
static public Driver getDriver(String url) throws SQLException
```

Description: Returns a driver than can handle the specified URL.

getDrivers()

```
static public Enumeration getDrivers()
```

Description: Returns a list of all registered drivers.

getLoginTimeout() and setLoginTimeout()

```
static public int getLoginTimeout()
static public int setLoginTimeout()
```

Description: The login timeout is the maximum time in seconds that a driver can wait in attempting to log in to a database.

getLogStream() and setLogStream()

```
#static public PrintStream getLogStream()
#static public void setLogStream(PrintStream out)
static public PrintWriter getLogWriter()
static public void setLogWriter(PrintWriter out)
```

Description: Sets the stream used by the DriverManager and all drivers. The LogStream variant is the old JDK 1.1 version and should be avoided in favor of log writers

println()

```
static public void println(String message)
```

Description: Prints a message to the current log stream.

registerDriver()

```
static public synchronized void registerDriver(Driver driver)
    throws SQLException
```

Description: This method allows a newly loaded Driver to register itself with the DriverManager class.

Copyright © 2001 O'Reilly & Associates, Inc.

DriverPropertyInfo

Synopsis

Class Name: java.sql.DriverPropertyInfo
Superclass: java.lang.Object
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

This class provides information required by a driver in order to connect to a database. Only development tools are likely ever to require this class. It has no methods, simply a list of public attributes.

Class Summary

```
public class DriverPropertyInfo {
   public String[] choices;
   public String description;
   public String name;
   public boolean required;
   public String value;
   public DriverPropertyInfo(String name, String value);
}
```

Object Attributes

choices

```
public String[] choices
```

Description: A list of choices from which a user may be prompted to specify a value for this property. This value can be null.

description

```
public String description
```

Description: A brief description of the property or null.

name

```
public String name
```

Description: The name of the property.

required

```
public boolean required
```

Description: Indicates whether or not this property must be set in order to make a connection.

value

```
public String value
```

Description: The current value of the property or null if no current value is set.

Copyright © 2001 O'Reilly & Associates, Inc.

Object Constructors

DriverPropertyInfo()

```
public DriverPropertyInfo(String name, String value)
```

Description: Constructs a new DriverPropertyInfo object with the name and value attributes set to the specified parameters. All other values are set to their default values.

PreparedStatement

Synopsis

Class Name: java.sql.PreparedStatement
Superclass: java.sql.Statement
Immediate Subclasses: java.sql.CallableStatement
Interfaces Implemented: None
Availability: JDK 1.1

Description

This class represents a precompiled SQL statement.

Class Summary

```
public interface PreparedStatement extends Statement {
  void addBatch() throws SQLException;
  void clearParameters() throws SQLException;
  boolean execute() throws SQLException;
  ResultSet executeQuery() throws SQLException;
  int executeUpdate() throws SQLException;
  ResultSetMetaData getMetaData() throws SQLException;
  void setArray(int index, Array arr)
    throws SQLException;
  void setAsciiStream(int index, InputStream is,
    int length) throws SQLException;
  void setBigDecimal(int index, BigDecimal d)
    throws SQLException;
  void setBinaryStream(int index, InputStream is,
    int length) throws SQLException;
  void setBlob(int index, Blob b) throws SQLException;
  void setBoolean(int index, boolean b)
    throws SQLException;
  void setByte(int index, byte b) throws SQLException;
  void setBytes(int index, byte[] bts)
    throws SQLException;
  void setCharacterStream(int index, Reader rdr,
    int length) throws SQLException;
  void setClob(int index, Clob c) throws SQLException;
  void setDate(int index, Date d) throws SQLException;
  void setDate(int index, Date d, Calendar cal)
    throws SQLException;
  void setDouble(int index, double x)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
        throws SQLException;
    void setFloat(int index, float f) throws SQLException;
    void setInt(int index, int x) throws SQLException;
    void setLong(int index, long x) throws SQLException;
    void setNull(int index, int type) throws SQLException;
    void setNull(int index, int type, String tname)
        throws SQLException;
    void setObject(int index, Object ob)
        throws SQLException;
    void setObject(int index, Object ob, int type)
        throws SQLException;
    void setObject(int index, Object ob, int type,
        int scale) throws SQLException;
    void setRef(int index, Ref ref) throws SQLException;
    void setShort(int index, short s) throws SQLException;
    void setString(int index, String str)
        throws SQLException;
    void setTime(int index, Time t) throws SQLException;
    void setTime(int index, Time t, Calendar cal)
        throws SQLException;
    void setTimestamp(int index, Timestamp ts)
        throws SQLException;
    void setTimestamp(int index, Timestamp ts, Calendar cal)
        throws SQLException;
    #void setUnicodeStream(int index, InputStream is,
        int length) throws SQLException;
}
```

Object Methods

addBatch()

```
public void addBatch() throws SQLException
```

Description: Adds a set of parameters to the batch for batch processing.

clearParameters()

```
public abstract void clearParameters() throws SQLException
```

Description: Once set, a parameter value remains bound until either a new value is set for the parameter or until clearParameters() is called. This method clears all parameters associated with the PreparedStatement.

execute(), executeQuery(), and executeUpdate()

```
public abstract boolean execute() throws SQLException
public abstract ResultSet executeQuery() throws SQLException
public abstract int executeUpdate() throws SQLException
```

Description: Executes the PreparedStatement. The first method, execute(), allows you to execute the PreparedStatement when you do not know if it is a query or an update. It returns true if the statement has result sets to process.

The executeQuery() method is used for executing queries. It returns a result set for processing.

Copyright © 2001 O'Reilly & Associates, Inc.

The executeUpdate() statement is used for executing updates. It returns the number of rows affected by the update.

getMetaData()

```
public ResultSetMetaData getMetaData() throws SQLException;
```

Description: Retrieves the number, types, and properties of a ResultSet's columns.

setArray(), setAsciiStream(), setBigDecimal(), setBinaryStream(), setBlob(), setBoolean(), setByte(), setBytes(), setCharacterStream(), setClob(), setDate(), setDouble(), setFloat(), setInt(), setLong(), setNull(), setObject(), setRef(), setShort(), setString(), setTime(), setTimestamp(), and setUnicodeStream()

```
public void setArray(int index, Array arr)
  throws SQLException
public void setAsciiStream(int index, InputStream is,
  int length) throws SQLException
public void setBigDecimal(int index, BigDecimal d)
  throws SQLException
public void setBinaryStream(int index, InputStream is,
  int length) throws SQLException
public void setBlob(int index, Blob b)
  throws SQLException
public void setBoolean(int index, boolean b)
  throws SQLException
public void setByte(int index, byte b)
  throws SQLException
public void setBytes(int index, byte[] bts)
  throws SQLException
public void setCharacterStream(int index, Reader rdr,
  int length) throws SQLException
public void setClob(int index, Clob c)
  throws SQLException
public void setDate(int index, Date d)
  throws SQLException
public void setDate(int index, Date d, Calendar cal)
  throws SQLException
public void setDouble(int index, double d)
  throws SQLException
public void setFloat(int index, float f)
  throws SQLException
public void setInt(int index, int x)
  throws SQLException
public void setLong(int index, long x)
  throws SQLException
public void setNull(int index, int type)
  throws SQLException
public void setNull(int index, int type, String tname)
  throws SQLException
public void setObject(int index, Object ob)
  throws SQLException
public void setObject(int index, Object ob, int type)
  throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
public void setObject(int index, Object ob, int type,
  int scale) throws SQLException
public void setRef(int index, Ref ref)
  throws SQLException
public void setShort(int index, short s)
  throws SQLException
public void setString(int index, String str)
  throws SQLException
public void setTime(int index, Time t)
  throws SQLException
public void setTime(int index, Time t, Calendar cal)
  throws SQLException
public void setTimestamp(int index, Timestamp ts)
  throws SQLException
public void setTimestamp(int index, Timestamp ts,
  Calendar cal) throws SQLException
#public void setUnicodeStream(int index, InputStream is,
  int length) throws SQLException
```

Description: Binds a value to the specified parameter.

Ref

Synopsis

Class Name: java.sql.Ref
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: New as of JDK 1.2

Description

A Ref is a reference to a value of an SQL structured type in the database. You can dereference a Ref by passing it as a parameter to an SQL statement and executing the statement.

Class Summary

```
public interface Ref {
  String getBaseTypeName() throws SQLException;
}
```

Object Methods

getBaseTypeName()

```
public String getBaseTypeName() throws SQLException
```

Description: Provides the SQL structured type name for the referenced item.

Copyright © 2001 O'Reilly & Associates, Inc.

ResultSet

Synopsis

Class Name: java.sql.ResultSet
Superclass: None
Immediate Subclasses: None
Interfaces Implemented: None
Availability: JDK 1.1

Description

This class represents a database result set. It provides an application with access to database queries one row at a time. During query processing, a ResultSet maintains a pointer to the current row being manipulated. The application then moves through the results sequentially until all results have been processed or the ResultSet is closed. A ResultSet is automatically closed when the Statement that generated it is closed, reexecuted, or used to retrieve the next ResultSet in a multiple result set query.

Class Summary

```
public interface ResultSet {
  static public final int CONCUR_READ_ONLY;
  static public final int CONCUR_UPDATABLE;
  static public final int FETCH_FORWARD;
  static public final int FETCH_REVERSE;
  static public final int FETCH_UNKNOWN;
  static public final int TYPE_FORWARD_ONLY;
  static public final int TYPE_SCROLL_INSENSITIVE;
  static public final int TYPE_SCROLL_SENSITIVE;
  boolean absolute(int row) throws SQLException;
  void afterLast() throws SQLException;
  void beforeFirst() throws SQLException;
  void cancelRowUpdates() throws SQLException;
  void clearWarnings() throws SQLException;
  void close() throws SQLException;
  void deleteRow() throws SQLException;
  int findColumn(String cname) throws SQLException;
  boolean first() throws SQLException;
  Array getArray(int index) throws SQLException;
  Array getArray(String cname) throws SQLException;
  InputStream getAsciiStream(int index)
    throws SQLException;
  InputStream getAsciiStream(String cname)
    throws SQLException;
  InputStream getBinaryStream(int index)
    throws SQLException;
  InputStream getBinaryStream(String cname)
    throws SQLException;
  BigDecimal getBigDecimal(int index)
    throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
#BigDecimal getBigDecimal(int index, int scale)
   throws SQLException;
BigDecimal getBigDecimal(String cname)
   throws SQLException;
#BigDecimal getBigDecimal(String cname, int scale)
   throws SQLException;
InputStream getBinaryStream(int index)
   throws SQLException;
InputStream getBinaryStream(String cname)
   throws SQLException;
Blob getBlob(int index) throws SQLException;
Blob getBlob(String cname) throws SQLException;
boolean getBoolean(int index) throws SQLException;
boolean getBoolean(String cname) throws SQLException;
byte getByte(int index) throws SQLException;
byte getByte(String cname) throws SQLException;
byte[] getBytes(int index) throws SQLException;
byte[] getBytes(String cname) throws SQLException;
Reader getCharacterStream(int index)
   throws SQLException;
Reader getCharacterStream(String cname)
   throws SQLException;
Clob getClob(int index) throws SQLException;
Clob getClob(String cname) throws SQLException;
int getConcurrency() throws SQLException;
String getCursorName() throws SQLException;
Date getDate(int index) throws SQLException;
Date getDate(int index, Calendar cal)
   throws SQLException;
Date getDate(String cname) throws SQLException;
Date getDate(String cname, Calendar cal)
   throws SQLException;
double getDouble(int index) throws SQLException;
double getDouble(String cname) throws SQLException;
int getFetchDirection() throws SQLException;
int getFetchSize() throws SQLException;
float getFloat(int index) throws SQLException;
float getFloat(String cname) throws SQLException;
int getInt(int index) throws SQLException;
int getInt(String cname) throws SQLException;
long getLong(int index) throws SQLException;
long getLong(String cname) throws SQLException;
ResultSetMetaData getMetaData() throws SQLException;
Object getObject(int index) throws SQLException;
Object getObject(int index, Map map)
   throws SQLException;
Object getObject(String cname) throws SQLException;
Object getObject(String cname, Map map)
   throws SQLException;
Ref getRef(int index) throws SQLException;
Ref getRef(String cname) throws SQLException;
int getRow() throws SQLException;
short getShort(int index) throws SQLException;
short getShort(String cname) throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
Statement getStatement() throws SQLException;
String getString(int index) throws SQLException;
String getString(String cname) throws SQLException;
Time getTime(int index) throws SQLException;
Time getTime(int index, Calendar cal)
   throws SQLException;
Time getTime(String cname) throws SQLException;
Time getTime(String cname, Calendar cal)
   throws SQLException;
Timestamp getTimestamp(int index) throws SQLException;
Timestamp getTimestamp(int index, Calendar cal)
   throws SQLException;
Timestamp getTimestamp(String cname) throws SQLException;
Timestamp getTimestamp(String cname, Calendar cal)
   throws SQLException;
int getType() throws SQLException;
#InputStream getUnicodeStream(int index)
   throws SQLException;
#InputStream getUnicodeStream(String cname)
   throws SQLException;
SQLWarning getWarnings() throws SQLException;
void insertRow() throws SQLException;
boolean isAfterLast() throws SQLException;
boolean isBeforeFirst() throws SQLException;
boolean isFirst() throws SQLException;
boolean isLast() throws SQLException;
boolean last() throws SQLException;
void moveToCurrentRow() throws SQLException;
void moveToInsertRow() throws SQLException;
boolean next() throws SQLException;
boolean previous() throws SQLException;
void refreshRow() throws SQLException;
boolean relative(int rows) throws SQLException;
boolean rowDeleted() throws SQLException;
boolean rowInserted() throws SQLException;
boolean rowUpdated() throws SQLException;
void setFetchDirection(int dir) throws SQLException;
void setFetchSize(int rows) throws SQLException;
void updateAsciiStream(int index, InputStream is,
   int length) throws SQLException;
void updateAsciiStream(String cname, InputStream is,
   int length) throws SQLException;
void updateBigDecimal(int index, BigDecimal d)
   throws SQLException;
void updateBigDecimal(String cname, BigDecimal d)
   throws SQLException;
void updateBinaryStream(int index, InputStream is)
   throws SQLException;
void updateBinaryStream(String cname, InputStream is)
   throws SQLException;
void updateBoolean(int index, boolean b)
   throws SQLException;
void updateBoolean(String cname, boolean b)
   throws SQLException;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
void updateByte(int index, byte b)
  throws SQLException;
void updateByte(String cname, byte b)
  throws SQLException;
void updateBytes(int index, byte[] bts)
  throws SQLException;
void updateBytes(String cname, byte[] bts)
  throws SQLException;
void updateCharacterStream(int index, Reader rdr,
  int length) throws SQLException;
void updateCharacterStream(String cname, Reader rdr,
  int length) throws SQLException;
void updateDate(int index, Date d)
  throws SQLException;
void updateDate(String cname, Date d)
  throws SQLException;
void updateDouble(int index, double d)
  throws SQLException;
void updateDouble(String cname, double d)
  throws SQLException;
void updateFloat(int index, float f)
  throws SQLException;
void updateFloat(String cname, float f)
  throws SQLException;
void updateInt(int index, int x) throws SQLException;
void updateInt(String cname, int x)
  throws SQLException;
void updateLong(int index, long x)
  throws SQLException;
void updateLong(String cname, long x)
  throws SQLException;
void updateNull(int index) throws SQLException;
void updateNull(String cname) throws SQLException;
void updateObject(int index, Object ob)
  throws SQLException;
void updateObject(int index, Object ob, int scale)
void updateObject(String cname, Object ob)
  throws SQLException;
void updateObject(String cname, Object ob, int scale)
  throws SQLException;
void updateRow() throws SQLException;
void updateShort(int index, short s)
  throws SQLException;
void updateShort(String cname, short s)
  throws SQLException;
void updateString(int index, String str)
  throws SQLException;
void updateString(String cname, String str)
  throws SQLException;
void updateTime(int index, Time t)
  throws SQLException;
void updateTime(String cname, Time t)
  throws SQLException;
void updateTimestamp(int index, Timestamp ts)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    throws SQLException;
  void updateTimestamp(String cname, Timestamp ts)
    throws SQLException;
  boolean wasNull() throws SQLException;
}
```

Class Attributes

CONCUR_READ_ONLY

```
static public final int CONCUR_READ_ONLY
```

Description: The concurrency mode that specifies that a result set may not be updated.

CONCUR_UPDATABLE

```
static public final int CONCUR_UPDATABLE
```

Description: The concurrency mode that specifies that a result set is updatable.

FETCH_FORWARD

```
static public final int FETCH_FORWARD
```

Description: This value specifies that a result set's fetch direction is in the forward direction, from first to last.

FETCH_REVERSE

```
static public final int FETCH_REVERSE
```

Description: This value specifies that a result set's fetch direction is in the reverse direction, from last to first.

FETCH_UNKNOWN

```
static public final int FETCH_UNKNOWN
```

Description: This value specifies that the order of result set processing is unknown.

TYPE_FORWARD_ONLY

```
static public final int TYPE_FORWARD_ONLY
```

Description: This result set type specifies that a result set can only be navigated in the forward direction.

TYPE_SCROLL_INSENSITIVE

```
static public final int TYPE_SCROLL_INSENSITIVE
```

Description: This result set type specifies that a result set may be navigated in any direction, but it is not sensitive to changes made by others.

TYPE_SCROLL_SENSITIVE

```
static public final int TYPE_SCROLL_SENSITIVE
```

Description: This result set type specifies that a result set may be navigated in any direction and that changes made by others will be seen in the result set.

Copyright © 2001 O'Reilly & Associates, Inc.

Object Methods

absolute()

public boolean absolute(int row) throws SQLException

Description: This method moves the cursor to the specified row number starting from the beginning for a positive number or from the end for a negative number.

afterLast()

public void afterLast() throws SQLException

Description: This method moves the cursor to the end of the result set, after the last row.

beforeFirst()

public void beforeFirst() throws SQLException

Description: Moves the cursor to the beginning of the result set, before the first row.

cancelRowUpdates()

public void cancelRowUpdates() throws SQLException

Description: Cancels any updates made to this row.

clearWarnings()

public void clearWarnings() throws SQLException

Description: Clears all warnings from the SQLWarning chain. Subsequent calls to getWarnings() then returns null until another warning occurs.

close()

public void close() throws SQLException

Description: Performs an immediate, manual close of the ResultSet. This is generally never required, as the closure of the Statement associated with the ResultSet will automatically close the ResultSet.

deleteRow()

public void deleteRow() throws SQLException

Description: Deletes the current row from this result set and from the database.

findColumn()

public int findColumn(String cname) throws SQLException

Description: For the specified column name, this method will return the column number associated with it.

first()

public boolean first() throws SQLException

Description: Moves the cursor to the first row of a result set.

Copyright © 2001 O'Reilly & Associates, Inc.

getAsciiStream(), getBinaryStream(), getCharacterStream(), and getUnicodeStream()

```
public InputStream getAsciiStream(int index)
  throws SQLException
public InputStream getAsciiStream(String cname)
  throws SQLException
public InputStream getBinaryStream(int index)
  throws SQLException
public InputStream getBinaryStream(String cname)
  throws SQLException
public Reader getCharacterStream(int index)
  throws SQLException
public Reader getCharacterStream(String cname)
  throws SQLException
#public InputStream getUnicodeStream(int index)
  throws SQLException
#public InputStream getUnicodeStream(String cname)
  throws SQLException
```

Description: In some cases, it may make sense to retrieve large pieces of data from the database as a Java InputStream. These methods allow an application to retrieve the specified column from the current row in this manner. You should notice that the getUnicodeStream() method has been deprecated in favor of the new getCharacterStream() method.

getArray(), getBlob(), getBoolean(), getByte(), getBytes(), getClob(), getDate(), getDouble(), getFloat(), getInt(), getLong(), getRef(), getShort(), getString(), getTime(), and getTimestamp()

```
public Array getArray(int index) throws SQLException
public Array getArray(String cname) throws SQLException
public Blob getBlob(int index) throws SQLException
public Blob getBlob(String cname) throws SQLException
public boolean getBoolean(int index) throws SQLException
public boolean getBoolean(String cname) throws SQLException
public byte getByte(int index) throws SQLException
public byte getByte(String cname) throws SQLException
public byte[] getBytes(int index) throws SQLException
public byte[] getBytes(String cname) throws SQLException
public Clob getClob(int index) throws SQLException
public Clob getClob(String cname) throws SQLException
public Date getDate(int index) throws SQLException
public Date getDate(String cname) throws SQLException
public double getDouble(int index) throws SQLException
public double getDouble(String cname) throws SQLException
public float getFloat(int index) throws SQLException
public float getFloat(String cname) throws SQLException
public int getInt(int index) throws SQLException
public int getInt(String cname) throws SQLException
public long getLong(int index) throws SQLException
public long getLong(String cname) throws SQLException
public Ref getRef(int index) throws SQLException
public Ref getRef(String cname) throws SQLException
public short getShort(int index) throws SQLException
public short getShort(String cname) throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
public String getString(int index) throws SQLException
public String getString(String cname) throws SQLException
public Time getTime(int index) throws SQLException
public Time getTime(String cname) throws SQLException
public Timestamp getTimestamp(int index)
  throws SQLException
public Timestamp getTimestamp(String cname)
  throws SQLException
```

Description: These methods return the specified column value for the current row as the Java datatype that matches the method name.

getConcurrency(), and setConcurrency()

```
public int getConcurrency() throws SQLException
```

Description: These methods access the result set concurrency mode. It initially takes its value from the statement that generated this result set.

getCursorName()

```
public String getCursorName() throws SQLException
```

Description: Because some databases allow positioned updates, an application needs the cursor name associated with a ResultSet in order to perform those positioned updates. This method provides the cursor name.

getMetaData()

```
public ResultSetMetaData getMetaData() throws SQLException
```

Description: Provides the meta-data object for this ResultSet.

getFetchDirection(), setFetchDirection(), getFetchSize(), and setFetchSize()

```
public int getFetchDirection() throws SQLException
public void setFetchDirection(int dir) throws SQLException
public int getFetchSize() throws SQLException
public void setFetchSize(int rows) throws SQLException
```

Description: These methods provide optimization hints for the driver. The driver is free to ignore these hints. The fetch size is the suggested number of rows the driver should prefetch for each time it grabs data from the database. The direction is a hint to the driver about the direction in which you intend to work.

getObject()

```
public Object getObject(int index) throws SQLException
public Object getObject(int index, Map map)
  throws SQLException
public Object getObject(String cname) throws SQLException
public Object getObject(String cname, Map map)
  throws SQLException
```

Description: Returns the specified column value for the current row as a Java object. The type returned will be the Java object that most closely matches the SQL type for the column. It is also useful for columns with database-specific datatypes.

Copyright © 2001 O'Reilly & Associates, Inc.

getRow()

public int getRow() throws SQLException

Description: Returns the current row number.

getStatement()

public Statement getStatement() throws SQLException

Description: Returns the Statement instance that generated this result set.

getType()

public int getType() throws SQLException

Description: Returns the result set type for this result set.

getWarnings()

public SQLWarning getWarnings() throws SQLException

Description: Returns the first SQLWarning object in the warning chain.

insertRow()

public void insertRow() throws SQLException

Description: Inserts the contents of the insert row into the result set and into the database.

isAfterLast()

public boolean isAfterLast() throws SQLException

Description: Returns true if this result set is positioned after the last row in the result set.

isBeforeLast()

public boolean isBeforeFirst() throws SQLException

Description: Returns true if this result set is positioned before the first row in the result set.

isFirst()

public boolean isFirst() throws SQLException

Description: Returns true if the result set is positioned on the first row of the result set.

isLast()

public boolean isLast() throws SQLException

Description: Returns true if result set is positioned after the last row in the result set.

last()

public boolean last() throws SQLException

Description: Moves the cursor to the last row in the result set.

moveToCurrentRow()

public void moveToCurrentRow() throws SQLException

Copyright © 2001 O'Reilly & Associates, Inc.

Description: Moves the result set to the current row. This is used after you are done inserting a row.

moveToInsertRow()

```
public void moveToInsertRow() throws SQLException
```

Description: Moves the result to a new insert row. You need to call moveToCurrentRow() to get back.

next() and previous()

```
public boolean next() throws SQLException
public boolean previous() throws SQLException
```

Description: These methods navigate one row forward or one row backward in the ResultSet. Under a newly created result set, the result set is positioned before the first row. The first call to next() would thus move the result set to the first row. These methods return true as long as there is a row to move to. If there are no further rows to process, it returns false. If an InputStream from the previous row is still open, it is closed. The SQLWarning chain is also cleared.

refreshRow()

```
public void refreshRow() throws SQLException
```

Description: Refreshes the current row with its most recent value from the database.

relative()

```
public boolean relative(int rows) throws SQLException
```

Description: Moves the cursor the specified number of rows forwards or backwards. A positive number indicates that the cursor should be moved forwards and a negative number indicates it should be moved backwards.

rowDeleted(), rowInserted(), and rowUpdated()

```
public boolean rowDeleted() throws SQLException
public boolean rowInserted() throws SQLException
public boolean rowUpdated() throws SQLException
```

Description: Returns true if the current row has been deleted, inserted, or updated.

updateAsciiStream(), updateBigDecimal(), updateBinaryStream(), updateBoolean(), updateByte(), updateBytes(), updateCharacterStream(), updateDate(), updateDouble(), updateFloat(), updateInt(), updateLong(), updateNull(), updateObject(), updateShort(), updateString(), updateTime(), and updateTimestamp()

```
public void updateAsciiStream(int index, InputStream is,
  int length) throws SQLException
public void updateAsciiStream(String cname, InputStream is,
  int length) throws SQLException
public void updateBigDecimal(int index, BigDecimal d)
  throws SQLException
public void updateBigDecimal(String cname, BigDecimal d)
  throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
public void updateBinaryStream(int index, InputStream is)
  throws SQLException
public void updateBinaryStream(String cname, InputStream is)
  throws SQLException
public void updateBoolean(int index, boolean b)
  throws SQLException
public void updateBoolean(String cname, boolean b)
  throws SQLException
public void updateByte(int index, byte b)
  throws SQLException
public void updateByte(String cname, byte b)
  throws SQLException
public void updateBytes(int index, byte[] bts)
  throws SQLException
public void updateBytes(String cname, byte[] bts)
  throws SQLException
public void updateCharacterStream(int index, Reader rdr,
  int length) throws SQLException
public void updateCharacterStream(String cname, Reader rdr,
  int length) throws SQLException
public void updateDate(int index, Date d)
  throws SQLException
public void updateDate(String cname, Date d)
  throws SQLException
public void updateDouble(int index, double d)
  throws SQLException
public void updateDouble(String cname, double d)
  throws SQLException
public void updateFloat(int index, float f)
  throws SQLException
public void updateFloat(String cname, float f)
  throws SQLException
public void updateInt(int index, int x)
  throws SQLException
public void updateInt(String cname, int x)
  throws SQLException
public void updateLong(int index, long x)
  throws SQLException
public void updateLong(String cname, long x)
  throws SQLException
public void updateNull(int index) throws SQLException
public void updateNull(String cname) throws SQLException
public void updateObject(int index, Object ob)
  throws SQLException
public void updateObject(int index, Object ob, int scale)
  throws SQLException
public void updateObject(String cname, Object ob)
  throws SQLException
public void updateObject(String cname, Object ob, int scale)
  throws SQLException
public void updateShort(int index, short s)
  throws SQLException
public void updateShort(String cname, short s)
  throws SQLException
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
public void updateString(int index, String str)
  throws SQLException
public void updateString(String cname, String str)
  throws SQLException
public void updateTime(int index, Time t)
  throws SQLException
public void updateTime(String cname, Time t)
  throws SQLException
public void updateTimestamp(int index, Timestamp ts)
  throws SQLException
public void updateTimestamp(String cname, Timestamp ts)
  throws SQLException
```

Description: These methods update column by column in the current row of your result set as long as your result set supports updating. Once you are done modifying the row, you can call insertRow() or updateRow() to save the changes to the database.

updateRow()

```
public void updateRow() throws SQLException
```

Description: Updates any changes made to the current row to the database.

wasNull()

```
public boolean wasNull() throws SQLException
```

Description: This method returns true if the last column read was null; otherwise it returns false.

ResultSetMetaData

Synopsis

Class Name:java.sql.ResultSetMetaData
Superclass:None
Immediate Subclasses:None
Interfaces Implemented:None
Availability:JDK 1.1

Description

This class provides meta-information about the types and properties of the columns in a ResultSet instance.

Class Summary

```
public interface ResultSetMetaData {
  static public final int columnNoNulls;
  static public final int columnNullable;
  static public final int columnNullableUnknown;
  String getCatalogName(int index)
    throws SQLException;
  String getColumnClassName(int index)
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
        throws SQLException;
    public int getColumnCount() throws SQLException;
    public int getColumnDisplaySize(int index)
        throws SQLException;
    public String getColumnLabel(int index)
        throws SQLException;
    public String getColumnName(int index)
        throws SQLException;
    public int getColumnType(int index) throws SQLException;
    public String getColumnTypeName(int index)
        throws SQLException;
    public int getPrecision(int index) throws SQLException;
    public int getScale(int index) throws SQLException;
    public String getSchemaName(int index)
        throws SQLException;
    public String getTableName(int index)
        throws SQLException;
    public boolean isAutoIncrement(int index)
        throws SQLException;
    public isCaseSensitive(int index)
        throws SQLException;
    public boolean isCurrency(int index)
        throws SQLException;
    public boolean isDefinitelyWritable(int index)
        throws SQLException;
    public int isNullable(int index) throws SQLException;
    public boolean isReadOnly(int index)
        throws SQLException;
    public boolean isSearchable(int index)
        throws SQLException;
    public boolean isSigned(int index) throws SQLException;
    public boolean isWritable(int index)
        throws SQLException;
}
```

Class Attributes

columnNoNulls

```
static public final int columnNoNulls
```

Description: The column in question does not allow NULL values.

columnNullable

```
static public final int columnNullable
```

Description: The column in question allows NULL values.

columnNullableUnknown

```
static public final int columnNullableUnknown
```

Description: It is not known if the column in question can accept NULL values.

Copyright © 2001 O'Reilly & Associates, Inc.

Object Methods

getCatalogName()

```
public String getCatalogName(int index) throws SQLException
```

Description: Provides the catalog name associated with the specified column's table.

getColumnClassName()

```
public String getColumnClassName(int index)
  throws SQLException
```

Description: Provides the fully-qualified name of the Java class that will be instantiated by a call to ResultSet.getObject() for this column.

getColumnCount()

```
public int getColumnCount() throws SQLException
```

Description: Returns the number of columns in the result set.

getColumnDisplaySize()

```
public int getColumnDisplaySize(int column)
  throws SQLException
```

Description: Returns the maximum width for displaying the column's values.

getColumnLabel()

```
public String getColumnLabel(int column) throws SQLException
```

Description: Returns the display name for the column.

getColumnName()

```
public String getcname(int column) throws SQLException
```

Description: Returns the database name for the column.

getColumnType()

```
public int getColumnType(int column) throws SQLException
```

Description: Returns the SQL type for the specified column as a value from java.sql.Types.

getColumnTypeName()

```
public String getColumnTypeName(int column)
  throws SQLException
```

Description: Returns the name of the SQL type for the specified column.

getPrecision()

```
public int getPrecision(int column) throws SQLException
```

Description: Returns the number of decimal digits for the specified column.

getScale()

```
public int getScale(int column) throws SQLException
```

Description: Returns the number of digits to the right of the decimal for this column.

Copyright © 2001 O'Reilly & Associates, Inc.

getSchemaName()

```
public String getSchemaName(int column) throws SQLException
```

Description: Returns the schema for the table for the specified column.

getTableName()

```
public String getTableName(int column) throws SQLException
```

Description: Returns the name of the table for the specified column.

isAutoIncrement()

```
public boolean isAutoIncrement(int column) throws SQLException
```

Description: Returns true if the column is automatically numbered and therefore read-only.

isCaseSensitive()

```
public boolean isCaseSensitive(int column) throws SQLException
```

Description: Returns true if the column's case is important.

isCurrency()

```
public boolean isCurrency(int column) throws SQLException
```

Description: Returns true if the value for the specified column represents a currency value.

isDefinitelyWritable()

```
public boolean isDefinitelyWritable(int column)
  throws SQLException
```

Description: Returns true if a write operation on the column will definitely succeed.

isNullable()

```
public int isNullable(int column) throws SQLException
```

Description: Returns true if null values are allowed for the column.

isReadOnly()

```
public boolean isReadOnly(int column) throws SQLException
```

Description: Returns true if the column is read-only.

isSearchable()

```
public boolean isSearchable(int column) throws SQLException
```

Description: Returns true if the column may be used in a WHERE clause.

isSigned()

```
public boolean isSigned(int column) throws SQLException
```

Description: Returns true if the column contains a signed number.

isWritable()

```
public boolean isWritable(int column) throws SQLException
```

Description: Returns true if it is possible for a write on a column to succeed.

Copyright © 2001 O'Reilly & Associates, Inc.

Statement

Synopsis

Class Name:java.sql.Statement
Superclass:None
Immediate Subclasses:java.sql.PreparedStatement
Interfaces Implemented:None
Availability:JDK 1.1

Description

This class represents an embedded SQL statement and is used by an application to perform database access. The closing of a Statement automatically closes any open ResultSet associated with the Statement.

Class Summary

```
public interface Statement {
  void addBatch(String sql) throws SQLException;
  void cancel() throws SQLException;
  void clearBatch() throws SQLException;
  void clearWarnings() throws SQLException;
  void close() throws SQLException;
  boolean execute(String sql) throws SQLException;
  int[] executeBatch() throws SQLException;
  ResultSet executeQuery(String sql)
    throws SQLException;
  int executeUpdate(String sql) throws SQLException;
  Connection getConnection() throws SQLException;
  int getFetchDirection() throws SQLException;
  int getFetchSize() throws SQLException;
  int getMaxFieldSize() throws SQLException;
  int getMaxRows() throws SQLException;
  boolean getMoreResults() throws SQLException;
  int getQueryTimeout() throws SQLException;
  ResultSet getResultSet() throws SQLException;
  int getResultSetConcurrency() throws SQLException;
  int getResultSetType() throws SQLException;
  int getUpdateCount() throws SQLException;
  SQLWarning getWarnings() throws SQLException;
  void setCursorName(String name) throws SQLException;
  void setEscapeProcessing(boolean enable)
    throws SQLException;
  void setFetchDirection(int dir) throws SQLException;
  void setFetchSize(int rows) throws SQLException;
  void setMaxFieldSize(int max) throws SQLException;
  void setMaxRows(int max) throws SQLException;
  void setQueryTimeout(int seconds)
    throws SQLException;
}
```

Copyright © 2001 O'Reilly & Associates, Inc.

Object Methods

addBatch()

public void addBatch(String sql) throws SQLException

Description: Adds the specified SQL statement to the current set of batch commands.

cancel()

public void cancel() throws SQLException

Description: In a multithreaded environment, you can use this method to indicate that any processing for this Statement should be canceled. In this respect, it is similar to the stop() method for Thread objects.

clearBatch()

public void clearBatch() throws SQLException

Description: Clears out any batch statements.

clearWarnings() and getWarnings()

public void clearWarnings() throws SQLException
public SQLWarning getWarnings() throws SQLException

Description: The clearWarnings() method allows you to clear all warnings from the warning chain associated with this class. The getWarnings() method retrieves the first warning on the chain. You can retrieve any subsequent warnings on the chain using that first warning.

close()

public void close() throws SQLException

Description: Manually closes the Statement. This is generally not required because a Statement is automatically closed whenever the Connection associated with it is closed.

execute(), executeQuery(), and executeUpdate()

public boolean execute(String sql) throws SQLException
public ResultSet executeQuery(String sql) throws SQLException
public int executeUpdate(String sql) throws SQLException

Description: Executes the Statement by passing the specified SQL to the database. The first method, execute(), allows you to execute the Statement when you do not know if it is a query or an update. It will return true if the statement has result sets to process.

The executeQuery() method is used for executing queries. It returns a result set for processing.

The executeUpdate() statement is used for executing updates. It returns the number of rows affected by the update.

executeBatch()

public int[] executeBatch(String sql) throws SQLException

Copyright © 2001 O'Reilly & Associates, Inc.

Description: Submits the batched list of SQL statements to the database for execution. The return value is an array of numbers that describe the number of rows affected by each SQL statement.

getConnection()

```
public Connection getConnection() throws SQLException
```

Description: Returns the Connection object associated with this Statement.

getFetchDirection(), setFetchDirection(), getFetchSize(), and setFetchSize()

```
public int getFetchDirection() throws SQLException
public void setFetchDirection(int dir) throws SQLException
public int getFetchSize() throws SQLException
public void setFetchSize(int rows) throws SQLException
```

Description: These methods provide optimization hints for the driver. The driver is free to ignore these hints. The fetch size is the suggested number of rows the driver should prefetch for each time it grabs data from the database. The direction is a hint to the driver about the direction in which you intend to work.

getMaxFieldSize() and setMaxFieldize()

```
public int getMaxFieldSize() throws SQLException
public void setMaxFieldSize(int max) throws SQLException
```

Description: These methods support the maximum field size attribute that determines the maximum amount of data for any BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR column value. If the limit is exceeded, the excess is silently discarded.

getMaxRows() and setMaxRows()

```
public int getMaxRows() throws SQLException
public void setMaxRows(int max) throws SQLException
```

Description: This attribute represents the maximum number of rows a ResultSet can contain. If this number is exceeded, then any excess rows are silently discarded.

getMoreResults()

```
public boolean getMoreResults() throws SQLException
```

Description: This method moves to the next result and returns true if that result is a ResultSet. Any previously open ResultSet for this Statement is then implicitly closed. If the next result is not a ResultSet or if there are no more results, this method will return false. You can test explicitly for no more results using:

```
(!getMoreResults() && (getUpdateCount() == -1)
```

getQueryTimeout() and setQueryTimeout()

```
public int getQueryTimeout() throws SQLException
public void setQueryTimeout(int seconds) throws SQLException
```

Description: This attribute is the amount of time a driver will wait for a Statement to execute. If the limit is exceeded, an SQLException is thrown.

Copyright © 2001 O'Reilly & Associates, Inc.

getResultSet()

public ResultSet getResultSet() throws SQLException

Description: This method returns the current ResultSet. You should call this only once per result. You never need to call this for executeQuery() calls that return a single result.

getResultSetConcurrency()

public int getResultSetConcurrency() throws SQLException

Description: Returns the concurrency for the result sets generated by this Statement.

getResultSetType()

public int getResultSetType() throws SQLException

Description: Returns the result set type for any result sets generated by this Statement.

getUpdateCount()

public int getUpdateCount() throws SQLException

Description: If the current result was an update, this method returns the number of rows affected by the update. If the result is a ResultSet or if there are no more results, –1 is returned. As with getResultSet(), this method should only be called once per result.

getWarnings()

public SQLWarning getWarnings() throws SQLException

Description: Retrieves the first warning associated with this object.

setCursorName()

public void setCursorName(String name) throws SQLException

Description: This method specifies the cursor name to be used by subsequent Statement executions. For databases that support positioned updates and deletes, you can then use this cursor name in coordination with any ResultSet objects returned by your execute() or executeQuery() calls to identify the current row for a positioned update or delete. You must use a different Statement object to perform those updates or deletes. This method does nothing for databases that do not support positioned updates or deletes.

setEscapeProcessing()

```
public void setEscapeProcessing(boolean enable)
    throws SQLException
```

Description: Escape processing is on by default. When enabled, the driver will perform escape substitution before sending SQL to the database.

Copyright © 2001 O'Reilly & Associates, Inc.

Struct

Synopsis

Class Name:java.sql.Struct
Superclass:None
Immediate Subclasses:None
Interfaces Implemented:None
Availability:New as of JDK 1.2

Description

This class maps to a SQL3 structured type. A Struct instance has values that map to each of the attributes in its associated structured value in the database.

Class Summary

```
public interface Struct {
  Object[] getAttributes() throws SQLException;
  Object[] getAttributes(Map map) throws SQLException;
  String getSQLTypeName() throws SQLException;
}
```

Object Methods

getAttributes()

```
public Object[] getAttributes() throws SQLException
public Object[] getAttributes(Map map) throws SQLException
```

Description: Provides the values for the attributes in the SQL structured type in order. If you pass a type map, it will use that type map to construct the Java values.

getSQLTypeName()

```
public String getSQLTypeName() throws SQLException
```

Description: Provides the SQL type name for this structured type.

Time

Synopsis

Class Name:java.sql.Time
Superclass:java.util.Date
Immediate Subclasses:None
Interfaces Implemented:None
Availability:JDK 1.1

Description

This version of the java.util.Date class maps to an SQL TIME datatype.

Copyright © 2001 O'Reilly & Associates, Inc.

Class Summary

```
public class Time extends java.util.Date {
   static public Time valueOf(String s);
   public Time(int hour, int minute, int second);
   public Time(long time);
   #public int getDate();
   #public int getDay();
   #public int getMonth();
   #public int getYear();
   #public int setDate(int i);
   #public int setMonth(int i);
   public void setTime(long time);
   #public void setYear(int i);
   public String toString();
}
```

Object Constructors

Time()

```
public Timestamp(int hour, int minute, intsecond)
public Timestamp(long time)
```

Description: Constructs a new Time object. The first prototype constructs a Time for the hour, minute, and seconds specified. The second constructs one based on the number of seconds since 12:00:00 January 1, 1970 GMT.

Object Methods

getDate(), setDate(), getDay(), getMonth(), setMonth(), getYear(), and setYear()

```
#public int getDate()
#public int getDay()
#public int getMonth()
#public int getYear()
#public int setDate(int i)
#public int setMonth(int i)
#public void setYear(int i)
```

Description: These attributes represent the individual segments of a Time object.

setTime()

```
public void setTime(long time)
```

Description: This method sets the Time object to the specified time as the number of seconds since 12:00:00 January 1, 1970 GMT.

toString()

```
public String toString()
```

Description: Formats the Time into a String in the form of hh:mm:ss.

valueOf()

```
static public Timestamp valueOf(String s)
```

Description: Create a new Time based on a String in the form of hh:mm:ss.

Copyright © 2001 O'Reilly & Associates, Inc.

Timestamp

Synopsis

Class Name:java.sql.Timestamp
Superclass:java.util.Date
Immediate Subclasses:None
Interfaces Implemented:None
Availability:JDK 1.1

Description

This class serves as an SQL representation of the Java Date class specifically designed to serve as an SQL TIMESTAMP. It also provides the ability to hold nanoseconds as required by SQL TIMESTAMP values. You should keep in mind that this class uses the java.util.Date version of hashcode(). This means that two timestamps that differ only by nanoseconds will have identical hashcode() return values.

Class Summary

```
public class Timestamp extends java.util.Date {
  static public Timestamp valueOf(String s);
  #public Timestamp(int year, int month, int date,
    int hour, int minute, int second, int nano);
  public Timestamp(long time);
  public boolean after(Timestamp t);
  public boolean before(Timestamp t);
  public boolean equals(Timestamp t);
  public int getNanos();
  public void setNanos(int n);
  public String toString();
}
```

Object Constructors

Timestamp()

```
#public Timestamp(int year, int month, int date, int hour, int minute,
    int second, int nano)
public Timestamp(long time)
```

Description: Constructs a new Timestamp object. The first prototype constructs a Timestamp for the year, month, date, hour, minute, seconds, and nanoseconds specified. The second prototype constructs one based on the number of seconds since 12:00:00 January 1, 1970 GMT.

Object Methods

after()

```
public boolean after(Timestamp t)
```

Description: Returns true if this Timestamp is later than the argument.

Copyright © 2001 O'Reilly & Associates, Inc.

before()

```
public boolean before(Timestamp t)
```

Description: Returns true if this Timestamp is earlier than the argument.

equals()

```
public boolean equals(Timestamp t)
```

Description: Returns true if the two timestamps are equivalent.

getNanos() and setNanos()

```
public int getNanos()
public void setNanos(int n)
```

Description: This attribute represents the number of nanoseconds for this Timestamp.

toString()

```
public String toString()
```

Description: Formats the Timestamp into a String in the form of yyyy-mm-dd hh:mm:ss.fffffffff.

valueOf()

```
static public Timestamp valueOf(String s)
```

Description: Creates a new Timestamp based on a String in the form of yyyy-mm-dd hh:mm:ss.fffffffff.

Types

Synopsis

Class Name:java.sql.Types
Superclass:java.lang.Object
Immediate Subclasses:None
Interfaces Implemented:None
Availability:JDK 1.1

Description

This class holds static attributes representing SQL data types. These values are the actual constant values defined in the XOPEN specification.

Class Summary

```
public class Types {
  static public final int ARRAY;
  static public final int BIGINT;
  static public final int BINARY;
  static public final int BIT;
  static public final int BLOB;
  static public final int CHAR;
  static public final int CLOB;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
    static public final int DATE;
    static public final int DECIMAL;
    static public final int DISTINCT;
    static public final int DOUBLE;
    static public final int FLOAT;
    static public final int INTEGER;
    static public final int JAVA_OBJECT;
    static public final int LONGVARBINARY;
    static public final int LONGVARCHAR;
    static public final int NULL;
    static public final int NUMERIC;
    static public final int OTHER;
    static public final int REAL;
    static public final int REF;
    static public final int SMALLINT;
    static public final int STRUCT;
    static public final int TIME;
    static public final int TIMESTAMP;
    static public final int TINYINT;
    static public final int VARBINARY;
    static public final int VARCHAR;
}
```

Copyright © 2001 O'Reilly & Associates, Inc.

Copyright © 2001 O'Reilly & Associates, Inc.