CHAPTER 3

A Tour of the MySQL Source Code

This chapter presents a complete introduction to the MySQL source, along with an explanation of how to obtain and build the system. I'll introduce you to the mechanics of the source code as well as coding guidelines and best practices for how to maintain the code. I'll focus on the parts of the code that deal with processing queries; this will set the stage for topics introduced in Chapter 7 and beyond.

Getting Started

In this section, I examine the principles behind modifying the MySQL source code and how you can obtain the source code. Let's begin with a review of the available licensing options.

Understanding the Licensing Options

When planning your modifications to open source software, consider how you're going to use those modifications. More specifically, how are you going to acquire the source code and work with it? Depending on your intentions for the modifications, your choices will be very different from others. There are three principal ways you may want to modify the source code:

- You may be modifying the source code to gain insight on how MySQL is constructed and therefore you are following the examples in this book or working on your own experiments.
- You may want to develop a capability for you or your organization that will not be distributed outside your organization.
- You may be building an application or extension that you plan to share or market to others.

In the first chapter I discussed the responsibilities of an open source developer modifying software under an open source license. Since MySQL uses the GPL and a commercial license (called a *dual license*), we must consider these uses of the source code under *both* licenses. I'll begin our discussion with the GPL.

Modifying the source code in a purely academic session is permissible under the GPL. The GPL clearly gives you the freedom to change the source code and experiment with it. The value of your contribution may contribute to whether your code is released under the GPL. For example, if your code modifications are considered singular in focus (they only apply to a limited set of users for a special purpose), the code may not be included in the source code base. In a similar way, if your code was focused on the exploration of an academic exercise, the code may not be of value to anyone other than yourself. Few at MySQL AB would consider an academic exercise in which you test options and features implemented in the source code as adding value to the MySQL system. On the other hand, if your experiments lead to a successful and meaningful addition to the system, most would agree you're obligated to share your findings. For the purposes of this book, you'll proceed with modifying the source code as if you will not be sharing your modifications. Although I hope that you find the experiments in this book enlightening and entertaining, I don't think they would be considered for adoption into the MySQL system without further development. If you take these examples and make something wonderful out of them, you have my blessing. Just be sure to tell everyone where you got the idea.

If you're modifying the MySQL source code for use by you or your organization and you do not want to share your modifications, you should purchase the appropriate MySQL Network support package. MySQL's commercial licensing terms give you the option of making the modifications (and even getting MySQL AB to help you) and keeping them to yourself.

Similarly, if you're modifying the source code and intend to distribute the modifications, you're required by the GPL to distribute the modified source code free of charge (but you may charge a media fee). Furthermore, your changes cannot be made proprietary and you cannot own the rights to the modifications under the GPL. If you choose not to publish your changes yourself, you should contribute the code to MySQL for incorporation into their products, at which point that code becomes the property of MySQL AB. On the other hand, if you want to make proprietary changes to MySQL for use in an embedded system or similar installation, you should contact MySQL AB and discuss your plans prior to launching your project. MySQL AB will work with you to come up with a solution that meets your needs and protects their interests.

Getting the Source Code

You can obtain the MySQL source code in one of two ways. You could use the source control application that MySQL uses (BitKeeper) and get the latest version, or you can download the code without ties to the source control application and obtain a copy of a specific version release. You should use the source control application if you want to make modifications that will be candidates for inclusion into the MySQL system. If you're making academic changes or changes you're not going to share, you should download the source code directly from MySQL AB either through the MySQL Network site or via the developer pages on the MySQL AB site.

■ **Tip** I recommend downloading the source code from `http://dev.mysql.com` for all cases except when you either want the latest version of the source code or want to contribute to the MySQL project.

WHAT IS SOURCE CONTROL?

Source control (also known as version or revision control, code repository, or source tree) is a mechanism that stores documents in a central location and tracks changes to those documents. The version control technology is represented as a tree structure (or a similar hierarchical view) and was originally designed for engineering drawings and word processing files. The technology also works for source code. In this case, the technology allows developers to store and retrieve source code, modify it, and resave it to the repository. This process is called *checking in* and *checking out*.

Not only does source control preserve the source code by managing the files, but it also allows tracking of changes to the files. Most source control applications allow diversions of the files to permit alternative modifications (called *branching*) and then resolve the conflicts at a later time (called *merging*). Source control therefore allows organizations to manage and track changes to the files in the repository. Source control is one of the many tools bundled in most configuration management tool suites. The source control used for the MySQL source code allows MySQL AB to permit many developers to work on the source code and make changes, then later manage which changes get placed in the final source build.

If you're using the MySQL Network licensing, you should contact a MySQL Network representative for assistance in choosing the correct version of the source code and location from which to download it. Your MySQL Network representative will also assign you a login and password for read-only access to the source code.

Using BitKeeper

Obtaining the source code from the source control application involves using a program called BitKeeper (see `www.bitkeeper.com` for more details). BitKeeper is a configuration management suite that permits developers to store and share source code and documents in a distributed environment (over the Internet).

■ **Caution** BitKeeper stores the very latest version, forks (branches), and all other development artifacts for the MySQL source code. It is what the MySQL AB developers use on a daily basis to store their revisions to the code. As such, it isn't always in the most stable of states. Use caution when choosing this method. Most of the new features will be incomplete or in some stage of refinement. If you have to have a stable build, use the code snapshots (described later in this chapter) or a release of the source code.

The first thing you need to do is remember the old adage about patience. This process can be a bit frustrating as it is very error prone. Although it seems to work well for most people, some users have had trouble getting and using the BitKeeper client. If you stick to my instructions, you shouldn't have any problems.

Installing BitKeeper

What you'll need to do is download the free BitKeeper client. This client is only available for POSIX-compatible Unix systems, but can be run from Windows clients if you use Cygwin. See the sidebar "Using BitKeeper on Windows Platforms" for information on obtaining and using Cygwin on Windows. Once the client is installed, you can point it to the MySQL repository and download the code. The entire process takes only a few minutes to complete on a broadband connection. Slower connection speeds will see significant delays as the entire source tree is downloaded. To download the BitKeeper client, open your browser and go to `www.bitkeeper.com/Hosted.html`.

On this page you'll find a link for downloading the client. Click on `Download the client` and save the `bk-client.shar` file in your home folder (or one of your choosing). The client is not built so you must make the executables. Do this using these commands:

```
%> /bin/sh bk-client.shar
%> cd bk_client-1.1
%> make
```

If your system is configured correctly and you have `gcc` and `make` installed, you should see a successful compilation of the BitKeeper client. Getting the source tree is easy. Just enter the following command:

```
sfioball -r+ bk://mysql.bkbits.net/mysql-5.1-new mysql-5.1
```

This command instructs the BitKeeper client to connect to the MySQL source tree named `mysql-5.1-new` and download the files into a new folder named `mysq-5.1`. You should see a long list of messages indicating that the source code is being transferred to your system. When the transfer is complete, you'll have the most recent copy of the MySQL source code. You're now ready to start exploring.

■ **Tip** A list of all of the available MySQL source trees can be found at `http://mysql.bkbits.net`.

The instructions I have presented will only allow you to get a copy of the source tree; they don't permit you to update the repository with your changes. To do that, you must have a license key and permission to update the tree. If you want to pursue this option, you must download a copy of the commercial BitKeeper software. The process is detailed here:

1. Go to the BitKeeper site (`www.bitkeeper.com`) and click on `Downloads`.
2. Click on the link `evaluation and download form`.
3. Fill out the form with your request, checking the Eval Key and Download Instructions option. Be sure to include your justification for the request as well as a brief description of what you intend to do with the source code.
4. If your request is granted, you will receive an e-mail from BitKeeper detailing the steps for downloading and installing the commercial BitKeeper client.

Fortunately, the commercial BitKeeper client is available as a GUI on many platforms. They even offer a client that interfaces with Visual Studio. From here, you should use the client documentation to learn how to synchronize and update the source tree.

USING BITKEEPER ON WINDOWS PLATFORMS

Using the free BitKeeper client on Windows is a bit tricky. To do so, you must first download and install Cygwin. Cygwin is a Linux-like environment that permits you to compile and run Linux programs on Windows platforms (NT, XP, etc.). Once Cygwin is installed, you can download the free BitKeeper client using the instructions I gave you earlier and create your source tree copy. Follow these steps on Windows to download, install, and use the free BitKeeper client:

1. Download the Cygwin `setup.exe` executable from `www.cygwin.com`. You should see a link for `Install or update now!`.
2. Follow the onscreen instructions and leave the default installation folders (trust me, it's easier that way). Be sure to install `gcc`, `make` (located inside the `Devel` package on the Select Packages screen during setup), and all of the development install packages.
3. Download the BitKeeper client from `www.bitkeeper.com/Hosted.Downloading.html`.
4. Save the file in the `c:\cygwin\home\`*`username`*`\` folder (where *`username`* represents your home directory name).
5. Open a Cygwin command window (the installer placed a shortcut on your desktop).
6. Enter the command `sh bk-client.shar`.
7. Change the working directory using `cd bk_client-1.1`.
8. Use WordPad to open a file named `makefile` in the `c:\cygwin\home\`*`username`*`\bk_client-1.1` folder. Change the line that reads `$(CC) $(CFLAGS) -o -sfio -lz -sfio.c` to `$(CC) $(CLFAGS) -o sfio sfio.c -lz`. Note: do not remove the tab character before the `$`!
9. Compile the client using `make all`. Note: if this step fails, see the Caution on modifying the source to overcome an error with the `getline` function.
10. Change your path to the current folder (or use the referencing directives in the next steps) using `PATH=$PWD:$PATH`.
11. If you want to place the source tree somewhere other than the current directory, navigate there now.
12. Copy the source tree to your folder using `sfioball -r+ bk://mysql.bkbits.net/mysql-5.1 /home/`*`username`*`/mysql-5.1`.

You should now have a full copy of the source tree copied to your Windows client. Unfortunately, what you have is probably not going to be very Windows friendly. MySQL AB provides the Windows source code Visual Studio project files as a courtesy. As such, they are often not created until just before or soon after the source code build is released to the public (GA). However, you can still compile the code on Windows if you have a fully functional GNU development environment. I find it easier to use the GA source code whenever I explore the source code on Windows.

■ **Caution** The BitKeeper client did not compile correctly on my Windows machine and it turns out to be a common problem for many. If this happens to you, you may have to modify the source code in a file named `sfioball.c`. The problem on my Windows client was that the function named `getline` was already defined in my copy of the `stdio.h` include file. Fixing this error is really easy. Simply do a search and replace `getline` with `getline_fix`. Then try the `make all` command again and you should have success. This and similar silly errors are the things that make life difficult for folks new to using BitKeeper, `gcc`, Cygwin, and the other tools necessary to get a copy of the source tree on Windows.

MySQL recommends that you update your copy of the source tree periodically. Once you have established a copy of the source tree, updating it is easy. Simply start your command window (or Cygwin command window on Windows), navigate to the BitKeeper folder, and enter the command

```
update bk://mysql.bkbits.net/mysql-5.1 mysql-5.1
```

■ **Caution** This command may copy over any files you may have altered. See the BitKeeper web site for more details.

The free BitKeeper client permits you to examine the change log (what has changed since the last update) and the change history for any or all files in the source tree. You can open the file named `BK/ChangeLog` in the source tree and examine its contents for the history of the changes. Look for the section titled "ChangeSet." You'll find information on what file was changed when and the e-mail address of the person who changed it. This information is interesting as it gives you an opportunity to contact the developer who last worked on the file if you have any questions. MySQL AB is eager to hear from you, especially if you have suggestions for improvements or if you find new and better ways to code something.

■ **Tip** If you use Windows, you may need to generate the Visual Studio project files and solution file. Check in the directory `win` and read the `README` file for the latest information about how to generate these files.

Downloading the Source

Obtaining the source code for download without using the source control application is easy. MySQL AB posts the latest source code for its products on their web site (`http://dev.mysql.com/downloads`).

When you go to that site, you'll see information about the two licenses of the MySQL products. The open source GPL products are called "MySQL Community Edition" and the commercial license products are called "MySQL Network." For use with this book, you need the MySQL

Community Edition. If you scroll down a bit, you'll see that MySQL AB offers three sets of links for the Community Edition:

- The current release (also called the generally available or GA) for production use
- Upcoming releases (e.g., alpha, beta—see Chapter 1 for more details on the types of releases MySQL AB offers)
- Older releases of the software

Also on this page are links to the many supporting applications, including the database connectors, administrative tools, and much more.

You can also download the source code using source code snapshots. The snapshots are usually alpha, development, or GA releases. The beta release is normally available on the main page. Use the source code snapshot if you want the latest look at a new feature or if you want to keep up to date by using the latest available stable release but don't want or need to use the code repository. (*Stable* in this case means the system has been tested and no extraordinary bugs have been found.)

For the purpose of following the examples in this book, you should download version 5.1.7 or higher from the web site. I provide instructions for installing MySQL in the next section. The site contains all of the binaries and source code for all of the environments supported. Notice that many different platforms are supported. You'll find the source code located near the bottom of the page. Be sure to download both the source code and the binaries (two downloads) for your platform. In this book, I'll use examples from both Red Hat Linux Fedora Core 5 and Microsoft Windows XP Professional.

■ **Tip** If you're using Windows, be sure to download the file containing all of the binaries or code, not the "essentials" packages. The smaller packages may not include some of the folders shown in the next section.

OS/2 SUPPORT

As of this writing, discussions were under way concerning removing OS/2 support from version 5.1. It is unlikely OS/2 will continue to be supported by MySQL AB. Various posts on the Planet MySQL blog (`www.planetmysql.org`) indicated that OS/2 support may be provided via variants of the source code contributed by the global community of developers.

■ **Note** Unless otherwise stated, the examples in this book are taken from the Linux source code distribution (`mysql-5.1.7-beta.tar.gz`). While most of the code is the same for Linux and Windows distributions, I will highlight differences as they occur. Most notably, the Windows platform has a slightly different `vio` implementation.

The MySQL Source Code

Once you have downloaded the source code, unpack the files into a folder on your system. You can unpack them into the same directory if you want. When you do this, notice that there are a lot of folders and many source files. The main folder you'll need to reference is the `/sql` folder. This folder contains the main source files for the server. Table 3-1 lists the most commonly accessed folders and their contents.

Table 3-1. *MySQL Source Folders*

Folder	Contents
`/BUILD`	The compilation configuration and make files for all platforms supported. Use this folder for compilation and linking.
`/client`	The MySQL command-line client tool.
`/dbug`	Utilities for use in debugging (see Chapter 5 for more details).
`/Docs`	Documentation for the current release. Linux users should use `generate-text-files.pl` in the `support` subfolder to generate the documentation. Windows users are provided with a `manual.chm` file.
`/include`	The base system include files and headers.
`/libmysql`	The C client API used for creating embedded systems. (See Chapter 6 for more details.)
`/libmysqld`	The core server API files. Also used in creating embedded systems. (See Chapter 6 for more details.)
`/mysql-test`	The MySQL system test suite. (See Chapter 4 for more details.)
`/mysys`	The majority of the core operating system API wrappers and helper functions.
`/regex`	A regular expression library. Used in the query optimizer and execution to resolve expressions.
`/scripts`	A set of shell script-based utilities.
`/sql`	The main system code. You should start your exploration from this folder.
`/sql-bench`	A set of benchmarking utilities.
`/SSL`	A set of Secure Socket Layer utilities and definitions.
`/storage`	The MySQL pluggable storage engine source code is located inside this folder. Also included is the storage engine example code. (See Chapter 7 for more details.)
`/strings`	The core string handling wrappers. Use these for all of your string handling needs.
`/support-files`	A set of preconfigured configuration files for compiling with different options.
`/tests`	A set of test programs and test files.
`/vio`	The network and socket layer code.
`/zlib`	Data compression tools.

I recommend taking some time now to dig your way through some of the folders and acquaint yourself with the location of the files. You will find many makefiles and a variety of Perl scripts dispersed among the folders. While not overly simplistic, the MySQL source code is logically organized around the functions of the source code rather than the subsystems. Some subsystems, like the storage engines, are located in a folder hierarchy, but most are located in several places in the folder structure. For each subsystem discussed while examining the source code, I will list the associated source files and their locations.

Getting Started

The best way to understand the flow and control of the MySQL system is to follow the source code along from the standpoint of a typical query. I presented a high-level view of each of the major MySQL subsystems in Chapter 2. I'll use the same subsystem view now as I show you how a typical SQL statement is executed. The following is the sample SQL statement I'll use:

```
SELECT lname, fname, DOB FROM Employees WHERE Employees.department = 'EGR';
```

This query selects the names and date of birth for everyone in the engineering department. While not very interesting, the query will be useful in demonstrating almost all of the subsystems in the MySQL system. Let's begin with the query arriving at the server for processing.

Figure 3-1 shows the path the example query would take through the MySQL source code. I have pulled out the major lines of code that you should associate with the subsystems identified in Chapter 2. Although not part of a specific subsystem, the `main()` function is responsible for initializing the server and setting up the connection listener. The `main()` function is in the file `/sql/mysqld.cc`.

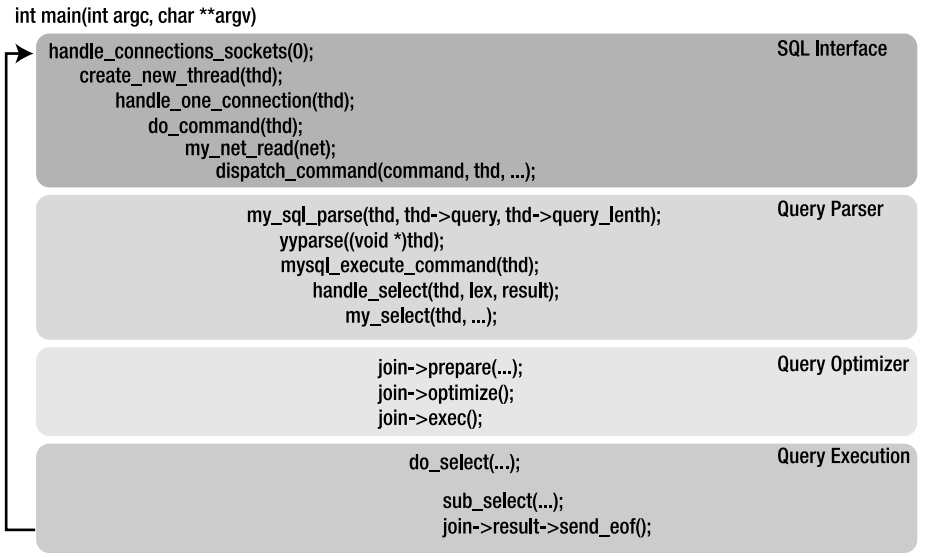

Figure 3-1. *Overview of the query path*

The path of the query begins in the SQL Interface subsystem (like most of the MySQL subsystems, the SQL Interface functions are distributed over a loosely associated set of source files). I'll tell you which files the methods are in as you go through this and the following sections. The `handle_connections_socket()` method (located in `/sql/mysqld.cc`) implements the listener loop, creating a thread for every connection detected. Once the thread is created, control flows to the `handle_one_connection()` function. The `handle_one_connection()` function identifies the command, then passes control to the `do_command` switch (located in `/sql/sql_parse.cc`). The `do_command` switch routes control to the proper network reading calls to read the query from the connection and passes the query to the parser via the `dispatch_command()` function (located in `/sql/sql_parse.cc`).

The query passes to the query parser subsystem, where the query is parsed and routed to the correct portion of the optimizer. The query parser is built in with Lex and YACC. Lex is used to identify tokens and literals as well as syntax of a language. YACC is used to build the code to interact with the MySQL source code. It captures the SQL commands storing the portions of the commands in an internal query representation and routes the command to a command processor called `mysql_execute_command()` (somewhat misnamed). This method then routes the query to the proper subfunction, in this case, `my_select()`. These methods are located in `/sql/sql_parse.cc`. This portion of the code enters the SELECT-PROJECT parts of the SELECT-PROJECT-JOIN query optimizer.

■ **Tip** A project is a relational database term describing the query operation that limits the result set to those columns defined in the column list on a SQL command. For example, the SQL command `SELECT fname, lname FROM employee` would "project" only the `fname` and `lname` columns from the employee table to the result set.

It is at this point that the query optimizer is invoked to optimize the execution of the query via the `join->prepare()` and `join->optimize()` functions. Query execution occurs next, with control passing to the lower-level `do_select()` function that carries out the restrict and projection operations. Finally, the `sub_select()` function invokes the storage engine to read the tuples, process them, and return results to the client. These methods are located in `/sql/sql_select.cc`. After the results are written to the network, control returns to the `hand_connections_sockets` loop (located in `/sql/mysqld.cc`).

■ **Tip** Classes, structures, classes, structures—it's all about classes and structures! Keep this in mind while you examine the MySQL source code. For just about any operation in the server, there is at least one class or structure that either manages the data or drives the execution. Learning the commonly used MySQL classes and structures is the key to understanding the source code, as you'll see in the "Important Classes and Structures" section later in this chapter.

You may be thinking that the code isn't as bad as you may have heard. That is largely true for simple `SELECT` statements like the example I am using, but as you'll soon see it can become

more complicated than that. Now that you have seen this path and have had an introduction to where some of the major functions fall in the path of the query and the subsystems, you should open the source code and look for those functions. You can begin your search in `/sql/mysqld.cc` (`/sql/mysqld.cpp` for Windows source code).

■ **Tip** The Windows source code often has different file extensions for the source files. Most times you can simply substitute `.cpp` for `.cc` to find the equivalent Windows source code file. I'll point out any differences between the Linux and Windows files in cases where this rule does not hold.

OK, so that was a whirlwind introduction, yes? From this point on, I'll slow things down a bit (OK, a lot) and navigate the source code in more detail. I'll also list the specific source files where the examples reside in the form of a table at the end of each section. So tighten those safety belts, we're going in!

I'll leave out sections that are not relevant to our tour. These sections could include conditional compilation directives, ancillary code, and other system-level calls. I'll annotate the missing sections with the following: `...`. I have left many of the original comments in place as I believe they will help you follow the source code and offer you a glimpse into the world of developing a world-class database system. Finally, I'll highlight the important parts of the code in bold so you can find them more easily while reading.

The main() Function

The `main()` function is where the server begins execution. It is the first function called when the server executable is loaded into memory. Several hundred lines of code in this function are devoted to operating system–specific startup tasks, and there's a good amount of system-level initialization code. Listing 3-1 shows a condensed view of the code, with the essential points in bold.

Listing 3-1. *The main() Function*

```
int main(int argc, char **argv)
{
  ...

  if (init_common_variables(MYSQL_CONFIG_NAME,
          argc, argv, load_default_groups))

  ...

  if (init_server_components())

  ...
```

```
  /*
   Initialize my_str_malloc() and my_str_free()
  */
  my_str_malloc= &my_str_malloc_mysqld;
  my_str_free= &my_str_free_mysqld;

  ...

  if (acl_init(opt_noacl) ||
      my_tz_init((THD *)0, default_tz_name, opt_bootstrap))

  ...

  create_shutdown_thread();
  create_maintenance_thread();

  ...

  handle_connections_sockets(0);

  ...

  (void) pthread_mutex_lock(&LOCK_thread_count);

  ...

  (void) pthread_mutex_unlock(&LOCK_thread_count);

  ...
}
```

The first interesting function is `init_common_variables()`. This function uses the command-line arguments to control how the server will perform. This is where the server interprets the arguments and starts the server in a variety of modes. This function takes care of setting up the system variables and places the server in the desired mode. The `init_server_components()` function initializes the database logs for use by any of the subsystems. These logs are the typical logs you see for events, statement execution, and so on.

I want to identify two of the most important `my_` library functions: `my_str_malloc()` and `my_str_free()`. It is as this point in the server startup code (near the beginning) that these two function pointers are set. You should always use these functions in place of the traditional C/C++ `malloc()` functions because the MySQL functions have additional error handling and therefore are safer than the base methods. The `acl_init()` function's job is to start the authentication and access control subsystem. This is a key system and appears early in the server startup code.

Now you're getting to what makes MySQL tick: threads. Two important helper threads are created. The `create_shutdown_thread()` function creates a thread whose job is to shut down the server on signal, and the `create_maintenance_thread()` function creates a thread to handle

any server-wide maintenance functions. I discuss threads in more detail in the "Process vs. Thread" sidebar.

At this point in the startup code, the system is just about ready to accept connections from clients. To do that, the `handle_connections_sockets(0)` function implements a listener that loops through the code waiting for connections. I'll discuss this function in more detail next.

The last thing I want to point out to you in the code is an example of the critical section protection code for mutually exclusive access during multithreading. A critical section is a block of code that must execute as a set and can only be accessed by a single thread at a time. Critical sections are usually areas that write to a shared memory variable and therefore must complete before another thread attempts to read the memory. MySQL AB has created an abstract of a common concurrency protection mechanism called a *mutex* (short for mutually exclusive). If you find an area in your code that you need to protect during concurrent execution, you can use the following functions to protect the code.

The first function you should call is `pthread_mutex_lock([resource reference])`. This function places a lock on the code execution at this point in the code. It will not permit another thread to access the memory location specified until your code calls the unlocking function `pthread_mutex_unlock([resource reference])`. In the example from the `main()` function, the mutex calls are locking the thread count global variable.

Well, that's your first dive under the hood. How did it feel? Do you want more? Keep reading—you've only just begun. In fact, you haven't seen where our example query enters the system. Let's do that next.

PROCESS VS. THREAD

The terms *process* and *thread* are often used interchangeably. This is incorrect as a *process* is an organized set of computer instructions that has its own memory and execution path. A *thread* is also a set of computer instructions, but threads execute in a host's execution path and do not have their own memory. (Some call threads lightweight processes. While a good description, calling them lightweight processes doesn't help the distinction.) They do store state (in MySQL, it is via the `THD` class). Thus, when talking about large systems that support processes, I mean systems that permit sections of the system to execute as a separate process and have their own memory. When talking about large systems that support threads, I mean systems that permit sections of the system to execute concurrently with other sections of the system and they all share the same memory space as the host.

Most database systems use the process model to manage concurrent connections and helper functions. MySQL uses the multithreaded model. There are a number of advantages to using threads over processes. Most notably, threads are easier to create and manage (no overhead for memory allocation and segregation). Threads also permit very fast switching because no context switching takes place. However, threads do have one severe drawback. If things go *wonky* (a highly technical term used to describe strange, unexplained behavior; in the case of threading, they are often very strange and harmful events) during a thread's execution, it is likely that if the trouble is severe, the entire system could be affected. Fortunately, MySQL AB and the global community of developers have worked very hard making MySQL's threading subsystem robust and reliable. This is why it is important for your modifications to be thread safe.

Handling Connections and Creating Threads

You saw in the previous section how the system is started and how the control flows to the listener loop that waits for user connections. The connections begin life at the client and are broken down into data packets, placed on the network by the client software, then flow across the network communications pathways where they are picked up by the server's network subsystems and reformed into data on the server. (A complete description of the communication packets is available in the MySQL Internals Manual.) This flow can be seen in Figure 3-2. I'll show you more details on the network communication methods in the next chapter. I'll also include examples of how to write code that returns results to the client using these functions.

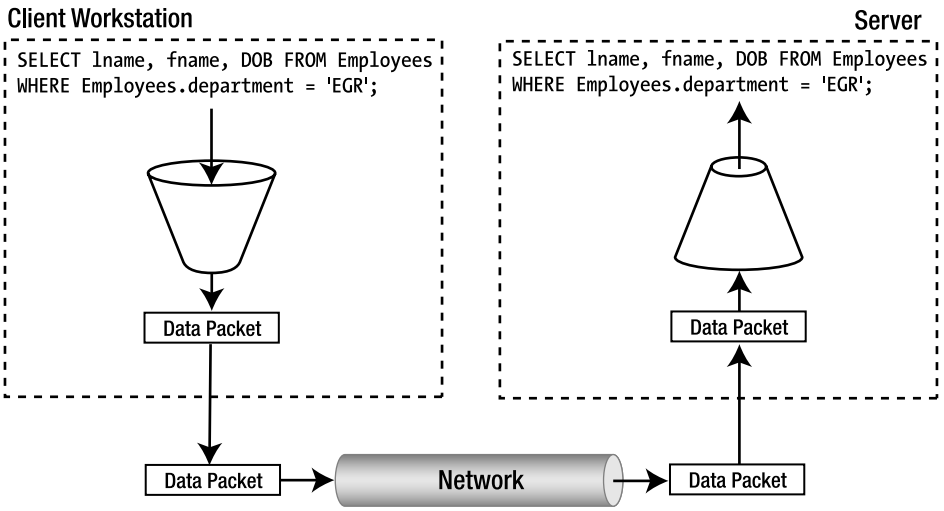

Figure 3-2. *Network communications from client to server*

At this point the system is in the SQL interface subsystem. That is, the data packets (containing the query) have arrived at the server and are detected via the `handle_connections_sockets()` function. This function enters a loop that waits until the variable `abort_loop` is set to `TRUE`. Table 3-2 shows the location of the files that manage the connection and threads.

Table 3-2. *Connections and Thread Management*

Source File	Description
`/sql/net_serv.cc`	Contains all of the network communications functions. Look here for information on how to communicate with the client or server via the network.
`/include/mysql_com.h`	Contains most of the structures used in communications.
`/sql/sql_parse.cc`	Contains the majority of the query routing and parsing functions except for the lexical parser.
`/sql/mysqld.cc`	Besides the main and server startup functions, this file also contains the methods for creating threads.

Listing 3-2 offers a condensed view of the connection-handling code. When a connection is detected (I've hidden that part of the code as it isn't helpful in learning how the system works), the function creates a new thread calling the aptly named `create_new_thread()` function. It is in this function that the first of the major structures is created. The `THD` class is responsible for maintaining all of the information for the thread. Although not allocated to the thread in a private memory space, the `THD` class allows the system to control the thread during execution. I'll expose some of the `THD` class in a later section.

Listing 3-2. *The Handle Connections Sockets Functions*

```
pthread_handler_t handle_connections_sockets(void *arg __attribute__((unused)))
{

  ...

  DBUG_PRINT("general",("Waiting for connections."));

  ...

  while (!abort_loop)
  {

  ...

    /*
    ** Don't allow too many connections
    */

    if (!(thd= new THD))

  ...

    if (sock == unix_sock)
      thd->security_ctx->host=(char*) my_localhost;

  ...

    create_new_thread(thd);
  }

  ...
}
```

OK, so now the client has connected to the server. What happens next? Let's see what happens inside the `create_new_thread()` function. Listing 3-3 shows a condensed view of the `create_new_thread()` function. The first thing you see is the mutex call to lock the thread count. As you saw in the `main()` function, this is necessary to keep other threads from potentially

competing for write access to the variable. When the thread is created, the associated unlock mutex call is made to unlock the resource.

Listing 3-3. *The create_new_thread() Function*

```
static void create_new_thread(THD *thd)
{

  ...

  pthread_mutex_lock(&LOCK_thread_count);

  ...

    if (cached_thread_count > wake_thread)
    {
      start_cached_thread(thd);
    }
    else
    {
      int error;
      thread_count++;
      thread_created++;
      threads.append(thd);
      if (thread_count-delayed_insert_threads > max_used_connections)
        max_used_connections=thread_count-delayed_insert_threads;
      DBUG_PRINT("info",(("creating thread %d"), thd->thread_id));
      thd->connect_time = time(NULL);
      if ((error=pthread_create(&thd->real_id,&connection_attrib,
        handle_one_connection,
        (void*) thd)))
      {
         DBUG_PRINT("error",
           ("Can't create thread to handle request (error %d)",
           error));

  ...

      }
    }
    (void) pthread_mutex_unlock(&LOCK_thread_count);
  }
  DBUG_PRINT("info",("Thread created"));

  ...
}
```

A very interesting thing occurs early in the function. Notice the `start_cached_thread()` function call. That function is designed to reuse a thread that may be residing in the connection pool. This helps speed things up a bit as creating threads, while faster than creating processes, can take some time to complete. Having a thread ready to go is a sort of caching mechanism for connections. The saving of threads for later use is called a *connection pool*.

If there isn't a connection (thread) ready for to reuse, the system creates one with the `pthread_create()` function call. Something really strange happens here. Notice the third parameter for this function call. What seems like a variable is actually the starting address of a function (a function pointer). `pthread_create()` uses this function pointer to associate the location in the server where execution should begin for the thread.

Now that the query has been sent from the client to the server and a thread has been created to manage the execution, control passes to the `handle_one_connection()` function. Listing 3-4 shows a condensed view of the `handle_one_connection()` function. In this view, I have commented out a large section of the code that deals with initializing the `THD` class for use. If you're interested, I encourage you to take a look at the code more closely later (located in `/sql/mysqld.cc`). For now, let's look at the essential work that goes on inside this function.

Listing 3-4. *The handle_one_connection() Function*

```
pthread_handler_t handle_one_connection(void *arg)
{
  THD *thd=(THD*) arg;

  ...

    while (!net->error && net->vio != 0 &&
           !(thd->killed == THD::KILL_CONNECTION))
    {
      net->no_send_error= 0;
      if (do_command(thd))
        break;
    }

  ...
}
```

In this case, the only function call of interest for our exploration is the `do_command(thd)` function. It is inside a loop that is looping once for each command read from the networking communications code. Although somewhat of a mystery at this point, this is of interest to those of us who have entered stacked SQL commands (more than one command on the same line). As you see here, this is where MySQL handles that eventuality. For each command read, the function passes control to the function that begins reads in the query from the network.

It is at this point where the system reads the query from the network and places it in the `THD` class for parsing. This takes place in the `do_command()` function. Listing 3-5 shows a condensed view of the `do_command()` function. I have left some of the more interesting comments and code bits in to demonstrate the robustness of the MySQL source code.

Listing 3-5. *The do_command() Function*

```
bool do_command(THD *thd)
{
  char *packet;
  uint old_timeout;
  ulong packet_length;
  NET *net;
  enum enum_server_command command;

  ...

  packet=0;

  ...

  net_new_transaction(net);
  if ((packet_length=my_net_read(net)) == packet_error)
  {
    DBUG_PRINT("info",("Got error %d reading command from socket %s",
           net->error,
           vio_description(net->vio)));

  ...

  }
  else
  {
    packet=(char*) net->read_pos;
    command = (enum enum_server_command) (uchar) packet[0];
    if (command >= COM_END)
      command= COM_END;        // Wrong command

  ...

  }
  net->read_timeout=old_timeout;    // restore it
  /*
    packet_length contains length of data, as it was stored in packet
    header. In case of malformed header, packet_length can be zero.
    If packet_length is not zero, my_net_read ensures that this number
    of bytes was actually read from network. Additionally my_net_read
    sets packet[packet_length]= 0 (thus if packet_length == 0,
    command == packet[0] == COM_SLEEP).
    In dispatch_command packet[packet_length] points beyond the end of packet.
  */
  DBUG_RETURN(dispatch_command(command,thd, packet+1, (uint) packet_length));
}
```

The first thing to notice is the creation of a packet buffer and a NET structure. This packet buffer is a character array and stores the raw query string as it is read from the network and stored in the NET structure. The next item that is created is a command structure, which will be used to route control to the appropriate parser functions. The my_net_read() function reads the packets from the network and stores them in the NET structure. The length of the packet is also stored in the packet_length variable of the NET structure. The last thing you see occurring in this function is a call to dispatch_command(), the point at which you can begin to see how commands are routed through the server code.

OK, so now you're starting to get somewhere. The job of the dispatch_command() function is to route control to a portion of the server that can best process the incoming command. Since you have a normal SELECT query on the way, the system has identified it as a query by setting the command variable to COM_QUERY. Other command types are used to identify statements, change user, generate statistics, and many other server functions. For this chapter, I will only look at query commands (COM_QUERY). Listing 3-6 shows a condensed view of the function. I have omitted the code for all of the other commands in the switch for the sake of brevity (I'm omitting the comment break too) but I'm leaving in the case statements for most of the commands. Take a moment and scan through the list. Most of the names are self-explanatory. If you were to conduct this exploration for another type of query, you could find your way by looking in this function for the type identified and following the code along in that case statement. I have also included the large function comment block that appears before the function code. Take a moment to look at that. I'll be getting more into that later in this chapter.

Listing 3-6. *The dispatch_command() Function*

```
/*
   Perform one connection-level (COM_XXXX) command.

  SYNOPSIS
    dispatch_command()
    thd             connection handle
    command         type of command to perform
    packet          data for the command, packet is always null-terminated
    packet_length   length of packet + 1 (to show that data is
                    null-terminated) except for COM_SLEEP, where it
                    can be zero.
  RETURN VALUE
    0   ok
    1   request of thread shutdown, i. e. if command is
        COM_QUIT/COM_SHUTDOWN
*/

bool dispatch_command(enum enum_server_command command, THD *thd,
          char* packet, uint packet_length)
{

  ...
```

```
switch (command) {
case COM_INIT_DB:
...
case COM_REGISTER_SLAVE:
...
case COM_TABLE_DUMP:
...
case COM_CHANGE_USER:
...
case COM_STMT_EXECUTE:
...
case COM_STMT_FETCH:
...
case COM_STMT_SEND_LONG_DATA:
...
case COM_STMT_PREPARE:
...
case COM_STMT_CLOSE:
...
case COM_STMT_RESET:
...
case COM_QUERY:
{
  if (alloc_query(thd, packet, packet_length))
    break;          // fatal error is set

...

  general_log_print(thd, command, "%s", thd->query);

...

  mysql_parse(thd,thd->query, thd->query_length);

...
}
case COM_FIELD_LIST:        // This isn't actually needed
...
case COM_QUIT:
...
case COM_BINLOG_DUMP:
...
case COM_REFRESH:
...
```

```
  case COM_STATISTICS:
  ...
  case COM_PING:
  ...
  case COM_PROCESS_INFO:
  ...
  case COM_PROCESS_KILL:
  ...
  case COM_SET_OPTION:
  ...
  case COM_DEBUG:
  ...
  case COM_SLEEP:
  ...
  case COM_DELAYED_INSERT:
  ...
  case COM_END:
  ...
  default:
  ...
}
```

The first thing that happens when control passes to the `COM_QUERY` handler is the query is copied from the `packet` array to the `thd->query` member variable via the `alloc_query()` function. In this way, the thread now has a copy of the query, which will stay with it all through its execution. Notice also that the code writes the command to the general log. This will help with debugging system problems and query issues later on. The last function call of interest in Listing 3-6 is the `mysql_parse()` function call. It is at this point that the code can officially transfer from the SQL Interface subsystem to the Query Parser subsystem. As you can see, this distinction is one of semantics rather than syntax.

Parsing the Query

Finally, the parsing begins. This is the heart of what goes on inside the server when it processes a query. The parser code is located in a couple of places (like so much of the rest of the system). It isn't that hard to follow if you realize that while being highly organized, the code is not structured to match the architecture.

The function you're examining now is the `mysql_parse()` function (located in `/sql/sql_parse.cc`). Its job is to check the query cache for the results of a previously executed query that has the same result set, then pass control to the lexical parser, and finally route the command to the query optimizer. Listing 3-7 shows a condensed view of the `mysql_parse()` function.

Listing 3-7. *The mysql_parse() Function*

```
void mysql_parse(THD *thd, char *inBuf, uint length)
{

  ...

  if (query_cache_send_result_to_client(thd, inBuf, length) <= 0)
  {
    LEX *lex= thd->lex;

  ...

    if (!yyparse((void *)thd) && ! thd->is_fatal_error)
    {

  ...

      mysql_execute_command(thd);
      query_cache_end_of_result(thd);

  ...
    }
  ...
}
```

The first thing to notice is the call to the query cache. The query cache stores all of the most frequently requested queries complete with the results. If the query is already in the query cache, you're done! All that is left is to return the results to the client. No parsing, optimizing, or even executing is necessary. How cool is that?

For the sake of our exploration, let's assume the query cache does not contain a copy of the example query. In this case, the function creates a new `LEX` structure to contain the internal representation of the query. This structure is filled out by the Lex/YACC parser, shown in Listing 3-8.

Listing 3-8. *The SELECT Lex/YACC Parsing Code Excerpt*

```
select:
  select_init
  {
    LEX *lex= Lex;
    lex->sql_command= SQLCOM_SELECT;
  }
  ;
```

```
/* Need select_init2 for subselects. */
select_init:
  SELECT_SYM select_init2
  |
  '(' select_paren ')' union_opt;

select_paren:
  SELECT_SYM select_part2
    {
      LEX *lex= Lex;
            SELECT_LEX * sel= lex->current_select;
      if (sel->set_braces(1))
      {
        yyerror(ER(ER_SYNTAX_ERROR));
        YYABORT;
      }
      if (sel->linkage == UNION_TYPE &&
        !sel->master_unit()->first_select()->braces)
      {
        yyerror(ER(ER_SYNTAX_ERROR));
        YYABORT;
      }
            /* select in braces, can't contain global parameters */
      if (sel->master_unit()->fake_select_lex)
              sel->master_unit()->global_parameters=
                 sel->master_unit()->fake_select_lex;
  }
  | '(' select_paren ')';

select_init2:
  select_part2
    {
      LEX *lex= Lex;
      SELECT_LEX * sel= lex->current_select;
      if (lex->current_select->set_braces(0))
      {
         yyerror(ER(ER_SYNTAX_ERROR));
         YYABORT;
      }
      if (sel->linkage == UNION_TYPE &&
          sel->master_unit()->first_select()->braces)
      {
        yyerror(ER(ER_SYNTAX_ERROR));
        YYABORT;
      }
  }
```

```
  union_clause
  ;

select_part2:
  {
    LEX *lex= Lex;
    SELECT_LEX *sel= lex->current_select;
    if (sel->linkage != UNION_TYPE)
      mysql_init_select(lex);
    lex->current_select->parsing_place= SELECT_LIST;
  }
  select_options select_item_list
  {
    Select->parsing_place= NO_MATTER;
  }
  select_into select_lock_type;

select_into:
  opt_order_clause opt_limit_clause {}
        | into
  | select_from
  | into select_from
  | select_from into;

select_from:
    FROM join_table_list where_clause group_clause having_clause
         opt_order_clause opt_limit_clause procedure_clause
        | FROM DUAL_SYM where_clause opt_limit_clause
          /* oracle compatibility: oracle always requires FROM clause,
             and DUAL is system table without fields.
             Is "SELECT 1 FROM DUAL" any better than "SELECT 1" ?
          Hmmm :) */
  ;

select_options:
  /* empty*/
  | select_option_list
    {
      if (Select->options & SELECT_DISTINCT && Select->options & SELECT_ALL)
      {
        my_error(ER_WRONG_USAGE, MYF(0), "ALL", "DISTINCT");
              YYABORT;
      }
    }
    ;
```

```
select_option_list:
  select_option_list select_option
  | select_option;

select_option:
  STRAIGHT_JOIN { Select->options|= SELECT_STRAIGHT_JOIN; }
  | HIGH_PRIORITY
    {
      if (check_simple_select())
        YYABORT;
      Lex->lock_option= TL_READ_HIGH_PRIORITY;
    }
  | DISTINCT         { Select->options|= SELECT_DISTINCT; }
  | SQL_SMALL_RESULT { Select->options|= SELECT_SMALL_RESULT; }
  | SQL_BIG_RESULT { Select->options|= SELECT_BIG_RESULT; }
  | SQL_BUFFER_RESULT
    {
      if (check_simple_select())
        YYABORT;
      Select->options|= OPTION_BUFFER_RESULT;
    }
  | SQL_CALC_FOUND_ROWS
    {
      if (check_simple_select())
        YYABORT;
      Select->options|= OPTION_FOUND_ROWS;
    }
  | SQL_NO_CACHE_SYM { Lex->safe_to_cache_query=0; }
  | SQL_CACHE_SYM
    {
      Lex->select_lex.options|= OPTION_TO_QUERY_CACHE;
    }
  | ALL        { Select->options|= SELECT_ALL; }
  ;

select_lock_type:
  /* empty */
  | FOR_SYM UPDATE_SYM
    {
      LEX *lex=Lex;
      lex->current_select->set_lock_for_tables(TL_WRITE);
      lex->safe_to_cache_query=0;
    }
```

```
  | LOCK_SYM IN_SYM SHARE_SYM MODE_SYM
    {
      LEX *lex=Lex;
      lex->current_select->
        set_lock_for_tables(TL_READ_WITH_SHARED_LOCKS);
      lex->safe_to_cache_query=0;
    }
  ;

select_item_list:
    select_item_list ',' select_item
  | select_item
  | '*'
    {
      THD *thd= YYTHD;
      if (add_item_to_list(thd,
                                  new Item_field(&thd->lex->current_select->
                                                 context,
                                                 NULL, NULL, "*")))
        YYABORT;
      (thd->lex->current_select->with_wild)++;
    };


select_item:
    remember_name select_item2 remember_end select_alias
    {
      if (add_item_to_list(YYTHD, $2))
        YYABORT;
      if ($4.str)
      {
        $2->set_name($4.str, $4.length, system_charset_info);
              $2->is_autogenerated_name= FALSE;
      }
      else if (!$2->name) {
        char *str = $1;
        if (str[-1] == '`')
          str--;
        $2->set_name(str,(uint) ($3 - str), YYTHD->charset());
      }
    };
```

I have included an excerpt from the Lex/YACC parser that shows how the `SELECT` token is identified and passed through the YACC code to be parsed. The way you should read this code

(in case you don't know Lex or YACC) is to watch for the keywords (or tokens) in the code (they are located flush left with a colon like `select:`). These keywords are used to direct flow of the parser. The placement of tokens to the right of these keywords defines the order of what must occur in order for the query to be parsed. For example, look at the `select:` keyword. To the right of that you will see a `select_init2` keyword, which isn't very informative. However, if you look down through the code you will see the `select_init:` keyword on the left. This allows the Lex/YACC author to specify certain behaviors in a sort of macro-like form. Also notice that there are curly braces under the `select_init` keyword. This is where the parser does its work of dividing the query into parts and placing the items in the `LEX` structure. Direct symbols such as `SELECT` are defined in a header file (`/sql/lex.h`) and appear in the parser as `SELECT_SYM`. Take a few moments now to skim through the code. You may want to run through this several times. It can be confusing if you haven't studied compiler construction or text parsing.

If you're thinking, "What a monster," then you can rest assured that you're normal. The Lex/YACC code is a challenge for most developers. I've highlighted a few of the important code statements that should help explain how the code works. Let's go through it. I've repeated the example `SELECT` statement again here for convenience:

```
SELECT lname, fname, DOB FROM Employees WHERE Employees.department = 'EGR';
```

Look at the first keyword again. Notice how the `select_init` code block sets the `LEX` structure's `sql_command` to `SQLCOM_SELECT`. This is important because the next function in the query path uses this in a large switch statement to further control the flow of the query through the server. The example `SELECT` statement has three fields in the field list. Let's try and find that in the parser code. Look for the `add_item_to_list()` function call. That is where the parser detects the fields and places them in the `LEX` structure. You will also see a few lines up from that call the parser code that identifies the * option for the field list. OK, now you've got the `sql_command` member variable set and the fields identified. So where does the `FROM` clause get detected? Look for the code statement that begins with `FROM join_table_list where_clause`. This code is the part of the parser that identifies the `FROM` and `WHERE` clause (and others). The code for the parser that processes these clauses is not included in Listing 3-8, but I think you get the idea. If you open the `sql_yacc.yy` source file (located in `/sql`), you should now be able to find all of those statements and see how the rest of the `LEX` structure is filled in with the table list in the `FROM` clause and the expression in the `WHERE` clause.

■ **Note** Some Windows distributions do not include the `sql_yacc.yy` file. If you use Windows and do not find this file in the `/sql` directory, you will need to download the Linux source code, extract the file, and place it in the `/sql` directory.

I hope that this tour of the parser code has helped mitigate the shock and horror that usually accompanies examining this part of the MySQL system. I will return to this part of the system later on when I demonstrate how to add your own commands the MySQL SQL lexicon (see Chapter 8 for more details). Table 3-3 lists the source files associated with the MySQL parser.

Table 3-3. *The MySQL Parser*

Source File	Description
/sql/lex.h	The symbol table for all of the keywords and tokens supported by the parser
/sql/lex_symbol.h	Type definitions for the symbol table
/sql/lex_hash.h	Mapping of symbols to functions used in the parser
/sql/sql_lex.h	Definition of LEX structure
/sql/sql_lex.cc	Definition of Lex class
/sql/sql_yacc.yy	The Lex/YACC parser code
/sql/sql_parse.cc	Contains the majority of the query routing and parsing functions except for the lexical parser

■ **Caution** Do not edit the files sql_yacc.cc, sql_yacc.h, or lex_hash.h. These files are generated by other utilities. See Chapter 8 for more details.

Preparing the Query for Optimization

Although the boundary of where the parser ends and the optimizer begins is not clear from the MySQL documentation (there are contradictions), it is clear from the definition of the optimizer that the routing and control parts of the source code can be considered part of the optimizer. To avoid confusion, I am going to call the next set of functions the *preparatory* stage of the optimizer.

The first of these preparatory functions is the mysql_execute_command() function (located in /sql/sql_parse.cc). The name leads you to believe you are actually executing the query, but that isn't the case. This function performs much of the setup steps necessary to optimize the query. The LEX structure is copied and several variables are set to help the query optimization and later execution. You can see some of these operations in a condensed view of the function shown in Listing 3-9.

Listing 3-9. *The mysql_execute_command() Function*

```
bool mysql_execute_command(THD *thd)
{
  bool  res= FALSE;
  int result= 0;
  LEX  *lex= thd->lex;
  /* first SELECT_LEX (have special meaning for many of non-SELECTcommands) */
  SELECT_LEX *select_lex= &lex->select_lex;
  /* first table of first SELECT_LEX */
  TABLE_LIST *first_table= (TABLE_LIST*) select_lex->table_list.first;
```

```
  /* list of all tables in query */
  TABLE_LIST *all_tables;
  /* most outer SELECT_LEX_UNIT of query */
  SELECT_LEX_UNIT *unit= &lex->unit;
  /* Saved variable value */
  DBUG_ENTER("mysql_execute_command");
  thd->net.no_send_error= 0;

  ...

switch (lex->sql_command) {
  case SQLCOM_SELECT:
  {

  ...

    select_result *result=lex->result;

  ...

      res= check_access(thd,
      lex->exchange ? SELECT_ACL | FILE_ACL : SELECT_ACL,
      any_db, 0, 0, 0, 0);

  ...

    if (!(res= open_and_lock_tables(thd, all_tables)))
    {
      if (lex->describe)
      {
        /*
          We always use select_send for EXPLAIN, even if it's an EXPLAIN
          for SELECT ... INTO OUTFILE: a user application should be able
          to prepend EXPLAIN to any query and receive output for it,
          even if the query itself redirects the output.
        */

  ...

  query_cache_store_query(thd, all_tables);
  res= handle_select(thd, lex, result, 0);

  ...
}
```

There are a number of interesting things happening in this function. You will notice another switch statement that has as its cases the `SQLCOM` keywords. In the case of the example query, you saw the parser set the `lex->sql_command` member variable to `SQLCOM_SELECT`. I have included a

condensed view of that case statement for you in Listing 3-9. What I did not include is the many other SQLCOM case statements. This function is a very large function. Since it is the central routing function for query processing, it contains a case for every possible command. Consequently, the source code is tens of pages long.

Let's see what this case statement does. Notice the statement `select_result *result=➥ lex->result`. This statement creates a result class that will be used to hold the results of the query for later transmission to the client. If you scan down, you will see the `check_table_➥ access()` function. This function is called to check the access control list for the resources used by the query. If access is granted, the function calls the `open_and_lock_tables()` function, which opens and locks the tables for the query. I left part of the code concerning the DESCRIBE (EXPLAIN) command for you to examine.

■ **Note** Once when I was modifying the code I needed to find all of the locations of the EXPLAIN calls so that I could alter them for a specific need. I looked everywhere until I found them in the parser. There in the middle of the Lex/YACC code was a comment that said something to the effect that DESCRIBE was left over from an earlier Oracle compatibility issue and that the correct term was EXPLAIN. Comments are useful. . . if you can find them.

The next function call is a call to the query cache. The `query_cache_store_query()` function stores the SQL statement in the query. As you will see later, when the results are ready they too are stored in the query cache. Finally you see that the function calls another function called `handle_select()`. You may be thinking, "Didn't we just do the handle thing?"

The `handle_select()` is a wrapper for another function named `mysql_select()`. Listing 3-10 shows the complete code for the `handle_select()` function. Near the top of the listing is the `select_lex->next_select()` operation, which is checking for the UNION command that appends multiple SELECT results into a single set of results. Other than that, the code just calls the next function in the chain, `mysql_select()`. It is at this point that you are finally close enough to transition to the query optimizer subsystem. Table 3-4 lists the source files associated with the query optimizer.

■ **Note** This is perhaps the part of the code that suffers most from ill-defined subsystems. While the code is still very organized, the boundaries of the subsystems are fuzzy at this point in the source code.

Listing 3-10. *The handle_select() Function*

```
bool handle_select(THD *thd, LEX *lex, select_result *result,
                   ulong setup_tables_done_option)
{
  bool res;
  register SELECT_LEX *select_lex = &lex->select_lex;
  DBUG_ENTER("handle_select");
```

```
  if (select_lex->next_select())
    res= mysql_union(thd, lex, result, &lex->unit, setup_tables_done_option);
  else
  {
    SELECT_LEX_UNIT *unit= &lex->unit;
    unit->set_limit(unit->global_parameters);
    /*
      'options' of mysql_select will be set in JOIN, as far as JOIN for
      every PS/SP execution new, we will not need to reset this flag if
      setup_tables_done_option changed for next execution
    */
    res= mysql_select(thd, &select_lex->ref_pointer_array,
          (TABLE_LIST*) select_lex->table_list.first,
          select_lex->with_wild, select_lex->item_list,
          select_lex->where,
          select_lex->order_list.elements +
          select_lex->group_list.elements,
          (ORDER*) select_lex->order_list.first,
          (ORDER*) select_lex->group_list.first,
          select_lex->having,
          (ORDER*) lex->proc_list.first,
          select_lex->options | thd->options |
                      setup_tables_done_option,
          result, unit, select_lex);
  }
  DBUG_PRINT("info",("res: %d  report_error: %d", res,
         thd->net.report_error));
  res|= thd->net.report_error;
  if (unlikely(res))
  {
    /* If we had another error reported earlier then this will be ignored */
    result->send_error(ER_UNKNOWN_ERROR, ER(ER_UNKNOWN_ERROR));
    result->abort();
  }
  DBUG_RETURN(res);
}
```

Table 3-4. *The Query Optimizer*

Source File	Description
/sql/sql_parse.cc	The majority of the parser code resides in this file
/sql/sql_select.cc	Contains some of the optimization functions and the implementation of the select functions
/sql/sql_parse.cc	Contains the majority of the query routing and parsing functions except for the lexical parser

Optimizing the Query

At last! You're at the optimizer. However, you won't find it if you go looking for a source file or class by that name. Although the `JOIN` class contains a method called `optimize()`, the optimizer is actually a collection of flow control and subfunctions designed to find the shortest path to executing the query. What happened to the fancy algorithms and query paths and compiled queries? Recall from our architecture discussion in Chapter 2 that the MySQL query optimizer is a nontraditional hybrid optimizer utilizing a combination of known best practices and cost-based path selection. It is at this point in the code that the best practices part kicks in.

An example of one of those best practices is standardizing the parameters in the `WHERE` clause expressions. The example query uses a `WHERE` clause with an expression, `Employees.department = 'EGR'`, but the clause could have been written as `'EGR' = Employees.department` and still be correct (it returns the same results). This is an example of where traditional cost-based optimizer could generate multiple plans—one for each of the expression variants. Just a few examples of the many best practices that MySQL uses follows:

- *Constant propagation*—The removal of transitive conjunctions using constants. For example, if you have `a=b='c'`, the transitive law states that `a='c'`. This optimization removes those inner equalities, thereby reducing the number of evaluations. For example, the SQL command `SELECT * FROM table1 WHERE column1 = 12 AND NOT (column3 = 17 OR column1 = column2)` would be reduced to `SELECT * FROM table1 WHERE column1 = 12 AND column3 <> 17 AND column2 <> 12`.

- *Dead code elimination*—The removal of always true conditions. For example, if you have `a=b AND 1=1`, the `AND 1=1` condition is removed. The same occurs for always false conditions where the false expression can be removed without affecting the rest of the clause. For example, the SQL command `SELECT * FROM table1 WHERE column1 = 12 AND column2 = 13 AND column1 < column2` would be reduced to `SELECT * FROM table1 WHERE column1 = 12 AND column2 = 13`.

- *Range queries*—The transformation of the `IN` clause to a list of disjunctions. For example, if you have an `IN (1,2,3)`, the transformation would be `a = 1 or a = 2 or a = 3`. This helps simplify the evaluation of the expressions. For example, the SQL command `SELECT * FROM table1 WHERE column1 = 12 OR column1 = 17 OR column1 = 21` would be reduced to `SELECT * FROM table1 WHERE column1 IN (12, 17, 21)`.

I hope this small set of examples has given you a glimpse into the inner workings of one of the world's most successful nontraditional query optimizers. In short, it works really well for a surprising amount of queries.

Well, I spoke too fast. There isn't much going on in the `mysql_select()` function in the area of optimization either. It seems the `mysql_select()` function just identifies joins and calls the `join->optimize()` function. Where are all of those best practices? They are in the `JOIN` class! A detailed examination of the optimizer source code in the `JOIN` class would take more pages than this entire book to present in any meaningful depth. Suffice to say that the optimizer is complex and also difficult to examine. Fortunately, few will ever need to venture that far down into the bowels of MySQL. However, you're welcome to! I will focus on a higher-level review of the `optimize()` function.

What you do see in the `optimize()` function is the definition of a local `JOIN` class with the code statement `JOIN *join`. The next thing you see is that the function checks to see if the

select_lex class already has a join class defined. Why? Because if you are executing another SELECT statement in a UNION or perhaps a reused thread from the connection pool, the select_lex class would already have been through this part of the code once and therefore we do not need to create another JOIN class. If there is no JOIN class in the select_lex class, a new one is created in the create statement join= new JOIN(). Finally, you see that the code calls the join->optimize() method.

However, once again you are at another fuzzy boundary. This time, it occurs in the middle of the mysql_select() function. The next major function call in this function is the join->exec() method. But first, let's take a look at what happens in the mysql_select() method in Listing 3-11. Table 3-5 lists the source files associated with query optimization.

Listing 3-11. *The mysql_select() Function*

```
bool mysql_select(THD *thd, Item ***rref_pointer_array,
       TABLE_LIST *tables, uint wild_num, List<Item> &fields,
       COND *conds, uint og_num,  ORDER *order, ORDER *group,
       Item *having, ORDER *proc_param, ulong select_options,
       select_result *result, SELECT_LEX_UNIT *unit,
       SELECT_LEX *select_lex)
{
  bool err;
  bool free_join= 1;
  DBUG_ENTER("mysql_select");

  select_lex->context.resolve_in_select_list= TRUE;
  JOIN *join;
  if (select_lex->join != 0)
  {
    join= select_lex->join;

  ...

    join->select_options= select_options;
  }
  else
  {
    if (!(join= new JOIN(thd, fields, select_options, result)))
  DBUG_RETURN(TRUE);

  ...

  }

  if ((err= join->optimize()))
  {
    goto err;
  }
```

```
  ...

  join->exec();

  ...
}
```

Table 3-5. *Query Optimization*

Source File	Description
/sql/sql_select.h	The definitions for the structures used in the select functions to support the SELECT commands
/sql/sql_select.cc	Contains some of the optimization functions and the implementation of the select functions

Executing the Query

In the same way as the optimizer, the query execution uses a set of best practices for executing the query. For example, the query execution subsystem detects special clauses like ORDER BY and DISTINCT and routes control of these operations to methods designed for fast sorting and tuple elimination.

Most of this activity occurs in the methods of the JOIN class. Listing 3-12 presents a condensed view of the join::exec() method. Notice that there is yet another function call to a function called by some name that includes select. Sure enough, there is another call that needs to be made to a function called do_select(). Take a look at the parameters for this function call. You are now starting to see things like field lists. Does this mean you're getting close to reading data? Yes, it does. In fact, the do_select() function is a high-level wrapper for exactly that.

Listing 3-12. *The join::exec() Function*

```
void JOIN::exec()
{
  List<Item> *columns_list= &fields_list;
  int      tmp_error;
  DBUG_ENTER("JOIN::exec");

  ...

  result->send_fields((procedure ? curr_join->procedure_fields_list :
                       *curr_fields_list),
                      Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF);
  error= do_select(curr_join, curr_fields_list, NULL, procedure);
  thd->limit_found_rows= curr_join->send_records;
  thd->examined_row_count= curr_join->examined_rows;
}
```

There is another function call that looks very interesting. Notice the code statement `result->send_fields()`. This function does what its name indicates. It is the function that sends the field headers to the client. As you can surmise, there are also methods to send the results to the client. I will look at these methods later in Chapter 4. Notice the `thd->limit_found_rows=` and `thd->examined_row_count=` assignments. These save record count values in the `THD` class. Let's take a look at that `do_select()` function.

You can see in the `do_select()` method shown in Listing 3-13 that something significant is happening. Notice the last highlighted code statement. The statement `join->result->send_eof()` looks like the code is sending an end-of-file flag somewhere. It is indeed sending an end-of-file signal to the client. So where are the results? They are generated in the `sub_select()` function. Let's look at that function next.

Listing 3-13. *The do_select() Function*

```
static int
do_select(JOIN *join,List<Item> *fields,TABLE *table,Procedure *procedure)
{
  int rc= 0;
  enum_nested_loop_state error= NESTED_LOOP_OK;
  JOIN_TAB *join_tab;
  DBUG_ENTER("do_select");

  ...

    error= sub_select(join,join_tab,0);

  ...

    if (join->result->send_eof())

  ...
}
```

Now you're getting somewhere! Take a moment to scan through Listing 3-14. This listing shows a condensed view of the `sub_select()` function. Notice that the code begins with an initialization of the `JOIN` class record. The `join_init_read_record()` function initializes any records available for reading in a structure named `JOIN_TAB` and populates the `read_record` member variable with another class named `READ_RECORD`. The `READ_RECORD` class contains the tuple read from the table. Inside this function are the abstraction layers to the storage engine subsystem. I will leave the discussion of the storage engine and how the system is used in a query until Chapter 7, where I present details on constructing your own storage engine. The system initializes the tables to begin reading records sequentially and then reads one record at a time until all of the records are read.

Listing 3-14. *The sub_select() Function*

```
enum_nested_loop_state
sub_select(JOIN *join,JOIN_TAB *join_tab,bool end_of_records)
{

  ...

  READ_RECORD *info= &join_tab->read_record;

  if (join->resume_nested_loop)
  {

  ...

  }
  else
  {

  ...

    join->thd->row_count= 0;

    error= (*join_tab->read_first_record)(join_tab);
    rc= evaluate_join_record(join, join_tab, error, report_error);
  }

  while (rc == NESTED_LOOP_OK)
  {
    error= info->read_record(info);
    rc= evaluate_join_record(join, join_tab, error, report_error);
  }

  ...
}
```

■ **Note** The code presented in Listing 3-14 is more condensed than the other examples I have shown. The main reason is this code uses a fair number of advanced programming techniques, such as recursion and function pointer redirection. However, the concept as presented is accurate for the example query.

Control returns to the `JOIN` class for evaluation of the expressions and execution of the relational operators. After the results are processed, they are transmitted to the client and then control returns to the `sub_select()` function, where the end-of-file flag is sent to tell the client there are no more results. Table 3-6 lists the source file associated with query execution.

Table 3-6. *Query Execution*

Source File	Description
`/sql/sql_select.cc`	Contains some of the optimization functions and the implementation of the select functions

I hope that this tour has satisfied your curiosity and if nothing else boosted your appreciation for the complexities of a real-world database system. Feel free to go back through this tour again until you're comfortable with the basic flow. I will discuss a few of the more important classes and structures in the next section.

Supporting Libraries

There are many additional libraries in the MySQL source tree. MySQL AB has long worked diligently to encapsulate and optimize many of the common routines used to access the supported operating systems and hardware. Most of these libraries are designed to render the code both operating system and hardware agnostic. These libraries make it possible to write code so that specific platform characteristics do not force you to write specialized code. Among these libraries are libraries for managing efficient string handling, hash tables, linked lists, memory allocation, and many others. Table 3-7 lists the purpose and location of a few of the more common libraries.

■ **Tip** The best way to discover if a library exists for a routine that you're trying to use is to look through the source code files in the `/mysys` directory using a text search tool. Most of the wrapper functions have a name similar to their original function. For example, `my_alloc.c` implements the `malloc` wrapper.

Table 3-7. *Supporting Libraries*

Source File	Utilities
`/mysys/array.c`	Array operations
`/mysys/hash.h and /mysys/hash.c`	Hash tables
`/mysys/list.c`	Linked lists
`/mysys/my_alloc.c`	Memory allocation
`/strings/*.c`	Base memory and string manipulation routines
`/mysys/string.c`	String operations
`/mysys/my_pthread.c`	Threading

Important Classes and Structures

Quite a few classes and structures in the MySQL source code can be considered key elements to the success of the system. To become fully knowledgeable about the MySQL source code, you should learn the basics of all of the key classes and structures used in the system. Knowing what is stored in which class or what the structures contain can help you make your modifications integrate well. The following sections describe these key classes and structures.

The ITEM_ Class

One class that permeates throughout the subsystems is the `ITEM_` class. I called it `ITEM_` because a number of classes are derived from the base `ITEM` class and even classes derived from those. These derivatives are used to store and manipulate a great many data (items) in the system. These include parameters (like in the `WHERE` clause), identifiers, time, fields, function, num, string, and many others. Listing 3-15 shows a condensed view of the `ITEM` base class. The structure is defined in the `/sql/item.h` source file and implemented in the `/sql/item.cc` source file. Additional subclasses are defined and implemented in files named after the data it encapsulates. For example, the function subclass is defined in `/sql/item_func.h` and implemented in `/sql/item_func.cc`.

Listing 3-15. *The ITEM_ Class*

```
class Item {
  Item(const Item &);      /* Prevent use of these */
  void operator=(Item &);
public:
  static void *operator new(size_t size)
  { return (void*) sql_alloc((uint) size); }
  static void *operator new(size_t size, MEM_ROOT *mem_root)
  { return (void*) alloc_root(mem_root, (uint) size); }
  static void operator delete(void *ptr,size_t size) { TRASH(ptr, size); }
  static void operator delete(void *ptr, MEM_ROOT *mem_root) {}

  enum Type {FIELD_ITEM= 0, FUNC_ITEM, SUM_FUNC_ITEM, STRING_ITEM,
       INT_ITEM, REAL_ITEM, NULL_ITEM, VARBIN_ITEM,
       COPY_STR_ITEM, FIELD_AVG_ITEM, DEFAULT_VALUE_ITEM,
       PROC_ITEM,COND_ITEM, REF_ITEM, FIELD_STD_ITEM,
       FIELD_VARIANCE_ITEM, INSERT_VALUE_ITEM,
             SUBSELECT_ITEM, ROW_ITEM, CACHE_ITEM, TYPE_HOLDER,
             PARAM_ITEM, TRIGGER_FIELD_ITEM, DECIMAL_ITEM,
             XPATH_NODESET, XPATH_NODESET_CMP,
             VIEW_FIXER_ITEM};

  ...
```

```
  /*
    str_values's main purpose is to be used to cache the value in
    save_in_field
  */
  String str_value;
  my_string name;      /* Name from select */
  /* Original item name (if it was renamed)*/
  my_string orig_name;
  Item *next;
  uint32 max_length;
  uint name_length;                     /* Length of name */
  uint8 marker, decimals;
  my_bool maybe_null;      /* If item may be null */
  my_bool null_value;      /* if item is null */
  my_bool unsigned_flag;
  my_bool with_sum_func;
  my_bool fixed;                        /* If item fixed with fix_fields */
  my_bool is_autogenerated_name;        /* indicate was name of this Item
                                           autogenerated or set by user */
  DTCollation collation;

  // alloc & destruct is done as start of select using sql_alloc
  Item();
  /*
     Constructor used by Item_field, Item_ref & aggregate (sum) functions.
     Used for duplicating lists in processing queries with temporary
     tables
     Also it used for Item_cond_and/Item_cond_or for creating
     top AND/OR structure of WHERE clause to protect it of
     optimisation changes in prepared statements
  */
  Item(THD *thd, Item *item);
  virtual ~Item()
  {
#ifdef EXTRA_DEBUG
    name=0;
#endif
  }    /*lint -e1509 */
  void set_name(const char *str, uint length, CHARSET_INFO *cs);
  void rename(char *new_name);
  void init_make_field(Send_field *tmp_field,enum enum_field_types type);
  virtual void cleanup();
  virtual void make_field(Send_field *field);
  Field *make_string_field(TABLE *table);

  ...
};
```

The LEX Structure

The LEX structure is responsible for being the internal representation (in-memory storage) of a query and its parts. It is more than that, though. The LEX structure is used to store all of the parts of a query in an organized manner. There are lists for fields, tables, expressions, and all of the parts that make up any query.

The LEX structure is filled in by the parser as it discovers the parts of the query. Thus, when the parser is done the LEX structure contains everything needed to optimize and execute the query. Listing 3-16 shows a condensed view of the LEX structure. The structure is defined in the /sql/lex.h source file.

Listing 3-16. *The LEX Structure*

```
typedef struct st_lex
{
  uint   yylineno,yytoklen;      /* Simulate lex */
  LEX_YYSTYPE yylval;
  SELECT_LEX_UNIT unit;                         /* most upper unit */
  SELECT_LEX select_lex;                        /* first SELECT_LEX */
  /* current SELECT_LEX in parsing */
  SELECT_LEX *current_select;
  /* list of all SELECT_LEX */
  SELECT_LEX *all_selects_list;
  const uchar *buf;    /* The beginning of string, used by SPs */
  const uchar *ptr,*tok_start,*tok_end,*end_of_query;

  /* The values of tok_start/tok_end as they were one call of yylex before */
  const uchar *tok_start_prev, *tok_end_prev;

  char *length,*dec,*change,*name;
  char *help_arg;
  char *backup_dir;        /* For RESTORE/BACKUP */
  char* to_log;                                 /* For PURGE MASTER LOGS TO */
  char* x509_subject,*x509_issuer,*ssl_cipher;
  char* found_semicolon;                        /* For multi queries - next query */
  String *wild;
  sql_exchange *exchange;
  select_result *result;
  Item *default_value, *on_update_value;
  LEX_STRING comment, ident;
  LEX_USER *grant_user;
  XID *xid;
  gptr yacc_yyss,yacc_yyvs;
  THD *thd;
  CHARSET_INFO *charset;
  TABLE_LIST *query_tables;  /* global list of all tables in this query */

  ...
} LEX;
```

The NET Structure

The NET structure is responsible for storing all of the information concerning communication to and from a client. Listing 3-17 shows a condensed view of the NET structure. The buff member variable is used to store the raw communication packets (that when combined form the SQL statement). As you will see in later chapters, there are helper functions that fill in, read, and transmit the data packets to and from the client. Two examples are

- my_net_write(),which writes the data packets to the network protocol from the NET structure
- my_net_read(), which reads the data packets from the network protocol into the NET structure

You can find the complete set of network communication functions in /include/mysql_com.h.

Listing 3-17. *The NET Structure*

```
typedef struct st_net {
#if !defined(CHECK_EMBEDDED_DIFFERENCES) || !defined(EMBEDDED_LIBRARY)
  Vio* vio;
  unsigned char *buff,*buff_end,*write_pos,*read_pos;
  my_socket fd;          /* For Perl DBI/dbd */
  unsigned long max_packet,max_packet_size;
  unsigned int pkt_nr,compress_pkt_nr;
  unsigned int write_timeout, read_timeout, retry_count;
  int fcntl;
  my_bool compress;
  /*
    The following variable is set if we are doing several queries in one
    command ( as in LOAD TABLE ... FROM MASTER ),
    and do not want to confuse the client with OK at the wrong time
  */
  unsigned long remain_in_buf,length, buf_length, where_b;
  unsigned int *return_status;
  unsigned char reading_or_writing;
  char save_char;
  my_bool no_send_ok;  /* For SPs and other things that do multiple stmts */
  my_bool no_send_eof; /* For SPs' first version read-only cursors */
  /*
    Set if OK packet is already sent, and we do not need to send error
    messages
  */
  my_bool no_send_error;
  /*
    Pointer to query object in query cache, do not equal NULL (0) for
    queries in cache that have not stored its results yet
  */
#endif
```

```
  char last_error[MYSQL_ERRMSG_SIZE], sqlstate[SQLSTATE_LENGTH+1];
  unsigned int last_errno;
  unsigned char error;
  gptr query_cache_query;
  my_bool report_error; /* We should report error (we have unreported error) */
  my_bool return_errno;
} NET;
```

The THD Class

In the preceding tour of the source code, you saw many references to the `THD` class. In fact, there is exactly one `THD` object for every connection. The thread class is paramount to successful thread execution and is involved in every operation from implementing access control to returning results to the client. As a result, the `THD` class shows up in just about every subsystem or function that operates within the server. Listing 3-18 shows a condensed view of the `THD` class. Take a moment and browse through some of the member variables and methods. As you can see, this is a large class (I've omitted a great many of the methods). The class is defined in the `/sql/sql_class.h` source file and implemented in the `/sql/sql_class.cc` source file.

Listing 3-18. *The THD Class*

```
class THD :public Statement,
           public Open_tables_state
{
public:

  ...

  String  packet;      // dynamic buffer for network I/O
  String  convert_buffer;              // buffer for charset conversions
  struct  sockaddr_in remote;    // client socket address
  struct  rand_struct rand;    // used for authentication
  struct  system_variables variables;  // Changeable local variables
  struct  system_status_var status_var; // Per thread statistic vars
  THR_LOCK_INFO lock_info;             // Locking info of this thread
  THR_LOCK_OWNER main_lock_id;         // To use for conventional queries
  THR_LOCK_OWNER *lock_id;             // If not main_lock_id, points to
                                       // the lock_id of a cursor.
  pthread_mutex_t LOCK_delete;    // Locked before thd is deleted

  ...

  char   *db, *catalog;
  Security_context main_security_ctx;
  Security_context *security_ctx;

  ...
```

```
  enum enum_server_command command;
  uint32     server_id;
  uint32     file_id;      // for LOAD DATA INFILE

  ...

  const char *where;
  time_t     start_time,time_after_lock,user_time;
  time_t     connect_time,thr_create_time; // track down slow pthread_create
  thr_lock_type update_lock_default;
  delayed_insert *di;

  ...

  table_map  used_tables;

  ...

  ulong      thread_id, col_access;

  ...

  inline time_t query_start() { query_start_used=1; return start_time; }
  inline void  set_time()    { if (user_time) start_time=time_after_lock=user_time;
                               else time_after_lock=time(&start_time); }
  inline void  end_time()    { time(&start_time); }
  inline void  set_time(time_t t) { time_after_lock=start_time=user_time=t; }

  ...
};
```

Now that you have had a tour of the source code and have examined some of the important classes and structures used in the system, I will shift the focus to items that will help you implement your own modifications to the MySQL system. Let's take a break from the source code and consider the coding guidelines and documentation aspects of software development.

Coding Guidelines

If the source code I've described seems to have a strange format, it may be because you have a different style than the authors of the source code. Consider the case where there are many developers writing a large software program like MySQL, each with their own style. As you can imagine, the code would quickly begin to resemble a jumbled mass of statements. To avoid this, MySQL AB has published coding guidelines in various forms. However, as you will see when you begin exploring the code yourself, it seems there are a few developers who aren't following the coding guidelines. The only plausible explanation is that the guidelines have changed over time, which can happen over the lifetime of a large project. Regardless of the

reasons why the guidelines are not being followed, most developers do adhere to the guidelines. More importantly, MySQL AB expects you to follow them.

The coding guidelines are included in the MySQL Internals Manual available online at `http://dev.mysql.com/doc`. Chapter 2 of the internals document lists all of the coding guidelines as a huge bulleted list containing the do's and don'ts of writing C/C++ code for the MySQL server. I have captured the most important guidelines and summarized them for you in the following paragraphs.

General Guidelines

One of the most stressed aspects of the guidelines is that you should write code that is as optimized as possible. This goal is counter to agile development methodologies, where you code only what you need and leave refinement and optimization to refactoring. If you develop using agile methodologies, you may want to wait to check in your code until you have refactored it.

Another very important overall goal is to avoid the use of direct API or operating system calls. You should always look in the associated libraries for wrapper functions. Many of these functions are optimized for fast and safe execution. For example, you should never use the C `malloc()` function. Instead, use the `sql_alloc()` or `my_alloc()` function.

All lines of code must be fewer than 80 characters long. If you need to continue a line of code onto another line, you should align the code so that parameters are aligned vertically or the continuation code is aligned with the indention space count.

Comments are written using the standard C-style comments, for example, `/* this is a comment */`. You should use comments liberally through your code.

■ **Tip** Resist the urge to use the C++ `// comment` option. The MySQL coding guidelines specifically discourage this technique.

Documentation

The language of choice for the source code is English. This includes all variables, function names, constants, and comments. The developers who write and maintain the MySQL source code are located throughout Europe and the United States. The choice of English as the default language in the source code is largely due to the influence of American computer science developments. English is also taught as a second language in many primary and secondary education programs in many European countries.

When writing functions, you should use a comment block that describes the function, its parameters, and the expected return values. The content of the comment block should be written in sections, with section names in all caps. You should include a short descriptive name of the function on the first line after the comment and, at a minimum, include the sections, synopsis, description, and return value. You may also include optional sections such as WARNING, NOTES, SEE ALSO, TODO, ERRORS, and REFERENCED_BY. The sections and content are described here:

- *SYNOPSIS* (required)—Presents a brief overview of the flow and control mechanisms in the function. It should permit the reader to understand the basic algorithm of the function. This helps readers understand the function and provide an at-a-glance glimpse of what it does. This section also includes a description of all of the parameters (indicated by IN for input, OUT for output, and IN/OUT for referenced parameters whose values may be changed).
- *DESCRIPTION* (required)—A narrative of the function. It should include the purpose of the function and a brief description of its use.
- *RETURN VALUE* (required)—Presents all of the possible return values and what they mean to the caller.
- *WARNING*—Include this section to describe any unusual side effects that the caller should be aware of.
- *NOTES*—Include this section to provide the reader with any information you feel is important.
- *SEE ALSO*—Include this section when you're writing a function that is associated with another function or requires specific outputs of another function or that is intended to be used by another function in a specific calling order.
- *TODO*—Include this section to communicate any unfinished features of the function. Be sure to remove the items from this section as you complete them. I tend to forget to do this and it often results in a bit of head scratching to figure out I've already completed the TODO item.
- *ERRORS*—Include this section to document any unusual error handling that your function has.
- *REFERENCED_BY*—Include this section to communicate specific aspects of the relationship this function has with other functions or objects—for example, whenever your function is called by another function, the function is a primitive of another function, or the function is a friend method or even a virtual method.

■ **Tip** MySQL AB suggests it isn't necessary to provide a comment block for short functions that have only a few lines of code, but I recommend writing a comment block for all of the functions you create. You will appreciate this advice as you explore the source code and encounter numerous small (and some large) functions with little or no documentation.

A sample of a function comment block is shown in Listing 3-19.

Listing 3-19. *Example Function Comment Block*

```
/*
  Find tuples by key.

  SYNOPSIS
    find_by_key()
    string key            IN     A string containing the key to find.
    Handler_class *handle IN     The class containing the table to be searched.
    Tuple *               OUT    The tuple class containing the key passed.

    Uses B Tree index contained in the Handler_class. Calls Index::find()
    method then returns a pointer to the tuple found.

  DESCRIPTION
    This function implements a search of the Handler_class index class to find
    a key passed.

  RETURN VALUE
    SUCCESS (TRUE)                Tuple found.
    != SUCCESS (FALES)            Tuple not found.

  WARNING
    Function can return an empty tuple when a key hit occurs on the index but
    the tuple has been marked for deletion.

  NOTES
    This method has been tested for empty keys and keys that are greater or
    less than the keys in the index.

  SEE ALSO
    Query:;execute(), Tuple.h

  TODO
    * Change code to include error handler to detect when key passed in exceeds
    the maximum length of the key in the index.

  ERRORS
    -1                             Table not found.
    1                              Table locked.

  REFERENCED_BY
    This function is called by the Query::execute() method.
*/
```

Functions and Parameters

I want to call these items out specifically because some inconsistencies exist in the source code. If you use the source code as a guide for formatting, you may wander astray of the coding guidelines. Functions and their parameters should be aligned so that the parameters are in vertical alignment. This applies to both defining the function and calling it from other code. In a similar way, variables should be aligned when you declare them. The spacing of the alignment isn't such an issue as the vertical appearance of these items. You should also add line comments about each of the variables. Line comments should begin in column 49 and not exceed the maximum 80-column rule. In the case where a comment for a variable exceeds 80 columns, you should place that comment on a separate line. Listing 3-20 shows examples of the type of alignment expected for functions, variables, and parameters.

Listing 3-20. *Variable, Function, and Parameter Alignment Examples*

```
int     var1;                                  /* comment goes here */
long    var2;                                  /* comment goes here too */
/* variable controls something of extreme interest and is documented well */
bool    var3;

return_value *classname::classmethod(int  var1,
                                     int  var2
                                     bool var3);

if (classname->classmethod(myreallylongvariablename1,
                           myreallylongvariablename2,
                           myreallylongvariablename3) == -1)
{
  /* do something */
}
```

■ **Warning** If you're developing on Windows, the line break feature of your editor may be set incorrectly. Most editors in Windows issue a CRLF (`/r/n`) when you place a line break in the file. MySQL AB requires you to use a single LF (`/n`), not a CRLF. This is a common incompatibility between files created on Windows versus files created in UNIX or Linux. If you're using Windows, check your editor and make the appropriate changes to its configuration.

Naming Conventions

MySQL AB prefers that you assign your variables meaningful names using all lowercase letters with underscores instead of initial caps. The exception is the use of class names, which are required to have initial caps. Enumerations should be prefixed with the phrase `enum_`. All structures and defines should be written with uppercase letters. Examples of the naming conventions are shown in Listing 3-21.

Listing 3-21. *Sample Naming Conventions*

```
class My_classname;
int   my_integer_counter;
bool  is_saved;

#define CONSTANT_NAME 12;

int my_function_name_goes_here(int variable1);
```

Spacing and Indenting

The MySQL coding guidelines state that spacing should always be two characters for each indention level. You should never use tabs. If your editor permits, you should change the default behavior of the editor to turn off automatic formatting and replace all tabs with two spaces. This is especially important when using documentation utilities like Doxygen (which I'll discuss in a moment) or line parsing tools to locate strings in the text.

When spacing between identifiers and operators, you should include no spaces between a variable and an operator and a single space between the operator and an operand (the right side of the operator). In a similar way, no space should follow the open parenthesis in functions, but include one space between parameters and no space between the last parameter name and the closing parenthesis. Lastly, you should include a single blank line to delineate variable declarations from control code, and control code from method calls, and block comments from other code, and functions from other declarations. Listing 3-22 depicts a properly formatted excerpt of code that contains an assignment statement, a function call, and a control statement.

Listing 3-22. *Spacing and Indention*

```
return_value= do_something_cool(i, max_limit, is_found);
if (return_value)
{
  int var1;
  int var2;

  var1= do_something_else(i);

  if (var1)
  {
    do_it_again();
  }
}
```

The alignment of the curly braces is also inconsistent in some parts of the source code. The MySQL coding guidelines state that the curly braces should align with the control code above it as I have shown in all of our examples. However, if you need to indent another level you should indent using the same column alignment as the code within the curly braces (two spaces). It is also not necessary to use curly braces if you're executing a single line of code in the code block.

An oddity of sorts in the curly braces area is the switch statement. A switch statement should be written to align the open curly brace after the switch condition and align the closing curly brace with the switch keyword. The case statements should be aligned in the same column as the switch keyword. Listing 3-23 illustrates this guideline.

Listing 3-23. *Switch Statement Example*

```
switch (some_var) {
case 1:
   do_something_here();
   do_something_else();
   break;
case 2:
   do_it_again();
   break;
}
```

■ **Note** The last `break` in the previous code is not needed. I usually include it in my code for the sake of completeness.

Documentation Utilities

Another useful method of examining source code is to use an automated documentation generator that reads the source code and generates function- and class-based lists of methods. These programs list the structures used and provide clues as to how and where they are used in the source code. This is important for investigating MySQL because of the many critical structures that the source code relies on to operate and manipulate data.

One such program is called Doxygen. The nice thing about Doxygen is that it too is open source and governed by the GPL. When you invoke Doxygen, it reads the source code and produces a highly readable set of HTML files that pull the comments from the source code preceding the function and lists the function primitives. Doxygen can read programming languages such as C, C++, and Java, among several others. Doxygen can be a useful tool for investigating a complex system such as MySQL—especially when you consider that the base library functions are called from hundreds of locations throughout the code.

Doxygen is available for both UNIX and Windows platforms. To use the program on Linux, download the source code from the Doxygen web site at `www.stack.nl/~dimitri/doxygen`.

Once you have downloaded the installation, follow the installation instructions (also on the web site). Doxygen uses configuration files to generate the look and feel of the output as well as what gets included in the input. To generate a default configuration file, issue the following command:

```
doxygen -g -s /path_to_new_file/doxygen_config_filename
```

The path specified should be the path you want to store the documentation in. Once you have a default configuration file, you can edit the file and change the parameters to meet your specific needs. See the Doxygen documentation for more information on the options and their parameters. You would typically specify the folders to process, the project name, and other project-related settings. Once you have set the configurations you want, you can generate documentation for MySQL by issuing this command:

```
doxygen </path_to_new_file/Doxygen_config_filename>
```

■ **Caution** Depending on your settings, Doxygen could run for a long time. Avoid using advanced graphing commands if you want Doxygen to generate documentation in a reasonable time period.

The latest version of Doxygen can be run from Windows using a supplied GUI. The GUI allows you to use create the configuration file using a wizard that steps you through the process and creates a basic configuration file, an expert mode that allows you to set your own parameters, and the ability to load a config file. I found the output generated by using the wizard interface sufficient for casual to in-depth viewing.

I recommend spending some time running Doxygen and examining the output files prior to diving into the source code. It will save you tons of lookup time. The structures alone are worth tacking up on the wall next to your monitor or pasting into your engineering logbook. A sample of the type of documentation Doxygen can generate is shown in Figure 3-3.

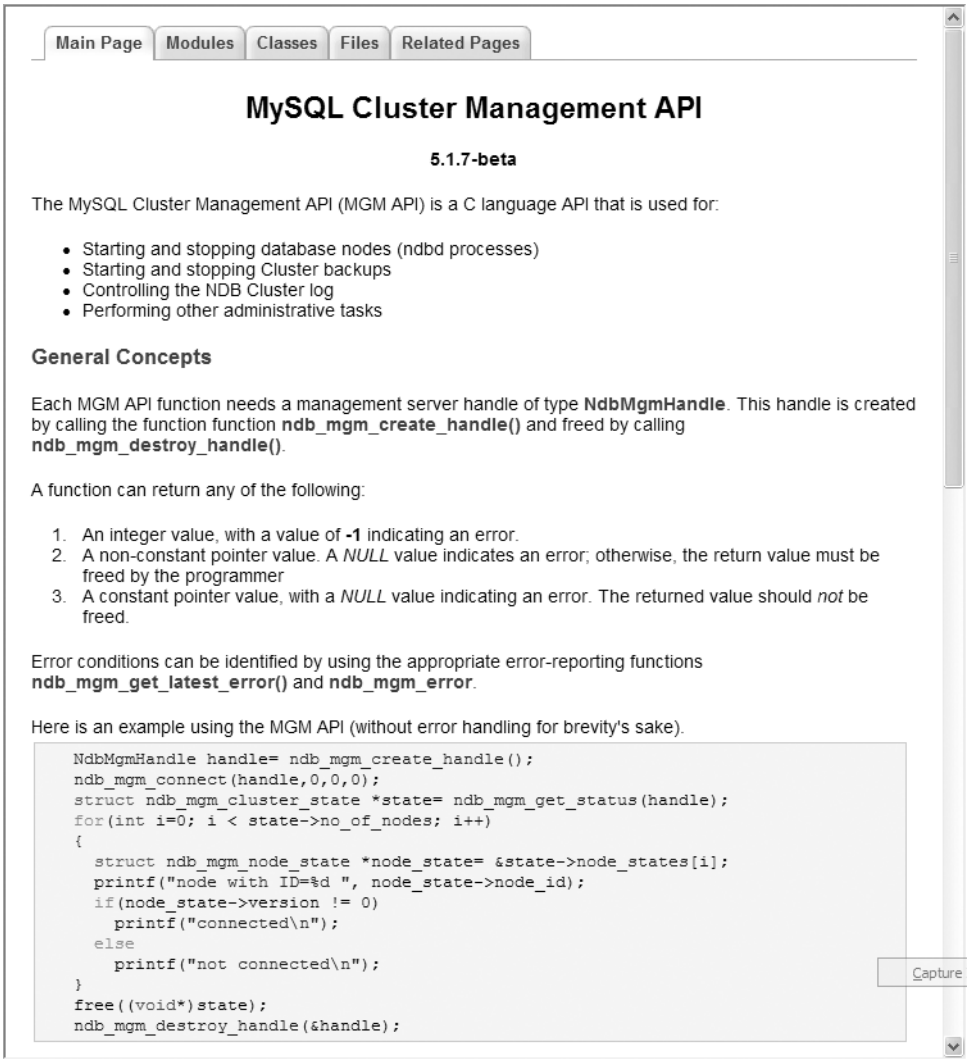

Figure 3-3. *Sample MySQL Doxygen output*

Keeping an Engineering Logbook

Many developers keep notes of their projects. Some are more detailed than others, but most take notes during meetings and phone conversations, thereby providing a written record for verbal communications. However, if you aren't in the habit of keeping an engineering logbook, you should consider doing so. I have found a logbook to be a vital tool in my work. Yes, it does require more effort to write things down and the log can get messy if you try to include all of the various drawings and e-mails you find important (mine are often bulging with clippings from important documents taped in place like some sort of engineer's scrapbook). However, the payoff is potentially huge.

This is especially true when you're doing the sort of investigative work you will be doing while studying the MySQL source code. Keep a logbook of each discovery you make. Write down every epiphany, important design decision, snippets from important paper documents, and even the occasional ah-ha! Over time you will build up a paper record of your findings (a former boss of mine called it her paper brain!) that will prove invaluable for reviews and your own documentation efforts. If you do use a logbook and make journal entries or paste in important document snippets, you will soon discover that logbooks of the journal variety do not lend themselves to being organized well. Most engineers (like me) prefer lined hardbound journals that cannot be reorganized (unless you use lots of scissors and glue). Others prefer loose-leaf logbooks that permit easy reorganization. If you plan to use a hardbound journal, consider building a "living" index as you go.

■ **Tip** If your journal pages aren't numbered, take a few minutes and place page numbers on each page.

There are many ways to build the living index. You could write any interesting keywords at the top of the page or in a specific place the margin. This would allow you to quickly skim through your logbook and locate items of interest. What makes a living index is the ability to add references over time. The best way I have found to create the living index is to use a spreadsheet to list all of the terms you write on the logbook pages and write the page number next to it. I update the spreadsheet every week or so and print it out and tape it into my logbook near the front. I have seen some journals that have a pocket in the front, but the tape approach works too. Over time you can reorder the index items and reference page numbers to make the list easier to read; you can also place an updated list in the front of your logbook so you can locate pages more easily.

I encourage you again to consider using an engineering logbook. You won't be sorry when it comes time to give your report to your superiors on your progress. It can also save you tons of rework later when you are asked to report on something you did six months or more ago.

Tracking Your Changes

You should always use comments when you create code that is not intuitive to the reader. For example, the code statement `if (found)` is pretty self-explanatory. The code following the control statement will be executed if the variable evaluates to `TRUE`. However, the code `if (func_call_17(i, x, lp))` requires some explanation. Of course, you would want to write all of your code to be self-explanatory, but sometimes that isn't possible. This is particularly true when you're accessing supporting library functions. Some of the names are not intuitive and the parameter lists can be confusing. Document these situations as you code them, and your life will be enhanced.

When writing comments, you can choose to use inline comments, single-line comments, or multiline comments. Inline comments are written beginning in column 49 and cannot exceed 80 columns. A single-line comment should be aligned with the code it is referring to (the indention mark) and also should not exceed 80 columns. Likewise, multiline comments should align with the code they are explaining, should not exceed 80 columns, but should have

the opening and closing comment markers placed on separate lines. Listing 3-24 illustrates these concepts.

Listing 3-24. *Comment Placement and Spacing Examples*

```
if (return_value)
{
  int     var1;                                /* comment goes here */
  long    var2;                                /* comment goes here too */

  /* this call does something else based on i */
  var1= do_something_else(i);

  if (var1)
  {
    /*
      This comment explains
      some really interesting thing
      about the following statement(s).
    */
    do_it_again();
  }
}
```

■ **Tip** Never use repeating *s to emphasize portions of code. It distracts the reader from the code and makes for a cluttered look. Besides, it's too much work to get all those things to line up—especially when you edit your comments later.

If you are modifying the MySQL source code using the source control application BitKeeper, you don't have to worry about tracking your changes. BitKeeper provides several ways in which you can detect and report on which changes are yours versus others. However, if you are not using BitKeeper, you could lose track of which changes are yours, particularly if you make changes directly to existing system functions. In this case, it becomes difficult to distinguish what you wrote from what was already there. Keeping an engineering logbook helps immensely with this problem, but there is a better way.

You could add comments before and after your changes to indicate which lines of code are your modifications. For example, you could place a comment like `/* BEGIN CAB MODIFICATION */` before the code and a comment like `/* END CAB MODIFICATION */` after the code. This allows you to bracket your changes and helps you search for the changes easily using a number of text and line parsing utilities. An example of this technique is shown in Listing 3-25.

Listing 3-25. *Commenting Your Changes to the MySQL Source Code*

```
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds my revision note to the MySQL version number. */
   /* original code:   */
   /*strmov(end, "."); */
   strmov(end, "-CAB Modifications");
/* END CAB MODIFICATION */
```

Notice I have also included the reason for the modification and the commented-out lines of the original code (the example is fictional). Using this technique will help you quickly access your changes and enhance your ability to diagnose problems later.

This technique can also be helpful if you make modifications for use in your organization and you are not going to share the changes with MySQL AB. If you do not share the changes, you will be forced to make the modifications to the source code every time MySQL AB releases a new build of the system you want to use. Having comment markers in the source code will help you quickly identify which files need changes and what those changes are. Chances are that if you create some new functionality you will eventually want to share that functionality if for no other reason than to avoid making the modifications every time a new version of MySQL is released.

■ **Caution** Although this technique isn't prohibited when using source code under configuration control (BitKeeper), it is usually discouraged. In fact, developers may later remove your comments altogether. Use this technique when you make changes that you are not going to share with anyone.

Building the System for the First Time

Now that you've seen the inner workings of the MySQL source code and followed the path of a typical query through the source code, it is time for you to take a turn at the wheel. If you are already working with the MySQL source code and you are reading this book to learn more about the source code and how to modify it, you can skip this section.

I recommend, before you get started, that you download the source code if you haven't already and then download and install the executables for your chosen platform. It is important to have the compiled binaries handy in case things go wrong during your experiments. Attempting to diagnose a problem with a modified MySQL source code build without a reference point can be quite challenging. You will save yourself a lot of time if you can revert to the base compiled binary when you encounter a difficult debugging problem. I will cover debugging in more detail in Chapter 5. If you ever find yourself with that system problem, you can always reinstall the binaries and return your MySQL system to normal.

Compiling the source is easy. If you are using Linux, open a command shell, change to the root of your source tree, and run the `configure`, `make`, and `make install` commands.

■ **Note** If are using Linux and the configure file does not exist, you need to generate the file using one of the platform scripts in the `BUILD` directory. For example, to create the configure file for a Pentium-class machine using `debug`, run the command `./BUILD/compile-pentium-debug` from the root of the source tree. Once the file is created, you can run the `./configure`, `make`, and `make install` commands to build the server.

The configure script will check the system for dependencies and create the appropriate makefiles. The `make` and `make install` commands build the system for the first time and build the installation. Most developers run these commands when building the MySQL source code. If compiling for the first time, you may need to change the owner of the files (if you aren't using `root`) and make group adjustments (for more details see "Source Installation Overview" in the MySQL Reference Manual at `http://dev.mysql.com/doc/refman/5.1/en/quick-install.html`). The following outlines a typical build process for building the source code on Linux for the first time:

```
%> groupadd mysql
%> useradd -g mysql mysql
%> gunzip < mysql-VERSION.tar.gz | tar -xvf -
%> cd mysql-VERSION
%> ./configure --prefix=/usr/local/mysql
%> make
%> make install
%> cp support-files/my-medium.cnf /etc/my.cnf
%> cd /usr/local/mysql
%> bin/mysql_install_db --user=mysql
%> chown -R root  .
%> chown -R mysql var
%> chgrp -R mysql .
%> bin/mysqld_safe --user=mysql &
```

You can compile the Windows platform source code using Microsoft Visual Studio 2005 (some have had great success using Visual Studio 6.0 and 2005 Express Edition with the Microsoft platform development kit, but I have found Visual Studio 2005 to be more stable). To compile the system for the first time, open the `mysql.dsw` project workspace in the root of the source distribution tree and set the active project to `mysqld` classes and the project configuration to `mysqld - Win32 nt`. When you click Build mysqld, the project is designed to compile any necessary libraries and link them to the project you specified. Take along a fresh beverage to entertain yourself as it can take a while to build all of the libraries the first time. Regardless of which platform you use, your compiled executable will be placed in the `client_release` or `client_debug` folder depending on which compile option you chose. To run the new executable, simply stop the server service, copy the file to the bin folder under the MySQL installation, and restart the server service.

■ **Caution** Most compilation problems can be traced to improperly configured development tools or missing libraries. Consult the MySQL forums for details on how to resolve the most common compilation problems.

The first thing you will notice about your newly compiled binary (unless there were problems) is that you cannot tell that the binary is the one you compiled! You could check the date of the file to see that the executable is the one you just created, but there isn't a way to know that from the client side. Although this approach is not recommended by MySQL AB and probably shunned by others as well, you could alter the version number of the MySQL compilation to indicate it is the one you compiled.

Let's assume you want to identify your modifications at a glance. For example, you want to see in the client window some indication that the server is your modified version. You could change the version number to show that. Figure 3-4 is an example of such a modification.

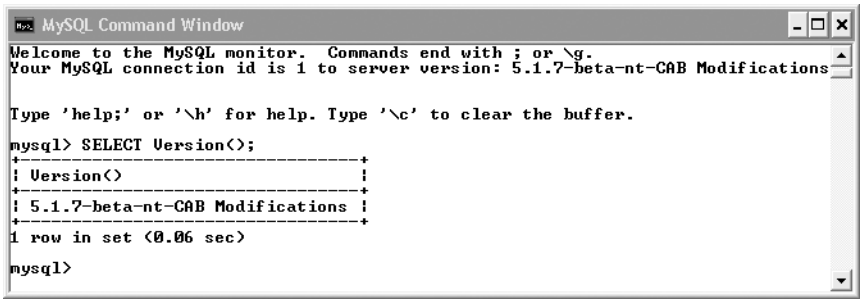

Figure 3-4. *Sample MySQL command cient with version modification*

Notice in both the header and the result of issuing the command, `SELECT Version();`, the version number returned is the same version number of the server you compiled plus an additional label I placed in the string. To make this change yourself, simply edit the `set_server_version()` function in the `mysqld.cpp` file, as shown in Listing 3-26. In the example, I have bolded the one line of code you can add to create this effect.

Listing 3-26. *Modified set_server_version Function*

```
static void set_server_version(void)
{
  char *end= strxmov(server_version, MYSQL_SERVER_VERSION,
                     MYSQL_SERVER_SUFFIX_STR, NullS);
#ifdef EMBEDDED_LIBRARY
  end= strmov(end, "-embedded");
#endif
#ifndef DBUG_OFF
  if (!strstr(MYSQL_SERVER_SUFFIX_STR, "-debug"))
      end= strmov(end, "-debug");
```

```
#endif
  if (opt_log || opt_update_log || opt_slow_log || opt_bin_log)
      strmov(end, "-log");                        // This may slow down system
  /* BEGIN CAB MODIFICATION */
  /* Reason for Modification: */
  /* This section adds my revision note to the MySQL version number. */
  strmov(end, "-CAB Modifications");
  /* END CAB MODIFICATION */
}
```

Note also that I have included the modification comments I referred to earlier. This will help you determine which lines of code you have changed. This change also has the benefit that the new version number will be shown in other MySQL tools such as the MySQL Administrator. Figure 3-5 shows the results of running the MySQL Administrator against the code compiled with this change.

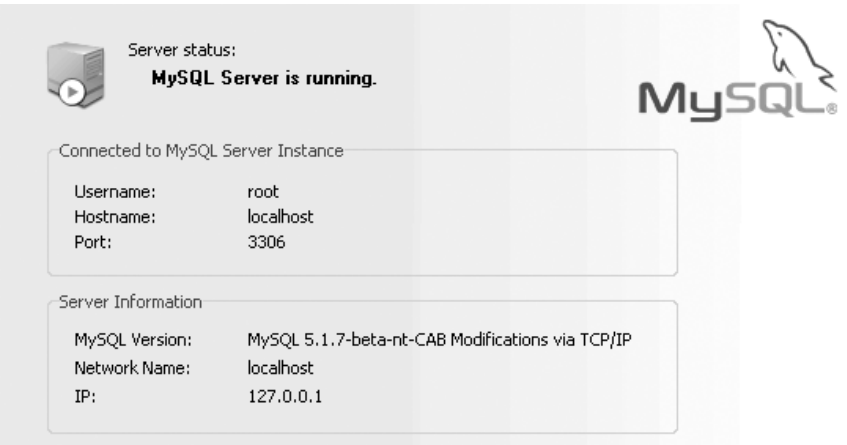

Figure 3-5. *Accessing the modified MySQL server using MySQL Administrator*

■ **Caution** Did I mention this wasn't an approved method? If you are using MySQL to conduct your own experiments or you are modifying the source code for your own use, you can get away with doing what I have suggested. However, if you are using the code under source code control or you are creating modifications that will be added to the base source code at a later date, you should *not* implement this technique.

Summary

In this chapter, you have learned several methods to get the source code. Whether you choose to download a snapshot of the source tree, a copy of the GA release source code, or use the BitKeeper client software to gain access to the latest and greatest version, you can get and start using the source code. Now that is the beauty of open source!

Perhaps the most intriguing aspect of this chapter is your guided tour of the MySQL source code. I hope that by following a simple query all the way through the system and back, you gained a lot of ground on your quest to understanding the MySQL source code. I also hope that you haven't tossed the book down in frustration if you've encountered issues with compiling the source code. Much of what makes a good open source developer is her ability to systematically diagnose and adapt her environment to the needs of the current project. Do not despair if you had issues come up. Solving issues is a natural part of the learning cycle.

You also explored the major elements from the MySQL Coding Guidelines document and saw examples of some of the code formatting and documentation guidelines. While not complete, the coding guidelines I presented are enough to give you a feel for how MySQL AB wants you to write the source code for your modifications. If you follow these simple guidelines, you should not be asked to conform later.

In the next two chapters, I will take you through two very important concepts of software development that are often overlooked. The next chapter will show you how to apply a test-driven development methodology to exploring and extending the MySQL system, and the chapter that follows will discuss debugging the MySQL source code.