Pro MySQL

MICHAEL KRUCKENBERG AND JAY PIPES

Apress®

ISBN (pbk): 1-59059-505-X

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

Lead Editors: Jason Gilmore, Matthew Moodie
Technical Reviewer: Chad Russell
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith, Susannah Pfalzer
Assistant Production Director: Kari Brooks-Copony
Production Editor: Linda Marousek
Compositor, Artist, and Interior Designer: Diana Van Winkle, Van Winkle Design Group
Proofreader: Patrick Vincent, Write Ideas Editorial Consulting
Indexer: Ann Rogers
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

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

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 4

MySQL System Architecture

In this chapter, we're going to take a look at MySQL internals. It will be a fun, informative examination of how all the different pieces and parts of the MySQL server operate together. MySQL's implementation is a fascinating mix of technology that is truly a remarkable achievement—an achievement born from the sweat and inspiration of numerous developers over many years.

One of the joys of open-source technology is just that: it's open source! On a system as large as MySQL,[1] taking a look at the source code gives you a true understanding of the dilemmas faced by the developers over the course of the software evolution. In this chapter, we'll investigate the source code of the server itself, so put on your hard hat. We encourage you to take a look at the source code, even if you have no intention of making any changes. You will gain an appreciation of the enormity of the tasks undertaken by the development team at MySQL AB, as well as gain a much deeper understanding of how the software works, and thus how you can optimize your programs to best utilize MySQL's strengths and avoid its weaknesses.

The information presented in this chapter comes from an analysis of both the internal system documentation and the actual source code for the MySQL database server system. Because MySQL is an evolving code repository, since press time, some of the design features explained here have likely changed and will continue to do so over time. If you look at the source code from one version to the next, you'll notice variations in the implementations of subsystems and how they interrelate; however, much of the way in which the system generally functions has persisted over the evolution of the software.

Even if you're not a C programming guru, you should be able to follow most of what we'll cover here. The focus will be less on the code itself and more on the structure and flow of operations within the server, and how the different code libraries interact with each other. Our intention is to provide a basic road map from which you can navigate the source code and documentation yourself. However, there are a few sections of this chapter that require a significant knowledge of C and C++ programming, and are meant for more advanced readers. If you don't have a whole lot of experience in C programming, just do your best to follow along, and don't worry about it too much!

1. At the time of this writing, the MySQL server consists of roughly 500,000 lines of source code.

In this discussion, we'll cover the following topics:

- How to access the MySQL source code and documentation
- An overview of the MySQL architecture and base function library
- The process, thread, and resource management subsystem
- The storage engine abstraction subsystem
- The caching and memory management subsystem
- The network and connection management subsystem
- The security and access control subsystem
- The log management subsystem
- The query parsing and execution subsystem
- The query cache
- The execution of a typical query

The MySQL Source Code and Documentation

Since we're going to be looking at the MySQL server source code, you'll want to download a copy of the latest MySQL source code so that you can follow along, as well as embark on your own code review adventures. The source code used in this chapter comes from a copy of the source code for version 5.0.2. To download a copy of the source code, head over to MySQL's download site (`http://dev.mysql.com/downloads/mysql/5.0.html`) and download the version of interest to you.

■ **Caution** The source distribution we used for this chapter's analysis came from the 5.0.2-alpha source tree. Bear in mind that MySQL is an *evolving* piece of software, and as such, various implementation details discussed in this chapter may change over time. *Always* obtain the proper source versions of development documentation before you assume anything is the case for a particular version of MySQL.

The Source Code

The source code is organized into a shallow directory tree containing the major libraries of the MySQL server and its different extensions.

Top-Level Directories

Table 4-1 shows all the major top-level directories, with a brief description of the files contained in each directory and the general purpose of those files. As we progress through the chapter, we'll break down the information in this table into smaller groups that relate to each subsystem, but you may use this larger table as a reference.

Table 4-1. *Main Top-Level Directories in the Source Tree*

Directory	Contents
/bdb	The Berkeley DB storage engine handler implementation files
/BUILD	Program compilation files
/client	The mysql command tool (client program) implementation files
/data	The mysql database (system database) schema, data, and index files
/dbug	Debugging utility code
/Docs	The documentation, both internal developer documents and the MySQL online manual
/heap	The MEMORY storage engine handler implementation files
/include	Core system header files and type definitions
/innobase	The InnoDB storage engine handler implementation files
/isam	The old ISAM storage engine handler implementation files
/libmysql	The MySQL C client API (all C source and header files)
/libmysqld	The MySQL server core library (C, C++, and some header files)
/libmysqltest	A simple program to test MySQL
/merge	The old Merge storage engine handler implementation files
/myisam	The MyISAM storage engine handler implementation files
/myisammrg	The MyISAM Merge storage engine handler implementation files
/mysys	The core function library, with basic low-level functions
/regex	The regular expression function library
/scripts	Shell scripts for common utilities
/share	Internationalized error messages
/sql	The meat of the server's implementation, with core classes and implementations for all major server and client activity
/sql-bench	MySQL benchmarking shell scripts
/strings	Lower-level string-handling functions
/support-files	Preconfigured MySQL configuration files (such as my-huge.cnf)
/tests	Test programs and scripts
/vio	Network/socket utility functions, virtual I/O, SSL, and so on
/zlib	Compression function source files

You can take some time now to dig through the source code a bit, for fun, but you will most likely find yourself quickly lost in the maze of classes, structs, and C functions that compose the source distribution. The first place you will want to go is the documentation for the distribution, located in the /Docs directory. Then follow along with us as we discuss the key subsystems and where you can discover the core files that correspond to the different system functionality.

C AND C++ PROGRAMMING TERMS

We'll be referring to a number of C and C++ programming paradigms in this chapter. C source code files are those files in the distribution that end in `.c`. C++ source files end in `.cc`, or on some Windows systems, `.cpp`. Both C and C++ source files can include (using the `#include` directive) *header* files, identified by an `.h` extension. In C and C++, it is customary to *define* the functions and variables used in the source files in a header file. Typically, the header file is named the same as the source file, but with an `.h` extension, but this is not always the case. One of the first tasks you'll attempt when looking at the source code of a system is identifying *where* the variables and functions are defined. Sometimes, this task involves looking through a vast hierarchy of header files in order to find where a variable or function is officially defined.

Undoubtedly, you're familiar with what variables and functions are, so we won't go into much depth about that. In C and C++ programming, however, some other data types and terms are frequently used. Most notably, we'll be using the following terms in this chapter:

- Struct
- Class
- Member variable
- Member method

A *struct* is essentially a container for a bunch of data. A typical definition for a *struct* might look something like this:

```
typedef struct st_heapinfo /* Struct from heap_info */
{
  ulong records; /* Records in database */
  ulong deleted; /* Deleted records in database */
  ulong max_records;
  ulong data_length;
  ulong index_length;
  uint reclength; /* Length of one record */
  int errkey;
  ulonglong auto_increment;
} HEAPINFO;
```

This particular definition came from `/include/heap.h`. It defines a `struct` (`st_heapinfo`) as having a number of member variables of various data types (such as `records`, `max_records`) and `typedef`s (aliases) the word `HEAPINFO` to represent the `st_heapinfo` struct. Comments in C code are marked with the `//` or `/* … */` characters.

A *class*, on the other hand, is a C++ object-oriented structure that is similar to a C struct, but can also have *member methods*, as well as *member variables*. The member methods are functions of the class, and they can be called through an *instance* of the class.

Doxygen for Source Code Analysis

A recommended way to analyze the source code is to use a tool like Doxygen (`http://www.stack.nl/~dimitri/doxygen/index.html`), which enables you to get the code structure from a source distribution. This tool can be extremely useful for navigating through functions in a large source distribution like MySQL, where a single execution can call hundreds of class members and functions. The documented output enables you to see where the classes or structs are defined and where they are implemented.

Doxygen provides the ability to configure the output of the documentation produced by the program, and it even allows for UML inheritance and collaboration diagrams to be produced. It can show the class hierarchies in the source code and provide links to where functions are defined and implemented.

On Unix machines, download the source code from the Doxygen web site, and then follow the manual instructions for installation (also available online at the web site). To produce graphical output, you'll want to first download and install the Graph visualization toolkit from `http://www.graphviz.org/`. After installing Doxygen, you can use the following command to create a default configuration file for Doxygen to process:

```
# doxygen -g -s /path/to/newconfig.file
```

The option `/path/to/newconfig.file` should be the directory in which you want to eventually produce your Doxygen documentation. After Doxygen has created the configuration file for you, simply open the configuration file in your favorite editor and edit the sections you need. Usually, you will need to modify only the `OUTPUT_DIRECTORY`, `INPUT`, and `PROJECT_NAME` settings. Once you've edited the configuration file, simply execute the following:

```
# doxygen </path/to/config-file>
```

For your convenience, a version of the MySQL 5.0.2 Doxygen output is available at `http://www.jpipes.com/mysqldox/`.

The MySQL Documentation

The internal system documentation is available to you if you download the source code of MySQL. It is in the `Docs` directory of the source tree, available in the `internals.texi` TEXI document.

The TEXI documentation covers the following topics in detail:

- Coding guidelines
- The optimizer (highly recommended reading)
- Important algorithms and structures
- Charsets and related issues
- How MySQL performs different `SELECT` operations (very useful information)
- How MySQL transforms queries
- Communication protocol
- Replication

- The MyISAM record structure
- The `.MYI` file structure
- The InnoDB record structure
- The InnoDB page structure

Although the documentation is extremely helpful in researching certain key elements of the server (particularly the query optimizer), it is worth noting that the internal documentation does not directly address how the different subsystems interact with each other. To determine this interaction, it is necessary to examine the source code itself and the comments of the developers.[2]

■ **Caution** Even the most recent `internals.texi` documentation has a number of bad hyperlinks, references, and incorrect filenames and paths, so do your homework before you take everything for granted. The `internals.texi` documentation may not be as up-to-date as your MySQL server version!

TEXI and texi2html Viewing

TEXI is the GNU standard documentation format. A number of utilities can convert the TEXI source documentation to other, perhaps more readable or portable, formats. For those of you using Emacs or some variant of it, that editor supports a TEXI major mode for easy reading.

If you prefer an HTML version, you can use the free Perl-based utility `texi2html`, which can generate a highly configurable HTML output of a TEXI source document. `texi2html` is available for download from `https://texi2html.cvshome.org/`. Once you've downloaded this utility, you can install it, like so:

```
# tar -xzvf texi2html-1.76.tar.gz
# cd texi2html-1.6
# ./configure
# make install
```

Here, we've untarred the latest (as of this writing) `texi2html` version and installed the software on our Linux system. Next, we want to generate an HTML version of the `internals.texi` document available in our source download:

```
# cd /path/to/mysql-5.0.2-alpha/
# texi2html Docs/internals.texi
```

After installation, you'll notice a new HTML document in the `/Docs` directory of your source tree called `internals.html`. You can now navigate the internal documentation via a web browser. For your convenience, this HTML document is also available at `http://www.jpipes.com/mysqldox/`.

---

2. Whether the developers chose to purposefully omit a discussion on the subsystem's communication in order to allow for changes in that communication is up for debate.

MySQL Architecture Overview

MySQL's architecture consists of a web of interrelated function sets, which work together to fulfill the various needs of the database server. A number of authors[3] have implied that these function sets are indeed components, or entirely encapsulated packages; however, there is little evidence in the source code that this is the case.

Indeed, the architecture includes separate function libraries, composed of functions that handle similar tasks, but there is not, in the traditional object-oriented programming sense, a full component-level separation of functionality. By this, we mean that you will be disappointed if you go into the source code looking for classes called `BufferManager` or `QueryManager`. They don't exist. We bring this point up because some developers, particularly ones with Java backgrounds, write code containing a number of "manager" objects, which fulfill the requests of client objects in a very object-centric approach. In MySQL, this simply isn't the case.

In some cases—notably in the source code for the query cache and log management subsystems—a more object-oriented *approach* is taken to the code. However, in most cases, system functionality is run through the various function libraries (which pass along a core set of structs) and classes (which do the dirty work of code execution), as opposed to an encapsulated approach, where components manage their internal execution and provide an API for other components to use the component. This is due, in part, to the fact that the system architecture is made up of both C and C++ source files, as well as a number of Perl and shell scripts that serve as utilities. C and C++ have different functional capabilities; C++ is a fully object-oriented language, and C is more procedural. In the MySQL system architecture, certain libraries have been written entirely in C, making an object-oriented component type architecture nearly impossible. For sure, the architecture of the server subsystems has a lot to do with performance and portability concerns as well.

■ **Note** As MySQL is an evolving piece of software, you will notice variations in both coding and naming style and consistency. For example, if you compare the source files for the older MyISAM handler files with the newer query cache source files, you'll notice a marked difference in naming conventions, commenting by the developers, and function-naming standards. Additionally, as we go to print, there have been rumors that significant changes to the directory structure and source layout will occur in MySQL 5.1.

Furthermore, if you analyze the source code and internal documentation, you will find little mention of components or packages.[4] Instead, you will find references to various task-related functionality. For instance, the internals TEXI document refers to "The Optimizer," but you will find no component or package in the source code called Optimizer. Instead, as the internals TEXI document states, "The Optimizer is a set of routines which decide what execution path the RDBMS should take for queries." For simplicity's sake, we

---

3. For examples, see *MySQL: The Complete Reference*, by Vikram Vaswani (McGraw-Hill/Osborne) and `http://wiki.cs.uiuc.edu/cs427/High-Level+Component+Diagram+of+the+MySQL+Architecture`.

4. The function init_server_components() in /sql/mysqld.cpp is the odd exception. Really, though, this method runs through starting a few of the functional subsystems and initializes the storage handlers and core buffers.

will refer to each related set of functionality by the term *subsystem*, rather than *component*, as it seems to more accurately reflect the organization of the various function libraries.

Each subsystem is designed to both accept information from and feed data into the other subsystems of the server. In order to do this in a standard way, these subsystems expose this functionality through a well-defined *function application programming interface* (*API*).[5] As requests and data funnel through the server's pipeline, the subsystems pass information between each other via these clearly defined functions and data structures. As we examine each of the major subsystems, we'll take a look at some of these data structures and methods.

MySQL Server Subsystem Organization

The overall organization of the MySQL server architecture is a layered, but not particularly hierarchical, structure. We make the distinction here that the subsystems in the MySQL server architecture are quite independent of each other.

In a hierarchical organization, subsystems depend on each other in order to function, as components *derive* from a tree-like set of classes. While there are indeed tree-like organizations of classes within some of the subsystems—notably in the SQL parsing and optimization subsystem—the subsystems themselves do not follow a hierarchical arrangement.

A base function library and a select group of subsystems handle lower-level responsibilities. These libraries and subsystems serve to support the abstraction of the storage engine systems, which feed data to requesting client programs. Figure 4-1 shows a general depiction of this layering, with different subsystems identified. We'll cover each of the subsystems separately in this chapter.

Note that client programs interact with an *abstracted* API for the storage engines. This enables client connections to issue statements that are storage-engine agnostic, meaning the client does not need to know which storage engine is handling the data request. No special client functions are required to return InnoDB records versus MyISAM records. This arrangement enables MySQL to extend its functionality to different storage requirements and media. We'll take a closer look at the storage engine implementation in the "Storage Engine Abstraction" section later in this chapter, and discuss the different storage engines in detail in the next chapter.

5. This abstraction generally leads to a loose *coupling*, or dependence, of related function sets to each other. In general, MySQL's components are loosely coupled, with a few exceptions.

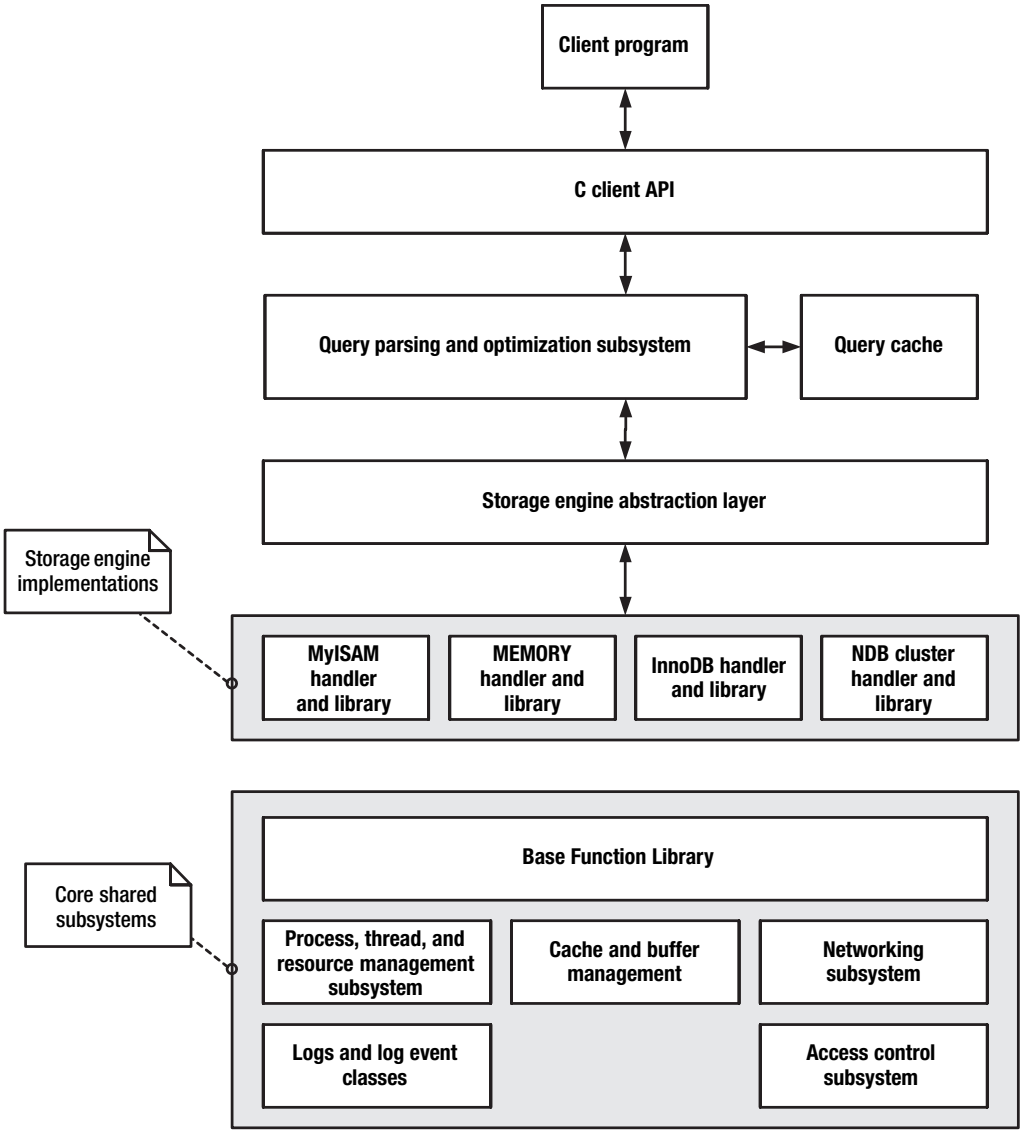

Figure 4-1. *MySQL subsystem overview*

Base Function Library

All of MySQL's subsystems share the use of a base library of common functions. Many of these functions exist to shield the subsystem (and the developers) from needing to operate directly with the operating system, main memory, or the physical hardware itself.[6] Additionally, the base function library enables code reuse and portability. Most of the functions in this base library are found in the C source files of the `/mysys` and `/strings` directories. Table 4-2 shows a sampling of core files and locations for this base library.

Table 4-2. *Some Core Function Files*

File	Contents
`/mysys/array.c`	Dynamic array functions and definitions
`/mysys/hash.c/.h`	Hash table functions and definitions
`/mysys/mf_qsort.c`	Quicksort algorithms and functions
`/mysys/string.c`	Dynamic string functions
`/mysys/my_alloc.c`	Some memory allocation routines
`/mysys/mf_pack.c`	Filename and directory path packing routines
`/strings/*`	Low-level string and memory manipulation functions, and some data type definitions

Process, Thread, and Resource Management

One of the lowest levels of the system architecture deals with the management of the various processes that are responsible for various activities on the server. MySQL happens to be a thread-based server architecture, which differs dramatically from database servers that operate on a process-based system architecture, such as Oracle and Microsoft SQL Server. We'll explain the difference in just a minute.

The library of functions that handles these various threads of execution is designed so that all the various executing threads can access key shared *resources*. These resources—whether they are simple variables maintained for the entire server system or other resources like files and certain data caches—must be monitored to avoid having multiple executing threads conflict with each other or overwriting critical data. This function library handles the coordination of the many threads and resources.

Thread-Based vs. Process-Based Design

A *process* can be described as an executing set of instructions the operating system has allocated an address space in which to conduct its operations. The operating system grants the process control over various resources, like files and devices. The operations conducted by the process have been given a certain priority by the operating system, and, over the course of its execution, the process maintains a given state (sleeping, running, and so on).

6. Certain components and libraries, however, will still interact directly with the operating system or hardware where performance or other benefits may be realized.

A *thread* can be thought of as a sort of lightweight process, which, although not given its own address space in memory, does execute a series of operations and does maintain its own state. A thread has a mechanism to save and restore its resources when it changes state, and it has access to the resources of its parent process. A *multithreaded* environment is one in which a process can create, or *spawn*, any number of threads to handle—sometimes synchronously[7]—its needed operations.

Some database servers have multiple processes handling multiple requests. However, MySQL uses multiple threads to accomplish its activities. This strategy has a number of different advantages, most notably in the arena of performance and memory use:

- It is less costly to create or destroy threads than processes. Because the threads use the parent process's address space, there is no need to allocate additional address space for a new thread.
- Switching between threads is a relatively inexpensive operation because threads are running in the same address space.
- There is little overhead involved in shared resources, since threads automatically have access to the parent's resources.

■ **Tip** Since each instance of a MySQL database server—that is, each execution of the `mysqd` server daemon—executes in its own address space, it is possible to simulate a multiprocess server by creating multiple instances of MySQL. Each instance will run in its own process and have a set of its own threads to use in its execution. This arrangement is useful when you need to have separate configurations for different instances, such as in a shared hosting environment, with different companies running different, separately configured and secured MySQL servers on the same machine.

Implementation Through a Library of Related Functions

A set of functions handles the creation of a myriad threads responsible for running the various parts of the server application. These functions are optimized to take advantage of the ability of the underlying operating system resource and process management systems. The process, thread, and resource management subsystem is in charge of creating, monitoring, and destroying threads. Specifically, threads are created by the server to manage the following main areas:

- A thread is created to handle each new user connection. This is a special thread we'll cover in detail later in the upcoming "User Connection Threads and THD Objects" section. It is responsible for carrying out both query execution and user authentication, although, as you will see, it passes this responsibility to other classes designed especially to handle those events.
- A global (instance-wide) thread is responsible for creating and managing each user connection thread. This thread can be considered a sort of user connection manager thread.

---

7. This depends on the available hardware; for instance, whether the system supports symmetric multiprocessing.

- A single thread handles all `DELAYED INSERT` requests separately.
- Another thread handles table flushes when requested by the system or a user connection.
- Replication requires separate threads for handling the synchronization of master and slave servers.
- A thread is created to handle shutdown events.
- Another thread handles signals, or alarms, inside the system.
- Another thread handles maintenance tasks.
- A thread handles incoming connection requests, either TCP/IP or Named Pipes.

The system is responsible for regulating the use of shared resources through an internal locking system. This locking system ensures that resources shared by all threads are properly managed to ensure the atomicity of data. Locks on resources that are shared among multiple threads, sometimes called *critical sections*, are managed using mutex structures.

MySQL uses the POSIX threads library. When this library is not available or not suited to the operating system, MySQL *emulates* POSIX threads by *wrapping* an operating system's available process or resource management library in a standard set of POSIX function definitions. For instance, Windows uses its own common resource management functions and definitions. Windows threads are known as *handles*, and so MySQL wraps, or redefines, a `HANDLE` struct to match a POSIX thread definition. Likewise, for locking shared resources, Windows uses functions like `InitializeCriticalSection()` and `EnterCriticalSection()`. MySQL wraps these function definitions to match a POSIX-style API: `pthread_mutex_init()` and `pthread_mutex_lock()`.

On server initialization, the function `init_thread_environment()` (in `/sql/mysqld.cc`) is called. This function creates a series of lock structures, called *mutexes*, to protect the resources used by the various threads executing in the server process. Each of these locks protects a specific resource or group of resources. When a thread needs to modify or read from the resource or resource group, a call is made to lock the resource, using `pthread_mutex_lock()`. The thread modifies the resource, and then the resource is unlocked using `pthread_mutex_unlock()`. In our walk-through of a typical query execution at the end of this chapter, you'll see an example of how the code locks and unlocks these critical resources (see Listing 4-10).

Additionally, the functions exposed by this subsystem are used by specific threads in order to allocate resources inside each thread. This is referred to as *thread-specific data* (TSD). Table 4-3 lists a sampling of files for thread and process management.

Table 4-3. *Some Thread and Process Management Subsystem Files*

File	Contents
`/include/my_pthread.h`	Wrapping definitions for threads and thread locking (mutexes)
`/mysys/my_pthread.c`	Emulation and degradation of thread management for nonsupporting systems
`/mysys/thr_lock.c` and `/mysys/thr_lock.h`	Functions for reading, writing, and checking status of thread locks
`/sql/mysqld.cc`	Functions like `create_new_thread()`, which creates a new user connection thread, and `close_connection()`, which removes (either destroys or sends to a pool) that user connection

User Connection Threads and THD Objects

For each user connection, a special type of thread, encapsulated in a class named `THD`, is responsible for handling the execution of queries and access control duties. Given its importance, you might think that it's almost ubiquitously found in the source code, and indeed it is. `THD` is defined in the `/sql/sql_class.h` file and implemented in the `/sql/sql_class.cc` file. The class represents everything occurring during a user's connection, from access control through returning a resultset, if appropriate. The following are just some of the class members of `THD` (some of them should look quite familiar to you):

- `last_insert_id`
- `limit_found_rows`
- `query`
- `query_length`
- `row_count`
- `session_tx_isolation`
- `thread_id`
- `user`

This is just a sampling of the member variables available in the substantial `THD` class. You'll notice on your own inspection of the class definition that `THD` houses all the functions and variables you would expect to find to maintain the state of a user connection and the statement being executed on that connection. We'll take a more in-depth look at the different parts of the `THD` class as we look further into how the different subsystems make use of this base class throughout this chapter.

The `create_new_thread()` function found in `/sql/mysqld.cc` spawns a new thread and creates a new user thread *object* (`THD`) for each incoming connection.[8] This function is called by the managing thread created by the server process to handle all incoming user connections. For each new thread, two global counters are incremented: one for the total number of threads created and one for the number of open threads. In this way, the server keeps track of the number of user connections created since the server started and the number of user connections that are currently open. Again, in our examination of a typical query execution at the end of this chapter, you'll see the actual source code that handles this user thread-spawning process.

Storage Engine Abstraction

The storage engine abstraction subsystem enables MySQL to use different handlers of the table data within the system architecture. Each storage engine implements the `handler` superclass defined in `/sql/handler.h`. This file indicates the standard API that the query parsing and execution subsystem will call when it needs to store or retrieve data from the engine.

---

8. This is slightly simplified, as there is a process that checks to see if an existing thread can be reused (pooling).

Not all storage engines implement the entire `handler` API; some implement only a small fraction of it. Much of the bulk of each handler's implementation details is concerned with converting data, schema, and index information into the format needed by MySQL's internal record format (in-memory record format).

■ **Note** For more information about the internal format for record storage, see the `internals.texi` document included with the MySQL internal system documentation, in the `Docs` directory of the source tree.

Key Classes and Files for Handlers

When investigating the storage engine subsystem, a number of files are important. First, the definition of the `handler` class is in `/sql/handler.h`. All the storage engines implement their own subclass of `handler`, meaning each subclass inherits all the functionality of the `handler` superclass. In this way, each storage engine's `handler` subclass follows the same API. This enables client programs to operate on the data contained in the storage engine's tables in an identical manner, even though the implementation of the storage engines—how and where they actually store their data—is quite different.

The `handler` subclass for each storage engine begins with `ha_` followed by the name of the storage engine. The definition of the subclass and its member variables and methods are available in the `/sql` directory of the source tree and are named after the `handler` subclass. The files that actually implement the `handler` class of the storage engine differ for each storage engine, but they can all be found in the directory named for the storage engine:

- The MyISAM storage engine handler subclass is `ha_myisam`, and it is defined in `/sql/ha_myisam.h`. Implementation files are in the `/myisam` directory.
- The MyISAM MERGE storage engine handler subclass is `ha_myisammrg`, and it is defined in `/sql/ha_myisammrg.h`. Implementation files are in the `/myisammrg` directory.
- The InnoDB storage engine handler subclass is `ha_innodb`, and it is defined in `/sql/ha_innodb.h`. Implementation files are in the `/innobase` directory.
- The MEMORY storage engine handler subclass is `ha_heap`, and it is defined in `/sql/ha_heap.h`. Implementation files are in the `/heap` directory.
- The NDB Cluster handler subclass is `ha_ndbcluster`, and it is defined in `/sql/ha_ndbcluster.h`. Unlike the other storage engines, which are implemented in a separate directory, the Cluster handler is implemented entirely in `/sql/ha_ndbcluster.cc`.

The Handler API

The storage engine `handler` subclasses must implement a base interface API defined in the `handler` superclass. This API is how the server interacts with the storage engine.

Listing 4-1 shows a stripped-out version (for brevity) of the `handler` class definition. Its member methods are the API of which we speak. We've highlighted the member method names to make it easier for you to pick them out. Out intention here is to give you a feel for the base class of each storage engine's implementation.

Listing 4-1. *handler Class Definition (Abridged)*

```
class handler // …
{
protected:
  struct st_table *table;       /* The table definition */

  virtual int index_init(uint idx) { active_index=idx; return 0; }
  virtual int index_end() { active_index=MAX_KEY; return 0; }
  // omitted ...
  virtual int rnd_init(bool scan) =0;
  virtual int rnd_end() { return 0; }

public:

  handler (TABLE *table_arg) {}
  virtual ~handler(void) {}
  // omitted ...
  void update_auto_increment();
  // omitted ...
  virtual bool has_transactions(){ return 0;}
  // omitted ...
  // omitted ...
  virtual int open(const char *name, int mode, uint test_if_locked)=0;
  virtual int close(void)=0;
  virtual int write_row(byte * buf) { return  HA_ERR_WRONG_COMMAND; }
  virtual int update_row(const byte * old_data, byte * new_data) {}
  virtual int delete_row(const byte * buf) {}
  virtual int index_read(byte * buf, const byte * key,
            uint key_len, enum ha_rkey_function find_flag) {}
  virtual int index_read_idx(byte * buf, uint index, const byte * key,
                uint key_len, enum ha_rkey_function find_flag);
  virtual int index_next(byte * buf) {}
  virtual int index_prev(byte * buf) {}
  virtual int index_first(byte * buf) {}
  virtual int index_last(byte * buf) {}
  // omitted ...
  virtual int rnd_next(byte *buf)=0;
  virtual int rnd_pos(byte * buf, byte *pos)=0;
  virtual int read_first_row(byte *buf, uint primary_key);
  // omitted ...
  virtual void position(const byte *record)=0;
  virtual void info(uint)=0;
  // omitted ...
  virtual int start_stmt(THD *thd) {return 0;}
  // omitted ...
  virtual ulonglong get_auto_increment();
  virtual void restore_auto_increment();
  virtual void update_create_info(HA_CREATE_INFO *create_info) {}
```

```
  /* admin commands - called from mysql_admin_table */
  virtual int check(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int backup(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int restore(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int repair(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int optimize(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int analyze(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int assign_to_keycache(THD* thd, HA_CHECK_OPT* check_opt) {}
  virtual int preload_keys(THD* thd, HA_CHECK_OPT* check_opt) {}
  /* end of the list of admin commands */

  // omitted ...
  virtual int add_index(TABLE *table_arg, KEY *key_info, uint num_of_keys) {}
  virtual int drop_index(TABLE *table_arg, uint *key_num, uint num_of_keys) {}
  // omitted ...
  virtual int rename_table(const char *from, const char *to);
  virtual int delete_table(const char *name);
  virtual int create(const char *name, TABLE *form, HA_CREATE_INFO *info)=0;
  // omitted ...
};
```

You should recognize most of the member methods. They correspond to features you may associate with your experience using MySQL. Different storage engines implement some or all of these member methods. In cases where a storage engine does not implement a specific feature, the member method is simply left alone as a placeholder for possible future development. For instance, certain administrative commands, like `OPTIMIZE` or `ANALYZE`, require that the storage engine implement a specialized way of optimizing or analyzing the contents of a particular table for that storage engine. Therefore, the `handler` class provides placeholder member methods (**`optimize()`** and **`analyze()`**) for the subclass to implement, *if it wants to*.

The member variable `table` is extremely important for the handler, as it stores a pointer to an `st_table` struct. This struct contains information about the table, its fields, and some meta information. This member variable, and four member methods, are in a *protected* area of the `handler` class, which means that only classes that inherit from the `handler` class—specifically, the storage engine `handler` subclasses—can use or see those member variables and methods.

Remember that not all the storage engines actually implement each of `handler`'s member methods. The `handler` class definition provides default return values or functional equivalents, which we've omitted here for brevity. However, certain member methods must be implemented by the specific storage engine subclass to make the handler at least useful. The following are some of these methods:

- `rnd_init()`: This method is responsible for preparing the handler for a scan of the table data.
- `rnd_next()`: This method reads the next row of table data into a buffer, which is passed to the function. The data passed into the buffer must be in a format consistent with the internal MySQL record format.

- `open()`: This method is in charge of opening the underlying table and preparing it for use.
- `info()`: This method fills a number of member variables of the handler by querying the table for information, such as how many records are in the table.
- `update_row()`: This member method replaces old row data with new row data in the underlying data block.
- `create ()`: This method is responsible for creating and storing the schema for a table definition in whatever format used by the storage engine. For instance, MyISAM's `ha_myisam::create()` member method implementation writes the `.frm` file containing the table schema information.

We'll cover the details of storage engine implementations in the next chapter.

■ **Note** For some light reading on how to create your *own* storage engine and handler implementations, check out John David Duncan's article at `http://dev.mysql.com/tech-resources/articles/creating-new-storage-engine.html`.

Caching and Memory Management Subsystem

MySQL has a separate subsystem devoted to the caching and retrieval of different types of data used by all the threads executing within the server process. These data caches, sometimes called *buffers*, enable MySQL to reduce the number of requests for disk-based I/O (an expensive operation) in return for using data already stored in memory (in buffers).

The subsystem makes use of a number of different types of caches, including the record, key, table, hostname, privilege, and other caches. The differences between the caches are in the type of data they store and why they store it. Let's briefly take a look at each cache.

Record Cache

The record cache isn't a buffer for just *any* record. Rather, the record cache is really just a set of function calls that mostly read or write data *sequentially* from a collection of files. For this reason, the record cache is used primarily during table scan operations. However, because of its ability to both read *and* write data, the record cache is also used for sequential writing, such as in some log writing operations.

The core implementation of the record cache can be found in `/mysys/io_cache.c` and `/sql/records.cc`; however, you'll need to do some digging around before anything makes much sense. This is because the key struct used in the record cache is called `st_io_cache`, aliased as `IO_CACHE`. This structure can be found in `/mysys/my_sys.h`, along with some very important macros, all named starting with `my_b_`. They are defined immediately after the `IO_CACHE` structure, and these macros are one of the most interesting implementation details in MySQL.

The IO_CACHE structure is essentially a structure containing a built-in buffer, which can be filled with record data structures.[9] However, this buffer is a fixed size, and so it can store only so many records. Functions throughout the MySQL system can use an IO_CACHE object to retrieve the data they need, using the my_b_ functions (like my_b_read(), which reads from the IO_CACHE internal buffer of records). But there's a problem.

What happens when somebody wants the "next" record, and IO_CACHE's buffer is full? Does the calling program or function need to switch from using the IO_CACHE's buffer to something else that can read the needed records from disk? No, the caller of my_b_read() does not. These macros, in combination with IO_CACHE, are sort of a built-in switching mechanism for other parts of the MySQL server to freely read data from a record cache, but not worry about whether or not the data actually exists in memory. Does this sound strange? Take a look at the definition for the my_b_read macro, shown in Listing 4-2.

Listing 4-2. *my_b_read Macro*

```
#define my_b_read(info,Buffer,Count) \
  ((info)->read_pos + (Count) <= (info)->read_end ? \
   (memcpy(Buffer,(info)->read_pos,(size_t) (Count)), \
    ((info)->read_pos+=(Count)),0) : \
   (*(info)->read_function)((info),Buffer,Count))
```

Let's break it down to help you see the beauty in its simplicity. The info parameter is an IO_CACHE object. The Buffer parameter is a reference to some output storage used by the caller of my_b_read(). You can consider the Count parameter to be the number of records that need to be read.

The macro is simply a ternary operator (that ? : thing). my_b_read() simply looks to see whether the request would read a record from before the end of the internal record buffer ((info)->read_pos + (Count) <= (info)->read_end). If so, the function copies (memcpy) the needed records from the IO_CACHE record buffer into the Buffer output parameter. If not, it calls the IO_CACHE read_function. This read function can be any of the read functions defined in /mysys/mf_iocache.c, which are specialized for the type of disk-based file read needed (such as sequential, random, and so on).

Key Cache

The implementation of the key cache is complex, but fortunately, a good amount of documentation is available. This cache is a repository for frequently used B-tree index data blocks for all MyISAM tables and the now-deprecated ISAM tables. So, the key cache stores key data for MyISAM and ISAM tables.

9. Actually, IO_CACHE is a generic buffer cache, and it can contain different data types, not just records.

The primary source code for key cache function definitions and implementation can be found in `/include/keycache.h` and `mysys/mf_keycache.c`. The `KEY_CACHE` struct contains a number of linked lists of accessed index data blocks. These blocks are a fixed size, and they represent a single block of data read from an `.MYI` file.

■ **Tip** As of version 4.1 you can change the key cache's block size by changing the `key_cache_block_size` configuration variable. However, this configuration variable is still not entirely implemented, as you cannot currently change the size of an index block, which is set when the `.MYI` file is created. See `http://dev.mysql.com/doc/mysql/en/key-cache-block-size.html` for more details.

These blocks are kept in memory (inside a `KEY_CACHE` struct instance), and the `KEY_CACHE` keeps track of how "warm"[10] the index data is—for instance, how frequently the index data block is requested. After a time, cold index blocks are purged from the internal buffers. This is a sort of least recently used (LRU) strategy, but the key cache is smart enough to retain blocks that contain index data for the root B-tree levels.

The number of blocks available inside the `KEY_CACHE`'s internal list of used blocks is controlled by the `key_buffer_size` configuration variable, which is set in multiples of the key cache block size.

The key cache is created the first time a MyISAM table is opened. The `multi_key_cache_search()` function (found in `/mysys/mf_keycaches.c`) is called during the storage engine's `mi_open()` function call.

When a user connection attempts to access index (key) data from the MyISAM table, the table's key cache is first checked to determine whether the needed index block is available in the key cache. If it is, the key cache returns the needed block from its internal buffers. If not, the block is read from the relevant `.MYI` file into the key cache for storage in memory. Subsequent requests for that index block will then come from the key cache, until that block is purged from the key cache because it is not used frequently enough.

Likewise, when changes to the key data are needed, the key cache first writes the changes to the internally buffered index block and marks it as dirty. If this dirty block is selected by the key cache for purging—meaning that it will be replaced by a more recently requested index block—that block is flushed to disk before being replaced. If the block is not dirty, it's simply thrown away in favor of the new block. Figure 4-2 shows the flow request between user connections and the key cache for requests involving MyISAM tables, along with the relevant function calls in `/mysys/mf_keycache.c`.

---

10. There is actually a `BLOCK_TEMPERATURE` variable, which places the block into warm or hot lists of blocks (`enum BLOCK_TEMPERATURE { BLOCK_COLD, BLOCK_WARM , BLOCK_HOT }`).

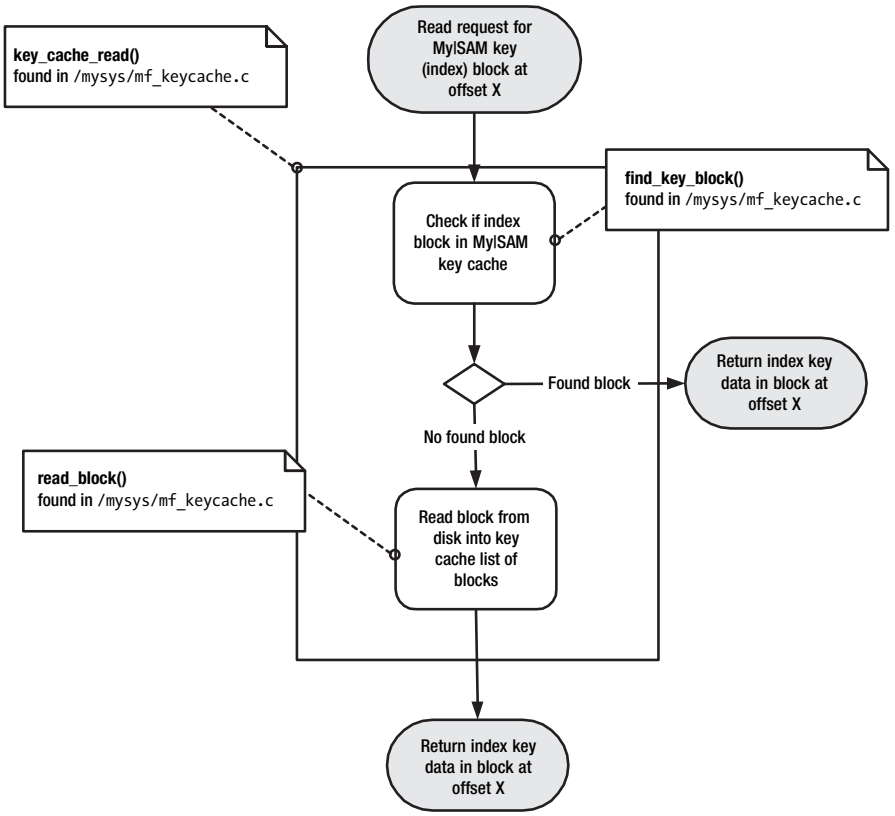

Figure 4-2. *The key cache*

You can monitor the server's usage of the key cache by reviewing the following server statistical variables:

- `Key_blocks_used`: This variable stores the number of index blocks currently contained in the key cache. This should be high, as the more blocks in the key cache, the less the server is using disk-based I/O to examine the index data.
- `Key_read_requests`: This variable stores the total number of times a request for index blocks has been received by the key cache, regardless of whether the key cache actually needed to read the block from disk.
- `Key_reads`: This variable stores the number of disk-based reads the key cache performed in order to get the requested index block.
- `Key_write_requests`: This variable stores the total number of times a write request was received by the key cache, regardless of whether the modifications (writes) of the key data were to disk. Remember that the key cache writes changes to the actual `.MYI` file only when the index block is deemed too cold to stay in the cache *and* it has been marked dirty by a modification.
- `Key_writes`: This variable stores the number of actual writes to disk.

Experts have recommended that the Key_reads to Key_read_requests and Key_writes to Key_write_requests should have, at a minimum, a 1:50–1:100 ratio.[11] If the ratio is lower than that, consider increasing the size of key_buffer_size and monitoring for improvements. You can review these variables by executing the following:

```
mysql> SHOW STATUS LIKE 'Key_%';
```

Table Cache

The table cache is implemented in /sql/sql_base.cc. This cache stores a special kind of structure that represents a MySQL table in a simple HASH structure. This hash, defined as a global variable called open_cache, stores a set of st_table structures, which are defined in /sql/table.h and /sql/table.cc.

■ **Note** For the implementation of the HASH struct, see /include/hash.h and /mysys/hash.c.

The st_table struct is a core data structure that represents the actual database table in memory. Listing 4-3 shows a small portion of the struct definition to give you an idea of what is contained in st_table.

Listing 4-3. *st_table Struct (Abridged)*

```
struct st_table {
  handler *file;
  Field **field;            /* Pointer to fields */
  Field_blob **blob_field;        /* Pointer to blob fields */
  /* hash of field names (contains pointers to elements of field array) */
  HASH name_hash;
  byte *record[2];    /* Pointer to records */
  byte *default_values;         /* Default values for INSERT */
  byte *insert_values;            /* used by INSERT ... UPDATE */
  uint fields;    /* field count */
  uint reclength;    /* Recordlength */
// omitted…
  struct st_table *next,*prev;
};
```

The st_table struct fulfills a variety of purposes, but its primary focus is to provide other objects (like the user connection THD objects and the handler objects) with a mechanism to find out meta information about the table's structure. You can see that some of st_table's member variables look familiar: fields, records, default values for inserts, a length of records, and a count of the number of fields. All these member variables provide the THD and other consuming classes with information about the structure of the underlying table source.

---

11. Jeremy Zawodny and Derrek Bailing, *High Performance MySQL* (O'Reilly, 2004), p 242.

This struct also serves to provide a method of linking the storage engine to the table, so that the THD objects may call on the storage engine to execute requests involving the table. Thus, one of the member variables (*file) of the st_table struct is a pointer to the storage engine (handler subclass), which handles the actual reading and writing of records in the table and indexes associated with it. Note that the developers named the member variable for the handler as file, bringing us to an important point: the handler represents a link for this in-memory table structure to the physical storage managed by the storage engine (handler). This is why you will sometimes hear some folks refer to the number of open *file descriptors* in the system. The handler class pointer represents this physical file-based link.

The st_table struct is implemented as a linked list, allowing for the creation of a list of used tables during executions of statements involving multiple tables, facilitating their navigation using the next and prev pointers. The table cache is a hash structure of these st_table structs. Each of these structs represents an in-memory representation of a table schema. If the handler member variable of the st_table is an ha_myisam (MyISAM's storage engine handler subclass), that means that the .frm file has been read from disk and its information dumped into the st_table struct. The task of initializing the st_table struct with the information from the .frm file is relatively expensive, and so MySQL caches these st_table structs in the table cache for use by the THD objects executing queries.

■ **Note** Remember that the key cache stores index blocks from the .MYI files, and the table cache stores st_table structs representing the .frm files. Both caches serve to minimize the amount of disk-based activity needed to open, read, and close those files.

It is very important to understand that the table cache does not share cached st_table structs *between* user connection threads. The reason for this is that if a number of concurrently executing threads are executing statements against a table whose schema may change, it would be possible for one thread to change the schema (the .frm file) while another thread is relying on that schema. To avoid these issues, MySQL ensures that each concurrent thread has its own set of st_table structs in the table cache. This feature has confounded some MySQL users in the past when they issue a request like the following:

```
mysql> SHOW STATUS LIKE 'Open_%';
```

and see a result like this:

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 200   |
| Open_files    | 315   |
| Open_streams  | 0     |
| Opened_tables | 216   |
+---------------+-------+
4 rows in set (0.03 sec)
```

knowing that they have only ten tables in their database.

The reason for the apparently mismatched open table numbers is that MySQL opens a new `st_table` struct for each concurrent connection. For each opened table, MySQL actually needs two file descriptors (pointers to files on disk): one for the `.frm` file and another for the `.MYD` file. The `.MYI` file is shared among all threads, using the key cache. But just like the key cache, the table cache has only a certain amount of space, meaning that a certain number of `st_table` structs will fit in there. The default is 64, but this is modifiable using the `table_cache` configuration variable. As with the key cache, MySQL provides some monitoring variables for you to use in assessing whether the size of your table cache is sufficient:

- `Open_tables`: This variable stores the number of table schemas opened by all storage engines for all concurrent threads.
- `Open_files`: This variable stores the number of actual file descriptors currently opened by the server, for all storage engines.
- `Open_streams`: This will be zero unless logging is enabled for the server.
- `Opened_tables`: This variable stores the total number of table schemas that have been opened since the server started, across all concurrent threads.

If the `Opened_tables` status variable is substantially higher than the `Open_tables` status variable, you may want to increase the `table_cache` configuration variable. However, be aware of some of the limitations presented by your operating system for file descriptor use. See the MySQL manual for some gotchas: `http://dev.mysql.com/doc/mysql/en/table-cache.html`.

■ **Caution** There is some evidence in the MySQL source code comments that the table cache is being redesigned. For future versions of MySQL, check the changelog to see if this is indeed the case. See the code comments in the `sql/sql_cache.cc` for more details.

Hostname Cache

The hostname cache serves to facilitate the quick lookup of hostnames. This cache is particularly useful on servers that have slow DNS servers, resulting in time-consuming repeated lookups. Its implementation is available in `/sql/hostname.cc`, with the following globally available variable declaration:

```
static hash_filo *hostname_cache;
```

As is implied by its name, `hostname_cache` is a first-in/last-out (FILO) hash structure. `/sql/hostname.cc` contains a number of functions that initialize, add to, and remove items from the cache. `hostname_cache_init()`, `add_hostname()`, and `ip_to_hostname()` are some of the functions you'll find in this file.

Privilege Cache

MySQL keeps a cache of the privilege (grant) information for user accounts in a separate cache. This cache is commonly called an *ACL*, for *access control list*. The definition and implementation of the ACL can be found in `/sql/sql_acl.h` and `/sql/sql_acl.cc`. These files

define a number of key classes and structs used throughout the user access and grant management system, which we'll cover in the "Access and Grant Management" section later in this chapter.

The privilege cache is implemented in a similar fashion to the hostname cache, as a FILO hash (see `/sql/sql_acl.cc`):

```
static hash_filo *acl_cache;
```

`acl_cache` is initialized in the `acl_init()` function, which is responsible for reading the contents of the `mysql` user and grant tables (`mysql.user`, `mysql.db`, `mysql.tables_priv`, and `mysql.columns_priv`) and loading the record data into the `acl_cache` hash. The most interesting part of the function is the sorting process that takes place. The sorting of the entries as they are inserted into the cache is important, as explained in Chapter 15. You may want to take a look at `acl_init()` after you've read that chapter.

Other Caches

MySQL employs other caches internally for specialized uses in query execution and optimization. For instance, the heap table cache is used when `SELECT…GROUP BY` or `DISTINCT` statements find all the rows in a MEMORY storage engine table. The join buffer cache is used when one or more tables in a `SELECT` statement cannot be joined in anything other than a `FULL JOIN`, meaning that all the rows in the table must be joined to the results of all other joined table results. This operation is expensive, and so a buffer (cache) is created to speed the returning of result sets. We'll cover `JOIN` queries in great detail in Chapter 7.

Network Management and Communication

The network management and communication system is a low-level subsystem that handles the work of sending and receiving network packets containing MySQL connection requests and commands across a variety of platforms. The subsystem makes the various communication protocols, such as TCP/IP or Named Pipes, transparent for the connection thread. In this way, it releases the query engine from the responsibility of interpreting the various protocol packet headers in different ways. All the query engine needs to know is that it will receive from the network and connection management subsystem a standard data structure that complies with an API.

The network and connection management function library can be found in the files listed in Table 4-4.

Table 4-4. *Network and Connection Management Subsystem Files*

File	Contents
`/sql/net_pkg.cc`	The client/server network layer API and protocol for communications between the client and server
`/include/mysql_com.h`	Definitions for common structs used in the communication between the client and server
`/include/my_net.h`	Addresses some portability and thread-safe issues for various networking functions

The main struct used in client/server communications is the `st_net` struct, aliased as `NET`. This struct is defined in `/include/mysql_com.h`. The definition for `NET` is shown in Listing 4-4.

Listing 4-4. *st_net Struct Definition*

```
typedef struct st_net {
  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 */
  /*
    Pointer to query object in query cache, do not equal NULL (0) for
    queries in cache that have not stored its results yet
  */
  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 `NET` struct is used in client/server communications as a handler for the communication protocol. The `buff` member variable of `NET` is filled with a packet by either the server or client. These packets, like all packets used in communications protocols, follow a rigid format, containing a fixed header and the packet data.

Different packet types are sent for the various legs of the trip between the client and server. The legs of the trip correspond to the diagram in Figure 4-3, which shows the communication between the client and server.

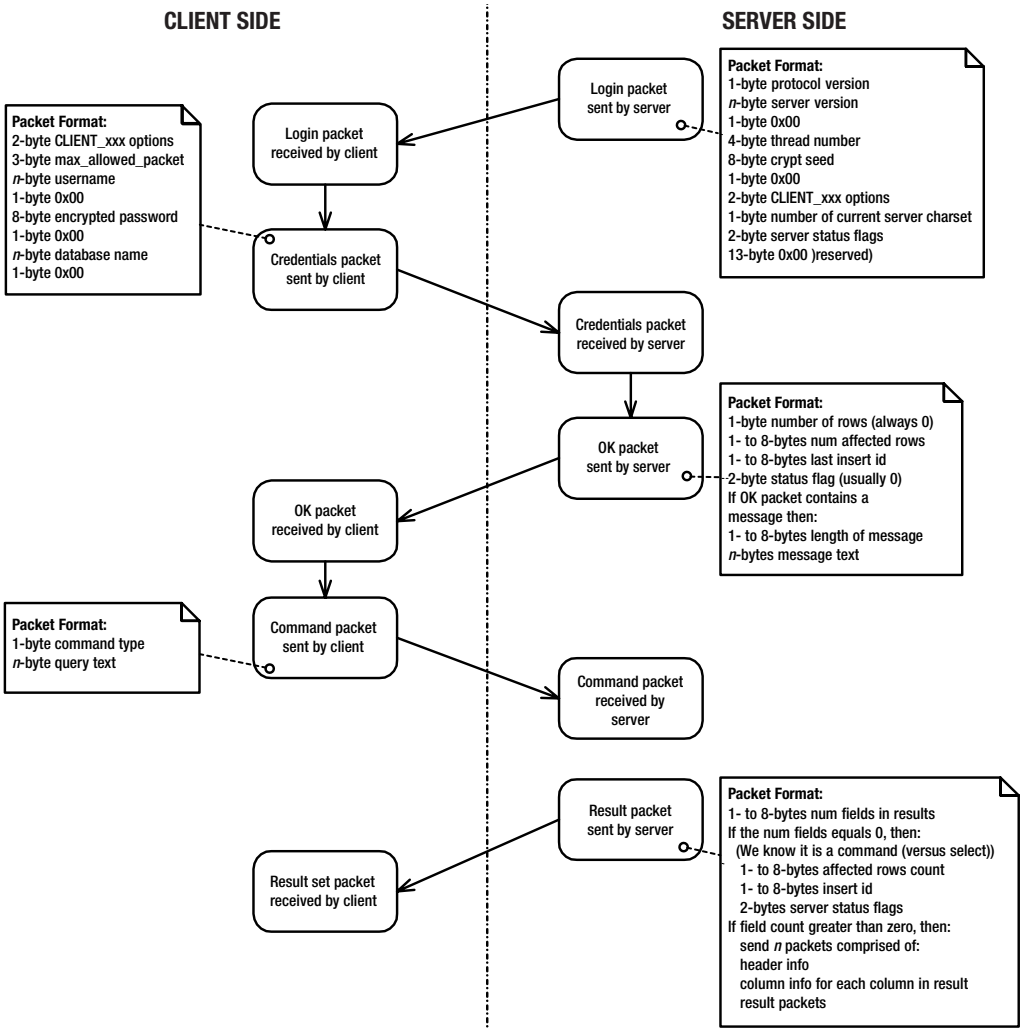

Figure 4-3. *Client/server communication*

In Figure 4-3, we've included some basic notation of the packet formats used by the various legs of the communication trip. Most are self-explanatory. The result packets have a standard header, described in the protocol, which the client uses to obtain information about how many result packets will be received to get all the information back from the server.

The following functions actually move the packets into the `NET` buffer:

- `my_net_write()`: This function stores a packet to be sent in the `NET->buff` member variable.
- `net_flush()`: This function sends the packet stored in the `NET->buff` member variable.

- `net_write_command()`: This function sends a command packet (1 byte; see Figure 4-3) from the client to the server.
- `my_net_read()`: This function reads a packet in the `NET` struct.

These functions can be found in the `/sql/net_serv.cc` source file. They are used by the various client and server communication functions (like `mysql_real_connect()`, found in `/libmysql/libmysql.c` in the C client API). Table 4-5 lists some other functions that operate with the `NET` struct and send packets to and from the server.

Table 4-5. *Some Functions That Send and Receive Network Packets*

Function	File	Purpose
`mysql_real_connect()`	`/libmysql/client.c`	Connects to the `mysqld` server. Look for the `CLI_MYSQL_REAL_CONNECT` function, which handles the connection from the client to the server.
`mysql_real_query()`	`/libmysql/client.c`	Sends a query to the server and reads the OK packet or columns header returned from the server. The packet returned depends on whether the query was a command or a resultset returning `SHOW` or `SELECT`.
`mysql_store_result()`	`/libmysql/client.c`	Takes a resultset sent from the server entirely into client-side memory by reading all sent packets definitions
various	`/include/mysql.h`	Contains some useful definitions of the structs used by the client API, namely `MYSQL` and `MYSQL_RES`, which represent the MySQL client session and results returned in it.

■ **Note** The `internals.texi` documentation thoroughly explains the client/server communications protocol. Some of the file references, however, are a little out-of-date for version 5.0.2's source distribution. The directories and filenames in Table 4-5 are correct, however, and should enable you to investigate this subsystem yourself.

Access and Grant Management

A separate set of functions exists solely for the purpose of checking the validity of incoming connection requests and privilege queries. The access and grant management subsystem defines all the `GRANT`s needed to execute a given command (see Chapter 15) and has a set of functions that query and modify the in-memory versions of the grant tables, as well as some utility functions for password generation and the like. The bulk of the subsystem is contained in the `/sql/sql_acl.cc` file of the source tree. Definitions are available in `/sql/sql_acl.h`, and the implementation is in `/sql/sql_acl.cc`. You will find all the actual `GRANT` constants defined at the top of `/sql/sql_acl.h`, as shown in Listing 4-5.

Listing 4-5. *Constants Defined in sql_acl.h*

```
#define SELECT_ACL    (1L << 0)
#define INSERT_ACL    (1L << 1)
#define UPDATE_ACL    (1L << 2)
#define DELETE_ACL    (1L << 3)
#define CREATE_ACL    (1L << 4)
#define DROP_ACL    (1L << 5)
#define RELOAD_ACL    (1L << 6)
#define SHUTDOWN_ACL    (1L << 7)
#define PROCESS_ACL    (1L << 8)
#define FILE_ACL    (1L << 9)
#define GRANT_ACL    (1L << 10)
#define REFERENCES_ACL    (1L << 11)
#define INDEX_ACL    (1L << 12)
#define ALTER_ACL    (1L << 13)
#define SHOW_DB_ACL    (1L << 14)
#define SUPER_ACL    (1L << 15)
#define CREATE_TMP_ACL    (1L << 16)
#define LOCK_TABLES_ACL    (1L << 17)
#define EXECUTE_ACL    (1L << 18)
#define REPL_SLAVE_ACL    (1L << 19)
#define REPL_CLIENT_ACL    (1L << 20)
#define CREATE_VIEW_ACL    (1L << 21)
#define SHOW_VIEW_ACL    (1L << 22)
```

These constants are used in the ACL functions to compare user and hostname privileges. The << operator is bit-shifting a long integer one byte to the left and defining the named constant as the resulting power of 2. In the source code, these constants are compared using Boolean operators in order to determine if the user has appropriate privileges to access a resource. If a user is requesting access to a resource that requires more than one privilege, these constants are ANDed together and compared to the user's own access integer, which represents all the privileges the user has been granted.

We won't go into too much depth here, because Chapter 15 covers the ACL in detail, but Table 4-6 shows a list of functions in this library.

Table 4-6. *Selected Functions in the Access Control Subsystem*

Function	Purpose
`acl_get()`	Returns the privileges available for a user, host, and database combination (database privileges).
`check_grant()`	Determines whether a user thread `THD`'s user has appropriate permissions on all tables used by the requested statement on the thread.
`check_grant_column()`	Same as `check_grant()`, but on a specific column.
`check_grant_all_columns()`	Checks all columns needed in a user thread's field list.
`mysql_create_user()`	Creates one or a list of users; called when a command received over a user thread creates users, such as `GRANT ALL ON *.* ➥ TO 'jpipes'@'localhost', 'mkruck'@'localhost'`.

Feel free to roam around the access control function library and get a feel for these core functions that handle the security between the client and server.

Log Management

In one of the more fully encapsulated subsystems, the log management subsystem implements an inheritance design whereby a variety of *log event* subclasses are consumed by a *log* class. Similar to the strategy deployed for storage engine abstraction, this strategy allows the MySQL developers to add different logs and log events as needed, without breaking the subsystem's core functionality.

The main log class, `MYSQL_LOG`, is shown in Listing 4-6 (we've stripped out some material for brevity and highlighted the member variables and methods).

Listing 4-6. *MYSQL_LOG Class Definition*

```
class MYSQL_LOG
 {
 private:
  /* LOCK_log and LOCK_index are inited by init_pthread_objects() */
  pthread_mutex_t LOCK_log, LOCK_index;
  // ... omitted
  IO_CACHE log_file;
  // ... omitted
  volatile enum_log_type log_type;
  // ... omitted
public:
  MYSQL_LOG();
  ~MYSQL_LOG();
  // ... omitted
  void set_max_size(ulong max_size_arg);
  void signal_update();
  void wait_for_update(THD* thd, bool master_or_slave);
  void set_need_start_event() { need_start_event = 1; }
  void init(enum_log_type log_type_arg,
        enum cache_type io_cache_type_arg,
        bool no_auto_events_arg, ulong max_size);
  void init_pthread_objects();
  void cleanup();
  bool open(const char *log_name,enum_log_type log_type,
        const char *new_name, const char *index_file_name_arg,
        enum cache_type io_cache_type_arg,
        bool no_auto_events_arg, ulong max_size,
            bool null_created);
  void new_file(bool need_lock= 1);
  bool write(THD *thd, enum enum_server_command command,
         const char *format,...);
  bool write(THD *thd, const char *query, uint query_length,
         time_t query_start=0);
```

```
  bool write(Log_event* event_info); // binary log write
  bool write(THD *thd, IO_CACHE *cache, bool commit_or_rollback);
  /*
    v stands for vector
    invoked as appendv(buf1,len1,buf2,len2,...,bufn,lenn,0)
  */
  bool appendv(const char* buf,uint len,...);
  bool append(Log_event* ev);
  // ... omitted
  int purge_logs(const char *to_log, bool included,
                 bool need_mutex, bool need_update_threads,
                 ulonglong *decrease_log_space);
  int purge_logs_before_date(time_t purge_time);
  // ... omitted
  void close(uint exiting);
  // ... omitted
  void report_pos_in_innodb();
  // iterating through the log index file
  int find_log_pos(LOG_INFO* linfo, const char* log_name,
           bool need_mutex);
  int find_next_log(LOG_INFO* linfo, bool need_mutex);
  int get_current_log(LOG_INFO* linfo);
  // ... omitted
};
```

This is a fairly standard definition for a logging class. You'll notice the various member methods correspond to things that the log must do: open, append stuff, purge records from itself, and find positions inside itself. Note that the `log_file` member variable is of type `IO_CACHE`. You may recall from our earlier discussion of the record cache that the `IO_CACHE` can be used for writing as well as reading. This is an example of how the `MYSQL_LOG` class uses the `IO_CACHE` structure for exactly that.

Three global variables of type `MYSQL_LOG` are created in `/sql/mysql_priv.h` to contain the three logs available in global scope:

```
extern MYSQL_LOG mysql_log,mysql_slow_log,mysql_bin_log;
```

During server startup, a function called `init_server_components()`, found in `/sql/mysqld.cc`, actually initializes any needed logs based on the server's configuration. For instance, if the server is running with the binary log enabled, then the `mysql_bin_log` global `MYSQL_LOG` instance is initialized and opened. It is also checked for consistency and used in recovery, if necessary. The function `open_log()`, also found in `/sql/mysqld.cc`, does the job of actually opening a log file and constructing a `MYSQL_LOG` object.

Also notice that a number of the member methods accept arguments of type `Log_event`, namely `write()` and `append()`. The `Log_event` class represents an event that is written to a `MYSQL_LOG` object. `Log_event` is a base (abstract) class, just like `handler` is for the storage engines, and a number of subclasses derive from it. Each of the subclasses corresponds to a specific event and contains information on *how* the event should be recorded (written) to the logs. Here are some of the `Log_event` subclasses:

- `Query_log_event`: This subclass logs when SQL queries are executed.
- `Load_log_event`: This subclass logs when the logs are loaded.
- `Intvar_log_event`: This subclass logs special variables, such as `auto_increment` values.
- `User_var_log_event`: This subclass logs when a user variable is set. This event is recorded *before* the `Query_log_event`, which actually sets the variable.

The log management subsystem can be found in the source files listed in Table 4-7. The definitions for the main log class (`MYSQL_LOG`) can be found in `/sql/sql_class.h`, so don't look for a `log.h` file. There isn't one. Developer's comments note that there are plans to move log-specific definitions into their own header file at some later date.

Table 4-7. *Log Management Source Files*

File	Contents
`/sql/sql_class.h`	The definition of the `MYSQL_LOG` class
`/sql/log_event.h`	Definitions of the various `Log_event` class and subclasses
`/sql/log_event.cc`	The implementation of `Log_event` subclasses
`/sql/log.cc`	The implementation of the `MYSQL_LOG` class
`/sql/ha_innodb.h`	The InnoDB-specific log implementation (covered in the next chapter)

Note that this separation of the logging subsystem allows for a variety of system activities—from startup, to multistatement transactions, to auto-increment value changes—to be logged via the subclass implementations of the `Log_event::write()` method. For instance, the `Intvar_log_event` subclass handles the logging of `AUTO_INCREMENT` values and partly implements its logging in the `Intvar_log_event::write()` method.

Query Parsing, Optimization, and Execution

You can consider the query parsing, optimization, and execution subsystem to be the brains behind the MySQL database server. It is responsible for taking the commands brought in on the user's thread and deconstructing the requested statements into a variety of data structures that the database server then uses to determine the best path to execute the requested statement.

Parsing

This process of deconstruction is called *parsing*, and the end result is sometimes referred to as an *abstract syntax tree*. MySQL's parser was actually generated from a program called Bison.[12] Bison generates the parser using a tool called YACC, which stands for Yet Another Compiler Compiler. YACC accepts a stream of rules. These rules consist of a regular expression and a snippet of C code designed to handle any matches made by the regular expression. YACC then produces an executable that can take an input stream and "cut it up" by matching on regular expressions. It then executes the C code paired with each regular expression in the order in which it matches the regular expression.[13] Bison is a complex program that uses the YACC compiler to generate a parser for a specific set of symbols, which form the *lexicon* of the parsable language.

■ **Tip** If you're interested in more information about YACC, Bison, and `Lex`, see `http://dinosaur.compilertools.net/`.

The MySQL query engine uses this Bison-generated parser to do the grunt work of cutting up the incoming command. This step of parsing not only standardizes the query into a tree-like request for tables and joins, but it also acts as an in-code representation of what the request needs in order to be fulfilled. This in-code representation of a query is a struct called `Lex`. Its definition is available in `/sql/sql_lex.h`. Each user thread object (`THD`) has a `Lex` member variable, which stores the *state* of the parsing.

As parsing of the query begins, the `Lex` struct fills out, so that as the parsing process executes, the `Lex` struct is filled with an increasing amount of information about the items used in the query. The `Lex` struct contains member variables to store lists of tables used by the query, fields used in the query, joins needed by the query, and so on. As the parser operates over the query statements and determines which items are needed by the query, the `Lex` struct is updated to reflect the needed items. So, on completion of the parsing, the `Lex` struct contains a sort of road map to get at the data. This road map includes the various objects of interest to the query. Some of `Lex`'s notable member variables include the following:

- `table_list` and `group_list` are lists of tables used in the `FROM` and `GROUP BY` clauses.
- `top_join_list` is a list of tables for the top-level join.
- `order_list` is a list of tables in the `ORDER BY` clause.
- `where` and `having` are variables of type `Item` that correspond to the `WHERE` and `HAVING` clauses.
- `select_limit` and `offset_limit` are used in the `LIMIT` clause.

---

12. Bison was originally written by Richard Stallman.
13. The order of matching a regular expression is not necessarily the order in which a particular word appears in the input stream.

■ **Tip** At the top of `/sql/sql_lex.h`, you will see an enumeration of all of the different SQL commands that may be issued across a user connection. This enumeration is used throughout the parsing and execution process to describe the activity occurring.

In order to properly understand what's stored in the `Lex` struct, you'll need to investigate the definitions of classes and structs defined in the files listed in Table 4-8. Each of these files represents the core units of the SQL query execution engine.

Table 4-8. *Core Classes Used in SQL Query Execution and Parsing*

File	Contents
`/sql/field.h` and `/sql/field.cc`	Definition and implementation of the `Field` class
`/sql/item.h` and `/sql/item.cc`	Definition and implementation of the `Item` class
`/sql/item_XXX.h` and `/sql/item_XXX.cc`	Definition and implementation of the specialized `Item_` classes used to represent various objects in database; for instance, `Item_row` and `Item_subselect`
`/sql/sql_class.h` and `/sql/sql_class.cc`	Definition and implementation of the various generic classes and `THD`

The different `Item_`*XXX* files implement the various components of the SQL language: its operators, expressions, functions, rows, fields, and so on.

At its source, the parser uses a table of symbols that correspond to the parts of a query or command. This symbol table can be found in `/sql/lex.h`, `/sql/lex_symbol.h`, and `/sql/lex_hash.h`. The symbols are really just the keywords supported by MySQL, including ANSI standard SQL and all of the extended functions usable in MySQL queries. These symbols make up the lexicon of the query engine; the symbols are the query engine's alphabet of sorts.

Don't confuse the files in `/sql/lex*` with the `Lex` class. They're not the same. The `/sql/lex*` files contain the symbol tables that act as tokens for the parser to deconstruct the incoming SQL statement into machine-readable structures, which are then passed on to the optimization processes.

You may view the MySQL-generated parser in `/sql/sql_yacc.cc`. Have fun. It's obscenely complex. The meat of the parser begins on line 11676 of that file, where the `yyn` variable is checked and a gigantic `switch` statement begins. The `yyn` variable represents the currently parsed symbol number. Looking at the source file for the parser will probably result in a mind melt. For fun, we've listed some of the files that implement the parsing functionality in Table 4-9.

Table 4-9. *Parsing and Lexical Generation Implementation Files*

File	Contents
/sql/lex.h	The base symbol table for parsing.
/sql/lex_symbol.h	Some more type definitions for the symbol table.
/sql/lex_hash.h	A mapping of symbols to functions.
/sql/sql_lex.h	The definition of the Lex class and other parsing structs.
/sql/sql_lex.cc	The implementation of the Lex class.
/sql/sql_yacc.h	Definitions used in the parser.
/sql/sql_yacc.cc	The Bison-generated parser implementation
/sql/sql_parse.cc	Ties in all the different pieces and parts of the parser, along with a huge library of functions used in the query parsing and execution stages.

Optimization

Much of the optimization of the query engine comes from the ability of this subsystem to "explain away" parts of a query, and to find the most efficient way of organizing how and in which order separate data sets are retrieved and merged or filtered. We'll go into the details of the optimization process in Chapters 6 and 7, so stay tuned. Table 4-10 shows a list of the main files used in the optimization system.

Table 4-10. *Files Used in the Optimization System*

File	Contents
/sql/sql_select.h	Definitions for classes and structs used in the SELECT statements, and thus, classes used in the optimization process
/sql/sql_select.cc	The implementation of the SELECT statement and optimization system
/sql/opt_range.h and /sql/opt_range.cc	The definition and implementation of range query optimization routines
/sql/opt_sum.cc	The implementation of aggregation optimization (MIN/MAX/GROUP BY)

For the most part, optimization of SQL queries is needed only for SELECT statements, so it is natural that most of the optimization work is done in /sql/sql_select.cc. This file uses the structs defined in /sql/sql_select.h. This header file contains the definitions for some of the most widely used classes and structs in the optimization process: JOIN, JOIN_TAB, and JOIN_CACHE. The bulk of the optimization work is done in the JOIN::optimize() member method. This complex member method makes heavy use of the Lex struct available in the user thread (THD) and the corresponding road map into the SQL request it contains.

JOIN::optimize() focuses its effort on "optimizing away" parts of the query execution by eliminating redundant WHERE conditions and manipulating the FROM and JOIN table lists into the smoothest possible order of tables. It executes a series of subroutines that attempt to optimize each and every piece of the JOIN conditions and WHERE clause.

Execution

Once the path for execution has been optimized as much as possible, the SQL commands must be executed by the *statement execution unit*. The statement execution unit is the function responsible for handling the execution of the appropriate SQL command. For instance, the statement execution unit for the SQL `INSERT` commands is `mysql_insert()`, which is found in `/sql/sql_insert.cc`. Similarly, the `SELECT` statement execution unit is `mysql_select()`, housed in `/sql/sql_select.cc`. These base functions all have a pointer to a `THD` object as their first parameter. This pointer is used to send the packets of result data back to the client. Take a look at the execution units to get a feel for how they operate.

The Query Cache

The query cache is not a subsystem, per se, but a wholly separate set of classes that actually do function as a component. Its implementation and documentation are noticeably different from other subsystems, and its design follows a cleaner, more component-oriented approach than most of the rest of the system code.[14] We'll take a few moments to look at its implementation and where you can view the source and explore it for yourself.

The purpose of the query cache is not just to cache the SQL commands executed on the server, but also to store the actual results of those commands. This special ability is, as far as we know, unique to MySQL. Its addition to the MySQL source distribution, as of version 4.0.1, greatly improves MySQL's already impressive performance. We'll take a look at how the query cache can be used. Right now, we'll focus on the internals.

The query cache is a single class, `Query_cache`, defined in `/sql/sql_cache.h` and implemented in `/sql/sql_cache.cc`. It is composed of the following:

- Memory pool, which is a cache of memory blocks (`cache` member variable) used to store the results of queries
- Hash table of queries (`queries` member variable)
- Hash table of tables (`tables` member variable)
- Linked lists of all the blocks used for storing queries, tables, and the root block

The memory pool (`cache` member variable) contains a directory of both the allocated (used) memory blocks and the free blocks, as well as all the actual blocks of data. In the source documentation, you'll see this directory structure referred to as *memory bins*, which accurately reflects the directory's hash-based structure.

A memory block is a specially defined allocation of the query cache's resources. It is not an index block or a block on disk. Each memory block follows the same basic structure. It has a header, represented by the `Query_cache_block` struct, shown in Listing 4-7 (some sections are omitted for brevity).

14. This may be due to a different developer or developers working on the code than in other parts of the source code, or simply a change of approach over time taken by the development team.

Listing 4-7. *Query_cache_block Struct Definition (Abridged)*

```
struct Query_cache_block
{
  enum block_type {FREE, QUERY, RESULT, RES_CONT, RES_BEG,
           RES_INCOMPLETE, TABLE, INCOMPLETE};
  ulong length;                 // length of all block
  ulong used;                   // length of data
  // … omitted
  Query_cache_block *pnext,*pprev,      // physical next/previous block
            *next,*prev;        // logical next/previous block
  block_type type;
  TABLE_COUNTER_TYPE n_tables;          // number of tables in query
  // ... omitted
};
```

As you can see, it's a simple header struct that contains a block type (`type`), which is one of the enum values defined as `block_type`. Additionally, there is a length of the whole block and the length of the block used for data. Other than that, this struct is a simple doubly linked list of other `Query_cache_block` structs. In this way, the `Query_cache.cache` contains a chain of these `Query_cache_block` structs, each containing different types of data.

When user thread (`THD`) objects attempt to fulfill a statement request, the `Query_cache` is first asked to see if it contains an identical query as the one in the `THD`. If it does, the `Query_cache` uses the `send_result_to_client()` member method to return the result in its memory pool to the client `THD`. If not, it tries to register the new query using the `store_query()` member method.

The rest of the `Query_cache` implementation, found in `/sql/sql_cache.cc`, is concerned with managing the freshness of the memory pool and invalidating stored blocks when a modification is made to the underlying data source. This invalidation process happens when an `UPDATE` or `DELETE` statement occurs on the tables connected to the query result stored in the block. Because a list of tables is associated with each query result block (look for the `Query_cache_result` struct in `/sql/sql_cache.h`), it is a trivial matter for the `Query_cache` to look up which blocks are invalidated by a change to a specific table's data.

A Typical Query Execution

In this section, we're going to explore the code execution of a typical user connection that issues a typical `SELECT` statement against the database server. This should give you a good picture of how the different subsystems work with each other to complete a request. The code snippets we'll walk through will be trimmed down, stripped editions of the actual source code. We'll highlight the sections of the code to which you should pay the closest attention.

For this exercise, we assume that the issued statement is a simple `SELECT * FROM ➥ some_table WHERE field_x = 200`, where `some_table` is a MyISAM table. This is important, because, as you'll see, the MyISAM storage engine will actually execute the code for the request through the storage engine abstraction layer.

We'll begin our journey at the starting point of the MySQL server, in the `main()` routine of `/sql/mysqld.cc`, as shown in Listing 4-8.

Listing 4-8. */sql/mysqld.cc main()*

```
int main(int argc, char **argv)
{
  init_common_variables(MYSQL_CONFIG_NAME,
                argc, argv, load_default_groups);
  init_ssl();
  server_init();
  init_server_components();
  start_signal_handler();               // Creates pidfile
  acl_init((THD *)0, opt_noacl);
  init_slave();
  create_shutdown_thread();
  create_maintenance_thread();
  handle_connections_sockets(0);
  DBUG_PRINT("quit",("Exiting main thread"));
  exit(0);
}
```

This is where the main server process execution begins. We've highlighted some of the more interesting sections. `init_common_variables()` works with the command-line arguments used on executing `mysqld` or `mysqld_safe`, along with the MySQL configuration files. We've gone over some of what `init_server_components()` and `acl_init()` do in this chapter. Basically, `init_server_components()` makes sure the `MYSQL_LOG` objects are online and working, and `acl_init()` gets the access control system up and running, including getting the privilege cache into memory. When we discussed the thread and resource management subsystem, we mentioned that a separate thread is created to handle maintenance tasks and also to handle shutdown events. `create_maintenance_thread()` and `create_shutdown_thread()` accomplish getting these threads up and running.

The `handle_connections_sockets()` function is where things start to really get going. Remember from our discussion of the thread and resource management subsystem that a thread is created for each incoming connection request, and that a separate thread is in charge of monitoring those connection threads?[15] Well, this is where it happens. Let's take a look in Listing 4-9.

15. A thread might be taken from the connection thread pool, instead of being created.

Listing 4-9. */sql/mysqld.cc handle_connections_sockets()*

```
handle_connections_sockets(arg attribute((unused)))
{
  if (ip_sock != INVALID_SOCKET)
  {
    FD_SET(ip_sock,&clientFDs);
    DBUG_PRINT("general",("Waiting for connections."));
    while (!abort_loop)
    {
        new_sock = accept(sock, my_reinterpret_cast(struct sockaddr *)
             (&cAddr), &length);
        thd= new THD;
        if (sock == unix_sock)
            thd->host=(char*) my_localhost;
        create_new_thread(thd);
    }
  }
}
```

The basic idea is that the `mysql.sock` socket is tapped for listening, and listening begins on the socket. While the listening is occurring on the port, if a connection request is received, a new `THD` struct is created and passed to the `create_new_thread()` function. The `if (sock==unix_sock)` checks to see if the socket is a Unix socket. If so, it defaults the `THD->host` member variable to be localhost. Let's check out what `create_new_thread()` does, in Listing 4-10.

Listing 4-10. */sql/mysqld.cc create_new_thread()*

```
static void create_new_thread(THD *thd)
{
  DBUG_ENTER("create_new_thread");
  /* don't allow too many connections */
  if (thread_count - delayed_insert_threads >= max_connections+1 || abort_loop)
  {
    DBUG_PRINT("error",("Too many connections"));
    close_connection(thd, ER_CON_COUNT_ERROR, 1);
    delete thd;
    DBUG_VOID_RETURN;
  }
  pthread_mutex_lock(&LOCK_thread_count);
  if (cached_thread_count > wake_thread)
  {
    start_cached_thread(thd);
  }
  else
  {
```

```
    thread_count++;
    thread_created++;
    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));
    pthread_create(&thd->real_id,&connection_attrib, \
handle_one_connection, (void*) thd))
    (void) pthread_mutex_unlock(&LOCK_thread_count);
  }
  DBUG_PRINT("info",("Thread created"));
}
```

In this function, we've highlighted some important activity. You see firsthand how the resource subsystem locks the `LOCK_thread_count` resource using `pthread_mutex_lock()`. This is crucial, since the `thread_count` and `thread_created` variables are modified (incremented) during the function's execution. `thread_count` and `thread_created` are global variables shared by all threads executing in the server process. The lock created by `pthread_mutex_lock()` prevents any other threads from modifying their contents while `create_new_thread()` executes. This is a great example of the work of the resource management subsystem.

Secondly, we highlighted `start_cached_thread()` to show you where the connection thread pooling mechanism kicks in. Lastly, and most important, `pthread_create()`, part of the thread function library, creates a new thread with the `THD->real_id` member variable and passes a function pointer for the `handle_one_connection()` function, which handles the creation of a single connection. This function is implemented in the parsing library, in `/sql/sql_parse.cc`, as shown in Listing 4-11.

Listing 4-11. */sql/sql_parse.cc handle_one_connection()*

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

We've removed most of this function's code for brevity. The rest of the function focuses on initializing the `THD` struct for the session. We highlighted two parts of the code listing within the function definition. First, we've made the `net->error` check bold to highlight the fact that the `THD->net` member variable struct is being used in the loop condition. This must mean that `do_command()` must be sending and receiving packets, right? `net` is simply a pointer to the `THD->net` member variable, which is the main structure for handling client/server communications, as we noted in the earlier section on the network subsystem. So, the main thing going on in `handle_one_connection()` is the call to `do_command()`, which we'll look at next in Listing 4-12.

Listing 4-12. */sql/sql_parse.cc do_command()*

```
bool do_command(THD *thd)
{
  char *packet;
  ulong packet_length;
  NET *net;
  enum enum_server_command command;
  packet=0;
  net_new_transaction(net);
  packet_length=my_net_read(net);
  packet=(char*) net->read_pos;
  command = (enum enum_server_command) (uchar) packet[0];
  DBUG_RETURN(dispatch_command(command,thd, packet+1, (uint) packet_length));
}
```

Now we're really getting somewhere, eh? We've highlighted a bunch of items in `do_command()` to remind you of topics we covered earlier in the chapter.

First, remember that packets are sent using the network subsystem's communication protocol. `net_new_transaction()` starts off the communication by initiating that first packet from the server to the client (see Figure 4-3 for a refresher). The client uses the passed `net` struct and fills the `net`'s buffers with the packet sent back to the server. The call to `my_net_read()` returns the length of the client's packet and fills the `net->read_pos` buffer with the packet string, which is assigned to the `packet` variable. Voilá, the network subsystem in all its glory!

Second, we've highlighted the `command` variable. This variable is passed to the `dispatch_command()` routine along with the `THD` pointer, the `packet` variable (containing our SQL statement), and the length of the statement. We've left the `DBUG_RETURN()` call in there to remind you that `do_command()` returns 0 when the command requests succeed to the caller, `handle_one_connection()`, which, as you'll recall, uses this return value to break out of the connection wait loop in case the request failed.

Let's now take a look at `dispatch_command()`, in Listing 4-13.

Listing 4-13. */sql/sql_parse.cc dispatch_command()*

```
bool dispatch_command(enum enum_server_command command, THD *thd,
              char* packet, uint packet_length)
{
  switch (command) {
      // ... omitted
  case COM_TABLE_DUMP:
  case COM_CHANGE_USER:
      // ... omitted
  case COM_QUERY:
  {
    if (alloc_query(thd, packet, packet_length))
      break;                    // fatal error is set
    mysql_log.write(thd,command,"%s",thd->query);
    mysql_parse(thd,thd->query, thd->query_length);
```

```
  }
  // ... omitted
}
```

Just as the name of the function implies, all we're doing here is dispatching the query to the appropriate handler. In the `switch` statement, we get `case`'d into the `COM_QUERY` block, since we're executing a standard SQL query over the connection. The `alloc_query()` call simply pulls the packet string into the `THD->query` member variable and allocates some memory for use by the thread. Next, we use the `mysql_log` global `MYSQL_LOG` object to record our query, as is, in the log file using the log's `write()` member method. This is the General Query Log (see Chapter 6) simply recording the query which we've requested.

Finally, we come to the call to `mysql_parse()`. This is sort of a misnomer, because besides parsing the query, `mysql_parse()` actually executes the query as well, as shown in Listing 4-14.

Listing 4-14. */sql/sql_parse.cc mysql_parse()*

```
void mysql_parse(THD *thd, char *inBuf, uint length)
{
  if (query_cache_send_result_to_client(thd, inBuf, length) <= 0)
  {
    LEX *lex= thd->lex;
    yyparse((void *)thd);
    mysql_execute_command(thd);
    query_cache_end_of_result(thd);
  }
  DBUG_VOID_RETURN;
}
```

Here, the server first checks to see if the query cache contains an identical query request that it may use the results from instead of actually executing the command. If there is no hit on the query cache, then the `THD` is passed to `yyparse()` (the Bison-generated parser for MySQL) for parsing. This function fills the `THD->Lex` struct with the optimized road map we discussed earlier in the section about the query parsing subsystem. Once that is done, we go ahead and execute the command with `mysql_execute_command()`, which we'll look at in a second. Notice, though, that after the query is executed, the `query_cache_end_of_result()` function awaits. This function simply lets the query cache know that the user connection thread handler (`thd`) is finished processing any results. We'll see in a moment how the query cache actually stores the returned resultset.

Listing 4-15 shows the `mysql_execute_command()`.

Listing 4-15. */sql/sql_parse.cc mysql_execute_command()*

```
bool mysql_execute_command(THD *thd)
{
  all_tables= lex->query_tables;
  statistic_increment(thd->status_var.com_stat[lex->sql_command],
              &LOCK_status);
  switch (lex->sql_command) {
```

```
  case SQLCOM_SELECT:
  {

    select_result *result=lex->result;
    check_table_access(thd,
                lex->exchange ? SELECT_ACL | FILE_ACL :
                SELECT_ACL,
                all_tables, 0);
    open_and_lock_tables(thd, all_tables);
    query_cache_store_query(thd, all_tables);
    res= handle_select(thd, lex, result);
    break;
  }
  case SQLCOM_PREPARE:
  case SQLCOM_EXECUTE:
// ...
  default:                  /* Impossible */
    send_ok(thd);
    break;
  }
}
```

In `mysql_execute_command()`, we see a number of interesting things going on. First, we highlighted the call to `statistic_increment()` to show you an example of how the server updates certain statistics. Here, the statistic is the `com_stat` variable for `SELECT` statements. Secondly, you see the access control subsystem interplay with the execution subsystem in the `check_table_access()` call. This checks that the user executing the query through `THD` has privileges to the list of tables used by the query.

Of special interest is the `open_and_lock_tables()` routine. We won't go into the code for it here, but this function establishes the table cache for the user connection thread and places any locks needed for any of the tables. Then we see `query_cache_store_query()`. Here, the query cache is storing the query text used in the request in its internal `HASH` of queries. And finally, there is the call to `handle_select()`, which is where we see the first major sign of the storage engine abstraction layer. `handle_select()` is implemented in `/sql/sql_select.cc`, as shown in Listing 4-16.

Listing 4-16. */sql/sql_select.cc handle_select()*

```
bool handle_select(THD *thd, LEX *lex, select_result *result)
{
    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,
              result, unit, select_lex);
  DBUG_RETURN(res);
}
```

As you can see in Listing 4-17, `handle_select()` is nothing more than a wrapper for the statement execution unit, `mysql_select()`, also in the same file.

Listing 4-17. */sql/sql_select.cc mysql_select()*

```
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)
{

  JOIN *join;
  join= new JOIN(thd, fields, select_options, result);
  join->prepare(rref_pointer_array, tables, wild_num,
              conds, og_num, order, group, having, proc_param,
              select_lex, unit));
  join->optimize();
  join->exec();
}
```

Well, it seems that `mysql_select()` has shrugged the responsibility of executing the `SELECT` statement off onto the shoulders of a `JOIN` object. We've highlighted the code sections in Listing 4-17 to show you where the optimization process occurs.

Now, let's move on to the `JOIN::exec()` implementation, in Listing 4-18.

Listing 4-18. */sql/sql_select.cc JOIN:exec()*

```
void JOIN::exec()
{
    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;
}
```

Oh, heck, it seems that we've run into another wrapper. `JOIN::exec()` simply calls the `do_select()` routine to do its dirty work. However, we do acknowledge that once `do_select()` returns, we have some information about record counts to populate some of the `THD` member variables. Let's take a look at `do_select()` in Listing 4-19. Maybe that function will be the answer.

Listing 4-19. */sql/sql_select.cc do_select()*

```
static int do_select(JOIN *join,List<Item> *fields,TABLE \
 *table,Procedure *procedure)
{
  JOIN_TAB *join_tab;
  sub_select(join,join_tab,0);
  join->result->send_eof())
}
```

This looks a little more promising. We see that `join` object's `result` member variable sends an end-of-file (EOF) marker after a call to another function called `sub_select()`, so we must be getting closer. From this behavior, it looks as though the `sub_select()` function should fill the `result` member variable of the `join` object with some records. Let's see whether we're right, in Listing 4-20.

Listing 4-20. */sql/sql_select.cc sub_select ()*

```
static int sub_select(JOIN *join,JOIN_TAB *join_tab,bool end_of_records)
{
    join_init_read_record(join_tab);
    READ_RECORD *info= &join_tab->read_record;

    join->thd->row_count= 0;
    do
    {
       join->examined_rows++;
       join->thd->row_count++;
    } while (info->read_record(info)));
  }
  return 0;
}
```

The key to the `sub_select()`[16] function is the `do…while` loop, which loops until a `READ_RECORD` struct variable (`info`) finishes calling its `read_record()` member method. Do you remember the record cache we covered earlier in this chapter? Does the `read_record()` function look familiar? You'll find out in a minute.

■ **Note** The `READ_RECORD` struct is defined in `/sql/structs.h`. It represents a record in the MySQL internal format.

---

16. We've admittedly taken a few liberties in describing the `sub_select()` function here. The real `sub_select()` function is quite a bit more complicated than this. Some very advanced and complex C++ paradigms, such as recursion through function pointers, are used in the real `sub_select()` function. Additionally, we removed much of the logic involved in the `JOIN` operations, since, in our example, this wasn't needed. In short, we kept it simple, but the concept of the function is still the same.

But first, the join_init_read_record() function, shown in Listing 4-21, is our link (finally!) to the storage engine abstraction subsystem. The function initializes the records available in the JOIN_TAB structure and populates the read_record member variable with a READ_RECORD object. Doesn't look like much when we look at the implementation of join_init_read_records(), does it?

Listing 4-21. */sql/sql_select.cc join_init_read_record()*

```
static int join_init_read_record(JOIN_TAB *tab)
{
  init_read_record(&tab->read_record, tab->join->thd, tab->table,
           tab->select,1,1);
  return (*tab->read_record.read_record)(&tab->read_record);
}
```

It seems that this simply calls the init_read_record() function, and then returns the record number read into the read_record member variable of tab. That's exactly what it is doing, so where do the storage engines and the record cache come into play? We thought you would never ask. Take a look at init_read_record() in Listing 4-22. It is found in /sql/records.cc (sound familiar?).

Listing 4-22. */sql/records.cc init_read_record ()*

```
void init_read_record(READ_RECORD *info,THD *thd, TABLE *table,
              SQL_SELECT *select,
              int use_record_cache, bool print_error)
{
    info->read_record=rr_sequential;
    table->file->ha_rnd_init(1);
}
```

Two important things are happening here. First, the info pointer to a READ_RECORD variable (passed in the arguments of init_read_records()) has had its read_record member variable changed to rr_sequential. rr_sequential is a function pointer, and setting this means that subsequent calls to info->read_record() will be translated into rr_sequential(READ_RECORD ➥ *info), which uses the record cache to retrieve data. We'll look at that function in a second. For now, just remember that all those calls to read_record() in the while loop of Listing 4-21 will hit the record cache from now on. First, however, notice the call to ha_rnd_init().

Whenever you see ha_ in front of a function, you know immediately that you're dealing with a table handler method (a storage engine function). A first guess might be that this function is used to scan a segment of records from disk for a storage engine. So, let's check out ha_rnd_init(), shown in Listing 4-23, which can be found in /sql/handler.h. Why just the header file? Well, the handler class is really just an interface for the storage engine's subclasses to implement. We can see from the class definition that a skeleton method is defined.

Listing 4-23. */sql/handler.h handler::ha_rnd_init()*

```
int ha_rnd_init(bool scan)
  {
    DBUG_ENTER("ha_rnd_init");
    DBUG_ASSERT(inited==NONE || (inited==RND && scan));
    inited=RND;
    DBUG_RETURN(rnd_init(scan));
  }
```

Since we are querying on a MyISAM table, we'll look for the *virtual* method declaration for `rnd_init()` in the `ha_myisam handler` class, as shown in Listing 4-24. This can be found in the `/sql/ha_myisam.cc` file.

Listing 4-24. */sql/ha_myisam.cc ha_myisam::rnd_init()*

```
int ha_myisam::rnd_init(bool scan)
{
  if (scan)
    return mi_scan_init(file);
// …
}
```

Sure enough, as we suspected, the `rnd_init` method involves a scan of the table's records. We're sure you've gotten tired of us saying this by now, but yes, the `mi_scan_init()` function is implemented in yet another file: `/myisam/mi_scan.c`, shown in Listing 4-25.

Listing 4-25. */myisam/mi_scan.c mi_scan_init()*

```
int mi_scan_init(register MI_INFO *info)
{
  info->nextpos=info->s->pack.header_length; /* Read first record */
// …
}
```

Unbelievable—all this work just to read in a record to a `READ_RECORD` struct! Fortunately, we're almost done. Listing 4-26 shows the `rr_sequential()` function of the record cache library.

Listing 4-26. */sql/records.cc rr_sequential()*

```
static int rr_sequential(READ_RECORD *info)
{
  while ((tmp=info->file->rnd_next(info->record)))
  {
    if (tmp == HA_ERR_END_OF_FILE)
      tmp= -1;
  }
  return tmp;
}
```

This function is now called whenever the `info` struct in `sub_select()` calls its `read_record()` member method. It, in turn, calls another MyISAM handler method, `rnd_next()`, which simply moves the current record pointer into the needed `READ_RECORD` struct. Behind the scenes, `rnd_next` simply maps to the `mi_scan()` function implemented in the same file we saw earlier, as shown in Listing 4-27.

Listing 4-27. */myisam/mi_scan.c mi_scan()*

```
int mi_scan(MI_INFO *info, byte *buf)
{
// …
  info->update&= (HA_STATE_CHANGED | HA_STATE_ROW_CHANGED);
  DBUG_RETURN ((*info->s->read_rnd)(info,buf,info->nextpos,1));
}
```

In this way, the record cache acts more like a wrapper library to the handlers than it does a cache. But what we've left out of the preceding code is much of the implementation of the shared `IO_CACHE` object, which we touched on in the section on caching earlier in this chapter. You should go back to `records.cc` and take a look at the record cache implementation now that you know a little more about how the handler subclasses interact with the main parsing and execution system. This advice applies for just about any of the sections we covered in this chapter. Feel free to go through this code execution over and over again, even branching out to discover, for instance, how an `INSERT` command is actually executed in the storage engine.

Summary

We've certainly covered a great deal of ground in this chapter. Hopefully, you haven't thrown the book away in frustration as you worked your way through the source code. We know it can be a difficult task, but take your time and read as much of the documentation as you can. It really helps.

So, what have we covered in this chapter? Well, we started off with some instructions on how to get your hands on the source code, and configure and retrieve the documentation in various formats. Then we outlined the general organization of the server's subsystems.

Each of the core subsystems was covered, including thread management, logging, storage engine abstraction, and more. We intended to give you an adequate road map from which to start investigating the source code yourself, to get an even deeper understanding of what's behind the scenes. Trust us, the more you dig in there, the more you'll be amazed at the skill of the MySQL development team to "keep it all together." There's a *lot* of code in there.

We finished up with a bit of a code odyssey, which took us from server initialization all the way through to the retrieval of data records from the storage engine. Were you surprised at just how many steps we took to travel such a relatively short distance?

We hope this chapter has been a fun little excursion into the world of database server internals. The next chapter will cover some additional advanced topics, including implementation details on the storage engines themselves and the differences between them. You'll learn the strengths and weaknesses of each of the storage engines, to gain a better understanding of when to use them.