High Performance MySQL, Third Edition
by Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko

Printed in the United States of America.

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

Editor: Andy Oram
Production Editor: Holly Bauer
Proofreader: Rachel Head
Indexer: Jay Marchand
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

March 2004:	First Edition.
June 2008:	Second Edition.
March 2012:	Third Edition.

Revision History for the Third Edition:

2012-03-01	First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449314286* for release details.

ISBN: 978-1-449-31428-6

[LSI]

1330630256

Table of Contents

CHAPTER 1
MySQL Architecture and History

MySQL is very different from other database servers, and its architectural characteristics make it useful for a wide range of purposes as well as making it a poor choice for others. MySQL is not perfect, but it is flexible enough to work well in very demanding environments, such as web applications. At the same time, MySQL can power embedded applications, data warehouses, content indexing and delivery software, highly available redundant systems, online transaction processing (OLTP), and much more.

To get the most from MySQL, you need to understand its design so that you can work with it, not against it. MySQL is flexible in many ways. For example, you can configure it to run well on a wide range of hardware, and it supports a variety of data types. However, MySQL's most unusual and important feature is its storage-engine architecture, whose design separates query processing and other server tasks from data storage and retrieval. This separation of concerns lets you choose how your data is stored and what performance, features, and other characteristics you want.

This chapter provides a high-level overview of the MySQL server architecture, the major differences between the storage engines, and why those differences are important. We'll finish with some historical context and benchmarks. We've tried to explain MySQL by simplifying the details and showing examples. This discussion will be useful for those new to database servers as well as readers who are experts with other database servers.

MySQL's Logical Architecture

A good mental picture of how MySQL's components work together will help you understand the server. Figure 1-1 shows a logical view of MySQL's architecture.

The topmost layer contains the services that aren't unique to MySQL. They're services most network-based client/server tools or servers need: connection handling, authentication, security, and so forth.

The second layer is where things get interesting. Much of MySQL's brains are here, including the code for query parsing, analysis, optimization, caching, and all the

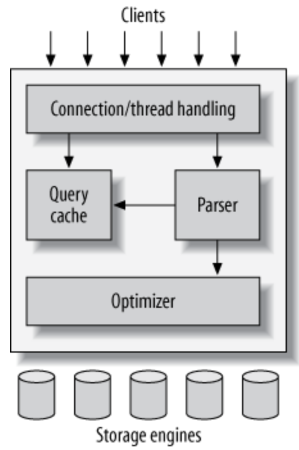

Figure 1-1. A logical view of the MySQL server architecture

built-in functions (e.g., dates, times, math, and encryption). Any functionality provided across storage engines lives at this level: stored procedures, triggers, and views, for example.

The third layer contains the storage engines. They are responsible for storing and retrieving all data stored "in" MySQL. Like the various filesystems available for GNU/Linux, each storage engine has its own benefits and drawbacks. The server communicates with them through the *storage engine API*. This interface hides differences between storage engines and makes them largely transparent at the query layer. The API contains a couple of dozen low-level functions that perform operations such as "begin a transaction" or "fetch the row that has this primary key." The storage engines don't parse SQL[1] or communicate with each other; they simply respond to requests from the server.

Connection Management and Security

Each client connection gets its own thread within the server process. The connection's queries execute within that single thread, which in turn resides on one core or CPU. The server caches threads, so they don't need to be created and destroyed for each new connection.[2]

When clients (applications) connect to the MySQL server, the server needs to authenticate them. Authentication is based on username, originating host, and password.

1. One exception is InnoDB, which does parse foreign key definitions, because the MySQL server doesn't yet implement them itself.
2. MySQL 5.5 and newer versions support an API that can accept thread-pooling plugins, so a small pool of threads can service many connections.

X.509 certificates can also be used across an SSL (Secure Sockets Layer) connection. Once a client has connected, the server verifies whether the client has privileges for each query it issues (e.g., whether the client is allowed to issue a `SELECT` statement that accesses the `Country` table in the `world` database).

Optimization and Execution

MySQL parses queries to create an internal structure (the parse tree), and then applies a variety of optimizations. These can include rewriting the query, determining the order in which it will read tables, choosing which indexes to use, and so on. You can pass hints to the optimizer through special keywords in the query, affecting its decision-making process. You can also ask the server to explain various aspects of optimization. This lets you know what decisions the server is making and gives you a reference point for reworking queries, schemas, and settings to make everything run as efficiently as possible. We discuss the optimizer in much more detail in Chapter 6.

The optimizer does not really care what storage engine a particular table uses, but the storage engine does affect how the server optimizes the query. The optimizer asks the storage engine about some of its capabilities and the cost of certain operations, and for statistics on the table data. For instance, some storage engines support index types that can be helpful to certain queries. You can read more about indexing and schema optimization in Chapter 4 and Chapter 5.

Before even parsing the query, though, the server consults the query cache, which can store only `SELECT` statements, along with their result sets. If anyone issues a query that's identical to one already in the cache, the server doesn't need to parse, optimize, or execute the query at all—it can simply pass back the stored result set. We write more about that in Chapter 7.

Concurrency Control

Anytime more than one query needs to change data at the same time, the problem of concurrency control arises. For our purposes in this chapter, MySQL has to do this at two levels: the server level and the storage engine level. Concurrency control is a big topic to which a large body of theoretical literature is devoted, so we will just give you a simplified overview of how MySQL deals with concurrent readers and writers, so you have the context you need for the rest of this chapter.

We'll use an email box on a Unix system as an example. The classic *mbox* file format is very simple. All the messages in an *mbox* mailbox are concatenated together, one after another. This makes it very easy to read and parse mail messages. It also makes mail delivery easy: just append a new message to the end of the file.

But what happens when two processes try to deliver messages at the same time to the same mailbox? Clearly that could corrupt the mailbox, leaving two interleaved messages at the end of the mailbox file. Well-behaved mail delivery systems use locking to prevent corruption. If a client attempts a second delivery while the mailbox is locked, it must wait to acquire the lock itself before delivering its message.

This scheme works reasonably well in practice, but it gives no support for concurrency. Because only a single process can change the mailbox at any given time, this approach becomes problematic with a high-volume mailbox.

Read/Write Locks

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

If you think of the mailbox as a database table and each mail message as a row, it's easy to see that the problem is the same in this context. In many ways, a mailbox is really just a simple database table. Modifying rows in a database table is very similar to removing or changing the content of messages in a mailbox file.

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

Without worrying about the actual locking technology, we can describe the concept as follows. Read locks on a resource are shared, or mutually nonblocking: many clients can read from a resource at the same time and not interfere with each other. Write locks, on the other hand, are exclusive—i.e., they block both read locks and other write locks—because the only safe policy is to have a single client writing to the resource at a given time and to prevent all reads when a client is writing.

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

Lock Granularity

One way to improve the concurrency of a shared resource is to be more selective about what you lock. Rather than locking the entire resource, lock only the part that contains the data you need to change. Better yet, lock only the exact piece of data you plan to change. Minimizing the amount of data that you lock at any one time lets changes to a given resource occur simultaneously, as long as they don't conflict with each other.

The problem is locks consume resources. Every lock operation—getting a lock, checking to see whether a lock is free, releasing a lock, and so on—has overhead. If the system spends too much time managing locks instead of storing and retrieving data, performance can suffer.

A locking strategy is a compromise between lock overhead and data safety, and that compromise affects performance. Most commercial database servers don't give you much choice: you get what is known as row-level locking in your tables, with a variety of often complex ways to give good performance with many locks.

MySQL, on the other hand, does offer choices. Its storage engines can implement their own locking policies and lock granularities. Lock management is a very important decision in storage engine design; fixing the granularity at a certain level can give better performance for certain uses, yet make that engine less suited for other purposes. Because MySQL offers multiple storage engines, it doesn't require a single general-purpose solution. Let's have a look at the two most important lock strategies.

Table locks

The most basic locking strategy available in MySQL, and the one with the lowest overhead, is *table locks*. A table lock is analogous to the mailbox locks described earlier: it locks the entire table. When a client wishes to write to a table (insert, delete, update, etc.), it acquires a write lock. This keeps all other read and write operations at bay. When nobody is writing, readers can obtain read locks, which don't conflict with other read locks.

Table locks have variations for good performance in specific situations. For example, `READ LOCAL` table locks allow some types of concurrent write operations. Write locks also have a higher priority than read locks, so a request for a write lock will advance to the front of the lock queue even if readers are already in the queue (write locks can advance past read locks in the queue, but read locks cannot advance past write locks).

Although storage engines can manage their own locks, MySQL itself also uses a variety of locks that are effectively table-level for various purposes. For instance, the server uses a table-level lock for statements such as `ALTER TABLE`, regardless of the storage engine.

Row locks

The locking style that offers the greatest concurrency (and carries the greatest overhead) is the use of *row locks*. Row-level locking, as this strategy is commonly known, is available in the InnoDB and XtraDB storage engines, among others. Row locks are implemented in the storage engine, not the server (refer back to the logical architecture diagram if you need to). The server is completely unaware of locks implemented in the