CHAPTER 6

Benchmarking and Profiling

This book departs from novice or intermediate texts in that we focus on using and developing for MySQL from a *professional* angle. We don't think the difference between a normal user and a professional user lies in the ability to recite every available function in MySQL's SQL extensions, nor in the capacity to administer large databases or high-volume applications.

Rather, we think the difference between a novice user and a professional is twofold. First, the professional has the desire to understand *why* and *how* something works. Merely knowing the steps to accomplish an activity is not enough. Second, the professional approaches a problem with an understanding that the circumstances that created the problem can and will change over time, leading to variations in the problem's environment, and consequently, a need for different solutions. The professional developer or administrator focuses on understanding how things work, and sets about to build a framework that can react to and adjust for changes in the environment.

The subject of benchmarking and profiling of database-driven applications addresses the core of this professional outlook. It is part of the foundation on which the professional's framework for understanding is built. As a professional developer, understanding how and why benchmarking is useful, and how profiling can save you and your company time and money, is critical.

As the size of an application grows, the need for a reliable method of measuring the application's performance also grows. Likewise, as more and more users start to query the database application, the need for a standardized framework for identifying bottlenecks also increases. Benchmarking and profiling tools fill this void. They create the framework on which your ability to identify problems and compare various solutions depends. Any reader who has been on a team scrambling to figure out why a certain application or web page is not performing correctly understands just how painful *not* having this framework in place can be.

Yes, setting up a framework for benchmarking your applications takes time and effort. It's not something that just happens by flipping a switch. Likewise, effectively profiling an application requires the developer and administrator to take a proactive stance. Waiting for an application to experience problems is *not* professional, but, alas, is usually the status quo, even for large applications. Above all, we want you to take from this chapter not only knowledge of how to establish benchmarks and a profiling system, but also a true understanding of the importance of each.

In this chapter, we don't assume you have any knowledge of these topics. Why? Well, one reason is that most novice and intermediate books on MySQL don't cover them. Another reason is that the vast majority of programmers and administrators we've met over the years (including ourselves at various points) have resorted to the old trial-and-error method of identifying bottlenecks and comparing changes to application code.

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

- Benefits of benchmarking
- Guidelines for conducting benchmarks
- Tools for benchmarking
- Benefits of profiling
- Guidelines for profiling
- Tools for profiling

What Can Benchmarking Do for You?

Benchmark tests allow you to measure your application's performance, both in execution speed and memory consumption. Before we demonstrate how to set up a reliable benchmarking framework, let's first examine what the results of benchmark tests can show you about your application's performance and in what situations running benchmarks can be useful. Here is a brief list of what benchmark tests can help you do:

- Make simple performance comparisons
- Determine load limits
- Test your application's ability to deal with change
- Find potential problem areas

BENCHMARKING, PROFILING—WHAT'S THE DIFFERENCE?

No doubt, you've all heard the terms *benchmarking* and *profiling* bandied about the technology schoolyard numerous times over the years. But what do these terms mean, and what's the difference between them?

Benchmarking is the practice of creating a set of performance results for a given set of tests. These tests represent the performance of an entire application or a piece of the application. The performance results are used as an indicator of how well the application or application piece performed given a specific configuration. These benchmark test results are used in comparisons between the application changes to determine the effects, if any, of that change.

Profiling, on the other hand, is a method of diagnosing the performance bottlenecks of an application. Like benchmark tests, profilers produce resultsets that can be analyzed in order to determine the pieces of an application that are problematic, either in their performance (time to complete) or their resource usage (memory allocation and utilization). But, unlike benchmark tools, which typically test the *theoretical limits* of the application, profilers show you a snapshot of what is *actually occurring* on your system.

Taken together, benchmarking and profiling tools provide a platform that can pinpoint the problem areas of your application. Benchmark tools provide you the ability to compare changes in your application, and profilers enable you to diagnose problems as they occur.

Conducting Simple Performance Comparisons

Suppose you are in the beginning phases of designing a toy store e-commerce application. You've mapped out a basic schema for the database and think you have a real winner on your hands. For the product table, you've determined that you will key the table based on the company's internal SKU, which happens to be a 50-character alphanumeric identifier. As you start to add more tables to the database schema, you begin to notice that many of the tables you're adding have foreign key references to this product SKU. Now, you start to question whether the 50-character field is a good choice, considering the large number of joined tables you're likely to have in the application's SQL code.

You think to yourself, "I wonder if this large character identifier will slow down things compared to having a trimmer, say, integer identifier?" Common sense tells you that it will, of course, but you don't have any way of determining *how* much slower the character identifier will perform. Will the performance impact be negligible? What if it isn't? Will you redesign the application to use a smaller key once it is in production?

But you don't need to just guess at the ramifications of your schema design. You can *benchmark test it and prove it*! You can determine that using a smaller integer key would result in an improvement of *x*% over the larger character key.

The results of the benchmark tests alone may not determine whether or not you decide to use an alphanumeric key. You may decide that the benefit of having a natural key, as opposed to a generated key, is worth the performance impact. But, when you have the results of your benchmarks in front of you, you're making an *informed* decision, not just a guess. The benchmark test results show you *specifically* what the impact of your design choices will be.

Here are some examples of how you can use benchmark tests in performance comparisons:

- A coworker complained that when you moved from MySQL 4.0.18 to MySQL 4.1, the performance of a specific query decreased dramatically. You can use a benchmark test against both versions of MySQL to test the claim.
- A client complained that the script you created to import products into the database from spreadsheets does not have the ability to "undo" itself if an error occurs halfway through. You want to understand how adding transactions to the script will affect its performance.
- You want to know whether replacing the normal B-tree index on your `product.name varchar(150)` field with a full-text index will increase search speeds on the product name once you have 100,000 products loaded into the database.
- How will the performance of a `SELECT` query against three of your tables be affected by having 10 concurrent client connections compared with 20, 30, or 100 client connections?

Determining Load Limits

Benchmarks also allow you to determine the limitations of your database server under load. By *load*, we simply mean a heavy level of activity from clients requesting data from your application. As you'll see in the "Benchmarking Tools" section later in this chapter, the benchmarking tools you will use allow you to test the limits, measured in the number of queries performed per second, given a supplied number of concurrent connections. This ability to provide insight into the stress level under which your hardware and application will most likely fail is an invaluable tool in assessing both your hardware and software configuration.

Determining load limits is particularly of interest to web application developers. You want to know *before a failure occurs* when you are approaching a problematic volume level for the web server and database server. A number of web application benchmarking tools, commonly called *load generators*, measure these limits effectively. Load generators fall into two general categories:

Contrived load generator: This type of load generator makes no attempt to simulate actual web traffic to a server. Contrived load generators use a sort of brute-force methodology to push concurrent requests for a specific resource through the pipeline. In this way, contrived load generation is helpful in determining a particular web page's limitations, but these results are often *theoretical*, because, as we all know, few web sites receive traffic to only a single web page or resource. Later in this chapter, we'll take a look at the most common contrived load generator available to open-source web application developers: ApacheBench.

Realistic load generator: On the flip side of the coin, realistic load generators attempt to determine load limitations based on actual traffic patterns. Typically, these tools will use actual web server log files in order to simulate typical user sessions on the site. These realistic load generation tools can be very useful in determining the limitations of the overall system, not just a specific piece of one, because the entire application is put through the ropes. An example of a benchmarking tool with the capability to do realistic load generation is httperf, which is covered later in this chapter.

Testing an Application's Ability to Deal with Change

To continue our online store application example, suppose that after running a few early benchmark tests, you determine that the benefits of having a natural key on the product SKU outweigh the performance impact you found—let's say, you discovered an 8% performance degradation. However, in these early benchmark tests, you used a test data set of 10,000 products and 100,000 orders.

While this might be a realistic set of test data for the first six months into production, it might be significantly less than the size of those tables in a year or two. Your benchmark framework will show you how your application will perform with a larger database size, and in doing so, will help you to be realistic about when your hardware or application design may need to be refactored.

Similarly, if you are developing commercial-grade software, it is imperative that you know how your database design will perform under varying database sizes and hardware configurations. Larger customers may often *demand* to see performance metrics that match closely their projected database size and traffic. Your benchmarking framework will allow you to provide answers to your clients' questions.

Finding Potential Problem Areas

Finally, benchmark tests give you the ability to identify potential problems on a broad scale. More than likely, a benchmark test result won't show you what's wrong with that faulty loop you just coded. However, the test can be very useful for determining which general parts of an application or database design are the weakest.

For example, let's say you run a set of benchmark tests for the main pages in your toy store application. The results show that of all the pages, the page responsible for displaying the order history has the worst performance; that is, the least number of concurrent requests for the order history page could be performed by the benchmark. This shows you the *area* of the application that could be a potential problem. The benchmark test results won't show you the specific code blocks of the order history page that take the most resources, but the benchmark points you in the direction of the problem. Without the benchmark test results, you would be forced to wait until the customer service department started receiving complaints about slow application response on the order history page.

As you'll see later in this chapter, profiling tools enable you to see which specific blocks of code are problematic in a particular web page or application screen.

General Benchmarking Guidelines

We've compiled a list of general guidelines to consider as you develop your benchmarking framework. This list highlights strategies you should adopt in order to most effectively diagnose the health and growth prospects of your application code:

- Set real performance standards.
- Be proactive.
- Isolate the changed variables.
- Use real data sets.
- Make small changes and then rerun benchmarks.
- Turn off unnecessary programs and the query cache.
- Repeat tests to determine averages.
- Save benchmark results.

Let's take a closer look at each of these guidelines.

Setting Real Performance Standards

Have you ever been on the receiving end of the following statement by a fellow employee or customer? "Your application is really slow today." (We bet just reading it makes some of you cringe. Hey, we've all been there at some point or another.) You might respond with something to the effect of, "What does 'really slow' mean, ma'am?"

As much as you may not want to admit it, this situation is *not* the customer's fault. The problem has arisen due to the fact that the customer's *perception* of the application's performance is that there has been a slowdown compared with the *usual* level of performance. Unfortunately for you, there isn't anything written down anywhere that states *exactly* what the usual performance of the application is.

Not having a clear understanding of the acceptable performance standards of an application can have a number of ramifications. Working with the project stakeholders to determine performance standards helps involve the end users at an early stage of the development and gives the impression that your team cares about their perceptions of the application's

performance and what an acceptable response time should be. As any project manager can tell you, setting expectations is one of the most critical components of a successful project. From a performance perspective, you should endeavor to set at least the following acceptable standards for your application:

Response times: You should know what the stakeholders and end users consider an acceptable response time for most application pieces from the outset of the project. For each application piece, work with business experts, and perhaps conduct surveys, to determine the threshold for how fast your application should return results to the user. For instance, for an e-commerce application, you would want to establish acceptable performance metrics for your shopping cart process: adding items to the cart, submitting an order, and so on. The more specific you can be, the better. If a certain process will undoubtedly take more time than others, as might be the case with an accounting data export, be sure to include *realistic* acceptable standards for those pieces.

Concurrency standards: Determining predicted levels of concurrency for a fledging project can sometimes be difficult. However, there is definite value to recording the stakeholders' expectation of how many users should be able to concurrently use the application under a normal traffic volume. For instance, if the company expects the toy store to be able to handle 50 customers simultaneously, then benchmark tests must test against those expectations.

Acceptable deviation: No system's traffic and load are static. Fluctuations in concurrency and request volumes naturally occur on all major applications, and it is important to set expectations with the stakeholders as to a normal deviation from acceptable standards. Typically, this is done by providing for a set interval during which performance standards may fluctuate a certain percentage. For instance, you might say that having performance degrade 10% over the course of an hour falls within acceptable performance standards. If the performance decrease lasts longer than this limit, or if the performance drops by 30%, then acceptable standards have not been met.

Use these performance indicators in constructing your baselines for benchmark testing. When you run entire application benchmarks, you will be able to confirm that the current database performance meets the acceptable standards set by you and your stakeholders. Furthermore, you can determine how the growth of your database and an increase in traffic to the site might threaten these goals.

The main objective here is to have these goals *in writing*. This is critical to ensuring that expectations are met. Additionally, having the performance standards on record allows your team to evaluate its work with a real set of guidelines. Without a record of acceptable standards and benchmark tests, you'll just be guessing that you've met the client's requirements.

Being Proactive

Being proactive goes to the heart of what we consider to be a professional outlook on application development and database administration. Your goal is to identify problems *before* they occur. Being *reactive* results in lost productivity and poor customer experience, and can significantly mar your development team's reputation. There is nothing worse than working in an IT department that is constantly "fighting fires." The rest of your company will come to view the team as inexperienced, and reach the conclusion that you didn't design the application properly in the first place.

Don't let reactive attitudes tarnish your project team. Take up the fight from the start by including benchmark testing as an integral part of your development process. By harnessing the power of your benchmarking framework, you can predict problems well before they rear their ugly heads.

Suppose early benchmark tests on your existing hardware have shown your e-commerce platform's performance will degrade rapidly once 50 concurrent users are consistently querying the database. Knowing that this limit will eventually be reached, you can run benchmarks against other hardware configurations or even different configurations of the MySQL server variables to determine if changes will make a substantial impact. You can then turn to the management team and show, certifiably, that without an expenditure of, say, $3,000 for new hardware, the web site will fall below the acceptable performance standards.

The management team will appreciate your ability to solve performance problems *before* they occur and provide real test results as opposed to a guess.

Isolating Changed Variables

When testing application code, or configurations of hardware or software, always isolate the variable you wish to test. This is an important scientific principle: in order to show a correlation between one variable and a test result, you must ensure that *all other things remain equal*.

You must ensure that the tests are run in an identical fashion, with no other changes to the test other than those tested for. In real terms, this means that when you run a benchmark to test that your integer product key is faster than your character product key, the only difference between the two benchmarks should be the product table's key field data type. If you make other changes to the schema, or run the tests against different data sets, you dilute the test result, and you cannot reliably state that the difference in the benchmark results is due to the change in the product key's data type.

Likewise, if you are testing to determine the impact of a SQL statement's performance given a twentyfold increase in the data set's size, the only difference between the two benchmarks should be the number of rows being operated upon.

Because it takes time to set up and to run benchmarks, you'll often be tempted to take shortcuts. Let's say you have a suspicion that if you increase the `key_buffer_size`, `query_cache_size`, and `sort_buffer_size` server system variables in your `my.cnf` file, you'll get a big performance increase. So, you run the test with and without those variable changes, and find you're absolutely right! The test showed a performance increase of 4% over the previous run. You've guessed correctly that your changes would increase throughput and performance, but, sadly, you're operating on false assumptions. You've assumed, because the test came back with an overall increase in performance, that increasing all three system variable values each improves the performance of the application. What if the changes to the `sort_buffer_size` and `query_cache_size` increased throughput by 5%, but the change in the `key_buffer_size` variable decreased performance by 1%? You wouldn't know this was the case. So, the bottom line is that you should try to isolate a single changed variable in your tests.

Using Real Data Sets

To get the most accurate results from your benchmark tests, try to use data sets from actual database tables, or at least data sets that represent a realistic picture of the data to be stored in your future tables. If you don't have actual production tables to use in your testing, you can use a data generator to produce sample data sets. We'll demonstrate a simple generation tool

(the `gen-data` program that accompanies Super Smack) a little later in this chapter, but you may find that writing your own homegrown data set generation script will produce test sets that best meet your needs.

When trying to create or collect realistic test data sets, consider key selectivity, text columns, and the number of rows.

Key Selectivity

Try to ensure that fields in your tables on which indexes will be built contain a distribution of key values that accurately depicts the real application. For instance, assume you have an `orders` table with a `char(1)` field called `status` containing one of ten possible values, say, the letters `A` through `J` to represent the various stages that order can be in during its lifetime. You know that once the `orders` table is filled with production data, more than 70% of the `status` field values will be in the `J` stage, which represents a closed, completed order.

Suppose you run benchmark tests for an order-report SQL statement that summarizes the orders filtered by their status, and this statement uses an index on the `status` field. If your test data set uses an equal distribution of values in the `status` column—perhaps because you used a data generation program that randomly chose the `status` value—your test will likely be skewed. In the real-world database, the likelihood that the optimizer would choose an index on the `status` column might be much less than in your test scenario. So, when you generate data sets for use in testing, make sure you investigate the selectivity of indexed fields to ensure the generated data set approximates the real-world distribution as closely as possible.

Text Columns

When you are dealing with larger text columns, especially ones with varying lengths, try to put a realistic distribution of text lengths into your data sets. This will provide a much more accurate depiction of how your database will perform in real-world scenarios.

If you load a test data set with similarly sized rows, the performance of the benchmark may not accurately reflect a true production scenario, where a table's data pages contain varying numbers of rows because of varying length text fields. For instance, let's say you have a table in your e-commerce database that stores customer product reviews. Clearly, these reviews can vary in length substantially. It would be imprudent to run benchmarks against a data set you've generated with 100,000 records, each row containing a text field with 1,000 bytes of character data. It's simply not a realistic depiction of the data that would actually fill the table.

Number of Rows

If you actually have millions of orders completed in your e-commerce application, but run benchmarks against a data set of only 100,000 records, your benchmarks will not represent the reality of the application, so they will be essentially useless to you. The benchmark run against 100,000 records may depict a scenario in which the server was able to cache in memory most or all of the order records. The same benchmark performed against two million order records may yield dramatically lower load limits because the server was not able to cache all the records.

Making Small Changes and Rerunning Benchmarks

The idea of making only small changes follows nicely from our recommendation of always isolating a single variable during testing. When you do change a variable in a test case, make small changes if you are adjusting settings. If you want to see the effects on the application's load limits given a change in the `max_user_connections` setting, adjust the setting in small increments and rerun the test, noting the effects. "Small" is, of course, relative, and will depend on the specific setting you're changing. The important thing is to continue making similar adjustments in subsequent tests.

For instance, you might run a baseline test for the existing `max_user_connections` value. Then, on the next tests, you increase the value of the `max_user_connections` value by 20 each time, noting the increase or decrease in the queries per second and concurrency thresholds in each run. Usually, your end goal will be to determine the optimal setting for the `max_user_connections`, given your hardware configuration, application design, and database size.

By plotting the results of your benchmark tests and keeping changes at a small, even pace, you will be able to more finely analyze where the optimal setting of the tested variable should be.

Turning Off Unnecessary Programs and the Query Cache

When running benchmark tests against your development server to determine the difference in performance between two methods or SQL blocks, make sure you turn off any unnecessary programs during testing, because they might interfere or obscure a test's results. For instance, if you run a test for one block of code, and, during the test for a comparison block of code a `cron` job is running in the background, the test results might be skewed, depending on how much processing power is being used by the job.

Typically, you should make sure only necessary services are running. Make sure that any backup jobs are disabled and won't run during the testing. Remember that the whole purpose is to isolate the test environment as much as possible.

Additionally, we like to turn off the query cache when we run certain performance comparisons. We want to ensure that one benchmark run isn't benefiting from the caching of resultsets inserted into the query cache during a previous run. To disable the query cache, you can simply set the `query_cache_size` variable to 0 before the run:

```
mysql> SET GLOBALS query_cache_size = 0;
```

Just remember to turn it back on when you need it!

Repeating Tests to Determine Averages

Always repeat your benchmark tests a number of times. You'll sometimes find that the test results come back with slightly different numbers each time. Even if you've shut down all nonessential processes on the testing server and eliminated the possibility that other programs or scripts may interfere with the performance tests, you still may find some discrepancies from test to test. So, in order to get an accurate benchmark result, it's often best to take a series of the same benchmark, and then average the results across all test runs.

Saving Benchmark Results

Always save the results of your benchmarks for future analysis and as baselines for future benchmark tests. Remember that when you do performance comparisons, you want a baseline test to compare the change to. Having a set of saved benchmarks also allows you to maintain a record of the changes you made to your hardware, application configuration, and so on, which can be a valuable asset in tracking where and when problems may have occurred.

Benchmarking Tools

Now that we've taken a look at how benchmarking can help you and some specific strategies for benchmarking, let's get our hands dirty. We're going to show you a set of tools that, taken together, can provide the start of your benchmarking framework. Each of these tools has its own strengths, and you will find a use for each of them in different scenarios. We'll investigate the following tools:

- MySQL benchmarking suite
- MySQL Super Smack
- MyBench
- ApacheBench
- httperf

MySQL's Benchmarking Suite

MySQL comes with its own suite of benchmarking tools, available in the source distribution under the `/sql-bench` directory. This suite of benchmarking shell and Perl scripts is useful for testing differences between installed versions of MySQL and testing differences between MySQL running on different hardware. You can also use MySQL's benchmarking tools to compare MySQL with other database server systems, like Oracle, PostgreSQL, and Microsoft SQL Server.

■ **Tip** Of course, many benchmark tests have already been run. You can find some of these tests in the source distribution in the `/sql-bench/Results` directory. Additionally, you can find other non-MySQL-generated benchmarks at `http://www.mysql.com/it-resources/benchmarks/`.

In addition to the benchmarking scripts, the `crash-me` script available in the `/sql-bench` directory provides a handy way to test the feature set of various database servers. This script is also available on MySQL's web site: `http://dev.mysql.com/tech-resources/features.html`.

However, there is one major flaw with the current benchmark tests: they run in a serial manner, meaning statements are issued one after the next in a brute-force manner. This means that if you want to test differences between hardware with multiple processes, you will need to use a different benchmarking toolset, such as MyBench or Super Smack, in order

to get reliable results. Also note that this suite of tools is *not* useful for testing your own specific applications, because the tools test only a specific set of generic SQL statements and operations.

Running All the Benchmarks

Running the MySQL benchmark suite of tests is a trivial matter, although the tests themselves can take quite a while to execute. To execute the full suite of tests, simply run the following:

```
#> cd /path/to/mysqlsrc/sql-bench
#> ./run-all-tests [options]
```

Quite a few parameters may be passed to the `run-all-tests` script. The most notable of these are outlined in Table 6-1.

Table 6-1. *Parameters for Use with MySQL Benchmarking Test Scripts*

Option	Description
`--server='`*`server name`*`'`	Specifies which database server the benchmarks should be run against. Possible values include `'MySQL'`, `'MS-SQL'`, `'Oracle'`, `'DB2'`, `'mSQL'`, `'Pg'`, `'Solid'`, `'Sybase'`, `'Adabas'`, `'AdabasD'`, `'Access'`, `'Empress'`, and `'Informix'`.
`--log`	Stores the results of the tests in a directory specified by the `--dir` option (defaults to `/sql-bench/output`). Result files are named in a format `RUN-`*`xxx`*, where *xxx* is the platform tested; for instance, `/sql-bench/output/RUN-mysql-Linux_2.6.10_1.766_FC3_i686`. If this looks like a formatted version of `#> uname -a`, that's because it is.
`--dir`	Directory for logging output (see `--log`).
`--use-old-result`	Overwrites any existing logged result output (see `--log`).
`--comment`	A convenient way to insert a comment into the result file indicating the hardware and database server configuration tested.
`--fast`	Lets the benchmark framework use non-ANSI-standard SQL commands if such commands can make the querying faster.
`--host='host'`	Very useful option when running the benchmark test from a remote location. `'Host'` should be the host address of the remote server where the database is located; for instance `'www.xyzcorp.com'`.
`--small-test`	Really handy for doing a short, simple test to ensure a new MySQL installation works properly on the server you just installed it on. Instead of running an exhaustive benchmark, this forces the suite to verify only that the operations succeeded.
`--user`	User login.
`--password`	User password.

So, if you wanted to run all the tests against the MySQL database server, logging to an output file and simply verifying that the benchmark tests worked, you would execute the following from the `/sql-bench` directory:

```
#> ./run-all-tests --small-test --log
```

Viewing the Test Results

When the benchmark tests are finished, the script states:

```
Test finished.  You can find the result in:
output/RUN-mysql-Linux_2.6.10_1.766_FC3_i686
```

To view the result file, issue the following command:

```
#> cat output/RUN-mysql-Linux_2.6.10_1.766_FC3_i686
```

The result file contains a summary of all the tests run, including any parameters that were supplied to the benchmark script. Listing 6-1 shows a small sample of the result file.

Listing 6-1. *Sample Excerpt from RUN-mysql-Linux_2.6.10_1.766_FC3_i686*

```
… omitted
alter-table: Total time:  2 wallclock secs ( 0.03 usr  0.01 sys +  0.00 cusr  0.00 \
 csys =  0.04 CPU)
ATIS: Total time:  6 wallclock secs ( 1.61 usr  0.29 sys +  0.00 cusr  0.00 \
 csys =  1.90 CPU)
big-tables: Total time:  0 wallclock secs ( 0.14 usr  0.05 sys +  0.00 cusr  0.00 \
 csys =  0.19 CPU)
connect: Total time:  2 wallclock secs ( 0.58 usr  0.16 sys +  0.00 cusr  0.00 \
 csys =  0.74 CPU)
create: Total time:  1 wallclock secs ( 0.08 usr  0.01 sys +  0.00 cusr  0.00 \
 csys =  0.09 CPU)
insert: Total time:  9 wallclock secs ( 3.32 usr  0.68 sys +  0.00 cusr  0.00 \
 csys =  4.00 CPU)
select: Total time: 14 wallclock secs ( 5.22 usr  0.63 sys +  0.00 cusr  0.00 \
 csys =  5.85 CPU)
… omitted
```

As you can see, the result file contains a summary of how long each test took to execute, in "wallclock" seconds. The numbers in parentheses, to the right of the wallclock seconds, show the amount of time taken by the script for some housekeeping functionality; they represent the part of the total seconds that should be disregarded by the benchmark as simply overhead of running the script.

In addition to the main `RUN-xxx` output file, you will also find in the `/sql-bench/output` directory nine other files that contain detailed information about each of the tests run in the benchmark. We'll take a look at the format of those detailed files in the next section (Listing 6-2).

Running a Specific Test

The MySQL benchmarking suite gives you the ability to run one specific test against the database server, in case you are concerned about the performance comparison of only a particular set of operations. For instance, if you just wanted to run benchmarks to compare connection operation performance, you could execute the following:

```
#> ./test-connect
```

This will start the benchmarking process that runs a series of loops to compare the connection process and various SQL statements. You should see the script informing you of various tasks it is completing. Listing 6-2 shows an excerpt of the test run.

Listing 6-2. *Excerpt from ./test-connect*

```
Testing server 'MySQL 5.0.2 alpha' at 2005-03-07  1:12:54

Testing the speed of connecting to the server and sending of data
Connect tests are done 10000 times and other tests 100000 times

Testing connection/disconnect
Time to connect (10000): 13 wallclock secs \
 ( 8.32 usr  1.03 sys +  0.00 cusr  0.00 csys =  9.35 CPU)

Test connect/simple select/disconnect
Time for connect+select_simple (10000): 17 wallclock secs \
 ( 9.18 usr  1.24 sys +  0.00 cusr  0.00 csys = 10.42 CPU)

Test simple select
Time for select_simple (100000): 10 wallclock secs \
 ( 2.40 usr  1.55 sys +  0.00 cusr  0.00 csys =  3.95 CPU)
… omitted

Total time: 167 wallclock secs \
 (58.90 usr 17.03 sys +  0.00 cusr  0.00 csys = 75.93 CPU)
```

As you can see, the test output shows a detailed picture of the benchmarks performed.

You can use these output files to analyze the effects of changes you make to the MySQL server configuration. Take a baseline benchmark script, like the one in Listing 6-2, and save it. Then, after making the change to the configuration file you want to test—for instance, changing the `key_buffer_size` value—rerun the same test and compare the output results to see if, and by how much, the performance of your benchmark tests have changed.

MySQL Super Smack

Super Smack is a powerful, customizable benchmarking tool that provides load limitations, in terms of queries per second, of the benchmark tests it is supplied. Super Smack works by processing a custom configuration file (called a *smack file*), which houses instructions on how to process one or more series of queries (called *query barrels* in smack lingo). These configuration files are the heart of Super Smack's power, as they give you the ability to customize the processing of your SQL queries, the creation of your test data, and other variables.

Before you use Super Smack, you need to download and install it, since it does not come with MySQL. Go to `http://vegan.net/tony/supersmack` and download the latest version of Super Smack from Tony Bourke's web site.[1] Use the following to install Super Smack, after

1. Super Smack was originally developed by Sasha Pachev, formerly of MySQL AB. Tony Bourke now maintains the source code and makes it available on his web site (`http://vegan.net/tony/`).

changing to the directory where you just downloaded the tar file to (we've downloaded version 1.2 here; there may be a newer version of the software when you reach the web site):

```
#> tar -xzf super-smack-1.2.tar.gz
#> cd super-smack-1.2
#> ./configure –with-mysql
#> make install
```

Running Super Smack

Make sure you're logged in as a root user when you install Super Smack. Then, to get an idea of what the output of a sample smack run is, execute the following:

```
#> super-smack -d mysql smacks/select-key.smack 10 100
```

This command fires off the `super-smack` executable, telling it to use MySQL (`-d mysql`), passing it the smack configuration file located in `smack/select-key.smack`, and telling it to use 10 concurrent clients and to repeat the tests in the smack file 100 times for each client.

You should see something very similar to Listing 6-3. The `connect` times and `q_per_s` values may be different on your own machine.

Listing 6-3. *Executing Super Smack for the First Time*

```
Error running query select count(*) from http_auth: \
Table 'test.http_auth' doesn't exist
Creating table 'http_auth'
Populating data file '/var/smack-data/words.dat' \
with # command 'gen-data -n 90000 -f %12-12s%n,%25-25s,%n,%d'
Loading data from file '/var/smack-data/words.dat' into table 'http_auth'
Table http_auth is now ready for the test
Query Barrel Report for client smacker1
connect: max=4ms  min=0ms avg= 1ms from 10 clients
Query_type      num_queries     max_time        min_time        q_per_s
select_index    2000            0               0               4983.79
```

Let's walk through what's going on here. Going from the top of Listing 6-3, you see that when Super Smack started the benchmark test found in `smack/select-key.smack`, it tried to execute a query against a table (`http_auth`) that didn't exist. So, Super Smack created the `http_auth` table. We'll explain how Super Smack knew how to create the table in just a minute. Moving on, the next two lines tell you that Super Smack created a test data file (`/var/smack-data/words.dat`) and loaded the test data into the `http_auth` table.

■ **Tip** As of this writing, Super Smack can also benchmark against the PostgreSQL database server (using the `-d pg` option). See the file `TUTORIAL` located in the `/super-smack` directory for some details on specifying PostgreSQL parameters in the smack files.

Finally, under the line `Query Barrel Report for client smacker1`, you see the output of the benchmark test (highlighted in Listing 6-3). The first highlighted line shows a breakdown of the times taken to connect for the clients we requested. The number of clients should match the number from your command line. The following lines contain the output results of each type of query contained in the smack file. In this case, there was only one query type, called `select_index`. In our run, Super Smack executed 2,000 queries for the `select_index` query type. The corresponding output line in Listing 6-3 shows that the minimum and maximum times for the queries were all under 1 millisecond (thus, 0), and that 4,982.79 queries were executed per second (`q_per_s`). This last statistic, `q_per_s`, is what you are most interested in, since this statistic gives you the best number to compare with later benchmarks.

■ **Tip** Remember to rerun your benchmark tests and average the results of the tests to get the most accurate benchmark results. If you rerun the smack file in Listing 6-3, even with the same parameters, you'll notice the resulting `q_per_s` value will be slightly different almost every time, which demonstrates the need for multiple test runs.

To see how Super Smack can help you analyze some useful data, let's run the following slight variation on our previous shell execution. As you can see, we've changed only the number of concurrent clients, from 10 to 20.

```
#> super-smack -d mysql smacks/select-key.smack 20 100
Query Barrel Report for client smacker1
connect: max=206ms  min=0ms avg= 18ms from 20 clients
Query_type      num_queries     max_time        min_time        q_per_s
select_index    4000            0               0               5054.71
```

Here, you see that increasing the number of concurrent clients actually *increased* the performance of the benchmark test. You can continue to increment the number of clients by a small amount (increments of ten in this example) and compare the `q_per_s` value to your previous runs. When you start to see the value of `q_per_s` decrease or level off, you know that you've hit your peak performance for this benchmark test configuration.

In this way, you perform a process of *determining an optimal condition*. In this scenario, the condition is the number of concurrent clients (the variable you're changing in each iteration of the benchmark). With each iteration, you come closer to determining the optimal value of a specific variable in your scenario. In our case, we determined that for the queries being executed in the `select-key.smack` benchmark, the optimal number of concurrent client connections would be around 30—that's where this particular laptop peaked in queries per second. Pretty neat, huh?

But, you might ask, how is this kind of benchmarking applicable to a real-world example? Clearly, `select-key.smack` doesn't represent much of anything (just a simple `SELECT` statement, as you'll see in a moment). The real power of Super Smack lies in the customizable nature of the smack configuration files.

Building Smack Files

You can build your own smack files to represent either your whole application or pieces of the application. Let's take an in-depth look at the components of the `select-key.smack` file, and you'll get a feel for just how powerful this tool can be. Do a simple `#> cat smacks/select-key.smack` to display the smack configuration file you used in the preliminary benchmark tests. You can follow along as we walk through the pieces of this file.

■ **Tip** When creating your own smack files, it's easiest to use a copy of the sample smack files included with Super Smack. Just do `#> cp smacks/select-key.smack smacks/mynew.smack` to make a new copy. Then modify the `mynew.smack` file.

Configuration smack files are composed of sections, formatted in a way that resembles C syntax. These sections define the following parts of the benchmark test:

- *Client configuration*: Defines a named client for the smack program (you can view this as a client connection to the database).
- *Table configuration*: Names and defines a table to be used in the benchmark tests.
- *Dictionary configuration*: Names and describes a source for data that can be used in generating test data.
- *Query definition*: Names one or more SQL statements to be run during the test and defines what those SQL statements should do, how often they should be executed, and what parameters and variables should be included in the statements.
- *Main*: The execution component of Super Smack.

Going from the top of the smack file to the bottom, let's take a look at the code.

First Client Configuration Section

Listing 6-4 shows the first part of `select-key.smack`.

Listing 6-4. *Client Configuration in select-key.smack*

```
// this is will be used in the table section
client "admin"
{
 user "root";
 host "localhost";
 db "test";
 pass "";
 socket "/var/lib/mysql/mysql.sock"; // this only applies to MySQL and is
// ignored for PostgreSQL
}
```

This is pretty straightforward. This section of the smack file is naming a new client for the benchmark called `admin` and assigning some connection properties for the client. You can create any number of named client components, which can represent various connections to the various databases. We'll take a look at the second client configuration in the `select-key.smack` file soon. But first, let's examine the next configuration section in the file.

Table Configuration Section

Listing 6-5 shows the first defined table section.

Listing 6-5. *Table Section Definition in select-key.smack*

```
// ensure the table exists and meets the conditions
table "http_auth"
{
  client "admin"; // connect with this client
 // if the table is not found or does not pass the checks, create it
 // with the following, dropping the old one if needed
  create "create table http_auth
    (username char(25) not null primary key,
     pass char(25),
     uid integer not null,
     gid integer not null
    )";
  min_rows "90000"; // the table must have at least that many rows
  data_file "words.dat"; // if the table is empty, load the data from this file
  gen_data_file "gen-data -n 90000 -f %12-12s%n,%25-25s,%n,%d";
// if the file above does not exist, generate it with the above shell command
// you can replace this command with anything that prints comma-delimited
// data to stdout, just make sure you have the right number of columns
}
```

Here, you see we're naming a new table configuration section, for a table called `http_auth`, and defining a create statement for the table, in case the table does not exist in the database. Which database will the table be created in? The database used by the client specified in the table configuration section (in this case the client `admin`, which we defined in Listing 6-4).

The lines after the create definition are used by Super Smack to populate the `http_auth` table with data, if the table has less than the `min_rows` value (here, 90,000 rows). The `data_file` value specifies a file containing comma-delimited data to fill the `http_auth` table. If this file does not exist in the `/var/smack-data` directory, Super Smack will use the command given in the `gen_data_file` value in order to create the data file needed.

In this case, you can see that Super Smack is executing the following command in order to generate the `words.dat` file:

```
#> gen-data -n 90000 -f %12-12s%n,%25-25s,%n,%d
```

`gen-data` is a program that comes bundled with Super Smack. It enables you to generate random data files using a simple command-line syntax similar to C's `fprintf()` function. The `-n [rows]` command-line option tells `gen-data` to create 90,000 rows in this case, and the `-f` option is followed by a formatting string that can take the tokens listed in Table 6-2. The

formatting string then outputs randomized data to the file in the `data_file` value, delimited by whichever delimiter is used in the format string. In this case, a comma was used to delimit fields in the data rows.

Table 6-2. *Super Smack gen-data -f Option Formatting Tokens*

Token	Used For	Comments
`%[min][-][max]s`	String fields	Prints strings of lengths between the *min* and *max* values. For example, `%10-25s` creates a character field between 10 and 25 characters long. For fixed-length character fields, simply set `min` equal to the maximum number of characters.
`%n`	Row numbers	Puts an integer value in the field with the value of the row number. Use this to simulate an auto-increment column.
`%d`	Integer fields	Creates a random integer number. The version of `gen-data` that comes with Super Smack 1.2 *does not* allow you to specify the length of the numeric data produced, so `%07d` does *not* generate a seven-digit number, but a random integer of a random length of characters. In our tests, `gen-data` simply generated 7-, 8-, 9-, and 10-character length positive integers.

You can optionally choose to substitute your own scripts or executables in place of the simple `gen-data` program. For instance, if you had a Perl script `/tests/create-test-data.pl`, which created custom test tables, you could change the `table` configuration section's `gen-data-file` value as follows:

```
gen-data-file "perl /tests/create-test-data.pl"
```

POPULATING TEST SETS WITH GEN-DATA

`gen-data` is a neat little tool that you can use in your scripts to generate randomized data. `gen-data` prints its output to the standard output (stdout) by default, but you can redirect that output to your own scripts or another file. Running `gen-data` in a console, you might see the following results:

```
#> gen-data -n 12 -f %10-10s,%n,%d,%10-40s
ilcpsklryv,1,1025202362,pjnbpbwllsrehfmxr
kecwitrsgl,2,1656478042,xvtjmxypunbqfgxmuvg
fajclfvenh,3,1141616124,huorjosamibdnjdbeyhkbsomb
ltouujdrbw,4,927612902,rcgbflqpottpegrwvgajcrgwdlpgitydvhedt
usippyvxsu,5,150122846,vfenodqasajoyomgsqcpjlhbmdahyvi
uemkssdsld,6,1784639529,esnnngpesdntrrvysuipywatpfoelthrowhf
exlwdysvsp,7,87755422,kfblfdfultbwpiqhiymmy
alcyeasvxg,8,2113903881,itknygyvjxnspubqjppj
brlhugesmm,9,1065103348,jjlkrmgbnwvftyveolprfdcajiuywtvg
fjrwwaakwy,10,1896306640,xnxpypjgtlhf
teetxbafkr,11,105575579,sfvrenlebjtccg
jvrsdowiix,12,653448036,dxdiixpervseavnwypdinwdrlacv
```

You can use a redirect to output the results to a file, as in this example:

```
#> gen-data -n 12 -f %10-10s,%n,%d,%10-40s > /test-data/table1.dat
```

A number of enhancements could be made to `gen-data`, particularly in the creation of more random data samples. You'll find that rerunning the `gen-data` script produces the same results under the same session. Additionally, the formatting options are quite limited, especially for the delimiters it's capable of producing. We tested using the standard `\t` character escape, which produces just a `"t"` character when the format string was left unquoted, and a literal `"\t"` when quoted. Using `";"` as a delimiter, you must remember to use double quotes around the format string, as your console will interpret the string as multiple commands to execute.

Regardless of these limitations, `gen-data` is an excellent tool for quick generation, especially of text data. Perhaps there will be some improvements to it in the future, but for now, it seems that the author provided a simple tool under the assumption that developers would generally prefer to write their own scripts for their own custom needs.

As an alternative to `gen-data`, you can always use a simple SQL statement to dump existing data into delimited files, which Super Smack can use in benchmarking. To do so, execute the following:

```
SELECT field1, field2, field3 INTO OUTFILE "/test-data/test.csv"
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY "\n"
FROM table1
```

You should substitute your own directory for our `/test-data/` directory in the code. Ensure that the `mysql` user has write permissions for the directory as well.

Remember that Super Smack looks for the data file in the `/var/smack-data` directory by default (you can configure it to look somewhere else during installation by using the `--datadir` configure option). So, copy your test file over to that directory before running a smack file that looks for it:

```
#> cp /test-data/test.csv /var/smack-data/test.csv
```

Dictionary Configuration Section

The next configuration section is to configure the dictionary, which is named `word` in `select-key.smack`, as shown in Listing 6-6.

Listing 6-6. *Dictionary Configuration Section in select-key.smack*

```
//define a dictionary
dictionary "word"
{
  type "rand"; // words are retrieved in random order
  source_type "file"; // words come from a file
  source "words.dat"; // file location
  delim ","; // take the part of the line before,
  file_size_equiv "45000"; // if the file is greater than this
//divive the real file size by this value obtaining N and take every Nth
//line skipping others. This is needed to be able to target a wide key
// range without using up too much memory with test keys
}
```

This structure defines a `dictionary` object named `word`, which Super Smack can use in order to find rows in a `table` object. You'll see how the `dictionary` object is used in just a moment. For now, let's look at the various options a dictionary section has. The variables are not as straightforward as you might hope.

The `source_type` variable is where to find or generate the dictionary entries; that is, where to find data to put into the array of entries that can be retrieved by Super Smack from the dictionary. The `source_type` can be one of the following:

- `"file"`: If `source_type = "file"`, the `source` value will be interpreted as a file path relative to the data directory for Super Smack. By default, this directory is `/var/smack-data`, but it can be changed with the `./configure --with-datadir=DIR` option during installation. Super Smack will load the dictionary with entries consisting of the *first* field in the row. This means that if the `source` file is a comma-delimited data set (like the one generated by `gen-data`), only the first character field (up to the comma) will be used as an entry. The rest of the row is discarded.

- `"list"`: When `source_type = "list"`, the `source` value must consist of a list of comma-separated values that will represent the entries in the dictionary. For instance, `source = "cat,dog,owl,bird"` with a `source_type` of `"list"` produces four entries in the dictionary for the four animals.

- `"template"`: If the `"template"` value is used for the `source_type` variable, the `source` variable must contain a valid `printf()`[2] format string, which will be used to generate the needed dictionary entries when the dictionary is called by a `query` object. When the `type` variable is also set to `"unique"`, the entries will be fed to the template defined in the `source` variable, along with an incremented integer ID of the entry generated by the dictionary. So, if you had set up the `source` template value as `"%05d"`, the generated entries would be five-digit auto-incremented integers.

The `type` variable tells Super Smack how to initialize the dictionary from the `source` variable. It can be any of the following:

- `"rand"`: The entries in the dictionary will be created by accessing entries in the `source` value or file in a random order. If the `source_type` is `"file"`, to load the dictionary, rows will be selected from the file randomly, and the characters in the row up to the delimiter (`delim`) will be used as the dictionary entry. If you used the same generated file in *populating* your table, you're guaranteed of finding a matching entry in your table.

- `"seq"`: Super Smack will read entries from the dictionary file in sequential order, for as many rows as the benchmark dictates (as you'll see in a minute). Again, you're guaranteed to find a match if you used the same generated file to populate the table.

- `"unique"`: Super Smack will generate fields in a unique manner similar to the way `gen-data` creates field values. You're not guaranteed that the uniquely generated field will match any values in your table. Use this `type` setting with the `"template"` `source_type` variable.

2. If you're unfamiliar with `printf()` C function, simply do a `#> man sprintf` from your console for instructions on its usage.

Query Definition Section

The next section in `select-key.smack` shows the `query` object definition being tested in the benchmark. The `query` object defines the SQL statements you will run for the benchmark. Listing 6-7 shows the definition.

Listing 6-7. *Query Object Definition in select-key.smack*

```
query "select_by_username"
{
  query "select * from http_auth where username = '$word'";
// $word will be substitute with the read from the 'word' dictionary
  type "select_index";
// query stats will be grouped by type
  has_result_set "y";
// the query is expected to return a result set
  parsed "y";
// the query string should be first processed by super-smack to do
// dictionary substitution
}
```

First, the `query` variable is set to a string housing a SQL statement. In this case, it's a simple `SELECT` statement against the `http_auth` table defined earlier, with a `WHERE` expression on the `username` field. We'll explain how the `'$word'` parameter gets filled in just a second. The `type` variable is simply a grouping for the final performance results output. Remember the output from Super Smack shown earlier in Listing 6-3? The `query_type` column corresponds to the `type` variable in the various `query` object definitions in your smack files. Here, in `select-key.smack`, there is only a single `query` object, so you see just one value in the `query_type` column of the output result. If you had more than one query, having distinct type values, you would see multiple rows in the output result representing the different query types. You can see an example of this in `update-key.smack`, the other sample smack file, which we encourage you to investigate.

The `has_result_set` value (either `"y"` or `"n"`) is fairly self-explanatory and simply informs Super Smack that the query will return a resultset. The `parsed` variable value (again, either `"y"` or `"n"`) is a little more interesting. It relates to the `dictionary` object definition we covered earlier. If the `parsed` variable is set to `"y"`, Super Smack will fill any placeholders of the style `$`*xxx* with a dictionary entry corresponding to *xxx*. Here, the placeholder `$word` in the `query` object's SQL statement will be replaced with an entry from the `"word"` dictionary, which was previously defined in the file.

You can define any number of named dictionaries, similar to the way we defined the `"word"` dictionary in this example. For each dictionary, you may refer to dictionary entries in your queries using the name of the dictionary. For instance, if you had defined two `dictionary` objects, one called `"username"` and one called `"password"`, which you had populated with usernames and passwords, you could have a query statement like the following:

```
query "userpass_select"
{
  query "SELECT * FROM http_auth WHERE username='$username' AND pass='$password'";
  has_result_set = "y";
  parsed = "y";
}
```

Second Client Configuration Section

In Listing 6-8, you see the next object definition, another `client` object. This time, it does the actual querying against the `http_auth` table.

Listing 6-8. *Second Client Object Definition in select-key.smack*

```
client "smacker1"
{
 user "test"; // connect as this user
 pass ""; // use this password
 host "localhost"; // connect to this host
 db "test"; // switch to this database
 socket "/var/lib/mysql/mysql.sock"; // this only applies to MySQL and is
// ignored for PostgreSQL
 query_barrel "2 select_by_username"; // on each round,
// run select_by_username query 2 times
}
```

This client is responsible for the brunt of the benchmark queries. As you can see, `"smacker1"` is a `client` object with the normal client variables you saw earlier, but with an extra variable called `query_barrel`.[3]

A *query barrel*, in smack terms, is simply a series of named queries run for the `client` object. The query barrel contains a string in the form of `"n query_object_name [...]"`, where `n` is the number of "shots" of the query defined in `query_object_name` that should be "fired" for each invocation of this client. In this case, the `"select_by_username"` `query` object is shot twice for each client during firing of the benchmark smack file. If you investigate the other sample smack file, `update-➥key.smack`, you'll see that Super Smack fires one shot for an `"update_by_username"` `query` object and one shot for a `"select_by_username"` `query` object in its own `"smacker1"` `client` object.

Main Section

Listing 6-9 shows the final main execution object for the `select-key.smack` file.

Listing 6-9. *Main Execution Object in select-key.smack*

```
main
{
  smacker1.init(); // initialize the client
  smacker1.set_num_rounds($2); // second arg on the command line defines
// the number of rounds for each client
  smacker1.create_threads($1);
// first argument on the command line defines how many client instances
// to fork. Anything after this will be done once for each client until
// you collect the threads
  smacker1.connect();
```

3. Super Smack uses a gun metaphor to symbolize what's going on in the benchmark runs. `super-smack` is the gun, which fires benchmark test bullets from its query barrels. Each query barrel can contain a number of shots.

```
// you must connect after you fork
  smacker1.unload_query_barrel(); // for each client fire the query barrel
// it will now do the number of rounds specified by set_num_rounds()
// on each round, query_barrel of the client is executed
  smacker1.collect_threads();
// the master thread waits for the children, each child reports the stats
// the stats are printed
  smacker1.disconnect();
// the children now disconnect and exit
}
```

This object describes the steps that Super Smack takes to actually run the benchmark using all the objects you've previously defined in the smack file.

■ **Note** It doesn't matter in which order you define objects in your smack files, with one exception. You must define the `main` executable object *last*.

The client "`smacker1`", which you've seen defined in Listing 6-8, is initialized (loaded into memory), and then the next two functions, `set_num_rounds()` and `create_threads()`, use arguments passed in on the command line to configure the test for the number of iterations you passed through and spawn the number of clients you've requested. The `$1` and `$2` represent the command-line arguments passed to Super Smack *after* the name of the smack file (those of you familiar with shell scripting will recognize the nomenclature here). In our earlier sample run of Super Smack, we executed the following:

```
#> super-smack -d mysql smacks/select-key.smack 10 100
```

The `10` would be put into the `$1` variable, and `100` goes into the `$2` variable.

Next, the `smacker1` client connects to the database defined in its `db` variable, passing the authentication information it also contains. The client's `query_barrel` variable is fired, using the `unload_query_barrel()` function, and finally some cleanup work is done with the `collect_threads()` and `disconnect()` functions. Super Smack then displays the results of the benchmark test to `stdout`.

When you're doing your own benchmarking with Super Smack, you'll most likely want to change the `client`, `dictionary`, `table`, and `query` objects to correspond to the SQL code you want to test. The `main` object definition will not need to be changed, unless you want to start tinkering with the C++ `super-smack` code.

■ **Caution** For each concurrent client you specify for Super Smack to create, it creates a *persistent* connection to the MySQL server. For this reason, unless you want to take a crack at modifying the source code, it's not possible to simulate nonpersistent connections. This constraint, however, is not a problem if you are using Super Smack simply to compare the performance results of various query incarnations. If, however, you wish to truly simulate a web application environment (and thus, nonpersistent connections) you should use either ApacheBench or httperf to benchmark the entire web application.

MyBench

Although Super Smack is a very powerful benchmarking program, it can be difficult to benchmark a complex set of logical instructions. As you've seen, Super Smack's configuration files are fairly limited in what they can test: basically, just straight SQL statements. If you need to test some complicated logic—for instance, when you need to benchmark a script that processes a number of statements inside a transaction, and you need to rely on SQL inline variables (`@variable...`)—you will need to use a more flexible benchmarking system.

Jeremy Zawodny, coauthor of *High Performance MySQL* (O'Reilly, 2004) has created a Perl module called MyBench (`http://jeremy.zawodny.com/mysql/mybench/`), which allows you to benchmark logic that is a little more complex. The module enables you to write your own Perl functions, which are fed to the MyBench benchmarking framework using a callback. The framework handles the chore of spawning the client threads and executing your function, which can contain any arbitrary logic that connects to a database, executes Perl and SQL code, and so on.

■ **Tip** For server and configuration tuning, and in-depth coverage of Jeremy Zawodny's various utility tools like MyBench and mytop, consider picking up a copy of *High Performance MySQL* (O'Reilly, 2004), by Jeremy Zawodny and Derek Bailing. The book is fairly focused on techniques to improve the performance of your hardware and MySQL configuration, the material is thoughtful, and the book is an excellent tuning reference.

The sample Perl script, called `bench_example`, which comes bundled with the software, provides an example on which you can base your own benchmark tests. Installation of the module follows the standard GNU make process. Instructions are available in the tarball you can download from the MyBench site.

■ **Caution** Because MyBench is not compiled (it's a Perl module), it can be more resource-intensive than running Super Smack. So, when you run benchmarks using MyBench, it's helpful to run them on a machine separate from your database, if that database is on a production machine. MyBench can use the standard Perl DBI module to connect to remote machines in your benchmark scripts.

ApacheBench (ab)

A good percentage of developers and administrators reading this text will be using MySQL for web-based applications. Therefore, we found it prudent to cover two web application stress-testing tools: ApacheBench (described here) and httperf (described in the next section).

ApacheBench (ab) comes installed on almost any Unix/Linux distribution with the Apache web server installed. It is a contrived load generator, and therefore provides a brute-force method of determining how many requests for a particular web resource a server can handle.

As an example, let's run a benchmark comparing the performance of two simple scripts, `finduser1.php` (shown in Listing 6-10) and `finduser2.php` (shown in Listing 6-11), which select records from the `http_auth` table we populated earlier in the section about Super Smack. The `http_auth` table contains 90,000 records and has a primary key index on `username`, which is a `char(25)` field. Each username has exactly 25 characters. For the tests, we've turned off the query cache, so that it won't skew any results. We know that the number of records that match both queries is exactly 146 rows in our generated table. However, here we're going to do some simple benchmarks to determine which method of retrieving the same information is faster.

■ **Note** If you're not familiar with the `REGEXP` function, head over to `http://dev.mysql.com/doc/mysql/en/regexp.html`. You'll see that the SQL statements in the two scripts in Listings 6-10 and 6-11 produce identical results.

Listing 6-10. *finduser1.php*

```
<?php
// finduser1.php
$conn = mysql_connect("localhost","test","") or die (mysql_error());

mysql_select_db("test", $conn) or die ("Can't use database 'test'");

$result = mysql_query("SELECT * FROM http_auth WHERE username LIKE 'ud%'");

if ($result)
 echo "found: " . mysql_num_rows($result);
else
 echo mysql_error();
?>
```

Listing 6-11. finduser2.php

```
<?php
// finduser2.php
$conn = mysql_connect("localhost","test","") or die (mysql_error());

mysql_select_db("test", $conn) or die ("Can't use database 'test'");

$result = mysql_query("SELECT * FROM http_auth WHERE username REGEXP '^ud'");

if ($result)
 echo "found: " . mysql_num_rows($result);
else
 echo mysql_error();
?>
```

You can call ApacheBench from the command line, in a fashion similar to calling Super Smack. Listing 6-12 shows an example of calling ApacheBench to benchmark a simple script and its output. The resultset shows the performance of the `finduser1.php` script from Listing 6-10.

Listing 6-12. *Running ApacheBench and the Output Results for finduser1.php*

```
# ab -n 100 -c 10 http://127.0.0.1/finduser1.php
Document Path:          /finduser1.php
Document Length:        84 bytes

Concurrency Level:      10
Time taken for tests:   1.797687 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      277000 bytes
HTML transferred:       84000 bytes
Requests per second:    556.27 [#/sec] (mean)
Time per request:       17.977 [ms] (mean)
Time per request:       1.798 [ms] (mean, across all concurrent requests)
Transfer rate:          150.19 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0       3
Processing:     1   15  62.2      6     705
Waiting:        1   11  43.7      5     643
Total:          1   15  62.3      6     708

Percentage of the requests served within a certain time (ms)
  50%      6
  66%      9
  75%     10
  80%     11
  90%     15
  95%     22
  98%     91
  99%    210
 100%    708 (longest request)
```

As you can see, ApacheBench outputs the results of its stress testing in terms of the number of requests per second it was able to sustain (along with the `min` and `max` requests), given a number of concurrent connections (the `-c` command-line option) and the number of requests per concurrent connection (the `-n` option).

We provided a high enough number of iterations and clients to make the means accurate and reduce the chances of an outlier skewing the results. The output from ApacheBench shows a number of other statistics, most notably the percentage of requests that completed within a certain time in milliseconds. As you can see, for `finduser1.php`, 80% of the requests completed in

11 milliseconds or less. You can use these numbers to determine whether, given a certain amount of traffic to a page (in number of requests and number of concurrent clients), you are falling within your acceptable response times in your benchmarking plan.

To compare the performance of `finduser1.php` with `finduser2.php`, we want to execute the same benchmark command, but on the `finduser2.php` script instead. In order to ensure that we were operating in the same environment as the first test, we did a quick reboot of our system and ran the tests. Listing 6-13 shows the results for `finduser2.php`.

Listing 6-13. *Results for finduser2.php (REGEXP)*

```
# ab -n 100 -c 10 http://127.0.0.1/finduser2.php
Document Path:          /finduser1.php
Document Length:        10 bytes

Concurrency Level:      10
Time taken for tests:   5.848457 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      203000 bytes
HTML transferred:       10000 bytes
Requests per second:    170.99 [#/sec] (mean)
Time per request:       58.485 [ms] (mean)
Time per request:       5.848 [ms] (mean, across all concurrent requests)
Transfer rate:          33.86 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.6      0       7
Processing:     3   57 148.3     30    1410
Waiting:        2   56 144.6     29    1330
Total:          3   57 148.5     30    1413

Percentage of the requests served within a certain time (ms)
  50%     30
  66%     38
  75%     51
  80%     56
  90%     73
  95%    109
  98%    412
  99%   1355
 100%   1413 (longest request)
```

As you can see, ApacheBench reported a substantial performance decrease from the first run: 556.27 requests per second compared to 170.99 requests per second, making `finduser1.php` more than 325% faster. In this way, ApacheBench enabled us to get *real* numbers in order to compare our two methods.

Clearly, in this case, we could have just as easily used Super Smack to run the benchmark comparisons, since we're changing only a simple SQL statement; the PHP code does very little. However, the example is meant only as a demonstration. The power of ApacheBench (and httperf, described next) is that you can use a single benchmarking platform to test both MySQL-specific code and PHP code. PHP applications are a mixture of both, and having a benchmark tool that can test and isolate the performance of both of them together is a valuable part of your benchmarking framework.

The ApacheBench benchmark has told us only that the `REGEXP` method fared poorly compared with the simple `LIKE` clause. The benchmark hasn't provided any insight into *why* the `REGEXP` scenario performed poorly. For that, we'll need to use some profiling tools in order to dig down into the root of the issue, which we'll do in a moment. But the benchmarking framework has given us two important things: real percentile orders of differentiation between two comparative methods of achieving the same thing, and knowledge of how many requests per second the web server can perform given this particular PHP script.

If we had supplied ApacheBench with a page in an actual application, we would have some numbers on the load limits our actual server could maintain. However, the load limits reflect a scenario in which users are requesting only a single page of our application in a brute-force way. If we want a more realistic tool for assessing a web application's load limitations, we should turn to httperf.

httperf

Developed by David Mosberger of HP Research Labs, httperf is an HTTP load generator with a great deal of features, including the ability to read Apache log files, generate sessions in order to simulate user behavior, and generate realistic user-browsing patterns based on a simple scripting format. You can obtain httperf from `http://www.hpl.hp.com/personal/David_Mosberger/httperf.html`. After installing httperf using a standard GNU make installation, go through the man pages thoroughly to investigate the myriad options available to you.

Running httperf is similar to running ApacheBench: you call the httperf program and specify a number of connections (`--num-conn`) and the number of calls per connection (`--num-calls`). Listing 6-14 shows the output of httperf running a benchmark against the same `finduser2.php` script (Listing 6-11) we used in the previous section.

Listing 6-14. *Output from httperf*

```
# httperf --server=localhost --uri=/finduser2.php --num-conns=10 --num-calls=100
Maximum connect burst length: 1

Total: connections 10 requests 18 replies 8 test-duration 2.477 s

Connection rate: 4.0 conn/s (247.7 ms/conn, <=1 concurrent connections)
Connection time [ms]: min 237.2 avg 308.8 max 582.7 median 240.5 stddev 119.9
Connection time [ms]: connect 0.3
Connection length [replies/conn]: 1.000

Request rate: 7.3 req/s (137.6 ms/req)
Request size [B]: 73.0
```

```
Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)
Reply time [ms]: response 303.8 transfer 0.0
Reply size [B]: header 193.0 content 10.0 footer 0.0 (total 203.0)
Reply status: 1xx=0 2xx=8 3xx=0 4xx=0 5xx=0

CPU time [s]: user 0.06 system 0.44 (user 2.3% system 18.0% total 20.3%)
Net I/O: 1.2 KB/s (0.0*10^6 bps)

Errors: total 10 client-timo 0 socket-timo 0 connrefused 0 connreset 10
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

As you've seen in our benchmarking examples, these tools can provide you with some excellent numbers in comparing the differences between approaches and show valuable information regarding which areas of your application struggle compared with others. However, benchmarks won't allow you to diagnose exactly what it is about your SQL or application code scripts that are causing a performance breakdown. For example, benchmark test results fell short in identifying *why* the `REGEXP` scenario performed so poorly. This is where profilers and profiling techniques enter the picture.

What Can Profiling Do for You?

Profilers and diagnostic techniques enable you to procure information about memory consumption, response times, locking, and process counts from the engines that execute your SQL scripts and application code.

PROFILERS VS. DIAGNOSTIC TECHNIQUES

When we speak about the topic of profiling, it's useful to differentiate between a *profiler* and a *profiling technique*.

A *profiler* is a full-blown application that is responsible for conducting what are called *traces* on application code passed through the profiler. These traces contain information about the breakdown of function calls within the application code block analyzed in the trace. Most profilers commonly contain the functionality of *debuggers* in addition to their profiling ability, which enables you to detect errors in the application code as they occur and sometimes even lets you step through the code itself. Additionally, profiler traces come in two different formats: *human-readable* and *machine-readable*. Human-readable traces are nice because you can easily read the output of the profiler. However, machine-readable trace output is much more extensible, as it can be read into analysis and graphing programs, which can use the information contained in the trace file because it's in a standardized format. Many profilers today include the ability to produce both types of trace output.

Diagnostic techniques, on the other hand, are not programs per se, but methods you can deploy, either manually or in an automated fashion, in order to grab information about the application code while it is being executed. You can use this information, sometimes called a *dump* or a *trace*, in diagnosing problems on the server as they occur.

From a MySQL perspective, you're interested in determining how many threads are executing against the server, what these threads are doing, and how efficiently your server is processing these requests. You should already be familiar with many of MySQL's status variables, which provide insight into the various caches and statistics that MySQL keeps available. However, aside from this information, you also want to see the statements that threads are actually running against the server as they occur. You want to see just how many resources are being consumed by the threads. You want to see if one particular type of query is consistently producing a bottleneck—for instance, locking tables for an extended period of time, which can create a domino effect of other threads waiting for a locked resource to be freed. Additionally, you want to be able to determine *how* MySQL is attempting to execute SQL statement requests, and perhaps get some insight into *why* MySQL chooses a particular path of execution.

From a web application's perspective, you want to know much the same kind of information. Which, if any, of your application blocks is taking the most time to execute? For a page request, it would be nice to see if one particular function call is demanding the vast majority of processing power. If you make changes to the code, how does the performance change?

Anyone can guess as to why an application is performing poorly. You can go on any Internet forum, enter a post about your particular situation, and you'll get 100 different responses, all claiming their answer is accurate. But, the fact is, until they or you run some sort of diagnostic routines or a profiler against your application while it is executing, everyone's answer is simply a guess. Guessing just doesn't cut it in the professional world. Using a profiler and diagnostic techniques, you can find out for yourself what specific parts of an application aren't up to snuff, and take corrective action based on your findings.

General Profiling Guidelines

There's a principle in diagnosing and identifying problems in application code that is worth repeating here before we get into the profiling tools you'll be using. When you see the results of a profiler trace, you'll be presented with information that will show you an application block broken down into how many times a function (or SQL statement) was called, and how long the function call took to complete. It is extremely easy to fall into the trap of *overoptimizing* a piece of application code, simply because you have the diagnostic tools that show you what's going on in your code. This is especially true for PHP programmers who see the function call stack for their pages and want to optimize every single function call in their application.

Basically, the rule of thumb is to start with the block of code that is taking the longest time to execute or is consuming the most resources. Spend your time identifying and fixing those parts of your application code that will have noticeable impact for your users. Don't waste your precious time optimizing a function call that executes in 4 milliseconds just to get the time down to 2 milliseconds. It's just not worth it, unless that function is called so often that it makes a difference to your users. Your time is much better spent going after the big fish.

That said, if you *do* identify a way to make your code faster, by all means document it and use that knowledge in your future coding. If time permits, perhaps think about refactoring older code bases with your newfound knowledge. But always take into account the value of your time in doing so versus the benefits, in real time, to the user.

Profiling Tools

Your first question might be, "Is there a MySQL profiler?" The flat answer is no, there isn't. Although MySQL provides some tools that enable you to do profiling (to a certain extent) of the SQL statements being run against the server, MySQL does not currently come bundled with a profiler program able to generate storable trace files.

If you are coming from a Microsoft SQL Server background and have experience using the SQL Server Profiler, you will still be able to use your basic knowledge of how traces and profiling work, but unfortunately, MySQL has no similar tool. There are some third-party vendors who make some purported profilers, but these merely display the binary log file data generated by MySQL and are not hooked in to MySQL's process management directly.

Here, we will go over some tools that you can use to simulate a true profiler environment, so that you can diagnose issues effectively. These tools will prove invaluable to you as you tackle the often-difficult problem of figuring out what is going on in your systems. We'll cover the following tools of the trade:

- The `SHOW FULL PROCESSLIST` and `SHOW STATUS` commands
- The `EXPLAIN` command
- The slow query and general query logs
- Mytop
- The Zend Advanced PHP Debugger extension

The SHOW FULL PROCESSLIST Command

The first tool in any MySQL administrator's tool belt is the `SHOW FULL PROCESSLIST` command. `SHOW FULL PROCESSLIST` returns the threads that are active in the MySQL server as a snapshot of the connection resources used by MySQL at the time the `SHOW FULL PROCESSLIST` command was executed. Table 6-3 lists the fields returned by the command.

Table 6-3. *Fields Returned from SHOW FULL PROCESSLIST*

Field	Comment
Id	ID of the user connection thread
User	Authenticated user
Host	Authenticating host
db	Name of database or `NULL` for requests not executing database-specific requests (like `SHOW FULL PROCESSLIST`)
Command	Usually either Query or Sleep, corresponding to whether the thread is actually performing something at the moment
Time	The amount of time in seconds the thread has been in this particular state (shown in the next field)
State	The status of the thread's execution (discussed in the following text)
Info	The SQL statement executing, if you ran your `SHOW FULL PROCESSLIST` at the time when a thread was actually executing a query, or some other pertinent information

Other than the actual query text, which appears in the Info column during a thread's query execution,[4] the State field is what you're interested in. The following are the major states:

Sending data: This state appears when a thread is processing rows of a `SELECT` statement in order to return the result to the client. Usually, this is a normal state to see returned, especially on a busy server. The Info field will display the actual query being executed.

Copying to tmp table: This state appears after the Sending data state when the server needs to create an in-memory temporary table to hold part of the result set being processed. This usually is a fairly quick operation seen when doing `ORDER BY` or `GROUP BY` clauses on a set of tables. If you see this state a lot and the state persists for a relatively long time, it might mean you need to adjust some queries or rethink a table design, or it may mean nothing at all, and the server is perfectly healthy. Always monitor things over an extended period of time in order to get the best idea of how often certain patterns emerge.

Copying to tmp table on disk: This state appears when the server needs to create a temporary table for sorting or grouping data, but, because of the size of the resultset, the server must use space on disk, as opposed to in memory, to create the temporary storage area. Remember from Chapter 4 that the buffer system can seamlessly switch from in-memory to on-disk storage. This state indicates that this operation has occurred. If you see this state appearing frequently in your profiling of a production application, we advise you to investigate whether you have enough memory dedicated to the MySQL server; if so, make some adjustments to the `tmp_table_size` system variable and run a few benchmarks to see if you see fewer Copying to tmp table on disk states popping up. Remember that you should make small changes incrementally when adjusting server variables, and test, test, test.

Writing to net: This state means the server is actually writing the contents of the result into the network packets. It would be rare to see this status pop up, if at all, since it usually happens *very* quickly. If you see this repeatedly cropping up, it usually means your server is getting overloaded or you're in the middle of a stress-testing benchmark.

Updating: The thread is actively updating rows you've requested in an `UPDATE` statement. Typically, you will see this state only on `UPDATE` statements affecting a large number of rows.

Locked: Perhaps the most important state of all, the Locked state tells you that the thread is waiting for another thread to finish doing its work, because it needs to `UPDATE` (or `SELECT ➥ FOR UPDATE`) a resource that the other thread is using. If you see a lot of Locked states occurring, it can be a sign of trouble, as it means that many threads are vying for the same resources. Using InnoDB tables for frequently updated tables can solve many of these problems (see Chapter 5) because of the finer-grained locking mechanism it uses (MVCC). However, poor application coding or database design can sometimes lead to frequent locking and, worse, deadlocking, when processes are waiting for *each other* to release the same resource.

4. By execution, we mean the query parsing, optimization, and execution, including returning the resultset and writing to the network packets.

Listing 6-15 shows an example of `SHOW FULL PROCESSLIST` identifying a thread in the Locked state, along with a thread in the Copying to tmp table state. (We've formatted the output to fit on the page.) As you can see, thread 71184 is waiting for the thread 65689 to finishing copying data in the `SELECT` statement into a temporary table. Thread 65689 is copying to a temporary table because of the `GROUP BY` and `ORDER BY` clauses. Thread 71184 is requesting an `UPDATE` to the `Location` table, but because that table is used in a `JOIN` in thread 65689's `SELECT` statement, it must wait, and is therefore locked.

■ **Tip** You can use the `mysqladmin` tool to produce a process list similar to the one displayed by `SHOW FULL PROCESSLIST`. To do so, execute `#> mysqladmin processlist`.

Listing 6-15. *SHOW FULL PROCESSLIST Results*

```
mysql> SHOW FULL PROCESSLIST;
+-------+--------+-----------+--------+---------+------+----------------------+-----
| Id    | User   | Host      | db     | Command | Time | State                | Info
+-------+--------+-----------+--------+---------+------+----------------------+-----
|    43 | job_db | localhost | job_db | Sleep   | 69   |                      | NULL
| 65378 | job_db | localhost | job_db | Sleep   | 23   |                      | NULL
| 65689 | job_db | localhost | job_db | Query   | 1    | Copying to tmp table |
SELECT e.Code, e.Name
FROM Job j
INNER JOIN Location l
ON j.Location = l.Code
INNER JOIN Employer e
ON j.Employer = e.Code
WHERE l.State = "NY"
AND j.ExpiresOn >= "2005-03-09"
GROUP BY  e.Code, e.Name
ORDER BY e.Sort ASC |
| 65713 | job_db | localhost | job_db | Sleep   | 60   |                      | NULL
| 65715 | job_db | localhost | job_db | Sleep   | 22   |                      | NULL
--- omitted ---
| 70815 | job_db | localhost | job_db | Sleep   | 12   |                      | NULL
| 70822 | job_db | localhost | job_db | Sleep   | 86   |                      | NULL
| 70824 | job_db | localhost | job_db | Sleep   | 62   |                      | NULL
| 70826 | root   | localhost | NULL   | Query   | 0    | NULL                 |  \
SHOW FULL PROCESSLIST
| 70920 | job_db | localhost | job_db | Sleep   | 17   |                      | NULL
| 70999 | job_db | localhost | job_db | Sleep   | 34   |                      | NULL
--- omitted ---
| 71176 | job_db | localhost | job_db | Sleep   | 39   |                      | NULL
| 71182 | job_db | localhost | job_db | Sleep   | 4    |                      | NULL
| 71183 | job_db | localhost | job_db | Sleep   | 17   |                      | NULL
| 71184 | job_db | localhost | job_db | Query   | 0    | Locked               |
```

```
UPDATE Job
SET   TotalViews = TotalViews + 1
WHERE Location = 55900
AND Position = 147
| 71185 | job_db | localhost | job_db | Sleep   | 6    |                      | NULL
+-------+--------+-----------+--------+---------+------+----------------------+-----
57 rows in set (0.00 sec)
```

■ **Note** You must be logged in to MySQL as a user with the `SUPER` privilege in order to execute the `SHOW FULL PROCESSLIST` command.

Running `SHOW FULL PROCESSLIST` is great for seeing a snapshot of the server at any given time, but it can be a bit of a pain to repeatedly execute the query from a client. The mytop utility, discussed shortly, takes away this annoyance, as you can set up mytop to reexecute the `SHOW FULL PROCESSLIST` command at regular intervals.

The SHOW STATUS Command

Another use of the `SHOW` command is to output the status and system variables maintained by MySQL. With the `SHOW STATUS` command, you can see the statistics that MySQL keeps on various activities. The status variables are all incrementing counters that track the number of times certain events occurred in the system. You can use a `LIKE` expression to limit the results returned. For instance, if you execute the command shown in Listing 6-16, you see the status counters for the various query cache statistics.

Listing 6-16. *SHOW STATUS Command Example*

```
mysql> SHOW STATUS LIKE 'Qcache%';
+-------------------------+----------+
| Variable_name           | Value    |
+-------------------------+----------+
| Qcache_queries_in_cache | 8725     |
| Qcache_inserts          | 567803   |
| Qcache_hits             | 1507192  |
| Qcache_lowmem_prunes    | 49267    |
| Qcache_not_cached       | 703224   |
| Qcache_free_memory      | 14660152 |
| Qcache_free_blocks      | 5572     |
| Qcache_total_blocks     | 23059    |
+-------------------------+----------+
8 rows in set (0.00 sec)
```

Monitoring certain status counters is a good way to track specific resource and performance measurements in real time and while you perform benchmarking. Taking before and after snapshots of the status counters you're interested in during benchmarking can show

you if MySQL is using particular caches effectively. Throughout the course of this book, as the topics dictate, we cover most of the status counters and their various meanings, and provide some insight into how to interpret changes in their values over time.

The EXPLAIN Command

The EXPLAIN command tells you how MySQL intends to execute a particular SQL statement. When you see a particular SQL query appear to take up a significant amount of resources or cause frequent locking in your system, EXPLAIN can help you determine if MySQL has been able to choose an optimal pattern for data access. Let's take a look at the EXPLAIN results from the SQL commands in the earlier finduser1.php and finduser2.php scripts (Listings 6-10 and 6-11) we load tested with ApacheBench. First, Listing 6-17 shows the EXPLAIN output from our LIKE expression in finduser1.php.

Listing 6-17. *EXPLAIN for finduser1.php*

```
mysql> EXPLAIN SELECT * FROM test.http_auth WHERE username LIKE 'ud%' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: http_auth
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 25
          ref: NULL
         rows: 128
        Extra: Using where
1 row in set (0.46 sec)
```

Although this is a simple example, the output from EXPLAIN has a lot of valuable information. Each row in the output describes an access strategy for a table or index used in the SELECT statement. The output contains the following fields:

id: A simple identifier for the SELECT statement. This can be greater than zero if there is a UNION or subquery.

select_type: Describes the type of SELECT being performed. This can be any of the following values:

- SIMPLE: Normal, non-UNION, non-subquery SELECT statement
- PRIMARY: Topmost (outer) SELECT in a UNION statement
- UNION: Second or later SELECT in a UNION statement
- DEPENDENT UNION: Second or later SELECT in a UNION statement that is dependent on the results of an outer SELECT statement
- UNION RESULT: The result of a UNION

- SUBQUERY: The first SELECT in a subquery
- DEPENDENT SUBQUERY: The first SELECT in a SUBQUERY that is dependent on the result of an outer query
- DERIVED: Subquery in the FROM clause

table: The name of the table used in the access strategy described by the row in the EXPLAIN result.

type: A description of the access strategy deployed by MySQL to get at the data in the table or index in this row. The possible values are system, const, eq_ref, ref, ref_or_null, index_merge, unique_subquery, index_subquery, range, index, and ALL. We go into detail about all the different access types in the next chapter, so stay tuned for an in-depth discussion on their values.

possible_keys: Lists the available indexes (or NULL if there are none available) that MySQL had to choose from in evaluating the access strategy for the table that the row describes.

key: Shows the actual key chosen to perform the data access (or NULL if there wasn't one available). Typically, when diagnosing a slow query, this is the first place you'll look, because you want to make sure that MySQL is using an appropriate index. Sometimes, you'll find that MySQL uses an index you didn't expect it to use.

key_len: The length, in bytes, of the key chosen. This number is often very useful in diagnosing whether a key's length is hindering a SELECT statement's performance. Stay tuned for Chapter 7, which has more on this piece of information.

ref: Shows the columns within the key chosen that will be used to access data in the table, or a constant, if the join has been optimized away with a single constant value. For instance, SELECT * FROM x INNER JOIN y ON x.1 = y.1 WHERE x.1 = 5 will be optimized away so that the constant 5 will be used instead of a comparison of key values in the JOIN between x and y. You'll find more on the topic of JOIN optimization in Chapter 7.

rows: Shows the number of rows that MySQL *expects* to find, based on the statistics it keeps on the table or index (key) chosen to be used and any preliminary calculations it has done based on your WHERE clause. This is a calculation MySQL does based on its knowledge of the distribution of key values in your indexes. The freshness of these statistics is determined by how often an ANALYZE TABLE command is run on the table, and, internally, how often MySQL updates its index statistics. In Chapter 7, you'll learn just how MySQL uses these key distribution statistics in determining which possible JOIN strategy to deploy for your SELECT statement.

Extra: This column contains extra information pertaining to this particular row's access strategy. Again, we'll go over all the possible things you'll see in the Extra field in our next chapter. For now, just think of it as any additional information that MySQL thinks you might find helpful in understanding how it's optimizing the SELECT statement you executed.

In the example in Listing 6-17, we see that MySQL has chosen to use the PRIMARY index on the http_auth table. It just so happens that the PRIMARY index is the only index on the table that contains the username field, so it decides to use this index. In this case, the access pattern is a range type, which makes sense since we're looking for usernames that begin with *ud* (LIKE 'ud%').

Based on its key distribution statistics, MySQL hints that there will be approximately 128 rows in the output (which isn't far off the actual number of 146 rows returned). In the Extra column, MySQL kindly informs us that it is using the WHERE clause on the index in order to find the rows it needs.

Now, let's compare that EXPLAIN output to the EXPLAIN on our second SELECT statement using the REGEXP construct (from finduser2.php). Listing 6-18 shows the results.

Listing 6-18. *EXPLAIN Output from SELECT Statement in finduser2.php*

```
mysql> EXPLAIN SELECT * FROM test.http_auth WHERE username REGEXP '^ud' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: http_auth
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 90000
        Extra: Using where
1 row in set (0.31 sec)
```

You should immediately notice the stark difference, which should explain the performance nightmare from the benchmark described earlier in this chapter. The possible_keys column is NULL, which indicates that MySQL was not able to use an index to find the rows in http_auth. Therefore, instead of 128 in the rows column, you see 90000. Even though the result of both SELECT statements is identical, MySQL did not use an index on the second statement. MySQL simply cannot use an index when the REGEXP construct is used in a WHERE condition.

This example should give you an idea of the power available to you in the EXPLAIN statement. We'll be using EXPLAIN extensively throughout the next two chapters to show you how various SQL statements and JOIN constructs can be optimized and to help you identify ways in which indexes can be most effectively used in your application. EXPLAIN's output gives you an insider's diagnostic view into how MySQL is determining a pathway to execute your SQL code.

The Slow Query Log

MySQL uses the slow query log to record any query whose execution time exceeds the long_query_time configuration variable. This log can be very helpful when used in conjunction with the bundled Perl script mysqldumpslow, which simply groups and sorts the logged queries into a more readable format. Before you can use this utility, however, you must enable the slow query log in your configuration file. Insert the following lines into /etc/my.cnf (or some other MySQL configuration file):

```
log-slow-queries
long_query_time=2
```

Here, we've told MySQL to consider all queries taking two seconds and longer to execute as a slow query. You can optionally provide a filename for the log-slow-queries argument. By

default, the log is stored in `/var/log/systemname-slow.log`. If you do change the log to a specific filename, remember that when you execute `mysqldumpslow`, you'll need to provide that filename. Once you've made the changes, you should restart `mysqld` to have the changes take effect. Then your queries will be logged if they exceed the `long_query_time`.

■ **Note** Prior to MySQL version 4.1, you should also include the `log-long-format` configuration option in your configuration file. This automatically logs any queries that aren't using any indexes at all, even if the query time does not exceed `long_query_time`. Identifying and fixing queries that are not using indexes is an easy way to increase the throughput and performance of your database system. The slow query log with this option turned on provides an easy way to find out which tables don't have any indexes, or any appropriate indexes, built on them. Version 4.1 and after have this option enabled by default. You can turn it off manually by using the `log-short-format` option in your configuration file.

Listing 6-19 shows the output of `mysqldumpslow` on the machine we tested our ApacheBench scripts against.

Listing 6-19. *Output from mysqldumpslow*

```
#> mysqldumpslow
Reading mysql slow query log from /var/log/mysql/slow-queries.log
Count: 1148  Time=5.74s (6585s)  \
Lock=0.00s (1s)  Rows=146.0 (167608), [test]@localhost
  SELECT * FROM http_auth WHERE username REGEXP 'S'

Count: 1  Time=3.00s (3s)  \
Lock=0.00s (0s)  Rows=90000.0 (90000), root[root]@localhost
  select * from http_auth
```

As you can see, `mysqldumpslow` groups the slow queries into buckets, along with some statistics on each, including an average time to execute, the amount of time the query was waiting for another query to release a lock, and the number of rows found by the query. We also did a `SELECT * FROM http_auth`, which returned 90,000 rows and took three seconds, subsequently getting logged to the slow query log.

In order to group queries effectively, `mysqldumpslow` converts any parameters passed to the queries into either `'S'` for string or `N` for number. This means that in order to actually see the query parameters passed to the SQL statements, you must look at the log file itself. Alternatively, you can use the `-a` option to force `mysqldumpslow` to not replace the actual parameters with `'S'` and `N`. Just remember that doing so will force many groupings of similar queries.

The slow query log can be very useful in identifying poorly performing queries, but on a large production system, the log can get quite large and contain many queries that may have performed poorly for only that one time. Make sure you don't jump to conclusions about any particular query in the log; investigate the circumstances surrounding its inclusion in the log. Was the server just started, and the query cache empty? Was an import or export process that caused long table locks running? You can use `mysqldumpslow`'s various optional arguments, listed in Table 6-4, to help narrow down and sort your slow query list more effectively.

Table 6-4. *mysqldumpslow Command-Line Options*

Option	Purpose
`-s=[t,at,l,al,r,ar]`	Sort the results based on time, total time, lock time, total lock time, rows, total rows
`-r`	Reverse sort order (list smallest values first)
`-t=n`	Show only the top *n* queries (based on sort value)
`-g=string`	Include only queries from the include "*string*" (`grep` option)
`-l`	Include the lock time in the total time numbers
`-a`	Don't abstract the parameter values passed to the query into `'S'` or `N`

For example, the `-g=string` option is very useful for finding slow queries run on a particular table. For instance, to find queries in the log using the `REGEXP` construct, execute `#> mysqldumpslow -g="REGEXP"`.

The General Query Log

Another log that can be useful in determining exactly what's going on inside your system is the general query log, which records most common interactions with the database, including connection attempts, database selection (the `USE` statement), and all queries. If you want to see a realistic picture of the activity occurring on your database system, this is the log you should use.

Remember that the binary log records only statements that change the database; it does not record `SELECT` statements, which, on some systems, comprise 90% or more of the total queries run on the database. Just like the slow query log, the general query log must first be enabled in your configuration file. Use the following line in your `/etc/my.cnf` file:

```
log=/var/log/mysql/localhost.general.log
```

Here, we've set up our log file under the `/var/log/mysql` directory with the name `general.log`. You can put the general log anywhere you wish; just ensure that the `mysql` user has appropriate write permissions or ownership for the directory or file.

Once you've restarted the MySQL server, all queries executed against the database server will be written to the general query log file.

■ **Note** There is a substantial difference between the way records are written to the general query log versus the binary log. Commands are recorded in the general query log in *the order they are received by the server*. Commands are recorded in the binary log in *the order in which they are executed by the server*. This variance exists because of the different purposes of the two logs. While the general query log serves as an information repository for investigating the activity on the server, the binary log's primary purpose is to provide an accurate recovery method for the server. Because of this, the binary log must write records in execution order so that the recovery process can rely on the database's state being restored properly.

Let's examine what the general query log looks like. Listing 6-20 shows an excerpt from our general query log during our ApacheBench benchmark tests from earlier in this chapter.

Listing 6-20. *Excerpt from the General Query Log*

```
# head -n 40 /var/log/mysql/mysqld.log
/usr/local/libexec/mysqld, Version: 4.1.10-log. started with:
Tcp port: 3306  Unix socket: /var/lib/mysql/mysql.sock
Time                 Id Command    Argument
050309 16:56:19       1 Connect     root@localhost on
050309 16:56:36       1 Quit
050309 16:56:52       2 Connect     test@localhost as anonymous on
                      3 Connect     test@localhost as anonymous on
                      4 Connect     test@localhost as anonymous on
                      5 Connect     test@localhost as anonymous on
                      6 Connect     test@localhost as anonymous on
                      7 Connect     test@localhost as anonymous on
                      8 Connect     test@localhost as anonymous on
                      9 Connect     test@localhost as anonymous on
                      2 Init DB     test
                      2 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      3 Init DB     test
                      3 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      4 Init DB     test
                      4 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      5 Init DB     test
                      5 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      6 Init DB     test
                      6 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      7 Init DB     test
                      7 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      8 Init DB     test
                      8 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      9 Init DB     test
                      9 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                     10 Connect     test@localhost as anonymous on
                     10 Init DB     test
                     10 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
050309 16:56:53      11 Connect     test@localhost as anonymous on
                     11 Init DB     test
                     11 Query      SELECT * FROM http_auth WHERE username LIKE 'ud%'
                      2 Quit
                      9 Quit
                      7 Quit
                      5 Quit
                      8 Quit
```

Using the `head` command, we've shown the first 40 lines of the general query log. The leftmost column is the date the activity occurred, followed by a timestamp, and then the ID of the thread within the log. The ID does *not* correspond to any system or MySQL process ID. The Command column will display the self-explanatory `"Connect"`, `"Init DB"`, `"Query"`, or `"Quit"` value. Finally, the Argument column will display the query itself, the user authentication information, or the database being selected.

The general query log can be a very useful tool in taking a look at exactly what's going on in your system, especially if you are new to an application or are unsure of which queries are typically being executed against the system.

Mytop

If you spent some time experimenting with `SHOW FULL PROCESSLIST` and the `SHOW STATUS` commands described earlier, you probably found that you were repeatedly executing the commands to see changes in the resultsets. For those of you familiar with the Unix/Linux `top` utility (and even those who aren't), Jeremy Zawodny has created a nifty little Perl script that emulates the `top` utility for the MySQL environment. The mytop script works just like the `top` utility, allowing you to set delays on automatic refreshing of the console, sorting of the resultset, and so on. Its benefit is that it summarizes the `SHOW FULL PROCESSLIST` and various `SHOW STATUS` statements.

In order to use mytop, you'll first need to install the Term::ReadKey Perl module from `http://www.cpan.org/modules/by-module/Term/`. It's a standard CPAN installation. Just follow the instructions after untarring the download. Then head over to `http://jeremy.zawodny.com/mysql/mytop/` and download the latest version. Follow the installation instructions and read the manual (`man mytop`) to get an idea of the myriad options and interactive prompts available to you.

Mytop has three main views:

- Thread view (default, interactive key `t`) shows the results of `SHOW FULL PROCESSLIST`.
- Command view (interactive key `c`) shows accumulated and relative totals of various commands, or command groups. For instance, `SELECT`, `INSERT`, and `UPDATE` are commands, and various administrative commands sometimes get grouped together, like the `SET` command (regardless of which `SET` is changing). This view can be useful for getting a breakdown of which types of queries are being executed on your system, giving you an overall picture.
- Status view (interactive key `S`) shows various status variables.

The Zend Advanced PHP Debugger Extension

If you're doing any substantive work in PHP, at some point, you'll want to examine the inner workings of your PHP applications. In most database-driven PHP applications, you will want to profile the application to determine where the bottlenecks are. Without a profiler, diagnosing why a certain PHP page is performing slowly is just guesswork, and that guesswork can involve long, tedious hours of trial-and-error debugging. How do you know if the bottleneck in your page stems from a long-running MySQL query or a poorly coded looping structure? How can you determine if there is a specific function or object call that is consuming the vast majority of the page's resources?

With the Zend Advanced PHP Debugger (APD) extension, help is at hand. Zend extensions are a little different from normal PHP extensions, in that they interact with the Zend Engine itself. The Zend Engine is the parsing and execution engine that translates PHP code into what's called Zend OpCodes (for operation codes). Zend extensions have the ability to interact, or hook into, this engine, which parses and executes the PHP code.

■ **Caution** Don't install APD on a production machine. Install it in a development or testing environment. The installation requires a source version of PHP (not the binary), which may conflict with some production concerns.

APD makes it possible to see the actual function call traces for your pages, with information on execution time and memory consumption. It can display the *call tree*, which is the tree organization of all subroutines executing on the page.

Setting Up APD

Although it takes a little time to set up APD, we think the reward for your efforts is substantial. The basic installation of APD is not particularly complicated. However, there are a number of shared libraries that, depending on your version of Linux or another operating system, may need to be updated. Make sure you have the latest versions of `gcc` and `libtools` installed on the server on which you'll be installing APD.

If you are running PHP 5, you'll want to download and install the latest version of APD. You can do so using PEAR's install process:

```
#> pear install apd
```

For those of you running earlier versions of PHP, or if there is a problem with the installation process through PEAR, you'll want to download the tarball designed for your version of PHP from the PECL repository: `http://pecl.php.net/package/apd/`.

Before you install the APD extension, however, you need to do a couple of things. First, you must have installed the source version of PHP (you will need the `phpize` program in order to install APD). `phpize` is available only in source versions of PHP. Second, while you don't need to provide any special PHP configuration options during installation (because APD is a Zend extension, not a loaded normal PHP extension), you *do* need to ensure that the CGI version of PHP is available. On most modern systems, this is the default.

After installing an up-to-date source version of PHP, install APD:

```
#> tar -xzf apd-0.9.1.tgz
#> cd apd-0.9.1
apd-0.9.1 #> phpize
apd-0.9.1 #> ./configure
apd-0.9.1 #> make
apd-0.9.1 #> make install
```

After the installation is completed, you will see a printout of the location of the APD shared library. Take a quick note of this location. Once APD is installed, you will need to change the `php.ini` configuration file, adding the following lines:

```
zend_extension = /absolute/path/to/apd.so
apd.dumpdir = /absolute/path/to/tracedir
apd.statement_trace = 0
```

Next, you'll want to create the `trace` directory for the APD trace files. On our system, we created the `apd.dumpdir` at `/var/apddumps`, but you can set it up anywhere. You want to create the directory and allow the public to write to it (because APD will be running in the public domain):

```
#> mkdir /var/apddumps
#> chmod 0766 /var/apddumps
```

Finally, restart the Apache server process to have your changes go into effect. On our system, we ran the following:

```
#> /etc/init.d/httpd restart
```

Profiling PHP Applications with APD

With APD set up, you're ready to see how it works. Listing 6-21 shows the script we'll profile in this example: `finduser3.php`, a modification of our earlier script that prints user information to the screen. We've used a variety of PHP functions for the demonstration, including a call to `sleep()` for one second every twentieth iteration in the loop.

■ **Note** If this demonstration doesn't work for you, there is more than likely a conflict between libraries in your system and APD's extension library. To determine if you have problems with loading the APD extension, simply execute `#> tail -n 20 /var/log/httpd/error_log` and look for errors on the Apache process startup (your Apache log file may be in a different location). The errors should point you in the right direction to fix any dependency issues that arise, or point out any typo errors in your `php.ini` file from your recent changes.

Listing 6-21. *finduser3.php*

```
<?php
apd_set_pprof_trace();
$conn = mysql_connect("localhost","test","") or die (mysql_error());
mysql_select_db("test", $conn) or die ("Can't use database 'test'");
$result = mysql_query("SELECT * FROM http_auth WHERE username REGEXP '^ud'");

if ($result) {
  echo '<pre>';
  echo "UserName\tPassword\tUID\tGID\n";
  $num_rows = mysql_num_rows($result);
```

```
  for ($i=0;$i<$num_rows;++$i) {
    mysql_data_seek($result, $i);
    if ($i % 20 == 0)
      sleep(1);
    $row = mysql_fetch_row($result);
    printf("%s\t%s\t%d\t%d\n", $row[0], $row[1], $row[2], $row[4]);
  }
  echo '</pre>';
}
?>
```

We've highlighted the `apd_set_pprof_trace()` function. This must be called at the top of the script in order to tell APD to trace the PHP page. The traces are dumped into `pprof.`*XXXXX* files in your `apd.dumpdir` location, where *XXXXX* is the process ID of the web page you trace. When we run the `finduser3.php` page through a web browser, nothing is displayed, which tells us the trace completed successfully. However, we can check the `apd.dumpdir` for files beginning with `pprof`. To display the `pprof` trace file, use the `pprofp` script available in your APD source directory (where you installed APD) and pass along one or more of the command-line options listed in Table 6-5.

Table 6-5. *pprofp Command-Line Options*

Option	Description
`-a`	Sort by alphabetic name of function
`-l`	Sort by number of calls to the function
`-r`	Sort by real time spent in function
`-R`	Sort by real time spent in function and all its child functions
`-s`	Sort by system time spent in function
`-S`	Sort by system time spent in function and all its child functions
`-u`	Sort by user time spent in function
`-U`	Sort by user time spent in function and all its child functions
`-v`	Sort by average amount of time spent in function (across all requests to function)
`-z`	Sort by total time spent in function (default)
`-c`	Display real time elapsed alongside call tree
`-i`	Suppress reporting for PHP built-in functions
`-m`	Display file/line number locations in trace
`-O [n]`	Display *n* number of functions (default = 15)
`-t`	Display compressed call tree
`-T`	Display uncompressed call tree

Listing 6-22 shows the output of `pprofp` when we asked it to sort our traced functions by the real time that was spent in the function. The trace file on our system, which resulted from browsing to `finduser3.php`, just happened to be called `/var/apddumps/pprof.15698` on our system.

Listing 6-22. *APD Trace Output Using pprofp*

```
# ./pprofp -r /var/apddumps/pprof.15698
Content-type: text/html
X-Powered-By: PHP/4.3.10


Trace for /var/www/html/finduser3.php
Total Elapsed Time = 8.28
Total System Time  = 0.00
Total User Time    = 0.00


         Real         User        System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls   call   s/call  Memory Usage Name
--------------------------------------------------------------------------------------
96.7 8.01 8.01 0.00 0.00  0.00 0.00     8  1.0012   1.0012         0 sleep
2.9 0.24 0.24  0.00 0.00  0.00 0.00     1  0.2400   0.2400         0 mysql_query
0.2 0.02 0.02  0.00 0.00  0.00 0.00     1  0.0200   0.0200         0 mysql_connect
0.1 0.01 0.01  0.00 0.00  0.00 0.00   146  0.0001   0.0001         0 mysql_data_seek
0.0 0.00 0.00  0.00 0.00  0.00 0.00   146  0.0000   0.0000         0 printf
0.0 0.00 0.00  0.00 0.00  0.00 0.00   146  0.0000   0.0000         0 mysql_fetch_row
0.0 0.00 0.00  0.00 0.00  0.00 0.00     1  0.0000   0.0000         0 mysql_num_rows
0.0 0.00 0.00  0.00 0.00  0.00 0.00     1  0.0000   0.0000         0 mysql_select_db
0.0 0.00 0.00  0.00 0.00  0.00 0.00     1  0.0000   0.0000         0 main
```

As you can see, APD supplies some very detailed and valuable information about the state of the page processing, which functions were used, how often they were called, and how much of a percentage of total processing time each function consumed. Here, you see that the `sleep()` function took the longest time, which makes sense because it causes the page to stop processing for one second at each call. Other than the `sleep()` command, only `mysql_query()`, `mysql_connect()`, and `mysql_data_seek()` had nonzero values.

Although this is a simple example, the power of APD is unquestionable when analyzing large, complex scripts. Its ability to pinpoint the bottleneck functions in your page requests relies on the `pprofp` script's numerous sorting and output options, which allow you to drill down into the call tree. Take some time to play around with APD, and be sure to add it to your toolbox of diagnostic tools.

Tip For those of you interested in the internals of PHP, writing extensions, and using the APD profiler, consider George Schlossnagle's *Advanced PHP Programming* (Sams Publishing, 2004). This book provides extensive coverage of how the Zend Engine works and how to effectively diagnose misbehaving PHP code.

Summary

In this chapter, we stressed the importance of benchmarking and profiling techniques for the professional developer and administrator. You've learned how setting up a benchmarking framework can enable you to perform comprehensive (or even just quick) performance comparisons of your design features and help you to expose general bottlenecks in your MySQL applications. You've seen how profiling tools and techniques can help you avoid the guesswork of application debugging and diagnostic work.

In our discussion of benchmarking, we focused on general strategies you can use to make your framework as reliable as possible. The guidelines presented in this chapter and the tools we covered should give you an excellent base to work through the examples and code presented in the next few chapters. As we cover various aspects of the MySQL query optimization and execution process, remember that you can fall back on your established benchmarking framework in order to test the theories we outline next. The same goes for the concepts and tools of profiling.

We hope you come away from this chapter with the confidence that you can test your MySQL applications much more effectively. The profilers and the diagnostic techniques we covered in this chapter should become your mainstay as a professional developer. Figuring out performance bottlenecks should no longer be guesswork or a mystery.

In the upcoming chapters, we're going to dive into the SQL language, covering `JOIN` and optimization strategies deployed by MySQL in Chapter 7. We'll be focusing on real-world application problems and how to restructure problematic SQL code. In Chapter 8, we'll take it to the next step, describing how you can structure your SQL code, database, and index strategies for various performance-critical applications. You'll be asked to use the information and tools you learned about here in these next chapters, so keep them handy!