6

Performance Tuning

Introduction

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

Performance Tuning Methodology

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

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

4. Hardware

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

Application Performance Tuning

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

- Normalize your database

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

- Let the MySQL server do what it does well

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

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

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

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

- Denormalize your database where appropriate

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

- Use persistent connections or connection pooling if possible

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

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

SQL Query Tuning

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

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

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

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

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

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

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

Index Guidelines

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

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

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

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

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

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

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

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

  would use an index, but

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

  would not.

- Use unique indexes where possible

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

- Take advantage of multi-column indexes

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

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

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

- Consider not indexing some columns

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

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

EXPLAIN SELECT

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

What does EXPLAIN tell us about this query?

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

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

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

How does our query look with that?

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



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

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

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

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

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

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

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

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

Other Options

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

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

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

Tuning the MySQL Server

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

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

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

- Table_cache

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

- Key_buffer_size

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

OS/Hardware Considerations

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

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

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

