Reading: Improving Performance of Slow Queries in MySQL

Estimated time needed: 20 minutes

In this reading, you'll learn how to improve the performance of slow queries in MySQL.

Objectives

After completing this reading, you will be able to:

1. Describe common reasons for slow queries in MySQL
2. Identify the reason for your query's performance with the `EXPLAIN` statement
3. Improve your query's performance with indexes and other best practices

Software Used

In this reading, you will see usage of MySQL. MySQL is a Relational Database Management System (RDBMS) designed to efficiently store, manipulate, and retrieve data.

Common Causes of Slow Queries

Sometimes when you run a query, you might notice that the output appears much slower than you expect it to, taking a few extra seconds, minutes or even hours to load. Why might that be happening?

There are many reasons for a slow query, but a few common ones include:

1. The size of the database, which is composed of the number of tables and the size of each table. The larger the table, the longer a query will take, particularly if you're performing scans of the entire table each time.
2. Unoptimized queries can lead to slower performance. For example, if you haven't properly indexed your database, the results of your queries will load much slower.

Each time you run a query, you'll see output similar to the following:

```
+--------+-----------+---------------+-----------------+--------+-----------+
300024 rows in set (0.34 sec)
```

As can be seen, the output includes the number of rows outputted and how long it took to execute, given in the format of `0.00` seconds.

One built-in tool that can be used to determine why your query might be taking a longer time to run is the `EXPLAIN` statement.

EXPLAIN Your Query's Performance

The `EXPLAIN` statement provides information about how MySQL executes your statement—that is, how MySQL plans on running your query. With `EXPLAIN`, you can check if your query is pulling more information than it needs to, resulting in a slower performance due to handling large amounts of data.

This statement works with `SELECT`, `DELETE`, `INSERT`, `REPLACE` and `UPDATE`. When run, it outputs a table that looks like the following:

```
mysql> EXPLAIN SELECT * FROM employees;
+----+-------------+-----------+------------+------+---------------+---
| id | select_type | table     | partitions | type | possible_keys | ke
+----+-------------+-----------+------------+------+---------------+---
|  1 | SIMPLE      | employees | NULL       | ALL  | NULL          | NU
+----+-------------+-----------+------------+------+---------------+---
1 row in set, 1 warning (0.00 sec)
```

As shown in the outputted table, with `SELECT`, the `EXPLAIN` statement tells you what type of select you performed, the table that select is being performed on, the number of rows examined, and any additional information.

In this case, the `EXPLAIN` statement showed us that the query performed a simple select (rather than, for example, a subquery or union select) and that 298,980 rows were examined (out of a total of about 300,024 rows).

The number of rows examined can be helpful when it comes to determining why a query is slow. For example, if you notice that your output is only 13 rows, but the query is examining about 300,000 rows—almost the entire table!—then that could be a reason for your query's slow performance.

In the earlier example, loading about 300,000 rows took less than a second to process, so that may not be a big concern with this database. However, that may not be the case with larger databases that can have up to a million rows in them.

One method of making these queries faster is by adding indexes to your table.

Indexing a Column

Think of indexes like bookmarks. Indexes point to specific rows, helping the query determine which rows match its conditions and quickly retrieves those results. With this process, the query avoids searching through the entire table and improves the performance of your query, particularly when you're using SELECT and WHERE clauses.

There are many types of indexes that you can add to your databases, with popular ones being regular indexes, primary indexes, unique indexes, full-text indexes and prefix indexes.

Type of Index	Description
Regular Index	An index where values do not have to be unique and can be NULL.
Primary Index	Primary indexes are automatically created for primary keys. All column values are unique and NULL values are not allowed.
Unique Index	An index where all column values are unique. Unlike the primary index, unique indexes can contain a NULL value.
Full-Text Index	An index used for searching through large amounts of text and can only be created for **char**, **varchar** and/or **text** datatype columns.
Prefix Index	An index that uses only the first N characters of a text value, which can improve performance as only those characters would need to be searched.

Now, you might be wondering: if indexes are so great, why don't we add them to each column?

Generally, it's best practice to avoid adding indexes to all your columns, only adding them to the ones that it may be helpful for, such as a column that is frequently accessed. While indexing can improve the performance of some queries, it can also slow down your inserts, updates and deletes because each index will need to be updated every time. Therefore, it's important to find the balance between the number of indexes and the speed of your queries.

In addition, indexes are less helpful for querying small tables or large tables where almost all the rows need to be examined. In the case where most rows need to be examined, it would be faster to read all those rows rather than using an index. As such, adding an index is dependent on your needs.

Be SELECTive With Columns

When possible, avoid selecting all columns from your table. With larger datasets, selecting all columns and displaying them can take much longer than selecting the one or two columns that you need.

For example, with a dataset of about 300,000 employee entries, the following query takes about 0.31 seconds to load:

```
SELECT * FROM employee;
```

```
| 499998 | 1956-09-05 | Patricia        | Breugel           | M      | 1993-10-13 |
| 499999 | 1958-05-01 | Sachin          | Tsukuda           | M      | 1997-11-30 |
+--------+------------+-----------------+-------------------+--------+------------+
300024 rows in set (0.31 sec)
```

But if we only wanted to see the employee numbers and their hire dates (2 out of the 6 columns) we could easily do so with this query that takes 0.12 seconds to load:

```
SELECT employee_number, hire_date FROM employee;
```

```
| 499998 | 1993-10-13 |
| 499999 | 1997-11-30 |
+--------+------------+
300024 rows in set (0.12 sec)
```

Notice how the execution time of the query is much faster compared to the when we selected them all. This method can be helpful when dealing with large datasets that you only need select specific columns from.

Avoid Leading Wildcards

Leading wildcards, which are wildcards (`"%abc"`) that find values that end with specific characters, result in full table scans, even with indexes in place.

If your query uses a leading wildcard and performs poorly, consider using a full-text index instead. This will improve the speed of your query while avoiding the need to search through every row.

Use the UNION ALL Clause

When using the `OR` operator with `LIKE` statements, a `UNION ALL` clause can improve the speed of your query, especially if the columns on both sides of the operator are indexed.

This improvement is due to the `OR` operator sometimes scanning the entire table and overlooking indexes, whereas the `UNION ALL` operator will apply them to the separate `SELECT` statements.

Next Steps

Congratulations! Now that you have a better understanding of why your query may be performing slow and how you can improve that performance, let's take a look at how we can do that with MySQL in the Skills Network Labs environment.

Author(s)

Kathy An