How to Optimize MySQL Queries for Speed and Performance

by Francis Ndungu MVB · **Oct. 03, 18 · Database Zone · Tutorial**

You can deploy fast, secure, and trusted MySQL database instances on Alibaba Cloud. Alibaba has an advanced network of cloud-based technologies and their breaking performance and flexible billing have enabled cloud without borders for its over one million paid customers.

Alibaba Cloud has continued to show enormous contribution to the open-source communities and has empowered developers worldwide. Alibaba Cloud was the winner of the prestigious 2018 MySQL Corporate Contributor Award and is also a platinum sponsor of the MariaDB foundation.

In this guide, we will take you through the steps of optimizing SQL queries and databases on your Alibaba Cloud Elastic Compute Service (ECS) instance. This will guarantee stability, scalability, reliability and speed of applications and websites running on your Alibaba Cloud instance.

Prerequisites

1. A valid Alibaba cloud account. If you don't have one already, you can sign up for an Alibaba Cloud and enjoy $300 worth in Free Trial.
2. A server running your favorite operating system that can support MySQL (e.g. Ubuntu, Centos, Debian).
3. MySQL database server.
4. A MySQL user capable of running root commands.

Tip #1: Index All Columns Used in 'where', 'order by', and 'group by' Clauses

Apart from guaranteeing uniquely identifiable records, an index allows MySQL server to

fetch results faster from a database. An index is also very useful when it comes to sorting records.

MySQL indexes may take up more space and decrease performance on inserts, deletes, and updates. However, if your table has more than 10 rows, they can considerably reduce select query execution time.

It is always advisable to test MySQL queries with a "worst case scenario" sample amount of data to get a clearer picture of how the query will behave on production.

Consider a case where you are running the following SQL query from a database with 500 rows without an index:

```
mysql> select customer_id, customer_name from customers where customer_id='140385';
```

The above query will force MySQL server to conduct a full table scan (start to finish) to retrieve the record that we are searching.

Luckily, MySQL has a special '**EXPLAIN**' statement that you can use alongside select, delete, insert, replace and update statements to analyze your queries.

Once you append the query before an SQL statement, MySQL displays information from the optimizer about the intended execution plan.

If we run the above SQL one more time with the explain statement, we will get a full picture of what MySQL will do to execute the query:

```
mysql> explain select customer_id, customer_name from customers where customer_id='140385';

+----+-------------+-----------+------------+------+---------------+------+---------+------+------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows |
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  500 |
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+
```

As you can see, the optimizer has displayed very important information that can help us to fine-tune our database table. First, it is clear that MySQL will conduct a full table scan because key column is '**NULL**'. Second, MySQL server has clearly indicated that it's going to conduct a full scan on the 500 rows in our database.

To optimize the above query, we can just add an index to the '**customer_id**' field using the below syntax:

```
mysql> Create index customer_id ON customers (customer_Id);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

If we run the explain statement one more time, we will get the below results:

```
mysql> Explain select customer_id, customer_name from customers where customer_id='140385';

+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+
| id | select_type | table     | partitions | type | possible_keys | key         | key_len | ref   |
+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+
|  1 | SIMPLE      | customers | NULL       | ref  | customer_id   | customer_id | 13      | const |
+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+
```

From the above explain output, it's clear that MySQL server will use our index (customer_Id) to search the table. You can clearly see that the number of rows to scan will be 1. Although I run the above query in a table with 500 records, indexes can be very useful when you are querying a large dataset (e.g. a table with 1 million rows).

Tip 2: Optimize Like Statements With Union Clause

Sometimes, you may want to run queries using the comparison operator '**or**' on different fields or columns in a particular table. When the '**or**' keyword is used too much in where clause, it might make the MySQL optimizer to incorrectly choose a full table scan to retrieve a record.

A union clause can make the query run faster especially if you have an index that can optimize one side of the query and a different index to optimize the other side.

Example, consider a case where you are running the below query with the '**first_name**' and '**last_name**' indexed:

```
mysql> select * from students where first_name like  'Ade%'  or last_name like 'Ade%' ;
```

The query above can run far much slower compared to the below query which uses a union operator merge the results of 2 separate fast queries that takes advantage of the indexes.

```
mysql> select  from students where first_name like  'Ade%'  union all select  from students where la
```

Tip 3: Avoid Like Expressions With Leading Wildcards

MySQL is not able to utilize indexes when there is a leading wildcard in a query. If we take our example above on the students table, a search like this will cause MySQL to perform full table scan even if you have indexed the '**first_name**' field on the students table.

```
mysql> select * from students where first_name like  '%Ade'  ;
```

We can prove this using the explain keyword:

```
mysql> explain select * from students where first_name like  '%Ade'  ;

+----+-------------+----------+------------+------+---------------+------+---------+------+------+--
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | f
+----+-------------+----------+------------+------+---------------+------+---------+------+------+--
|  1 | SIMPLE      | students | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  500 |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+--
```

As you can see above, MySQL is going to scan all the 500 rows in our students table and make will make the query extremely slow.

Tip 4: Take Advantage of MySQL Full-Text Searches

If you are faced with a situation where you need to search data using wildcards and you don't want your database to underperform, you should consider using MySQL full-text search (FTS) because it is far much faster than queries using wildcard characters.

Furthermore, FTS can also bring better and relevant results when you are searching a huge database.

To add a full-text search index to the students sample table, we can use the below MySQL command:

```
mysql>Alter table students ADD FULLTEXT (first_name, last_name);
mysql>Select * from students where match(first_name, last_name) AGAINST ('Ade');
```

In the above example, we have specified the columns that we want to be matched (first_name and last_name) against our search keyword ('Ade').

If we query the optimizer about the execution plan of the above query, we will get the following results:

```
mysql> explain Select * from students where match(first_name, last_name) AGAINST ('Ade');
+----+-------------+----------+------------+----------+---------------+------------+---------+------
| id | select_type | table    | partitions | type     | possible_keys | key        | key_len | ref
+----+-------------+----------+------------+----------+---------------+------------+---------+------
|  1 | SIMPLE      | students | NULL       | fulltext | first_name    | first_name | 0       | const
+----+-------------+----------+------------+----------+---------------+------------+---------+------
```

It's clear that only a single row will be scanned even if our student's database has 500 rows and this will speed up the database.

Tip 6: Optimize Your Database Schema

Even if you optimize your MySQL queries and fail to come up with a good database structure, your database performance can still halt when your data increases.

Normalize Tables

First, normalize all database tables even if it will involve some trade-offs. For instance, if you are creating two tables to hold customers data and orders, you should reference the customer on the orders table using the customer id as opposed to repeating the customer's name on the orders table. The latter will cause your database to bloat.

The image below refers to a database schema that is designed for performance without any data redundancy. In MySQL database normalization, you should represent a fact only once in the entire database. Don't repeat the customer name in every table; instead just use the customer_Id for reference in other tables.

Also, always use the same data type for storing similar values even if they are on different tables, for instance, the schema above uses '**INT**' data type to store '**customer_id**' both in the customers and orders table.

Use Optimal Data Types

MySQL supports different data types including integer, float, double, date, date_time, Varchar, and text, among others. When designing your tables, you should know that "shorter is always better."

For instances, if you are designing a system user's table which will hold less than 100 users, you should use '**TINYINT**' data type for the '**user_id**' field because it will accommodate all your values from -128 to 128.

Also, if a field expects a date value (e.g. sales_order_date), using a date_time data type will be ideal because you don't have to run complicated functions to convert the field to date when retrieving records using SQL.

Use integer values if you expect all values to be numbers (e.g. in a student_id or a payment_id field). Remember, when it comes to computation, MySQL can do better with

integer values as compared to text data types such as Varchar

Avoid Null Values

Null is the absence of any value in a column. You should avoid this kind of values whenever possible because they can harm your database results. For instance, if you want to get the sum of all orders in a database but a particular order record has a null amount, the expected result might misbehave unless you use MySQL '**ifnull**' statement to return alternative value if a record is null.

In some cases, you might need to define a default value for a field if records don't have to include a mandatory value for that particular column/field.

Avoid Too Many Columns

Wide tables can be extremely expensive and require more CPU time to process. If possible, don't go above a hundred unless your business logic specifically calls for this.

Instead of creating one wide table, consider splitting it apart in to logical structures. For instance, if you are creating a customer table but you realize a customer can have multiple addresses, it is better to create a separate table for holding customers addresses that refer back to the customers table using the '**customer_id**' field.

Optimize Joins

Always include fewer tables in your join statements. An SQL statement with poorly designed pattern that involves a lot of joins may not work well. A rule of thumb is to have utmost a dozen joins for each query.

Tip 7: MySQL Query Caching

If your website or application performs a lot of select queries (e.g. WordPress), you should take advantage of MySQL query caching feature. This will speed up performance when read operations are conducted.

The technology works by caching the select query alongside the resulting data set. This makes the query run faster since they are fetched from memory if they are executed more than once. However, if your application updates the table frequently, this will invalidate any cached query and result set.

You can check if your MySQL server has query cache enabled by running the command below:

```
mysql> show variables like 'have_query_cache';
+------------------+-------+
| Variable_name    | Value |
```

```
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+
1 row in <b>set</b> (0.00 sec)
```

Setting the MySQL Server Query Cache

You can set the MySQL query cache values by editing the configuration file ('**/etc/mysql/my.cnf**' or '**/etc/mysql/mysql.conf.d/mysqld.cnf**'). This will depend on your MySQL installation. Don't set a very large query cache size value because this will degrade the MySQL server due to cached overhead and locking. Values in the range of tens of megabytes are recommended.

To check the current value, use the command below:

```
mysql> show variables like 'query_cache_%' ;
+------------------------------+----------+
| Variable_name                | Value    |
+------------------------------+----------+
| query_cache_limit            | 1048576  |
| query_cache_min_res_unit     | 4096     |
| query_cache_size             | 16777216 |
| query_cache_type             | OFF      |
| query_cache_wlock_invalidate | OFF      |
+------------------------------+----------+
5 rows in <b>set</b> (0.00 sec)
```

Then to adjust the values, include the following on the MySQL configuration file:

```
query_cache_type=1
query_cache_size = 10M
query_cache_limit=256k
```

You can adjust the above values according to your server needs.

The directive '**query_cache_type=1**' turns MySQL caching on if it was turned off by default.

The default '**query_cache_size**' is 1MB and like we said above a value a range of around 10 MB is recommended. Also, the value must be over 40 KB otherwise MySQL server will throw a warning, "**Query cache failed to set size**".

The default '**query_cache_limit**' is also 1MB. This value controls the amount of individual query result that can be can be cached.

Conclusion

In this guide, we have shown you how to optimize your MySQL server hosted on Alibaba