5

Understanding Query Components

Now that you understand what factors break your queries, it's time to understand their components – tasks inside of them. Here we come back to something I've mentioned dozens of times throughout this book already – *queries are tasks composed of other tasks.* To ensure these tasks will execute successfully, we have to understand their components. Sounds simple enough, but if it were that simple, we would have way fewer people complaining about query performance, right? That's why understanding query components is so crucial – a proper understanding of query internals can make or break your database instances.

SQL Queries and Stored Procedures

From the outset, SQL queries look simple and uncomplicated. Under the hood, they're not a child's play – they're composed of many small tasks that all need to work in unison so that your database can achieve perfection. Flick back to the last chapter and read through the Why Are Queries Slow? heading and you will quickly understand what I mean – with so many tasks to complete, it's reasonable to assume that queries reading through bigger data sets would take more time than those who haven't got as much data to sift through. Each type of query completes a different task (queries insert, select, update, or delete data), but under the hood, they all work similarly.

As I've already mentioned in the last chapter, there are multiple types of queries in MySQL – those include DML (Data Manipulation Language) queries, DDL (Data Definition Language) queries, and also DCL (Data Control Language) and

TCL (Transaction Control Language) queries. Most of these queries work similarly – they execute 20 or so subtasks before returning any results. Those queries are often bundled together in the sense that once the user using your application acts on a form or anything similar to it, an SQL query executes in the background. More savvy developers might even be aware of stored procedures to help them succeed!

To understand why query components are so crucial to your SQL queries, I'd like you to take a look at stored procedures as an example. In MySQL, stored procedures are implemented with the `CALL` statement – the statement calls (invokes) a stored procedure that has to be created using `CREATE PROCEDURE`. Procedures in MySQL are sets of SQL queries stored inside the database (and are thus known to the server) and executed once procedures are invoked using `CALL procedure` or `CALL procedure()`.

Procedures require two parameters – one that goes `IN`, and another that goes `OUT`. <...>

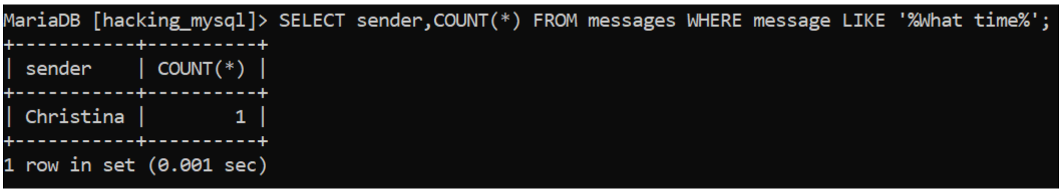
Query performance is directly dependent on how quickly your database can execute query components – which, remember, there are slightly over 20 of. Those components depend on the query execution plan, which depends on parsers and optimizers.

Parsers and Optimizers

Internally, every SQL query is turned into an execution plan. Before coming up with the execution plan, MySQL checks whether the syntax and grammar of the SQL query are correct during the parser phase – it checks for errors by examining all characters in the query one by one, matches them against rules, and if everything's OK, turns the query to the optimizer.

For you to imagine how a parser works within your database, take any query and split it into pieces. Take a look at this one if you want an example:

```
SELECT sender,COUNT(*) FROM messages WHERE message LIKE '%What time%';
```



```
MariaDB [hacking_mysql]> SELECT sender,COUNT(*) FROM messages WHERE message LIKE '%What time%';
+-----------+----------+
| sender    | COUNT(*) |
+-----------+----------+
| Christina |        1 |
+-----------+----------+
1 row in set (0.001 sec)
```

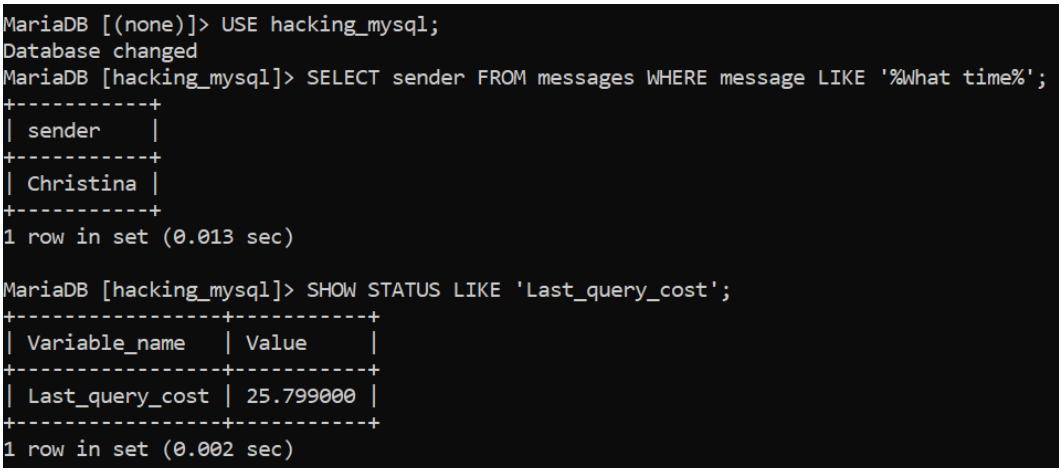
Image 1 – SQL Query in MariaDB

For such an SQL query, the parser would interpret everything. Yes, every word and character! These words and characters would be as follows:

1. SELECT
2. sender
3. ,
4. COUNT
5. (
6. *
7.)
8. FROM
9. messages
10. WHERE
11. message
12. LIKE
13. ‘
14. %
15. What time
16. %
17. ‘
18. ;

See a pattern? The query parser has to split any SQL query, no matter how simple or complex, into pieces and evaluate them one by one. After it has gotten the OK sign from MySQL, it turns to the query optimizer.

The query optimizer in MySQL tries to predict what query execution plans would execute the quickest and choose the best option available for MySQL to use. In this scenario, I'd like you to think of everything in terms of price – MySQL even has a variable called `Last_query_cost`:



```
MariaDB [(none)]> USE hacking_mysql;
Database changed
MariaDB [hacking_mysql]> SELECT sender FROM messages WHERE message LIKE '%What time%';
+-----------+
| sender    |
+-----------+
| Christina |
+-----------+
1 row in set (0.013 sec)

MariaDB [hacking_mysql]> SHOW STATUS LIKE 'Last_query_cost';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| Last_query_cost | 25.799000 |
+-----------------+-----------+
1 row in set (0.002 sec)
```

Image 2 – Last_query_cost in MariaDB

<...>

Factors Disliked by Your Queries

Query components are directly impacted by factors within your server, application, and database, and all queries you run come with a distinct weight towards your database too – you already know that to make your queries performant, you need to minimize their effort by minimizing or eliminating the number of tasks they perform, but that be made even more effective if you avoid certain things that SQL queries dislike.

Isolate Your Columns!

The first thing to keep in mind has to do with search – `SELECT` – queries. If you find that the column you run search queries on has an index, is properly partitioned, and you run `SELECT column` instead of `SELECT *` to select data inside of your database but it is still running slow, you may want to check if the column after the `WHERE` clause is isolated, too. That means that queries like the one below certainly won't fly:

```
SELECT * FROM `demo_table` WHERE column + 786 = 4425;
```

They won't fly because if the column after the `WHERE` clause isn't isolated ("left alone" if you will), your database won't be able to use the index. As such, your query may slow down. To avoid such an issue, make sure that the columns after the `WHERE` clause aren't "conspiring together with someone" to achieve a result. Leave them alone – MySQL will decide how best to return results.

Get Rid of Duplicate Indexes

MySQL or any of its flavors won't protect you from using duplicate indexes on the same column either. Take a look at this example:

```
CREATE TABLE `demo_table` (
`incrementing_id` INT(25) NOT NULL AUTO_INCREMENT PRIMARY KEY,
`column_2` VARCHAR(120) NOT NULL DEFAULT '',
`column_3` VARCHAR(125) NOT NULL DEFAULT '',
…
INDEX(incrementing_id),
UNIQUE(incrementing_id)
);
```

This one may become a tough nut to crack for inexperienced users – some may think that this SQL query makes the `incrementing_id` column increment automatically (it should, hence the name), adds an index on the same column once the columns are defined, and also makes sure that all values in the column are unique (there are no duplicate values.) All fun and roses, right?

Experienced DBAs will quickly notice that something's up. Indeed, we have *three* indexes on the same column! MySQL implements the PRIMARY KEY and UNIQUE constraints with indexes, so we have two indexes right then and there. And since our column is indexed, too, that adds another index on the same column. Three indexes on one column? And then devs wonder what's up with their database structure...

Keep in mind that MySQL doesn’t “protect you” from appending multiple indexes on the same column, so choose the types of indexes carefully. Alright, I’ve lied and some may get lucky – adding a `FULLTEXT` index on a column with a primary key cannot be done and such actions will result in MySQL screaming an error like so:

```
#1283 - Column ‘incrementing_id’ cannot be part of FULLTEXT index
```

I’ll walk you through the ins and outs of indexes in an upcoming chapter, but for now, please don’t make your columns (and your database as a result) suffer – employ indexes wisely.

<...>

Summary

Understanding SQL query components is an essential task for everyone who cares about their database. As each SQL query is a task composed of many different tasks that help your database, understanding how queries look under the hood and what they consist of, what their parsers and optimizers do, and walking yourself through the explanatory outputs and error codes provided by your database is an essential step towards a brighter and better future for your application, server, and database.

Contrary to popular belief though, components aren’t the only thing that makes your database tick; your database is only a part of the bigger puzzle that includes your application and your server. Understanding how both of these things and your database work in conjunction is an essential step to achieving database harmony – and that’s why I now invite you to understand your server before telling you how you should optimize your database and everything within.