SQL Query Optimization: Top 5 Tips for Modern Query Optimization

September 11, 2018 ·
Tomer Shay @ EverSQL

This is the first part of our SQL query optimization articles series. In this article, we'll focus on how to optimize MySQL queries, but the same concepts can be applied to many other relational databases.

Now more than ever, software engineers need to have vast knowledge in SQL query optimization.
The shift is happening in both small startups and large enterprises. Nowadays, developers are the ones writing the SQL queries and database access layer.

It doesn't really matter if you're using a database abstraction layer (Hibernate, JOOQ, Entity Framework, Sqlalchemy, Django, or others) or writing native SQL queries, you'll eventually be challenged with tuning the queries you're sending to your database.

So what can you do to optimize your SQL queries?

Create indexes, but do it wisely

Indexing is probably the most important part of the SQL optimization process. So first, make sure you're familiar with the different aspects you should consider when choosing the optimal indexes. for your database.

When thinking about which indexes to create, you should pay close attention to the query's WHERE clause and table JOINs, as those statements include the critical index-able parts of the query.

Also, major bottlenecks can originate in the GROUP BY and ORDER BY parts. Said that, a potential hiccup will be that you may not be able to index them in some cases, as we explained here. Therefore, you might need to re-think the design of your query before creating the indexes, to make sure you write great queries, but also write index-able queries.

Once you got indexing figured out for one query, don't stop there. Widen your view and look into other important queries in your application. Make sure you combine indexes whenever possible, and remove indexes which aren't used. Looking at the entire application's scope will always be better than looking at a single query's scope.

You should also keep in mind that creating more indexes than you need can also backfire on you, as they can slow down write operations (such as INSERT / UPDATE statements). So create indexes to optimize your SQL queries, but do it wisely.

Do not stand in the way of indexes

We're being approached a lot by customers who're asking us "why the database doesn't use my index?". Well, that's a great question, with endless possible answers. But, in this article, we'll try to cover several common scenarios which we see often, so hopefully, you'll find them useful for your own use case.

Example #1 - Avoid wrapping indexed columns with functions

Consider this query, which counts the number of hot dogs purchased in the US on 2018. Just in case you're curious, 18,000,000,000 hot dogs were sold in the US in 2018.

```
SELECT
    COUNT(*)
FROM
    us_hotdog_purchases
WHERE
    YEAR(purchase_time) = ‘2018’
```

As you can see, we are using the *YEAR* function to grab the year part from the *purchase_time* column. This function call will prevent the database from being able to use an index for the *purchase_time* column search, because we indexed the value of *purchase_time*, but not the return value of *YEAR(purchase_time).*

To overcome this challenge and tune this SQL query, you can index the function's result, by using Generated Columns, which are available starting MySQL 5.7.5.

Another solution can be to find an alternative way to write the same query, without using the function call. In this example, we can transform that condition to a 2-way range condition, which will return the same results:

```
SELECT
    COUNT(*)
FROM
    us_hotdog_purchases
WHERE
    purchased_at >= '2018-01-01'
        AND purchased_at < '2019-01-01'
```

Example #2 - avoid OR conditions

Consider this query, which selects the amount of posts on Facebook posted after new year's eve, or posted by a user named Mark.

```
SELECT
```

```
    COUNT(*)
FROM
    fb_posts
WHERE
    username = 'Mark'
        OR post_time > '2018-01-01'
```

Having an index on both the *username* and *post_time* columns might sound helpful, but in most cases, the database won't use it, at least not in full. The reason will be the connection between the two conditions - the OR operator, which makes the database fetch the results of each part of the condition separately.

An alternative way to look at this query can be to 'split' the OR condition and 'combine' it using a UNION clause. This alternative will allow you to index each of the conditions separately, so the database will use the indexes to search for the results and then combine the results with the UNION clause.

```
SELECT …
FROM …
WHERE username = 'Mark'
    UNION
SELECT …
FROM …
WHERE post_time > '2018-01-01'
```

Please note that if you don't mind duplicate records in your result set, you can also use UNION ALL (which will perform better than the default UNION DISTINCT).

Example #3 - Avoid sorting with a mixed order

Consider this query, which selects all posts from Facebook and sorts them by the username in an ascending order, and then by the post date in a descending order.

```
SELECT
    username, post_type
FROM
    fb_posts
ORDER BY username ASC , post_type DESC
```

MySQL (and so many other relational databases), cannot use indexes when sorting with a mixed order (both ASC and DESC in the same ORDER BY clause). This changed with the release of the reversed indexes functionality and MySQL 8.x.

So what can you do if you didn't upgrade to the latest MySQL version just yet? First, we'd recommend to re-consider the mixed order sort. Do you really need it? If not, avoid it.

So you decided you need it, or your product manager said: "No way we can manage without it"? Another option will be to use Generated columns (available on MySQL 5.7.5+) to create a reversed column and sort on that column instead of the original. As an example, assume you're sorting on a numeric column, you can create a generated column with the negative numeric value that correlates to the original number and sort on that new column in the opposite order. That way, all columns will have the same sort order in the ORDER BY clause, but the sort will happen as originally defined by your product's requirement.

The last potential solution won't always be an option, so your last resort will be upgrading to the latest MySQL version which supports mixed order sorting using indexes.

Example #4 - Avoid conditions with different column types

Consider this query, which selects the number of red fruits in a forest.

```
SELECT
    COUNT(*)
FROM
    forest
WHERE
    fruit_color = 5;    /* 5 = red */
```

Assuming the column *fruit_color*'s type is VARCHAR, or just anything non-numeric, indexing that column won't be very helpful, as the required implicit cast will prevent the database from using the index for the filtering process.

So how can you tune this SQL query? You have two options to optimize this query. The first one would be to compare the column to a constant value that matches the column's type, so if it's a VARCHAR column, compare it to *'5'* (with single quotes) and not to *5* (which is a numeric comparison which will result in an implicit cast).

A better option will be to adjust the column's type to match the most suitable type for the values the column holds. In this example, the column should be altered to an INT type. Please note that altering a column's type can be a complicated task, so read about the challenges of that task before heading towards it.

Avoid LIKE searches with prefix wildcards

Consider this query, which searches all Facebook posts from a username which includes the string *'Mar'*, so we are searching for all posts written by users named *Mark, Marcus*, *Almar,* etc.

```
SELECT
    *
FROM
    fb_posts
WHERE
    username LIKE '%Mar%'
```

Having a wildcard '%' at the beginning of the pattern will prevent the database from using an index for this column's search. Such searches can take a while..

In this case, there are two options to improve this query's performance. The first one is trivial - consider whether the prefix

wildcard is important enough. If you can manage without it, get rid of it.

Another option will be to use full-text indexes. Please note though, that these indexes and the MATCH ... AGAINST syntax aren't free from challenges and have some differences when compared to the familiar LIKE expressions in MySQL.

Conclusion

In this first part of our *SQL query optimization* series, we covered the importance of *wise* indexing, we went through several examples of possible obstacles while using indexed columns in queries, and we also detailed several other tips and tricks which can be helpful for better query performance. See you in the next post.

3 thoughts on “SQL Query Optimization: Top 5 Tips for Modern Query Optimization”

mark wusinich

SEPTEMBER 12, 2018 AT 1:58 AM

Great suggestions. Nothing we haven't heard before, but always good to be reminded of good recommendations.

Øystein Grøvlen

SEPTEMBER 13, 2018 AT 1:23 PM