SQL Performance Quick Guide

A collection of tactics you can use to improve the performance of your SQL queries

Ben Brumm

www.databasestar.com

Welcome

In this guide, you're going to learn about a range of tactics and techniques to improve the performance of your SQL queries.

This guide includes:

- a range of tactics for improving the performance of a query
- an explanation of each tactic and how it can help
- examples of SQL code to use the tactic
- links to references for more information

So if you want to improve the performance of your SQL queries, you'll love this guide.

Let’s get right into it.

Table of Contents

1 - Create Indexes 3

2 - Remove Distinct 4

3 - Remove Calculated Fields from WHERE and Joins 6

4 - Remove Unneeded Tables and Columns 8

5 - Use UNION ALL Instead Of Union 9

6 - Use EXISTS Instead Of IN 10

7 - Avoid Wildcard Matching 12

8 - Use Bind Variables 14

9 - Change UNION to CASE 16

10 - Use a Temporary Table 19

11 - Partition Your Data 21

12 - Change Correlated Subqueries to Joins 22

13 - Avoid Cursors 24

14 - Consider Joining to a Subquery 26

Conclusion 28

1 - Create Indexes

An index is a feature in SQL databases that allow you to speed up the execution of a SELECT query.

When you create an index, an object is created on the database. You specify the table and column (or columns) for the index to use.

Then, when a query uses these columns, it could use the index to find the data it needs, and is often much faster than without the index.

They aim to improve the performance of a SELECT query, so if that's the kind of query you are looking to improve, it could help. Indexes may have a small impact on performance for INSERT, DELETE, and UPDATE queries, as there is some overhead needed to also update the index when a row is changed, so that may be something to consider.

How To

Here's how to create an index in SQL. The syntax is the same for Oracle, SQL Server, MySQL, and Postgres.

```
CREATE INDEX index_name
ON table_name (columns);
```

Here's an example:

```
CREATE INDEX idx_emp_id
ON employee (id);
```

This creates an index called idx_emp_id on the id column in the employee table.

This will mean any query that mentions the id column may use this index.

A few other points to consider:

- Look into creating indexes on columns that are in the WHERE clause or JOIN clauses of your queries
- Consider creating an index on columns in the SELECT clause if it's an important query or only has a few columns
- You can create indexes on multiple columns

More Information

SQL Indexes - The Definitive Guide

YouTube - SQL Indexes

2 - Remove Distinct

The DISTINCT keyword in SQL will eliminate duplicate rows from your result.

It's helpful if you want unique rows. It doesn't eliminate duplicate values in each column - it's the combination of all columns in your result that is considered.

If you have a SELECT query that uses DISTINCT, and you have the DISTINCT there because you don't want duplicates, it may not be needed.

Often, a DISTINCT keyword is added because there are multiple tables being joined, and some WHERE clauses, but there are still duplicate rows. So, a DISTINCT keyword is added to remove duplicates.

Adding DISTINCT solves a problem of duplicate rows, but the problem is caused by your query and possibly the data. You may be missing a criteria in your join, or missing a table, or missing a WHERE clause. Any of these could be causing duplicate rows.

The database will perform an operation when it finds the DISTINCT keyword to remove duplicates, and this step can be expensive (aka it can take time).

If you remove the DISTINCT, and focus on finding an issue in your query and fixing it, you'll have a more correct query and will save execution time by avoiding the DISTINCT step.

How To

To remove the DISTINCT keyword, you can just remove it from your query.

If you're getting duplicate rows, investigate your query and tables to see where the duplicates are coming from.

It's often from columns in your tables that are not being displayed.

For example, if you want to see unique products and categories, you might have a query like this:

```
SELECT DISTINCT
p..product_name,
c.category_name,
p.price
FROM product p
INNER JOIN product_category c ON p.category_id = c.category_id
WHERE c.active = 1;
```

This would show products where the category is active.

However, you might be seeing duplicate data if you remove the distinct.

Let's say you've looked into the records, and you can see there are some products with the same name and price, but one has a product.active value of 1 and another has 0.

So, it seems we have forgotten to exclude inactive products. We excluded inactive categories, but not products.

Here's our updated query:

```
SELECT
p..product_name,
c.category_name,
p.price
FROM product p
INNER JOIN product_category c ON p.category_id = c.category_id
WHERE c.active = 1
AND p.active = 1;
```

More Information

SQL Distinct: The Complete Guide

YouTube - SQL Distinct

3 - Remove Calculated Fields from WHERE and Joins

Calculated fields are fields where a function or formula is applied to a column.

This can be useful if you want to display data in a different way,

It's possible to also include them in WHERE clauses or joins.

However, this can sometimes cause performance issues, even if an index is created.

You may have an index on a column, but if a query has a function or calculation on that column, then the index will probably not get used.

So, consider removing calculated fields from WHERE clases and joins, if you can find another way to write the query.

How To

There are two ways to avoid this issue.

You could remove the calculation from the WHERE clause or join, and use the uncalculated value. The ability to do this would depend on the query, but here's an example:

```
SELECT
first_name,
last_name,
salary,
start_date
FROM employee
WHERE annual_salary / 12 > 2000;
```

This will show all employees that have a monthly salary greater than 2000.

Another way to write this query is to change the other side of the WHERE clause to be an annual salary.

```
SELECT
first_name,
last_name,
salary,
start_date
FROM employee
WHERE annual_salary > 24000;
```

Another approach would be to create an index on the field that includes the function.

So, if you have a query on the lastname column that uses UPPER, consider creating an index on the upper function applied to that column:

Here's a query that filters on this value:

```
SELECT
first_name,
last_name
FROM employee
WHERE UPPER(last_name) = 'SMITH';
```

Here's an index that could be created to help this query:

```
CREATE INDEX idx_emp_upperlname
ON employee (UPPER(last_name));
```

More Information

SQL Indexes - The Definitive Guide

4 - Remove Unneeded Tables and Columns

One way to improve the performance of your query is to remove any tables and columns from your query that are not needed.

This may not apply to your query, and is more likely an issue on larger queries.

If you look into your query, you may find that there are some columns that are selected but are not used by your application/report/whatever is using the query.

Or, you might find that you are joining to some tables that are not necessary.

Including tables and columns in your query can add extra steps for the database to process the query, so if it's possible to remove them, then do that.

How To

Look into your query and the application, report, or another person that uses it, and see if there are any columns that can be removed or any tables that can be removed.

Test that removing these will still show the same results.

More Information

Not applicable.

5 - Use UNION ALL Instead Of Union

Both UNION ALL and UNION are set operators: they operate on sets of data.

They allow you to combine the results of two queries into one result.

When faced with the need to combine results, we often go straight to using UNION because it's shorter and likely simpler:

```
SELECT first_name, last_name
FROM employee
UNION
SELECT first_name, last_name
FROM customer;
```

However, the UNION keyword eliminates duplicate results, and the UNION ALL does not. The step of removing duplicates can be expensive.

So, if you don't need to remove duplicate rows, consider using UNION ALL instead:

```
SELECT first_name, last_name
FROM employee
UNION ALL
SELECT first_name, last_name
FROM customer;
```

You'll likely get the same results with better performance.

How To

Look at any queries that use UNION, and if you're sure they don't need to remove duplicate rows, change them to UNION ALL.

More Information

SQL Set Operators

6 - Use EXISTS Instead Of IN

The IN keyword and the EXISTS keyword are similar, but operate a little differently.

The IN keyword is used to check if a value exists in a list of values, and if so, the row is displayed.

The EXISTS keyword is used to check if any value is returned from a subquery, and if so, the row is displayed.

The main difference is that when EXISTS finds a match, the database stops looking for other values. Using an IN keyword, the database will keep searching through all values.

The EXISTS keyword may exit faster, and therefore be faster to use than a comparable IN keyword.

So, consider replacing an IN keyword with an EXISTS keyword. It can't be done in every situation, but it's possible in some.

How To

Consider this query that uses an IN keyword:

```
SELECT
e.first_name,
e.last_name
FROM employee e
WHERE e.department_id IN (
  SELECT d.id
  FROM department d
  WHERE d.active = 0
);
```

You may be able to replace it with an EXISTS keyword:

```
SELECT
e.first_name,
e.last_name
FROM employee e
WHERE EXISTS (
  SELECT 1
  FROM department d
  WHERE d.id = e.department_id
  AND d.active = 0
);
```

This may improve the performance of the query, especially if there is a lot of data.

This example is simple and just used to demonstrate this point. A better way to write this actual query may be to use a join instead of a subquery, which I've written more about in tactic 14 in this guide.

More Information

SQL IN and NOT IN Guide

SQL Operators: The Complete Guide

7 - Avoid Wildcard Matching

Wildcard matching is where you have a WHERE clause that uses LIKE and a wildcard character, such as _ or %.

This is often needed, such as searching for a string that the user provides.

Here's an example:

```
SELECT
p.id,
p.product_name,
c.category_name,
p.price
FROM product p
INNER JOIN category c ON p.category_id = c.id
WHERE c.name LIKE '%E';
```

The problem with this is that any indexes on the field being searched on can't be used, and the query may be slow.

You could add an index, but it may not be used.

A solution to this could be:

- Identify the reason you need to perform wildcard matching
- Look into the data you are matching on
- Update your query, if possible, to use a filter that does not use wildcard matching

You may be able to filter on specific values if there is a range of values that can be matched on. It depends on your data.

How To

To remove the wildcard matching, understand why it is needed. In the example above, we want to show products for categories starting with "E".

Next, look into the data you are matching on. In this example, we can look into the category table to find categories that start with "E". Let's say we do this, and find there are two: Electronics and Entertainment.

Next, we can update our query to filter on these values (assuming it meets our requirements):

```
SELECT
p.id,
p.product_name,
c.category_name,
p.price
FROM product p
INNER JOIN category c ON p.category_id = c.id
```

```
WHERE c.name IN ('Electronics', 'Entertainment');
```

This could cause an index on the name column to be used, and the query to be faster.

More Information

SQL Wildcards

YouTube: SQL LIKE Keyword

8 - Use Bind Variables

A bind variable is a feature of SQL that allows you to indicate that a part of a query will be provided when the query is run. When you run the query, you specify the value, and the query is run.

This is a good approach for queries that are similar, but a value can change, such as a criteria in a WHERE clause.

Here's an example of a query that gets cases assigned to a user:

```
SELECT
case_id,
case_name,
case_status,
created_date
FROM submitted_cases
WHERE assigned_to_id = 1;
```

Rather than changing the query each time depending on the assigned_to_id value, or using concatenation in your application code to add a value, you can use a bind variable.

Instead of the actual value, you add in a prompt for a bind variable.

Then, when you run the SQL in the editor, or when the application runs the SQL, the actual value needs to be provided.

This can improve security (by avoiding SQL injection), and also improve performance (because multiple similar queries are treated as the same query and not optimised from scratch).

How To

If you have any queries that use inputs from the application or the user, such as the example above, replace the input value with the bind variable syntax depending on your database vendor:

Vendor	Bind Variable Syntax
Oracle	:variable_name
SQL Server	@variable_name
MySQL	?
PostgreSQL	:variable_name

So, writing the same query above using bind variables in PostgreSQL would look like this:

```
SELECT
case_id,
case_name,
```

```
case_status,
created_date
FROM submitted_cases
WHERE assigned_to_id = :current_user_id;
```

The value of current_user_id is provided when the query is run.

More Information

A Guide to SQL Bind Variables

9 - Change UNION to CASE

We may sometimes see queries that use a UNION (or UNION ALL) keyword to combine data from two queries.

The two queries may be very similar, but have some slight differences. They could get some data from the same tables, and have a different WHERE clause.

Here's an example of a query that does this. It has two SELECT queries combined with UNION.

- The first query shows competitors and their medals who are from some countries in North or South America
- The second query shows competitors and their medals who competed in games in the US.

Here's the query:

```
SELECT p.full_name, gco.age, n.region_name, m.medal_name, g.games_year,
g.season, c.city_name, 'FC' AS data_code
FROM noc_region n
INNER JOIN person_region pr ON n.id = pr.region_id
INNER JOIN person p ON pr.person_id = p.id
INNER JOIN games_competitor gco ON p.id = gco.person_id
INNER JOIN games g ON gco.games_id = g.id
INNER JOIN games_city gci ON g.id = gci.games_id
INNER JOIN city c ON gci.city_id = c.id
INNER JOIN competitor_event ce ON gco.id = ce.competitor_id
INNER JOIN medal m ON ce.medal_id = m.id
WHERE n.noc IN ('USA', 'CAN', 'MEX', 'BRA', 'ARG')

UNION

SELECT p.full_name, gco.age, n.region_name, m.medal_name, g.games_year,
g.season, c.city_name, 'CC'
FROM noc_region n
INNER JOIN person_region pr ON n.id = pr.region_id
INNER JOIN person p ON pr.person_id = p.id
INNER JOIN games_competitor gco ON p.id = gco.person_id
INNER JOIN games g ON gco.games_id = g.id
INNER JOIN games_city gci ON g.id = gci.games_id
INNER JOIN city c ON gci.city_id = c.id
INNER JOIN competitor_event ce ON gco.id = ce.competitor_id
INNER JOIN medal m ON ce.medal_id = m.id
WHERE city_name IN (
     'Calgary',
     'Los Angeles',
     'Salt Lake City',
     'Lake Placid',
     'Atlanta',
     'Rio de Janeiro',
     'Mexico City',
```

```
      'Montreal',
      'Vancouver',
      'St. Louis'
);
```

The two queries are similar, but the WHERE clauses are different.

The database will get data from each of these tables twice, running similar queries and performing the filter.

A more efficient way would be to query the data from the tables once, and use logic in the WHERE clause to show the records you need.

Here's an updated query. It queries the tables once, and uses a WHERE clause to find rows that match condition 1 or condition 2.

```
SELECT p.full_name, gco.age, n.region_name, m.medal_name, g.games_year,
g.season, c.city_name,
CASE
WHEN n.noc IN ('USA', 'CAN', 'MEX', 'BRA', 'ARG') THEN 'FC'
WHEN c.city_name IN (
      'Calgary',
      'Los Angeles',
      'Salt Lake City',
      'Lake Placid',
      'Atlanta',
      'Rio de Janeiro',
      'Mexico City',
      'Montreal',
      'Vancouver',
      'St. Louis'
) THEN 'CC'
END AS data_code
FROM noc_region n
INNER JOIN person_region pr ON n.id = pr.region_id
INNER JOIN person p ON pr.person_id = p.id
INNER JOIN games_competitor gco ON p.id = gco.person_id
INNER JOIN games g ON gco.games_id = g.id
INNER JOIN games_city gci ON g.id = gci.games_id
INNER JOIN city c ON gci.city_id = c.id
INNER JOIN competitor_event ce ON gco.id = ce.competitor_id
INNER JOIN medal m ON ce.medal_id = m.id
WHERE (
      n.noc IN ('USA', 'CAN', 'MEX', 'BRA', 'ARG')
      OR c.city_name IN (
            'Calgary',
            'Los Angeles',
            'Salt Lake City',
            'Lake Placid',
            'Atlanta',
            'Rio de Janeiro',
```

```
            'Mexico City',
            'Montreal',
            'Vancouver',
            'St. Louis'
      )
);
```

This query should run more efficiently than the original.

How To

If you have any queries that use UNION or UNION ALL and are similar, look into combining them into one and using AND or OR keywords to capture your logic.

You may find using a CASE statement useful if you need to display logic in a different way.

More Information

SQL CASE Statement Explained

10 - Use a Temporary Table

A temporary table is a feature in SQL that allows you to create a table that's only used for a short time (such as your current session). Data can be added to the table, and when you are finished with the table, it is removed.

It's helpful in operations where you are working with a large amount of data, or doing complicated things to data, or using a set of data multiple times.

You can create a temporary table in your SQL script or stored procedure, populate it with data you are using, and use the temporary table later in your script or procedure.

It may not be useful for all situations, but it's worth trying if you are working with a lot of data or a complex operation.

How To

Creating a temporary table in your database is slightly different for each vendor.

Oracle

This query creates a global temporary table (the table can be seen by others but the data is removed):

```
CREATE GLOBAL TEMPORARY TABLE temp_customers (
id NUMBER(10),
cust_name VARCHAR2(100)
)
ON COMMIT DELETE ROWS;
```

This query creates a private temporary table (the table is deleted at the end of the session):

```
CREATE PRIVATE TEMPORARY TABLE ora$ptt_temp_customers (
id NUMBER(10),
cust_name VARCHAR2(100)
)
ON COMMIT DROP DEFINITION;
```

SQL Server

This query creates a local temporary table:

```
CREATE TABLE #temp_customers (
id INT,
cust_name VARCHAR(100)
);
```

To create a global temporary table, use ## instead of # at the start of the table name.

Local temporary tables are visible only in the current session, and global temporary tables are visible in other sessions.

MySQL

This query creates a temporary table in MySQL:

```
CREATE TEMPORARY TABLE temp_customers (
id INT,
cust_name VARCHAR(100)
);
```

PostgreSQL

This query creates a temporary table in PostgreSQL:

```
CREATE TEMPORARY TABLE temp_customers (
id NUMBER(8),
cust_name VARCHAR(100)
);
```

More Information

SQL Temp Tables: The Ultimate Guide

11 - Partition Your Data

If you work with tables that contain a lot of data, but you only look at some of the data at a time, you may benefit from partitions.

A common example is data that is captured for different times or days, but you are only interested in specific years or months.

If your table has many years of data, then your query would need to look at all data to find what it needs.

However, you can use a database feature called "partitions". This will change the way the data is grouped in the table. You can, for example, partition your tables by year. Any queries that refer to data for a single year will only look in data for that one partition, which can improve the performance of your query.

How To

Partitioning is done differently for each database.

Rather than providing an example to create partitions, I'll link to the documentation for each vendor which explains how to do it.

More Information

The official documentation for partitioning for each vendor:

Oracle

SQL Server

MySQL

PostgreSQL

12 - Change Correlated Subqueries to Joins

A correlated subquery is a subquery that refers to the outer table.

It's a handy feature of SQL and can help in many cases.

However, it can cause your query to run slower, depending on your query and the database vendor.

Sometimes, a correlated query can be replaced with a join to the table.

Here's a simple example, from an earlier tactic in this guide:

```
SELECT
e.first_name,
e.last_name
FROM employee e
WHERE EXISTS (
  SELECT 1
  FROM department d
  WHERE d.id = e.department_id
  AND d.active = 0
);
```

This query shows employees that have a department that is no longer active. It uses a correlated subquery because the `WHERE d.id = e.department_id` means that the row in the subquery refers to the "e" table in the outer query.

We can improve this by removing the subquery and using a join instead.

```
SELECT
e.first_name,
e.last_name
FROM employee e
INNER JOIN department d ON e.department_id = d.id
WHERE d.active = 0;
```

This query should perform better because the database can optimise for the data being used at the right time. However, you should test both versions as it could make your query worse (or return the wrong data).

How To

If your query uses a correlated subquery, try to change the query to use a join, by including the tables from the subquery in the main query, joining on the same field, and adding the same criteria.

More Information

SQL Subqueries: The Complete Guide

YouTube: When to Use a Subquery in SQL

13 - Avoid Cursors

There is a programmatic language that comes with each vendor's database, and the language is different. For Oracle, it is PL/SQL, for SQL Server it is T-SQL, and so on.

This language lets you do things you can't normally do in SQL, such as conditional logic or other behaviour.

Cursors are a feature of these programmatic languages. They are objects that allow you to store the results of a query.

The problem with working with cursors is that they are often misused. They are commonly used to store results of a query, and then these results are looped through or iterated through, with logic or operations being performed on each row individually.

This is useful in other programming languages, but SQL operates best when using sets of data instead of individual records.

If your script or stored procedure uses a cursor, and it has some code to loop through the items in the cursor, consider replacing it with operations that work on the entire set.

For example, if your code looks like this (pseudo code used for this example):

```
DECLARE cursor;
FETCH INTO cursor SELECT …;
FOR EACH row IN cursor
  IF (condition) THEN
  UPDATE table SET col = newvalue;
  END IF;
END LOOP;
CLOSE cursor;
```

You may want to consider changing it to just use an update statement:

```
UPDATE table SET col = newvalue
WHERE condition;
```

You can use data from one table to define the records to update in another table, if that's what your code does.

How To

Look at your procedural code and determine if cursors are being used and if they are being looped or iterated.

If they are, consider replacing them with pure SQL that does not loop through data.

More Information

A Beginner's Guide to Cursors

14 - Consider Joining to a Subquery

One way to improve the performance of certain queries is to join to a subquery instead of a table.

When a join is performed, the data being brought into the query (for example, from a table), could be large. Your query may then filter this data later on during the process, which means there was time spent on loading data that wasn't used.

A way to improve this is to limit the data before it is brought in. This can be done by joining to a subquery rather than a table.

This can help because the subquery does the operations first, which then limits the data that is being brought into the main query.

Here's an example:

```
SELECT
c.id,
c.customer_name,
SUM(o.order_value) AS total_order_value
FROM customer c
INNER JOIN all_orders o ON c.id = o.customer_id
WHERE o.order_date > 20220101
AND o.status != 'Cancelled'
GROUP BY c.id, c.customer_name;
```

If the all_orders table has a lot of data, there's a chance the database will read all of the data, and then filter it just to those where the order date is after 1 Jan 2022, and the status is not Cancelled, and then SUM it for each customer;

This could be improved by changing the query to join to a subquery:

```
SELECT
c.id,
c.customer_name,
SUM(o.order_value) AS total_order_value
FROM customer c
INNER JOIN (
  SELECT o.customer_id,
  SUM(o.order_value)
  FROM all_orders o
  WHERE o.order_date > 20220101
  AND o.status != 'Cancelled'
) o ON c.id = o.customer_id
GROUP BY c.id, c.customer_name;
```

This subquery may perform the filter and aggregation before passing the data back to the main query, and may result in a performance increase.

It may not improve the performance, but it's worth trying.

How To

Take a look at your query. For each table in your query, analyse how you are using the columns. If you are ignoring some of the data or summarising it, it may be better to replace the join of the table to a join of a subquery.

To do this:

1. Write a subquery that summarises or filters the data to what you need
2. Replace the table in your main query with this subquery
3. Test the performance is OK
4. Test that the data is correct

More Information

SQL Joins: The Complete Guide

Conclusion

That brings us to the end of this guide.

I hope you've found it useful.

There should be at least one tactic in this guide you can try for a query you're looking at or writing. Not all of them work in all situations, but having this library of tactics can help you now and in the future.

If you're interested in learning more about improving the performance of your SQL queries, check out my Write Faster SQL course inside the Database Star Academy.

It includes lessons on these concepts, how to find slow queries to work on, how to test the performance change, how to test the data is the same, and more.

You can find out more about Database Star Academy here: https://www.databasestar.com/dsa/

Thanks,

Ben Brumm

www.DatabaseStar.com