Sravya Madipalli

MASTERING SQL-QUERY OPTIMIZATION

Looking to improve your SQL skills?

Writing efficient queries is key to boosting performance and reducing database load.

Here are some powerful tips to help you write faster, more effective SQL queries.

Let's dive-in....

1. Join and Subquery Optimization

Why It's Important:

- The order in which tables are joined in a query can significantly impact performance. When you start with a large table, the database engine may process a large number of rows early on, which can lead to excessive memory and CPU usage.
- Conversely, starting with a smaller table reduces the initial data set that the database needs to handle, allowing for faster joins and reducing the overall workload.

Bad Example

In this query, the large table large_table is processed first. This can lead to the database handling a significant amount of data upfront, which increases memory usage and slows down the overall query performance.

1. Join and Subquery Optimization

Good Example

```sql
-- Efficient JOIN with small table first
SELECT * FROM small_table st
JOIN large_table lt ON st.id = lt.id;
```

By starting with the smaller table small_table, the database initially processes a smaller result set. This allows for more efficient use of resources, as fewer rows are handled during the initial join, leading to faster query execution.

Additional Tip: When optimizing joins, also consider the use of indexes on the join columns. Indexing the columns used in the join condition can further improve performance, especially when combined with an optimized join order.

2. Use EXISTS Instead of IN for Subqueries for large datasets

Why It's Important:

- The EXISTS clause is generally more efficient than IN for subqueries, particularly when dealing with large datasets.
- IN compares every value returned by the subquery with each row in the outer query, which can be slow if the subquery returns many rows.
- EXISTS, on the other hand, stops processing as soon as it finds a match, making it faster..

Bad Example

```sql
-- BAD: Using IN with a large subquery
SELECT first_name, last_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE
order_date > '2023-01-01');
```

This query forces the database to compare each customer_id in the customers table with all customer_id values returned by the subquery, which can be slow.

2. Use EXISTS Instead of IN for Subqueries for large datasets

Good Example

SQL

```sql
-- GOOD: Using EXISTS for better performance
SELECT first_name, last_name
FROM customers
WHERE EXISTS (SELECT 1 FROM orders WHERE orders.customer_id =
customers.customer_id AND order_date > '2023-01-01');
```

The EXISTS clause stops processing as soon as it finds a match, making the query faster, especially with large datasets.

3. Use Window Functions Instead of Self-Joins and Correlated Subqueries

Why It's Important:

- Window functions allow you to perform calculations across a set of table rows related to the current row, without reducing the number of rows in the result set.
- They are more efficient than self-joins or correlated subqueries because they often require fewer passes over the data and can be processed in a single scan.

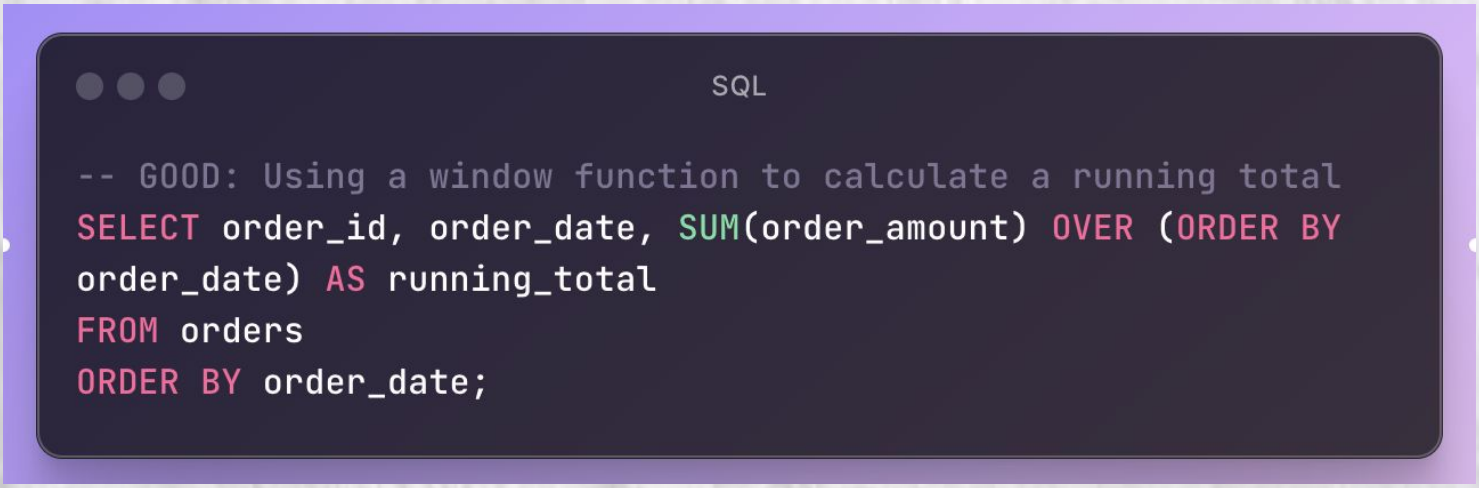
Bad Example

```sql
-- BAD: Using a self-join to calculate a running total
SELECT o1.order_id, o1.order_date, SUM(o2.order_amount) AS
running_total
FROM orders o1
JOIN orders o2 ON o2.order_date <= o1.order_date
GROUP BY o1.order_id, o1.order_date
ORDER BY o1.order_date;
```

This query requires the database to repeatedly scan the orders table, leading to inefficient processing.

3. Use Window Functions Instead of Self-Joins and Correlated Subqueries

Good Example



The window function calculates the running total in a single pass over the data, making the query more efficient.

4. Avoid Leading Wildcards in LIKE Clauses instead restructure your data

Why It's Important:

- Using wildcards at the start of a LIKE pattern (%value) prevents the database from using indexes effectively, leading to full table scans.
- This can significantly slow down your queries, especially when working with large datasets.

Bad Example

SQL

```sql
-- BAD: Using a leading wildcard in LIKE
SELECT * FROM products WHERE product_name LIKE '%widget';
```

Instead of relying on inefficient text searches, you can restructure your data to include additional columns that indicate whether a specific pattern exists, allowing for faster, more efficient queries.

4. Avoid Leading Wildcards in LIKE Clauses instead restructure your data

How to Restructure Your Data:

- One approach is to add a boolean column, such as isWIDGET, that flags whether the term "widget" appears in the product_name.
- This column can be indexed, allowing for fast, efficient queries without the need for text searching.

Good Example

```sql
-- GOOD: Using the indexed boolean column
SELECT * FROM products WHERE isWIDGET = TRUE;
```

This strategy can be applied to any frequently searched text pattern.

If you have multiple patterns to search for, consider adding multiple boolean columns, such as isWIDGET, isGADGET, etc., or using a more advanced data structuring approach like bitwise flags if appropriate.

5. Using Temporary Tables Instead of CTEs for Large Datasets

Why It's Important:

- CTEs (Common Table Expressions) are often recalculated each time they are referenced, which can be inefficient, especially with large datasets.
- If your query involves multiple CTEs handling large datasets, joining them can lead to poor performance.
- Temporary tables, however, are created once, can be indexed, and reused across multiple queries, making them more efficient and flexible.

5. Using Temporary Tables Instead of CTEs for Large Datasets

Good Example of using temp tables:

SQL

```sql
-- GOOD: Using temporary tables for large datasets
CREATE TEMPORARY TABLE temp_large_sales AS
SELECT customer_id, SUM(order_amount) AS total_sales
FROM sales
WHERE order_date > '2023-01-01'
GROUP BY customer_id;

CREATE TEMPORARY TABLE temp_large_returns AS
SELECT customer_id, SUM(return_amount) AS total_returns
FROM returns
WHERE return_date > '2023-01-01'
GROUP BY customer_id;

SELECT s.customer_id, s.total_sales, r.total_returns
FROM temp_large_sales s
JOIN temp_large_returns r ON s.customer_id = r.customer_id;
```

5. Using Temporary Tables Instead of CTEs for Large Datasets

Benefits of Temporary Tables Over CTEs

- Reusability: Unlike CTEs, which are only accessible within the context of a single query, temporary tables can be reused across multiple queries within the same session. This allows for greater flexibility in complex queries or when performing multiple operations on the same dataset.
- Indexing: Temporary tables can be indexed, which can greatly improve the performance of subsequent queries that filter or join on the indexed columns.
- Reduced Redundancy: Temporary tables avoid the need to recalculate the same large dataset multiple times, as is often the case with CTEs.
- Efficiency: Overall, using temporary tables reduces the computational load on your database, especially when dealing with large datasets or complex queries that need to be broken down into manageable steps.

6. Use Proper Indexing and Know When to Add Indexes

Why It's Important:

- Indexes are crucial for speeding up data retrieval, especially for queries that filter or join large datasets.
- Proper indexing can drastically improve performance, but it's also important to know when and where to add indexes.

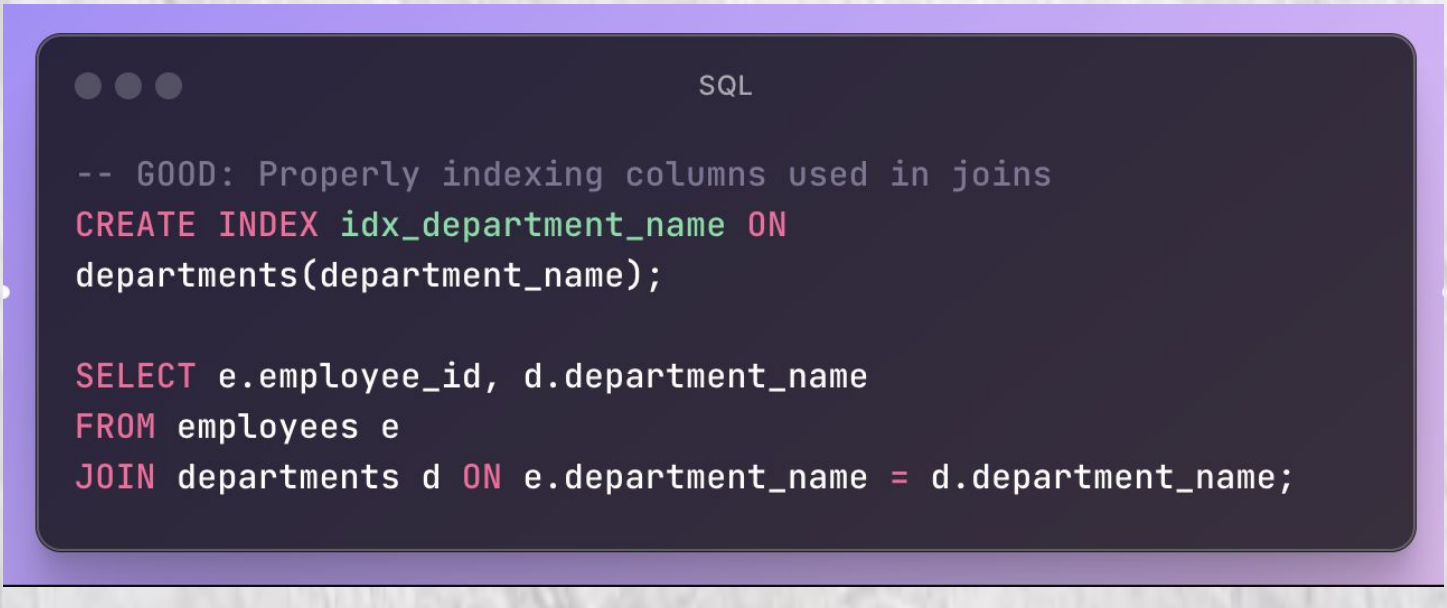
Bad Example

```sql
-- BAD: Running queries on non-indexed columns
SELECT e.employee_id, d.department_name
FROM employees e
JOIN departments d ON e.department_name = d.department_name;
```

Without an index on department_name, the database must perform a full table scan, which is inefficient.

6. Use Proper Indexing and Know When to Add Indexes

Good Example



Indexing department_name allows the database to quickly find matching rows, improving query performance.

Additional Tips:

- Avoid using scalar functions on indexed columns in your WHERE clauses, as they can prevent the index from being used effectively.
- Every time you use a data table check for the table's indexes in data dictionary and make major joins and group by's on those columns.

Additional Optimizations

- Simplify Complex Logic with CASE Statements
  - Complex logic using multiple OR conditions can be inefficient and difficult to optimize. CASE statements allow you to simplify these conditions, making your queries easier to read, maintain, and optimize.
- Avoid SELECT * and Specify Columns Explicitly
  - Using SELECT * retrieves all columns from a table, which can lead to unnecessary data retrieval, increased I/O operations, and slower query performance. By explicitly specifying the columns you need, you reduce the amount of data processed and transferred, which speeds up your queries.
- Filter Early with WHERE Clauses
  - Applying filters early in your query process reduces the dataset before it undergoes more expensive operations like joins, aggregations, or window functions. This minimizes the amount of data that the database engine needs to process, leading to faster execution times.
- Analyze and Use Execution Plans
  - Regularly analyze execution plans to identify performance bottlenecks and optimize your queries accordingly.

Additional Optimizations

- Use LIMIT to Restrict Results
  - Limit the number of rows returned by your query when you don't need the entire dataset to reduce processing and improve performance.
- Use IS NULL and IS NOT NULL for NULL Checks
  - When dealing with NULL values, use IS NULL or IS NOT NULL instead of equality checks.
- Eliminate Unnecessary DISTINCT
  - Only use DISTINCT when you need to remove duplicates; otherwise, it adds unnecessary processing.
  - Applying filters early in your query process reduces the dataset before it undergoes more expensive operations like joins, aggregations, or window functions. This minimizes the amount of data that the database engine needs to process, leading to faster execution times.
- Analyze and Use Execution Plans
  - Regularly analyze execution plans to identify performance bottlenecks and optimize your queries accordingly.

Sravya Madipalli

Was this Helpful?

- Save it
- Follow Me
- Repost and Share it with your friends