DOI 10.34132/mspc2025.01.14.26

Bohdan Somryakov,
Viktor Ralenko

SQL DATA ANALYSIS FOR BIG DATA OPTIMIZATION: BEST PRACTICES AND PITFALLS

The theses investigates SQL optimization techniques essential for enhancing query performance, reducing resource consumption, and ensuring efficient data retrieval in large-scale database systems. It outlines seven key strategies: avoiding functions on WHERE columns to leverage indexes, partitioning large tables for faster searches, filtering data early to minimize processing, utilizing indexes for frequent queries, applying TOP (LIMIT) for testing, optimizing ON clauses in joins, and avoiding unnecessary DISTINCT operations.

These methods enhance query execution speed, scalability, and system performance in big data environments. Practical examples and common pitfalls are highlighted, providing actionable insights for database administrators and developers to build responsive, resource-efficient applications while managing complex queries and large datasets effectively.

Key words*: SQL optimization, big data, query performance, indexing, partitioning, database scalability, data retrieval, query efficiency, performance tuning.*

SQL optimization is crucial for improving query performance, reducing resource consumption, and ensuring faster data retrieval [1]. By efficiently structuring queries, databases can handle large datasets more effectively, minimizing the time taken for processing and enhancing the overall system performance [2]. This helps in achieving more responsive applications, especially when dealing with complex operations or large volumes of data. Proper optimization ensures that queries execute quickly, making the system scalable and able to handle increasing workloads without sacrificing performance. The top seven SQL optimization techniques include:

1) Avoid Functions on WHERE Columns.

Using functions on columns in the WHERE clause can prevent Sybase IQ from utilizing indexes effectively, which forces the database to perform full table scans, leading to slower query performance.

```
SELECT *
FROM sales
WHERE YEAR(sale_date) = 2025;
```

YEAR(sale_date) applies a function to the sale_date column, and Sybase IQ might not use an index on sale_date. Avoid Functions on Indexed Columns. Instead, rewrite the query to perform a direct comparison.

```
SELECT *
FROM sales
WHERE sale_date BETWEEN '2025-01-01' AND '2025-12-31';
```

2) Partition Large Tables.

Partitioning improves performance by distributing the data across smaller, more manageable physical blocks, which makes searches faster and more efficient when querying specific ranges of data.

To partition a table by date, for example, you might partition it by month or year. This way, queries filtering on the sale_date column will only scan the relevant partition.

```
CREATE TABLE sales (
  sale_id INT,
  sale_date DATE,
  amount DECIMAL(10, 2)
)
PARTITION BY RANGE (sale_date)
 (PARTITION p2020 VALUES LESS THAN '2023-01-01',
  PARTITION p2021 VALUES LESS THAN '2024-01-01',
  PARTITION p2022 VALUES LESS THAN '2025-01-01');
```

Queries that filter by year can take advantage of partitioning by scanning only the relevant partition. This minimizes unnecessary I/O operations and leads to faster query execution and improved overall performance.

```
SELECT *
FROM sales
WHERE sale_date BETWEEN '2024-01-01' AND '2024-12-31';
```

3) Filter Early

Filtering early in the query reduces the size of the data being processed, which can lead to significant performance gains. Apply WHERE filters as early as possible, especially before joins or aggregations.

```
SELECT customer_name, SUM(amount)
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.status = 'completed'
GROUP BY customer_name;
```

Filtering the orders table first reduces the amount of data involved in the join operation.

```
SELECT customer_name, SUM(amount)
FROM customers
JOIN (
  SELECT customer_id, amount
  FROM orders
  WHERE status = 'completed'
) AS filtered_orders ON customers.customer_id = filtered_orders.customer_id
GROUP BY customer_name;
```

4) Use Indexes

If you often query by email in the customers table, create an index on that column. Queries filtering by email will use the index, significantly improving query performance.

```
CREATE INDEX idx_email ON customers(email);
SELECT *
FROM customers
WHERE email = 'bob@example.com';
```

That being said, too many indexes can slow down insert and update operations because the indexes must also be updated. Choose only relevant columns to index, especially those involved in frequent filtering and joining operations.

5) TOP (LIMIT)

TOP is useful for testing and to limit the result set in exploratory queries. It helps prevent full table scans and large result sets during query development.

```
SELECT TOP 10 *
FROM employees;
```

6) Filter early in ON clause

A LEFT JOIN ensures that all rows from the left table (e.g., employees) are returned, even if there is no matching row in the right table (e.g., departments). If there is no matching row in the right table, the columns from the right table are filled with NULL values. The WHERE clause is used to filter rows after the join has been completed. It is applied to the final result set, which means it comes into play after SQL performs the join operation and has already combined the rows from multiple tables.

```
SELECT e.name, d.department_name
FROM employees e
LEFT JOIN departments d
  ON e.department_id = d.department_id
WHERE d.department_name = 'Engineering';
```

The ON clause is used to specify the join condition itself. It dictates how the rows from the two tables are combined. The condition d.department_name = 'Engineering' is part of the join condition itself, which means only the rows where the department is 'Engineering' are considered for joining with the employees table.

```
SELECT  e.name, d.department_name
FROM  employees e
LEFT  JOIN departments d
  ON e.department_id = d.department_id
  AND d.department_name = 'Engineering';
```

The ON clause can be used to filter data before the join operation happens, so that unnecessary rows from the right table are excluded from the join process. This can help improve performance because you limit the amount of data being processed during the join. It also ensures that the result of a LEFT JOIN includes rows from the left table even if there is no match in the right table (i.e., it can preserve rows from the left table with NULL values in the columns of the right table)

7) Avoid DISTINCT when unnecessary

The query selects unique combinations of department_id and the count of active employees in each department.

```
SELECT  DISTINCT  department_id, COUNT(*)
FROM  employees
WHERE  status = 'active'
GROUP  BY department_id;
```

DISTINCT is unnecessary when you're already using GROUP BY. The GROUP BY will automatically ensure that you get one row per department_id, so there's no need to add DISTINCT.

```
SELECT  department_id, COUNT(*)
FROM  employees
WHERE  status = 'active'
GROUP  BY department_id;
```

Using DISTINCT is very resource-intensive because it requires the database to sort and remove duplicate rows, which can increase processing time and memory usage, especially on large datasets.

In conclusion, optimizing SQL queries is crucial for maintaining efficient and scalable database performance, especially as data volumes increase. By employing strategies such as avoiding functions on WHERE columns, filtering early, using indexes, partitioning large tables, avoiding DISTINCT when unnecessary, and limiting data for testing, you can significantly enhance query performance and reduce resource consumption. These techniques help ensure that your database can handle complex queries and large datasets swiftly, leading to faster response times, improved user experience, and overall better system performance.

References

1. "SQL Performance Explained" by Markus Winand, 2012.
2. "Joe Celko's SQL for Smarties: Advanced SQL Programming" (The Morgan Kaufmann Series in Data Management Systems) by Joe Celko, 2005.

DOI 10.34132/mspc2025.01.14.26

Богдан Сомряков
Віктор Раленко

АНАЛІЗ ДАНИХ SQL ДЛЯ ОПТИМІЗАЦІЇ ВЕЛИКИХ ДАНИХ: НАЙКРАЩІ ПРАКТИКИ ТА ПІДВОДНІ КАМЕНІ

Дослідження розглядає техніки оптимізації SQL, важливі для підвищення продуктивності запитів, зменшення споживання ресурсів та забезпечення ефективного отримання даних у великих базах даних. Визначено сім ключових стратегій: уникнення функцій у стовпцях WHERE для використання індексів, партиціонування великих таблиць для швидшого пошуку, рання фільтрація даних для зменшення обробки, використання індексів для частих запитів, застосування TOP (LIMIT) для тестування, оптимізація умов ON у з'єднаннях та уникнення непотрібного DISTINCT.

Ці методи підвищують швидкість виконання запитів, масштабованість і продуктивність системи у середовищах великих даних. Наведено практичні приклади та типові помилки, що надають корисні

рекомендації адміністраторам баз даних і розробникам для створення швидких, ефективних застосунків при роботі зі складними запитами та великими наборами даних..

Ключові слова: *оптимізація SQL, великі дані, продуктивність запитів, індексування, розділення на частини, масштабованість баз даних, отримання даних, ефективність запитів, налаштування продуктивності.*

Відомості про авторів / Information about the Authors

Богдан Сомряков, здобувач кафедри інтелектуальних інформаційних систем факультету комп'ютерних наук Чорноморського національного університету імені Петра Могили, м. Миколаїв, Україна.

Bohdan Somryakov, student of the Faculty of Computer Sciences at Petro Mohyla Black Sea National University, Mykolaiv, Ukraine.

Віктор Раленко, викладач кафедри інженерії програмного забезпечення факультету комп'ютерних наук Чорноморського національного університету імені Петра Могили, м. Миколаїв, Україна. E-mail: ralenko.v@chmnu.edu.ua, orcid: https://orcid.org/0009-0009-4161-8468.

Viktor Ralenko, Lecturer at the Department of Software Engineering, Petro Mohyla Black Sea National University, Mykolaiv, Ukraine. E-mail: ralenko.v@chmnu.edu.ua, orcid: https://orcid.org/0009-0009-4161-8468.

 Стаття отримана редакцією 19.05.2025.