International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

A Summary of SQL Optimization Methods for Optimizing Query Performance

Ramneet Singh Chadha[1], Prabhjeet Singh Raina[2]

[1]Scientist F, Embedded Systems, Centre for Development of Advanced Computing (C-DAC), Noida, Uttar Pradesh, India
[2]Project Engineer, Embedded Systems, Centre for Development of Advanced Computing (CDAC), Noida, Uttar Pradesh, India

Abstract:
Efficient data retrieval is crucial for database-driven applications due to the rapid expansion of data in modern information systems. The typical way to manage and change relational databases is with SQL. But if SQL queries aren't written correctly, they might take longer to run, require a lot of CPU power, and waste valuable resources. Optimizing SQL queries makes databases work more efficiently by changing slow queries into faster execution plans. A study on SQL query optimization techniques aims to improve query execution performance in relational database systems. The study examines query optimization techniques, including indexing strategies, query rewriting, join optimization, execution plan analysis, and cost-based optimization. Also, real examples show how optimized searches considerably reduce execution time compared to non-optimized queries. Effective optimization strategies can significantly enhance database efficiency and reduce query latency, based on experiments.

Keywords: SQL, Query Optimization, Database Systems, Indexing, Execution Plan, and Database Performance.

1. INTRODUCTION

Relational database management systems (RDBMS) are extensively utilized in contemporary applications, including banking systems, healthcare systems, e-commerce platforms and enterprise resource planning solutions. These systems store huge amounts of structured data that must be access effectively to help with decision-making and operational procedures. Structured Query Language (SQL) is the standardized language employed to engage with relational databases. SQL queries enable users to obtain, insert, update, and delete data within database tables. As database sizes increase significantly, the efficiency of SQL queries becomes an important factor in maintaining optimum system performance. Query optimization is the procedure that determines the most efficient method for performing a SQL query. An inaccurately optimized query could lead to unnecessary disk I/O operations, excess CPU cycles, and substantial memory usage.

Consequently, contemporary database management systems incorporate query optimizers that assess alternative execution plans and choose the most economical method. The main goal of query optimization is to decrease query execution time while decreasing system utilization of resources. Optimization approaches comprise of query rewriting, indexing, join ordering, and the implementation of efficient

execution plans. This paper presents an overview of SQL query optimization strategies and analyzes their effect on database performance.

2. A REVIEW OF THE LITERATURE

Several studies have investigated methods to improve SQL query performance.

Banubakode and Dakhode [1] performed a comparative analysis of query optimization techniques used with relational and object-oriented database systems. The research they conducted showed how query clauses like WHERE, GROUP BY, and HAVING affect whether a query works.

Yan and Larson [2] suggested an optimization method where GROUP BY operations are done before JOIN operations. This method reduces the number of tuples that need to be joined, which speeds up how quickly queries are executed.

Kim [3] introduced methods for changing nested SQL queries into equivalent, non-nested versions. These changes help database optimizers create more efficient execution plans.

Mithani and his team [4] created a model that automatically turns user requests into optimized SQL queries. This model uses smart strategies to improve how searches are done. The model can find missing indexes, inefficient join orders, and unnecessary operations that slow down queries.

Habimana [5] discussed several practical ways to write efficient SQL queries. These include avoiding SELECT *, using subqueries sparingly, and using UNION ALL instead of UNION.

3. PROCESSING SQL QUERIES

When a database management system (DBMS) gets a SQL query, it runs a series of steps to get the data that was requested for. The basic goal of query processing is to turn a high-level SQL query that the user wrote into a low-level execution plan that the database engine can use.

Query processing is a very important part of determining out how effectively a database system performs. MySQL, PostgreSQL, Oracle, and SQL Server are examples of modern relational database systems that use advanced query processing methods to make queries execute more quickly.

The process of processing a query usually has three main steps:

- Parsing and Translation
- Query Optimization
- Query Execution

Each of these steps helps make SQL queries work better.

A. Parsing and Translation:

Parsing and translation are the first steps in SQL query processing. The query parser checks to make sure that the SQL query is written correctly when it is sent to the database system. This step is like the parsing phase in a compiler.

During this stage, the parser does several types of checks:

- Checks the syntax of SQL query is written correctly.
- Makes sure that the tables and columns being referenced are there.
- Checks that data types and constraints are correct.
- Verifies that users have the right to access database items.

If the query satisfies these tests, it is then turned into an internal form called a query tree or relational algebra expression. This picture shows what needs to be done to get the data you want.

The following SQL query is an example:

```
SELECT name
FROM employee
WHERE salary > 50000;
```

can be translated into a relational algebra expression as:

πname(σsalary>50000(employee)) (1)

In this phrase:

- σ stands for the selection operation.
- π stands for the operation of projection.

The internal representation created at this point is then sent to the query optimizer.

B. Query Optimization:

Query optimization is one of the most critical stages in SQL query processing. The purpose of query optimization is to determine the most efficient way to execute a given SQL query. A SQL query can often be executed in multiple ways, each having different execution costs. The query optimizer evaluates different execution strategies and selects the one with the lowest estimated cost.[6]

Two major types of query optimization techniques are commonly used:

1) Rule-Based Optimization:

Rule-based optimization uses a set of predefined rules to transform queries into more efficient forms. These rules include:

- Performing selection operations as early as possible.
- Reducing the size of intermediate results.
- Reordering join operations.
- Eliminating redundant expressions.

Rule-based optimizers follow fixed heuristics rather than calculating actual execution costs.

2) Cost-Based Optimization:

Cost-based optimization evaluates multiple execution plans and estimates their cost based on various parameters such as:

- Number of disk I/O operations
- CPU processing time
- Memory usage
- Network communication cost

The optimizer uses database statistics such as table size, distribution of values, and index availability to estimate these costs. The execution plan with the lowest estimated cost is selected for execution.

C. Query Execution Plan:

After choosing the best query plan, the database system generates a thorough execution plan that lists the steps needed to get the data.[6]

A query execution plan usually has the following steps:

- Table scans
- Index scans
- Join operations
- Sorting operations
- Aggregation operations

The optimizer has to choose which join method is used when a query involves joining two tables. Some

common join algorithms are:

- Nested Loop Join
- Hash Join
- Merge Join

The performance of each join method depends on the size of the tables and if indexes are available or not.

D. Query Execution:

In the final stage, the query execution engine executes the selected query plan. The execution engine retrieves data from disk, performs necessary operations such as filtering, joining, and sorting, and produces the final result set. During execution, the DBMS may also apply additional runtime optimizations such as buffer management, caching, and parallel query execution. Efficient query execution ensures that database systems can handle large volumes of data while maintaining acceptable response times for users.

E. Factors Affecting Query Processing Performance:

Several factors influence the efficiency of SQL query pro-cessing:

- Database indexing strategy
- Query structure and complexity
- Table size and data distribution
- Hardware resources such as CPU, memory, and disk speed
- Availability of query execution statistics

Proper database design and query optimization techniques can significantly improve the performance of SQL query processing.

4. SQL OPTIMIZATION TECHNIQUES

Various strategies can be employed to optimize SQL queries and enhance database performance. SQL optimization aims to reduce query execution time, minimize disk I/O operations, and improve the overall efficiency of the system. Modern database management systems utilize several optimization techniques, including indexing, query rewriting, join optimization, and execution plan refinement, to enhance performance.[7]

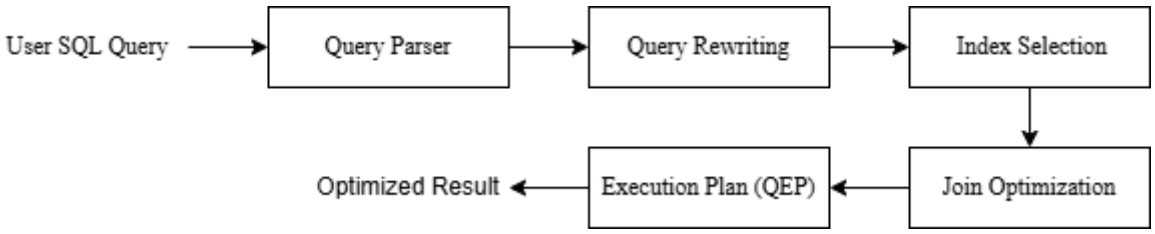

Figure 1: Workflow of SQL query optimization from input query to optimized execution.

A. Indexing:

One of the most common ways to speed up queries in relational databases is by using indexing. An index is a data structure that helps the database engine find rows fast without having to look through the whole table.

Most of the time, data structures like B-trees or hash tables are used to make indexes. If a query employs indexed columns in its search conditions, the database system can use an index lookup instead of a full table scan. This cuts down on the number of times the disk has to be accessed.

As an example, consider the following query:

```
SELECT name
FROM employee
WHERE employee_id = 1001;
```

The database can rapidly find the record if the employee ID column is indexed. The database has to look through every row of the table in search of the matching value if there is no index.

There are different types of indexes:

- Primary Index: This is created automatically on the primary key columns.
- Unique Index: makes sure that the values in the index are all different.
- Composite Index: Made up of more than one column to speed up searches that use multiple attributes at once.
- Clustered Index: This tells the table how the data is physically arranged.

Indexes can speed up queries, but too many of them can make storage costs go up and make inserting or updating data take a while. Because of this, indexes need to be properly planned.

B. Query Rewriting:

Query rewriting means changing a SQL query into a form that the database engine can run more quickly. This technique accelerates up the query without affecting what it logically returns.

One way for rewriting is to replace subqueries with join operations. In lots of circumstances, joins are faster because the query optimizer can come up with better execution plans.

For example, consider this subquery:

```
SELECT name
FROM employee
WHERE department_id IN (SELECT department_id FROM department);
```

You can also write this query using the following join operation:

```
SELECT e.name
FROM employee e
JOIN department d
ON e.department_id = d.department_id;
```

Other ways to rewrite a query are:

- Eliminating redundant conditions.
- Using joins instead of related subqueries.
- Making complicated expressions easier.
- Using UNION ALL instead of UNION when you don't need duplicates.

These changes support lower the cost of computing and improve in execution.

C. Join Optimization:

Join operations can be real resource hogs in SQL, particularly when dealing with massive datasets. Optimizing joins means finding the most efficient method for combining tables and determining the optimal order for those joins. Database systems employ several join algorithms:

- **Nested Loop Join:** This approach examines every row in one table, comparing it to every row in another. When dealing with smaller tables, it's a good match.

- **Hash Join:** A hash join uses a hash table to find matching rows in two tables. This method is particularly effective when dealing with large datasets.
- **Merge Join:** This method requires both input tables to be pre-sorted. It executes the join by simultaneously scanning both tables. The order in which joins are performed significantly impacts how efficiently they work. Query optimizers usually choose a join order that aims to reduce the size of the intermediate results.

For example, joining smaller tables initially can reduce the number of rows processed in subsequent joins.

D. Avoiding SELECT *:

When you use the SELECT * query type, it gets all of the columns from a table, even if you only need a few of them. This makes it take longer to transfer data, use memory, and execute queries. Instead, it is best to clearly state which columns are needed.

For example:

```
SELECT *
FROM employee;
```

can also be written as:

```
SELECT name, department_id
FROM employee;
```

This method reduces the quantity of data that the database engine has to deal with, which speeds up queries.

E. Using Query Execution Plans:

A query execution plan shows how a database system runs a SQL query. MySQL and PostgreSQL have tools like EXPLAIN that let you look into the execution plan of a query. By analyzing the execution plan, developers can find activities that are inefficient, like full table scans, joins that aren't needed, or indexes that are missing.

For instance:

```
EXPLAIN SELECT *
FROM employee
WHERE department_id = 5
```

Database managers might discover how queries work and use the correct optimization techniques by analyzing at execution plans.

F. Reducing Redundant Data Retrieval:

Another important method to optimize is to reduce the quantity of data that has to be processed while a query is running. Queries that get more rows or columns need more memory and computational cost. Using the **WHERE** clause to filter data early, using **LIMIT** conditions, and eliminating needless joins are all ways to speed up the execution of queries. To make a database work better overall, it's important to use efficient data retrieval methods.

Table1: Comparison Of SQL Optimization Techniques

Technique	Purpose	Benefit	Limitation
Indexing	Faster data retrieval	Reduces table scans	Extra storage required
Query Rewriting	Better query structure	Faster execution	Requires expertise
Join Optimization	Efficient table joins	Lower processing costs	Difficult for large datasets
Avoid SELECT *	Less unnecessary data	Saves memory and time	Needs manual column selection
Execution Plan Usage	Analyzes performance	Finds bottlenecks	Requires interpretation
Data Reduction	Minimizes processed data	Improves efficiency	May limit flexibility

5. QUERY OPTIMIZATION EXAMPLES

This section gives a few real-world examples of how query optimization can make a database perform better. Each example shows how an optimized SQL query is better than a non-optimized one and how the optimization makes the query run faster and with less work.[8]

A. Using REGEXP Instead of Multiple LIKE Clauses:

In many database queries, multiple LIKE conditions are used to search for different patterns within the same column. Although this method works, it could lead to inefficient query execution. This is because the database engine needs to evaluate each LIKE condition separately.

Using many pattern-matching conditions can make things more complex. This is because multiple OR operators can increase the computational load and possibly cause the same column to be scanned multiple times.[9] As a result, this can lead to higher CPU usage and longer query execution times.

Non-Optimized Query:

```
SELECT * FROM Phone_list
WHERE LOWER(item_name) LIKE '%apple%'
OR LOWER(item_name) LIKE '%blackberry%'
OR LOWER(item_name) LIKE '%LG%';
```

In this query, the database system evaluates three separate pattern-matching conditions. As the number of conditions increases, the cost of evaluating the query also increases.

Optimized Query:

```
SELECT * FROM Phone_list
WHERE REGEXP_LIKE(LOWER(item_name),
'apple|blackberry|LG');
```

The optimized query replaces multiple LIKE conditions with a single regular expression pattern using the REGEXP_LIKE function. This allows the database engine to perform pattern matching using a single expression, reducing the number of logical comparisons required.

Using regular expressions provides several advantages. It reduces the need for conditional checks, which simplifies the query. This also makes the query easier to read and maintain. In addition, regular expressions simplify pattern matching when there are many possible values. As a result, the optimized query can reduce execution time, especially when working with large datasets.

B. Using Temporary Tables Instead of Long IN Clauses:

Another typical performance problem happens when the IN clause of a SQL query has a large list of values. The database engine has to look at each value in the list separately when there are a lot of them, which makes the query more costly to run.

Non-Optimized Query:

```
SELECT * FROM Phone_list
WHERE item_id IN (1234,4567,7890)
```

This query tries to see if the column item_id matches any value in the list. This method doesn't work well when the list has a lot of values because the comparison has to be done over and over again.

Optimized Query:

```
SELECT * FROM Phone_list p
JOIN temp_ids t
ON p.item_id = t.item_id;
```

In the optimized version, the values are kept in a temporary table (temp_ids) and then added to the main table. This method lets the database optimizer employ fast indexing and join algorithms.

Some of the benefits of using temporary tables are:

- Reducing repeated comparisons in huge value lists.
- Allowing indexes to be used on the temporary table.
- Enhancing query execution plans generated by the optimizer.

This optimization is especially useful when the list of values is quite long or when the same list of values is used in a multiple query.

6. CONCLUSION

Given the large amount of data in today's applications, how well we process database queries is very important for organizations. Therefore, optimizing SQL queries is crucial for maintaining the performance and scalability of relational database systems.

This study offered a thorough examination of SQL query optimization methods, with a particular emphasis on improving the efficiency of how queries are executed. The analysis covered several critical aspects, including SQL query processing, query rewriting approaches, indexing strategies, join optimization techniques, and execution plan analysis. Moreover, the report presented practical examples to illustrate the enhanced performance of optimized queries relative to their unoptimised versions.[10] The experimental case study provided supplementary evidence that the application of optimization techniques, such as the substitution of multiple LIKE conditions with regular expressions and the use of temporary tables in place of extensive IN clauses, led to a significant reduction in query execution durations. As a

result, these changes led to reduced CPU usage, fewer disk input/output operations, and a more efficient way of allocating system resources.

Good query optimization requires a combination of effective database design, careful indexing, and the use of efficient query writing methods. Database administrators and developers must regularly analyze how queries are executed and use appropriate optimization techniques to maintain high system performance. Future investigations might focus on advanced optimization techniques tailored for distributed databases, cloud-based database designs, and big data processing frameworks. Given the escalating volumes and intricacies of data, the development of intelligent, automated query optimization strategies will be essential for contemporary database systems.

REFERENCES

1. A. Banubakode and V. Dakhode,"Comparative Analysis of Query Optimization in Object-Oriented and Relational Databases", 2014.
2. W. P. Yan and P. Larson, "Performing Group by before join", IEEE Conference on Data Engineering, 1994.
3. W. Kim, "On Optimizing an SQL-like Nested Query", ACM Transactions on Database Systems, 1982.
4. F. Mithani, S. Machchhar, and F. Jasdanwala, "A Novel Approach for SQL Query Optimization",IEEE Conference, 2016.
5. J. Habimana, "Query Optimization Techniques -Tips for Writing Efficient SQL Queries", IJSTR, 2015.
6. M. Muntjir, "A Novel Evaluation of Query Processing and Optimization in DBMS," International Journal of Engineering Research & Technology (IJERT),vol. 3, no. 11, pp. —, Nov. 2014.
7. S. Chaudhuri, "An overview of query optimization in relational systems," Stanford University, 1998.
8. H. Tm, U. K, M. Shafiulla and Dadapeer, "An Overview of SQL Optimization Techniques for Enhanced Query Performance," 2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), Ballar, India, 2023.
9. J. -H. Park and J. -H. Kang, "Optimization of XQuery Queries Including FOR Clauses," Second International Conference on Internet and Web Applications and Services (ICIW'07), Morne Mauritius, 2007.
10. V.S. Ramdoss, "Optimizing Database Queries: Cost and Performance Analysis," International Journal of Science and Research Archive (IJSRA), vol. 2, no. 2,pp. 293–297, 2021.