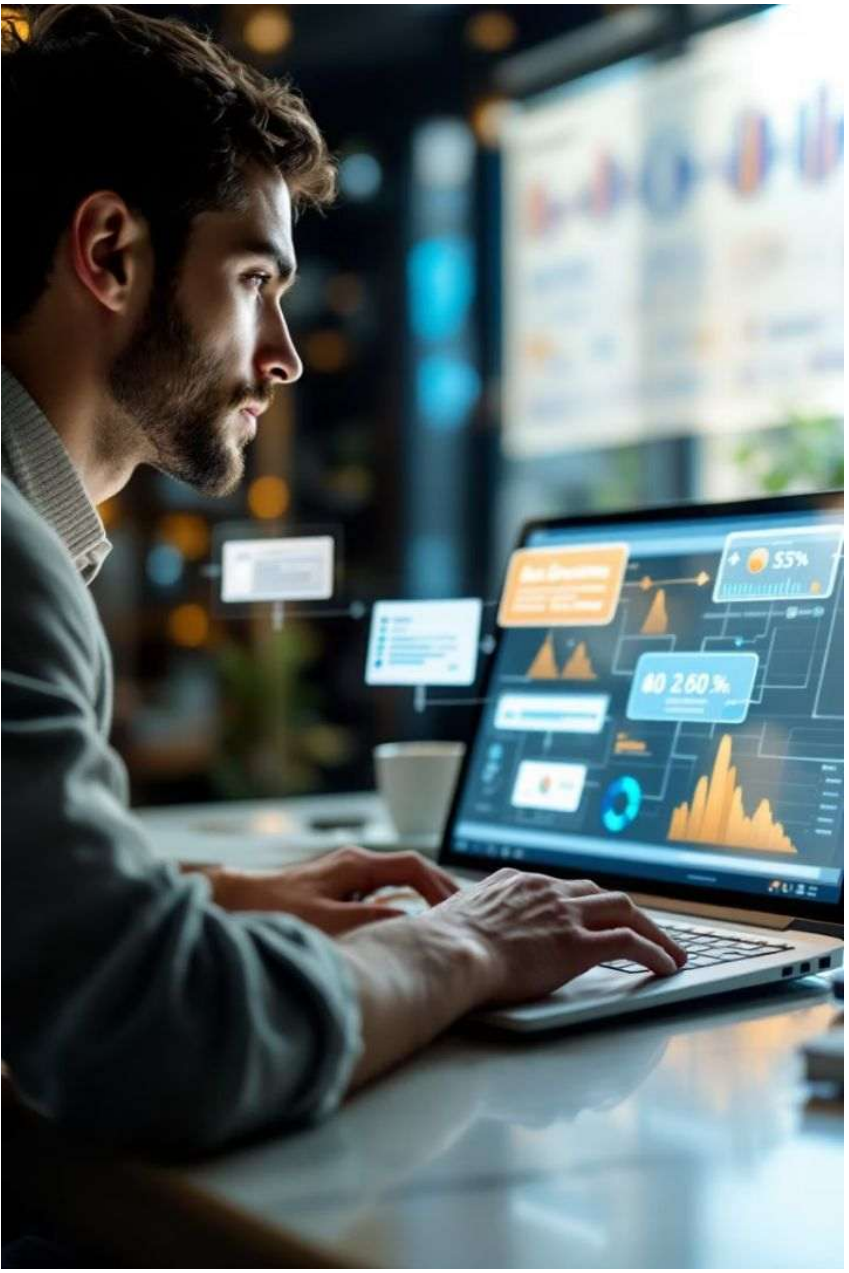


MERiT

Introduction to Query Optimization

Query optimization is a crucial process used by Database Management Systems to find the most efficient way to execute SQL queries. The goal is to reduce response time, minimize resource usage, and enhance overall database performance.

Database optimizers employ two main strategies: cost-based optimization, which calculates the efficiency of different execution plans, and rule-based optimization, which follows predefined rules to determine the best approach.

Understanding these optimization techniques is essential for database administrators and developers who want to create high-performing database applications.

Cost-Based vs. Rule-Based Optimizers

Rule-Based Optimizers

These optimizers select execution plans using a set of predefined rules that determine which access paths to use. They follow fixed heuristics regardless of the data distribution or volume.

- Simple implementation
- Predictable behavior
- Less adaptive to data changes

Cost-Based Optimizers

Modern optimizers that analyze statistics about data, indexes, and hardware to estimate the "cost" of different execution plans. They dynamically adjust to changing data patterns.

- Superior for complex queries
- Adapts to data distribution
- Requires updated statistics

Optimization Steps in the DBMS

Parse & Validate

Check SQL syntax and semantics to ensure the query is valid and can be executed

Generate Plans

Create multiple possible execution strategies using available indexes and join methods

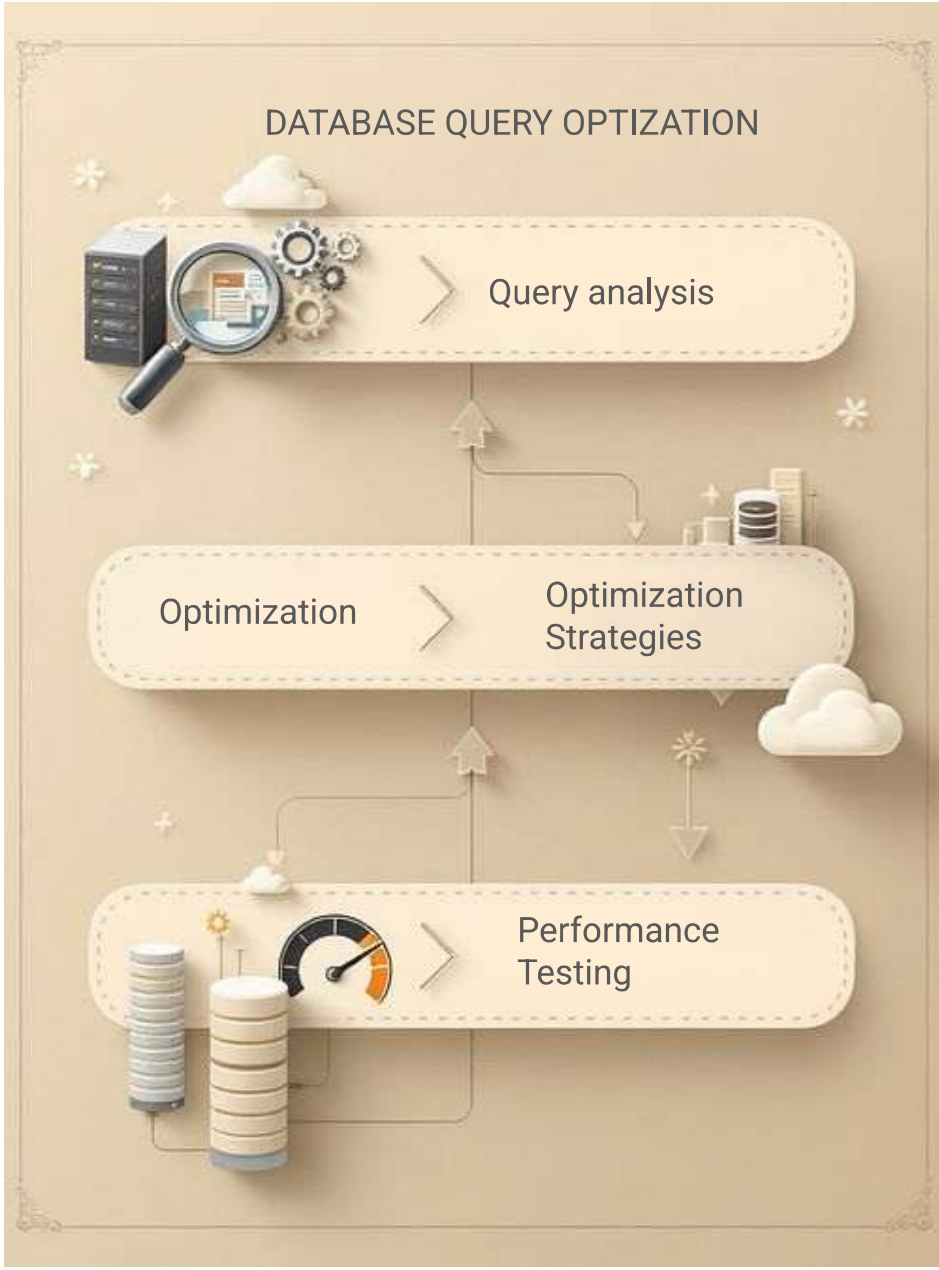
Estimate Costs

Calculate resource requirements for each plan based on statistics

Select Best Plan

Choose the plan with the lowest estimated cost for execution

During optimization, the DBMS may also employ rewriting techniques such as simplifying logical expressions or transforming subqueries into more efficient forms. This multi-step process ensures queries are executed in the most resource-efficient manner possible.



Understanding Execution Plans

Definition

A detailed roadmap showing how the DBMS will process your query, including all operations and their sequence

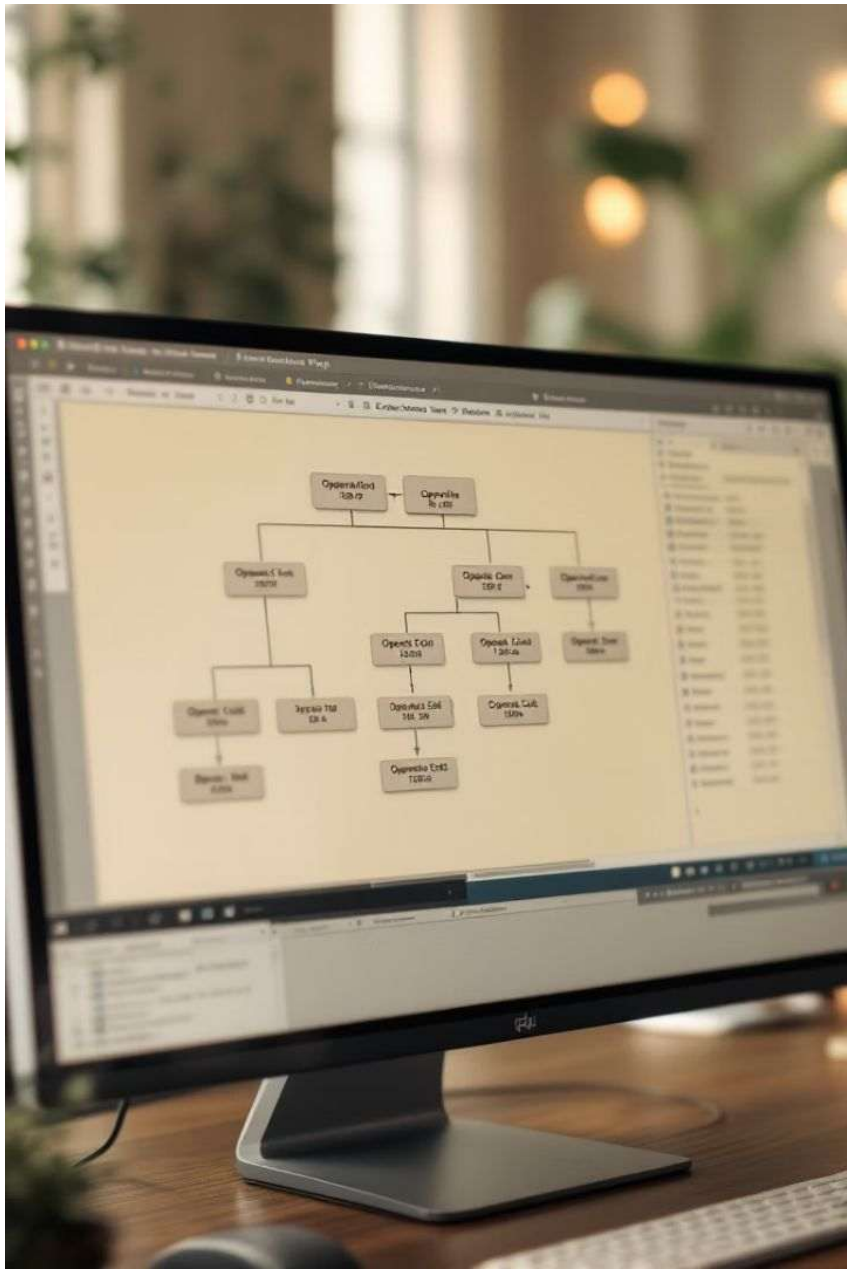
Components

Shows table scans, index usage, join methods, filters, and sorting operations with their estimated costs

Viewing Tools

EXPLAIN shows the planned strategy, while EXPLAIN ANALYZE executes the query and provides actual runtime statistics

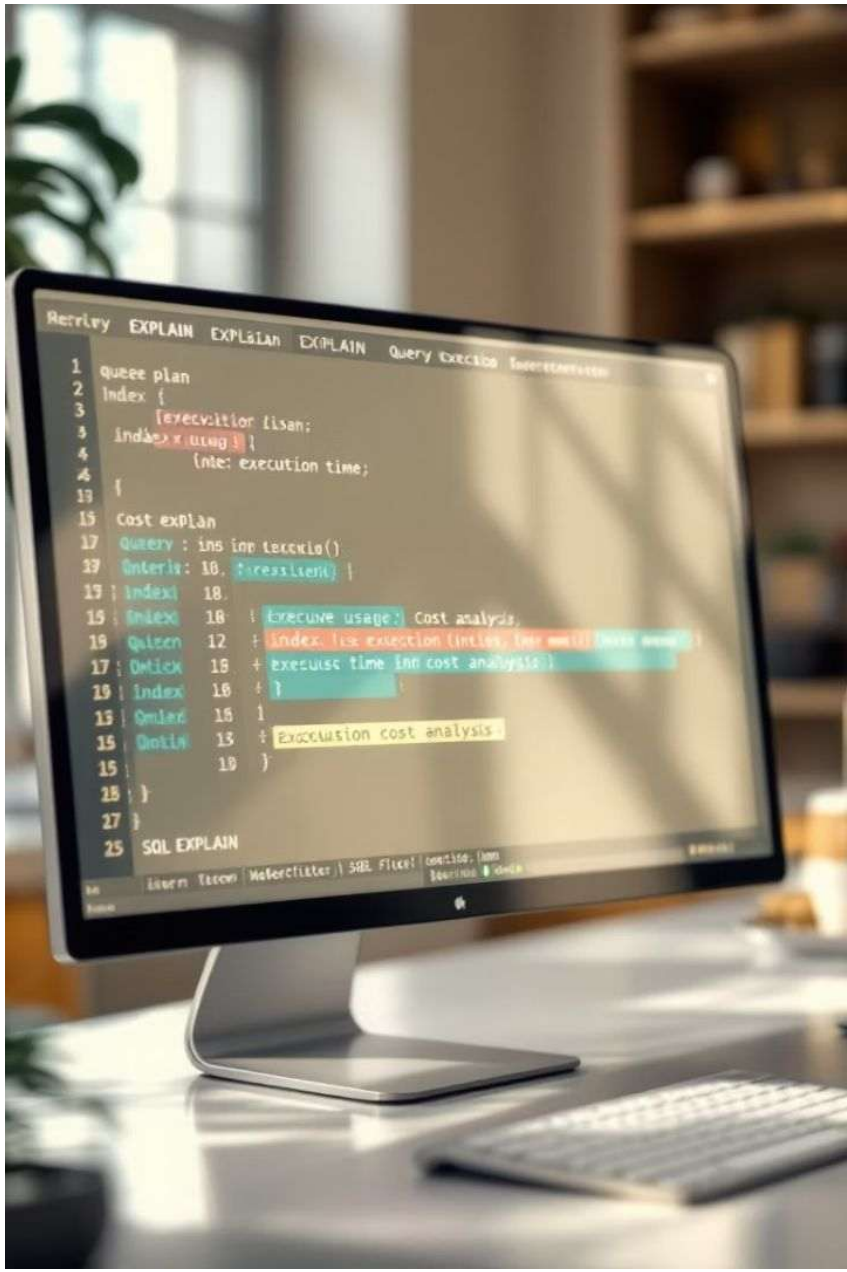
Execution plans are the key diagnostic tool for query performance analysis. They reveal exactly how the database interprets and processes your SQL statements, allowing you to identify inefficient operations and optimize accordingly.



Interpreting EXPLAIN and EXPLAIN ANALYZE

Operation	Description	Performance Implication
Seq Scan	Sequential table scan - reads all rows	Slow for large tables; may indicate missing indexes
Index Scan	Uses an index to locate specific rows	Efficient for selective queries
Nested Loop Join	For each row in outer table, scans inner table	Good for small tables or highly selective joins
Hash Join	Builds hash table from smaller table	Efficient for larger datasets with equality joins
Merge Join	Merges two pre-sorted inputs	Efficient when inputs are already sorted

When analyzing execution plans, focus on operations with high costs and look for opportunities to add or modify indexes. Compare estimated versus actual row counts to identify where the optimizer's statistics may be inaccurate.



Introduction to Indexes

B-Tree

Default index type that works well for equality and range queries

Excellent for sorting operations

Functional

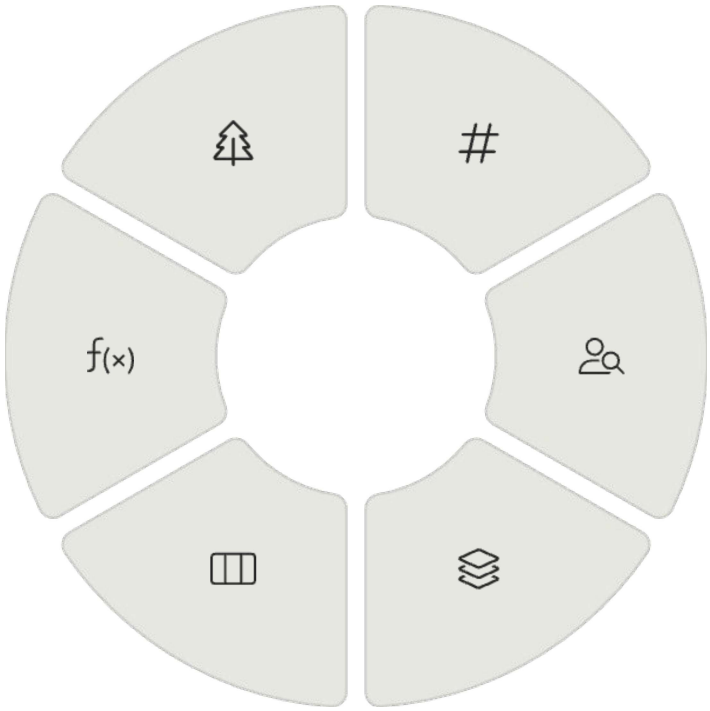
Indexes results of functions applied to columns

Useful for case-insensitive searches

Composite

Involves multiple columns in a specific order

Efficient for queries using those exact columns



Hash

Optimized for equality comparisons

Fast lookups but doesn't support ranges

GiST

Generalized Search Tree for complex data types

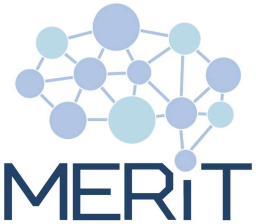
Supports geometric data and full-text search

BRIN

Block Range Index for large tables with ordered data

Low maintenance overhead for time-series data

Index Type Use Cases



B-Tree Indexes

Ideal for: Sorting, filtering with operators like =, >, <, BETWEEN

Example: **WHERE customer_id = 1000** or **WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31'**

Hash Indexes

Perfect for: Exact equality matches, dictionary-like lookups

Example: **WHERE username = 'jsmith'** or **WHERE product_code = 'ABC123'**

GiST Indexes

Specialized for: Geometric data, full-text search, custom data types

Example: **WHERE description @@ 'database & performance'** or **WHERE location && circle '((0,0),1)'**

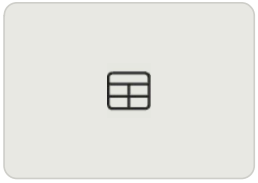
BRIN Indexes

Efficient for: Very large tables with naturally ordered data

Example: **WHERE log_timestamp > '2023-01-01'** on tables with billions of time-ordered rows

Choosing the right index type requires understanding your data access patterns. Always consider the specific queries your application runs most frequently and optimize for those particular patterns.

Query Rewriting Techniques

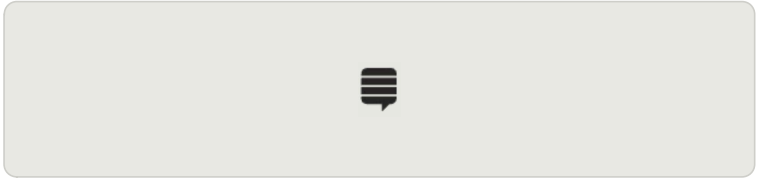


Common Table Expressions (CTEs)

Organize complex queries into manageable, readable blocks

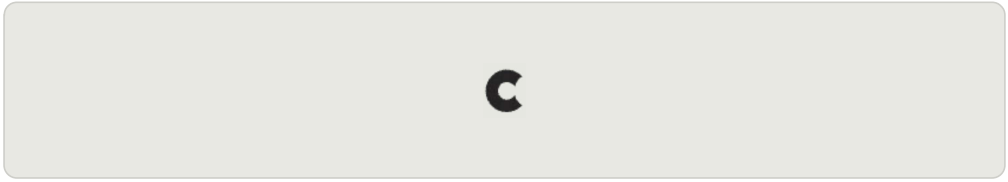
Table Partitioning

Divide large tables by column values for faster access



JOIN vs. Subquery Optimization

Choose the most efficient approach based on data volume



Expression Simplification

Remove unnecessary complexity and redundant operations

Query rewriting often yields significant performance improvements without requiring changes to database structure. By reorganizing the SQL logic, we can help the optimizer generate more efficient execution plans while maintaining the same logical results.

CTEs, Partitioning, Subqueries, and JOINs

CTE
Organize complex queries into manageable, readable blocks

```
WITH sales_employees AS (
  SELECT id, first_name, last_name, email
  FROM employees
  WHERE department = 'Sales'
)
SELECT *
FROM sales_employees
WHERE email LIKE '%@example.com';'
```

Partitioning
Divide large tables by column

```
CREATE TABLE logs (
  id SERIAL,
  event_time DATE,
  message TEXT
) PARTITION BY RANGE (event_time);
CREATE TABLE logs_2024 PARTITION OF logs
  FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
SELECT * FROM logs WHERE event_time = '2024-07-01';
```

SUBQUERY
Choose the most efficient approach

```
SELECT *
FROM employees
WHERE department_id IN (
  SELECT id
  FROM departments
  WHERE location = 'New York'
);
```

JOIN
Choose the most efficient approach

```
SELECT e.*
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE d.location = 'New York';
```

Best Practices for SQL Optimization

Analyze Before Optimizing

Always use EXPLAIN to understand the current execution plan before making changes

Strategic Indexing

Create indexes that match your most common query patterns, but avoid over-indexing

Maintain Statistics

Regularly update database statistics to ensure the optimizer has accurate information

Simplify Queries

Remove unnecessary joins, calculations, and columns from your queries

Continuous Improvement

Revisit query performance as data volumes grow and usage patterns change

Query optimization is an ongoing process rather than a one-time task. As your application evolves and data grows, previously efficient queries may begin to slow down. Establish a regular performance review cycle to maintain optimal database performance.