Query Optimization and Performance
Indexing Strategies, EXPLAIN Plans, Optimizing Window Functions and Aggregates: *Exercises*

Sequential SQL

May 21, 2025

Contents

1 Indexing Strategies

1.1 Dataset for Indexing Strategies

```sql
-- Drop tables if they exist to ensure a clean setup
DROP TABLE IF EXISTS EmployeeProjects CASCADE;
DROP TABLE IF EXISTS Employees CASCADE;
DROP TABLE IF EXISTS Projects CASCADE;
DROP TABLE IF EXISTS Departments CASCADE;

-- Create Departments Table
CREATE TABLE Departments (
    departmentId SERIAL PRIMARY KEY,
    departmentName VARCHAR(100) NOT NULL UNIQUE,
    location VARCHAR(100)
);

-- Create Projects Table
CREATE TABLE Projects (
    projectId SERIAL PRIMARY KEY,
    projectName VARCHAR(150) NOT NULL UNIQUE,
    startDate DATE,
    endDate DATE,
    projectDescription TEXT -- For GIN/GiST index example
);

-- Create Employees Table
CREATE TABLE Employees (
    employeeId SERIAL PRIMARY KEY,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE, -- Automatically indexed due to
    UNIQUE constraint
    departmentId INT,
    salary NUMERIC(10, 2),
    hireDate DATE,
    jobTitle VARCHAR(100),
    performanceScore REAL,
    status VARCHAR(20) DEFAULT 'Active', -- Low cardinality column
    example
    CONSTRAINT fkDepartment FOREIGN KEY (departmentId) REFERENCES
    Departments(departmentId)
);

-- Create EmployeeProjects Table (Junction Table)
CREATE TABLE EmployeeProjects (
    employeeProjectId SERIAL PRIMARY KEY,
    employeeId INT,
    projectId INT,
    roleInProject VARCHAR(100),
    CONSTRAINT fkEmployee FOREIGN KEY (employeeId) REFERENCES Employees
    (employeeId) ON DELETE CASCADE,
    CONSTRAINT fkProject FOREIGN KEY (projectId) REFERENCES Projects(
    projectId) ON DELETE CASCADE,
    UNIQUE (employeeId, projectId)
);

```

```sql
-- Populate Departments
INSERT INTO Departments (departmentName, location) VALUES
('Human Resources', 'Building A, Floor 1'), ('Engineering', 'Building B, Floor 2'),
('Marketing', 'Building A, Floor 2'), ('Sales', 'Building C, Floor 1'),
('Research and Development', 'Building D, Floor 3'), ('Customer Support', 'Building B, Floor 1'),
('Finance', 'Building A, Floor 3'), ('IT Operations', 'Building D, Floor 1'),
('Legal', 'Building A, Floor 4'), ('Product Management', 'Building B, Floor 3');

-- Populate Projects
INSERT INTO Projects (projectName, startDate, endDate, projectDescription)
SELECT
    'Project Alpha ' || i,
    CURRENT_DATE - (RANDOM() * 365)::INT,
    CURRENT_DATE + (RANDOM() * 730)::INT,
    'Detailed description for Project Alpha ' || i || '. Focuses on innovation and market disruption. Keywords: agile, development, beta, release.'
FROM generate_series(1, 25) s(i);
INSERT INTO Projects (projectName, startDate, endDate, projectDescription)
SELECT
    'Project Omega ' || i,
    CURRENT_DATE - (RANDOM() * 100)::INT,
    CURRENT_DATE + (RANDOM() * 200)::INT,
    'Strategic initiative for Project Omega ' || i || '. Aims to optimize core business processes. Keywords: optimization, strategy, core, efficiency.'
FROM generate_series(1, 25) s(i);
UPDATE Projects SET projectDescription = projectDescription || ' Contains sensitive data about future plans.' WHERE projectId % 7 = 0;
UPDATE Projects SET projectDescription = projectDescription || ' This project is critical for Q3 targets.' WHERE projectId % 5 = 0;

-- Populate Employees (e.g., 30,000 employees for noticeable performance differences)
INSERT INTO Employees (firstName, lastName, email, departmentId, salary, hireDate, jobTitle, performanceScore, status)
SELECT
    'FirstName' || i,
    'LastName' || (i % 2000), -- Creates some duplicate last names
    'user' || i || '@example.com',
    (i % 10) + 1,
    30000 + (RANDOM() * 90000)::INT,
    CURRENT_DATE - (RANDOM() * 365 * 15)::INT, -- Hired in the last 15 years
    CASE (i % 6)
        WHEN 0 THEN 'Software Engineer' WHEN 1 THEN 'Product Manager' WHEN 2 THEN 'Sales Representative'
        WHEN 3 THEN 'HR Specialist' WHEN 4 THEN 'Data Analyst' ELSE 'Support Technician'
    END,
    ROUND((1 + RANDOM() * 4)::NUMERIC, 1),
```

```sql
    CASE WHEN RANDOM() < 0.1 THEN 'Terminated' ELSE 'Active' END -- ~10% Terminated
FROM generate_series(1, 30000) s(i);

-- Populate EmployeeProjects
INSERT INTO EmployeeProjects (employeeId, projectId, roleInProject)
SELECT
    e.employeeId,
    p.projectId,
    CASE (p.projectId % 4)
        WHEN 0 THEN 'Developer' WHEN 1 THEN 'Team Lead'
        WHEN 2 THEN 'QA Engineer' ELSE 'Consultant'
    END
FROM Employees e
CROSS JOIN LATERAL (
    SELECT projectId FROM Projects ORDER BY RANDOM() LIMIT (1 + (RANDOM() * 3)::INT) -- 1 to 4 projects
) p
ON CONFLICT (employeeId, projectId) DO NOTHING;

-- Initial recommended indexes for exercises (some intentionally omitted for specific questions)
CREATE INDEX IF NOT EXISTS idxEmployeesLastName ON Employees (lastName);
CREATE INDEX IF NOT EXISTS idxEmployeesDepartmentId ON Employees (departmentId);
CREATE INDEX IF NOT EXISTS idxEmployeesHireDate ON Employees (hireDate);
```

Listing 1: PostgreSQL Dataset for Indexing Strategies

1.2 Exercises for Indexing Strategies

1.2.1 Exercise IS-1 (Meaning, Values, Advantages of B-tree Indexes)

Problem: The HR department frequently searches for employees by their exact `jobTitle`. Currently, this search is slow.

1. Write a query to find all employees with `jobTitle = 'Data Analyst'`.
2. Use `EXPLAIN ANALYZE` to observe its performance and note the scan type on `Employees`.
3. Create an appropriate B-tree index on the `jobTitle` column.
4. Re-run `EXPLAIN ANALYZE` on the same query. Describe the change in the execution plan (e.g., scan type) and explain why the B-tree index provides an advantage here.

1.2.2 Exercise IS-2 (Disadvantages of Indexes / When B-tree Indexes are Not Optimal)

Problem:

1. **Write Overhead:** You are adding 5,000 new employee records. If the `Employees` table has 10 indexes versus just 2 indexes, describe qualitatively how the `INSERT` performance would differ and why. What is the disadvantage being illustrated?

2. **Very Low Selectivity / Small Table:** The `Departments` table is small (10 rows). You want to find departments in 'Building A, Floor 1'. Create an index on `location`. Query for it and use `EXPLAIN ANALYZE`. Does the optimizer use the index? Why might it choose a Seq Scan even if an index exists on a very small table or for a very common value?

3. **Leading Wildcard `LIKE`:** You need to find employees whose `email` address *contains* 'user123'. An index exists on `email` (due to UNIQUE constraint). Write the query using `LIKE '%user123%'`. Use `EXPLAIN ANALYZE`. Does it use the B-tree index on `email` effectively for this pattern? Why or why not?

1.2.3 Exercise IS-3 (Inefficient Alternatives / GIN & GiST Indexes for Full-Text Search)

Problem: The company wants to search `projectDescription` for projects mentioning "innovation" and "strategy". A naive SQL approach might use multiple `LIKE` clauses.

1. Write a query using `projectDescription LIKE '%innovation%' AND projectDescription LIKE '%strategy%'`. Run `EXPLAIN ANALYZE`. Note its inefficiency.

2. Create a GIN index on `projectDescription` using `to_tsvector`.

3. Rewrite the query to use the GIN index with full-text search operators (e.g., `@@` and `to_tsquery`) to find projects containing both "innovation" AND "strategy".

4. Run `EXPLAIN ANALYZE` on the FTS query. Compare plan and performance. Briefly explain GIN's advantage for this type of search over B-trees and multiple LIKEs.

1.2.4 Exercise IS-4 (Hardcore Problem - Comprehensive Indexing Strategy for Complex Reporting Query)

Problem: Generate a report of all 'Active' employees in the 'Engineering' or 'Product Management' departments, hired between Jan 1, 2015, and Dec 31, 2020, with a `performanceScore` of 3.5 or higher. For these employees, list their full name, job title, department name, hire date, and the number of projects they are currently assigned to. Order the result by department name, then by number of projects (descending), then by hire date (most recent first).

1. Write the SQL query to generate this report. Use a CTE for clarity if it helps.

2. Analyze the query and list all columns from `Employees`, `Departments`, and `EmployeeProjects` that are involved in `WHERE` clauses, `JOIN` conditions, or `ORDER BY` clauses. These are candidates for indexing.

3. Propose a set of single-column B-tree indexes that would optimize this query. Explain your choices for each index. (Assume standard PK/FK indexes exist for join keys like `employeeId`, `departmentId`).

4. Create these indexes. Then run `EXPLAIN ANALYZE` on your query. Conceptually, describe how the plan might look with these indexes (e.g., types of scans, joins, and how filters are applied).

Previous Concepts Used: `SELECT`, `FROM`, `JOIN` (INNER, LEFT), `WHERE` (AND, OR, BETWEEN, ¿=), `GROUP BY`, `COUNT`, `ORDER BY` (multiple columns, DESC), `CTEs`, `Date Functions`.

1.2.5 Exercise IS-5 (BRIN Indexes for Time-Series Data)

Problem: The `Projects` table has grown significantly, and many queries filter projects by `startDate` to focus on recent projects (e.g., started after 2023). Due to the table's size and the sequential nature of `startDate`, a BRIN index could be more efficient than a B-tree.

1. Write a query to find all projects with `startDate > '2023-01-01'`.
2. Run `EXPLAIN ANALYZE` to observe the current performance and scan type.
3. Create a BRIN index on `startDate` in the `Projects` table.
4. Re-run `EXPLAIN ANALYZE` on the query. Compare the execution plan and performance. Explain why a BRIN index is advantageous for this scenario, considering the sequential nature of `startDate`.

1.2.6 Exercise IS-6 (Hash Indexes and Advanced Options for Equality Lookups)

Problem: The company frequently searches for employees by their exact `email` address for login verification. The `email` column already has a B-tree index (due to the `UNIQUE` constraint), but you want to test a Hash index for faster equality lookups and explore non-disruptive index creation.

1. Write a query to find an employee by `email = 'user100@example.com'`.
2. Run `EXPLAIN ANALYZE` to confirm the B-tree index is used.
3. Drop the existing `UNIQUE` constraint on `email` (to allow a new index), then create a Hash index on `email` using the `CONCURRENTLY` option to avoid locking the table. Restore the `UNIQUE` constraint afterward.
4. Re-run `EXPLAIN ANALYZE`. Does the planner use the Hash index? Explain why a Hash index might be more efficient for exact equality lookups compared to a B-tree, and why `CONCURRENTLY` is useful in a production environment.

1.2.7 Exercise IS-7 (Full-Text Search with Stored tsvector and Covering Indexes)

Problem: The project management team needs to frequently search `projectDescription` for keywords like "agile" and "release" and retrieve the `projectName` and `startDate` without accessing the table. You decide to use a stored `tsvector` column and a covering index to optimize performance.

1. Alter the `Projects` table to add a generated `tsvector` column for `projectDescription`.
2. Create a GIN index on the `tsvector` column, using the `INCLUDE` clause to cover `projectName` and `startDate`.

3. Write a query to search for projects containing both "agile" and "release" in `projectDescription`, selecting only `projectName` and `startDate`.

4. Run `EXPLAIN ANALYZE` to confirm an `Index Only Scan` is used. Explain how the stored `tsvector` and covering index improve performance compared to a regular GIN index on `to_tsvector('english', projectDescription)`.

2 EXPLAIN Plans

2.1 Dataset for EXPLAIN Plans

The exercises in this section use the same dataset as defined in Section 1.1 (Dataset for Indexing Strategies, see Listing 1 on page 3).

2.2 Exercises for EXPLAIN Plans

2.2.1 Exercise EP-1 (Meaning, Values of EXPLAIN - Basic Scan & Join Types)

Problem: You want to list employees from the 'Sales' department and their `jobTitle`.

1. Write a query joining `Employees` and `Departments` to achieve this.

2. Use `EXPLAIN` (not `ANALYZE` yet). Identify:

   - The scan type on `Employees` (e.g., Seq Scan, Index Scan).
   - The scan type on `Departments`.
   - The join type (e.g., Nested Loop, Hash Join, Merge Join).

3. What do "cost", "rows", and "width" represent in the `EXPLAIN` output for a node?

2.2.2 Exercise EP-2 (Disadvantages/Misinterpretations of EXPLAIN - Stale Statistics & Actual Time)

Problem: `EXPLAIN` provides estimates. `EXPLAIN ANALYZE` provides actuals.

1. Consider a query: `SELECT * FROM Employees WHERE salary > 150000;` (A high, rare salary).

2. Run `EXPLAIN` on this query. Note the estimated `rows`.

3. Now, run the following `INSERT` statement:

```sql
INSERT INTO Employees (firstName, lastName, email, departmentId,
    salary, hireDate, jobTitle, performanceScore, status)
VALUES ('High', 'Earner', 'high.earner@example.com',
        (SELECT departmentId FROM Departments WHERE departmentName
    = 'Finance' LIMIT 1),
        200000.00, CURRENT_DATE, 'CFO', 5.0, 'Active');
```

   Do NOT run `ANALYZE Employees;` yet.

4. Re-run `EXPLAIN` on the same query from step 1. Does the estimated `rows` change significantly? Why or why not? This illustrates a disadvantage of relying solely on `EXPLAIN` with potentially stale statistics.

5. Run `EXPLAIN ANALYZE` on the query from step 1. Compare `actual time` for nodes vs. estimated `cost`. Compare `actual rows` vs. estimated `rows`. What's the key value `ANALYZE` adds?

2.2.3 Exercise EP-3 (Inefficient Alternatives & EXPLAIN for Correlated Subqueries vs. JOINs)

Problem: You need to list each employee and the name of their project if they are working on 'Project Alpha 1'. A common inefficient way is a correlated subquery in the `SELECT` list.

1. Write this query using such a correlated subquery to fetch `projectName`. Filter for employees on 'Project Alpha 1'.

2. Run `EXPLAIN ANALYZE`. Observe the plan, especially how often the subquery might be executed (implied by loops and costs).

3. Rewrite using a `LEFT JOIN` to `EmployeeProjects` and `Projects`.

4. Run `EXPLAIN ANALYZE` on the `JOIN` version. Compare plan (e.g., join types, scan costs) and total `actual execution time`. Why is the `JOIN` generally better?

2.2.4 Exercise EP-4 (Hardcore Problem - Analyzing and Suggesting Improvements for Complex Query Plan)

Problem: A query is written to find departments where the average salary of 'Software Engineer' employees hired after Jan 1, 2018, exceeds $75,000. The query also lists the count of such engineers in those departments.

```sql
SELECT
    d.departmentName,
    COUNT(e.employeeId) as numEngineers,
    AVG(e.salary) as avgSalary
FROM
    Departments d
JOIN
    Employees e ON d.departmentId = e.departmentId
WHERE
    e.jobTitle = 'Software Engineer'
    AND e.hireDate > '2018-01-01'
GROUP BY
    d.departmentId, d.departmentName
HAVING
    AVG(e.salary) > 75000
ORDER BY
    avgSalary DESC;
```

1. Run `EXPLAIN (ANALYZE, BUFFERS)` on this query.

2. Identify the most time-consuming operations (nodes with high `actual total time`).

3. Check for discrepancies between estimated rows (`rows`) and `actual rows` in key filter or join nodes. What might this indicate?

4. Look at `Buffers: shared hit=... read=...`. What does a high `read` count suggest for a particular table scan?

5. Based on the plan, suggest **two distinct potential improvements**. These could be adding a specific type of index (single/composite), rewriting part of the query, or an environment tweak (like `work_mem` if a sort/hash is spilling to disk). Explain why your suggestions might help.

Previous Concepts Used: `SELECT`, `FROM`, `JOIN`, `WHERE` (AND, ¿), `GROUP BY`, `HAVING`, `AVG`, `COUNT`, `ORDER BY DESC`, Date comparisons.

3 Optimizing Window Functions and Aggregates

3.1 Dataset for Optimizing Window Functions and Aggregates

```sql
DROP TABLE IF EXISTS SalesTransactions CASCADE;
DROP TABLE IF EXISTS Products CASCADE;
DROP TABLE IF EXISTS Customers CASCADE;
DROP TABLE IF EXISTS Regions CASCADE;

CREATE TABLE Regions (
    regionId SERIAL PRIMARY KEY,
    regionName VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE Customers (
    customerId SERIAL PRIMARY KEY,
    customerName VARCHAR(150) NOT NULL,
    regionId INT,
    joinDate DATE,
    CONSTRAINT fkRegion FOREIGN KEY (regionId) REFERENCES Regions(regionId)
);

CREATE TABLE Products (
    productId SERIAL PRIMARY KEY,
    productName VARCHAR(100) NOT NULL,
    category VARCHAR(50),
    launchDate DATE
);

CREATE TABLE SalesTransactions (
    transactionId BIGSERIAL PRIMARY KEY,
    productId INT NOT NULL,
    customerId INT NOT NULL,
    transactionDate TIMESTAMP NOT NULL,
    quantitySold INT NOT NULL,
    unitPrice NUMERIC(10, 2) NOT NULL,
    totalAmount NUMERIC(12, 2) NOT NULL,
    CONSTRAINT fkProduct FOREIGN KEY (productId) REFERENCES Products(productId),
```

```sql
    CONSTRAINT fkCustomer FOREIGN KEY (customerId) REFERENCES Customers
   (customerId)
);

-- Populate Regions
INSERT INTO Regions (regionName) VALUES ('North'), ('South'), ('East'),
    ('West'), ('Central');

-- Populate Customers (2,000 customers)
INSERT INTO Customers (customerName, regionId, joinDate)
SELECT
    'Customer ' || i,
    (i % 5) + 1,
    CURRENT_DATE - (RANDOM() * 1000)::INT
FROM generate_series(1, 2000) s(i);

-- Populate Products (200 products, 10 categories)
INSERT INTO Products (productName, category, launchDate)
SELECT
    'Product ' || i,
    'Category ' || ((i % 10) + 1),
    CURRENT_DATE - (RANDOM() * 700)::INT
FROM generate_series(1, 200) s(i);

-- Populate SalesTransactions (e.g., 1,500,000 rows for significant
   window function workload)
INSERT INTO SalesTransactions (productId, customerId, transactionDate,
   quantitySold, unitPrice, totalAmount)
SELECT
    (RANDOM() * 199)::INT + 1 AS prodId,
    (RANDOM() * 1999)::INT + 1 AS custId,
    TIMESTAMP '2021-01-01 00:00:00' +
        make_interval(days => (RANDOM() * 365 * 2.5)::INT, hours => (
   RANDOM()*23)::INT, mins => (RANDOM() * 59)::INT),
    (RANDOM() * 5)::INT + 1 AS qty,
    ROUND((RANDOM() * 150 + 10)::NUMERIC, 2) AS price,
    0
FROM generate_series(1, 1500000) s(i);

UPDATE SalesTransactions SET totalAmount = quantitySold * unitPrice;

-- Indexes for optimization examples
CREATE INDEX IF NOT EXISTS idxSalesTransactionsDate ON
   SalesTransactions (transactionDate);
CREATE INDEX IF NOT EXISTS idxSalesTransactionsProdCustDate ON
   SalesTransactions (productId, customerId, transactionDate);
CREATE INDEX IF NOT EXISTS idxSalesTransactionsCustDate ON
   SalesTransactions (customerId, transactionDate);
CREATE INDEX IF NOT EXISTS idxProductsCategory ON Products (category);
CREATE INDEX IF NOT EXISTS idxCustomersRegionId ON Customers (regionId)
   ;
```

Listing 2: PostgreSQL Dataset for Optimizing Window Functions and Aggregates

3.2 Exercises for Optimizing Window Functions and Aggregates

3.2.1 Exercise OWA-1 (Meaning, Values, Advantages of Window Functions - Contextual Aggregation)

Problem: For each sale transaction, you want to display its `totalAmount` alongside the average `totalAmount` of all transactions made by that same `customerId`.

1. Write a query using a window function `AVG(...) OVER (PARTITION BY ...)` to achieve this efficiently. Select a few columns for readability and `LIMIT` the result.

2. Explain the "value" or "advantage" of using a window function here compared to, for example, a `LEFT JOIN` to a subquery that calculates average sales per customer.

3.2.2 Exercise OWA-2 (Disadvantages/Overhead of Window Functions - Cost of Sorting & Large Partitions)

Problem: You want to calculate, for every sales transaction, its rank based on `totalAmount` across *all* transactions in the entire `SalesTransactions` table (1.5M rows).

1. Write this query using `RANK() OVER (ORDER BY totalAmount DESC)`.

2. Run `EXPLAIN ANALYZE`. Focus on the "WindowAgg" node and any preceding "Sort" node. What is the primary disadvantage highlighted by the cost/time of these operations for such a large, unpartitioned window?

3. If you added `PARTITION BY productId` to the `OVER()` clause, how would that conceptually change the workload and potentially reduce the "disadvantage" observed in step 2 (even if total work is similar, how is it broken down)?

3.2.3 Exercise OWA-3 (Inefficient Alternatives vs. Optimized Approach - Using Window Functions for Running Totals)

Problem: For each customer, you want to see their monthly sales in 2022 and a running total of their sales month by month throughout 2022.

1. **Inefficient Sketch:** Briefly describe how you might achieve the running total *inefficiently* using a correlated subquery for each customer-month, summing up sales from the start of the year up to that month. Why is this approach bad?

2. **Optimized Query:** Write an efficient query. First, use a CTE to aggregate sales per customer per month in 2022. Then, in an outer query, use `SUM(...) OVER (PARTITION BY ... ORDER BY ...)` to calculate the running total.

3. What indexes on `SalesTransactions` and `Customers` would be most beneficial for the aggregation part (the CTE)?

3.2.4 Exercise OWA-4 (Hardcore Problem - Complex Analytics with Optimized Window Functions and Aggregates)

Problem: Management wants a detailed sales report for the year 2022. For each `Product Category` and `Region`:

1. Calculate the total sales amount for that category in that region for 2022.
2. Calculate the rank of this category-region combination based on its total sales, compared to all other category-region combinations in 2022.
3. For each category-region, also show its percentage contribution to the total sales of its `Region` in 2022.
4. For each category-region, show its percentage contribution to the total sales of its `Product Category` across all regions in 2022.

Filter the final result to show only combinations where the category-region total sales amount is greater than $10,000. Order by the overall rank.

Previous Concepts Used: `CTEs`, `Joins` (multiple), `Aggregate Functions` (SUM), `Window Functions` (RANK, SUM OVER for percentages), `Date Functions` (filtering by year), `GROUP BY` (multiple columns), `Arithmetic` for percentages, `Filtering` (HAVING or WHERE on CTE).