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

Sequential SQL

May 21, 2025

Contents

1 Solutions for Indexing Strategies

1.1 Solution for Exercise IS-1

1. Query:

```sql
SELECT employeeId, firstName, lastName, jobTitle FROM Employees
    WHERE jobTitle = 'Data Analyst';
```

2. Observe performance (before specific index on `jobTitle`):

```sql
EXPLAIN ANALYZE SELECT employeeId, firstName, lastName, jobTitle
    FROM Employees WHERE jobTitle = 'Data Analyst';
-- Expected: Seq Scan on Employees if no specific index on jobTitle
     exists or is chosen.
```

3. Create B-tree index:

```sql
CREATE INDEX idxEmployeesJobTitle ON Employees (jobTitle);
```

4. Re-run and observe improvement:

```sql
EXPLAIN ANALYZE SELECT employeeId, firstName, lastName, jobTitle
    FROM Employees WHERE jobTitle = 'Data Analyst';
-- Expected: Index Scan or Bitmap Heap Scan using
    idxEmployeesJobTitle.
```

Description: The plan changes from a Sequential Scan (reading all rows) to an Index-based scan. The B-tree index on `jobTitle` allows the database to quickly locate an ordered list of pointers to rows where `jobTitle` is 'Data Analyst', significantly reducing I/O and CPU by avoiding a full table scan. This is advantageous for equality predicates on moderately to highly selective columns.

1.2 Solution for Exercise IS-2

1. **Write Overhead Description:** With 10 indexes, each `INSERT` operation would require updating all 10 index structures in addition to writing the row to the table itself. With only 2 indexes, only those 2 index structures need updating. Each index update involves finding the correct position in the B-tree (or other structure) and inserting the new key/pointer, potentially causing page splits and other maintenance. Therefore, `INSERT` performance would be noticeably slower with 10 indexes compared to 2 indexes. **Disadvantage Illustrated:** Indexes add overhead to DML operations (INSERT, UPDATE, DELETE). Over-indexing can significantly degrade write performance.

2. **Very Low Selectivity / Small Table:**

```sql
CREATE INDEX IF NOT EXISTS idxDepartmentsLocation ON Departments (
    location);
EXPLAIN ANALYZE SELECT departmentName FROM Departments WHERE
    location = 'Building A, Floor 1';
```

Observation & Explanation: The optimizer might still choose a `Seq Scan`. For very small tables, the cost of reading the entire table (which might be just one or a few data blocks) can be less than the overhead of an index scan (reading index blocks then data blocks via pointers). If 'Building A, Floor 1' is very common (low selectivity), fetching many rows via an index (random I/Os) can be costlier than one sequential scan. The optimizer compares estimated costs.

3. **Leading Wildcard `LIKE`:**

```sql
EXPLAIN ANALYZE SELECT employeeId, email FROM Employees WHERE email
    LIKE '%user123%';
```

Observation & Explanation: The B-tree index on `email` will likely **not** be used effectively (or at all) for the `WHERE email LIKE '%user123%'` condition, resulting in a `Seq Scan`. B-tree indexes are structured based on the prefix of the indexed values. A leading wildcard (`%`) means the beginning of the string is unknown, so the database cannot use the B-tree's ordered structure to efficiently narrow down the search. It must check all values. (Note: For `email LIKE 'user123%'`, the index would be effective).

1.3 Solution for Exercise IS-3

1. Naive LIKE query:

```sql
EXPLAIN ANALYZE SELECT projectId, projectName FROM Projects
WHERE projectDescription LIKE '%innovation%' AND projectDescription
    LIKE '%strategy%';
-- Expected: Seq Scan on Projects, slow due to full text scanning
   for each LIKE.
```

2. Create GIN index:

```sql
DROP INDEX IF EXISTS idxGinProjectDescription;
CREATE INDEX idxGinProjectDescription ON Projects USING GIN (
   to_tsvector('english', projectDescription));
```

3. FTS query:

```sql
EXPLAIN ANALYZE SELECT projectId, projectName FROM Projects
WHERE to_tsvector('english', projectDescription) @@ to_tsquery('
   english', 'innovation & strategy');
```

4. **Comparison and GIN Advantage:** The FTS query using the GIN index will be significantly faster and use a plan like Bitmap Heap Scan on `Projects` with a Bitmap Index Scan on `idxGinProjectDescription`. GIN (Generalized Inverted Index) is designed for indexing composite values where elements within the value are of interest (like words in text). `to_tsvector` breaks text into lexemes. The GIN index stores these lexemes and lists of documents they appear in. **Advantage:** Instead of scanning all text, FTS quickly finds documents containing "innovation" and documents containing "strategy" using the index, then finds the intersection. This is vastly more efficient than B-trees (unsuitable for `'%text%'`) or repeated full string scans with LIKE.

1.4 Solution for Exercise IS-4

1. SQL Query:

```sql
WITH EligibleEmployees AS (
    SELECT
        e.employeeId,
        e.firstName,
        e.lastName,
        e.jobTitle,
        e.hireDate,
        e.departmentId
    FROM
        Employees e
    WHERE
        e.status = 'Active'
        AND e.departmentId IN (SELECT departmentId FROM Departments WHERE departmentName IN ('Engineering', 'Product Management'))
        AND e.hireDate BETWEEN '2015-01-01' AND '2020-12-31'
        AND e.performanceScore >= 3.5
),
EmployeeProjectCounts AS (
    SELECT
        ee.employeeId,
        ee.firstName,
        ee.lastName,
        ee.jobTitle,
        ee.hireDate,
        d.departmentName,
        COUNT(ep.projectId) AS projectCount
    FROM
        EligibleEmployees ee
    JOIN
        Departments d ON ee.departmentId = d.departmentId
    LEFT JOIN
        EmployeeProjects ep ON ee.employeeId = ep.employeeId
    GROUP BY
        ee.employeeId, ee.firstName, ee.lastName, ee.jobTitle, ee.hireDate, d.departmentName
)
SELECT
    firstName,
    lastName,
    jobTitle,
    departmentName,
    hireDate,
    projectCount
FROM
    EmployeeProjectCounts
ORDER BY
    departmentName ASC, projectCount DESC, hireDate DESC;
```

2. Candidate Columns for Indexing:

    - `Employees` table: `status` (WHERE), `departmentId` (WHERE, JOIN), `hireDate` (WHERE, ORDER BY), `performanceScore` (WHERE), `employeeId` (JOIN, GROUP BY - PK).
    - `Departments` table: `departmentId` (JOIN - PK), `departmentName` (WHERE subquery, ORDER BY - UNIQUE).
    - `EmployeeProjects` table: `employeeId` (JOIN, COUNT), `projectId` (COUNT).

3. Proposed B-tree Indexes & Choices:
   - On `Employees`:
     - `CREATE INDEX IF NOT EXISTS idxEmployeesStatus ON Employees (status);`
     - `CREATE INDEX IF NOT EXISTS idxEmployeesPerformanceScore ON Employees (performanceScore);`
     - (Ensure `idxEmployeesHireDate ON Employees(hireDate)` exists)
     - (Ensure `idxEmployeesDepartmentId ON Employees(departmentId)` exists)
   - On `Departments`: (PK `departmentId` and UNIQUE `departmentName` are already indexed.)
   - On `EmployeeProjects`:
     - `CREATE INDEX IF NOT EXISTS idxEmployeeProjectsEmpId ON EmployeeProjects (employeeId);`
4. Create indexes and Analyze:

```sql
CREATE INDEX IF NOT EXISTS idxEmployeesStatus ON Employees (status)
    ;
CREATE INDEX IF NOT EXISTS idxEmployeesPerformanceScore ON
    Employees (performanceScore);
CREATE INDEX IF NOT EXISTS idxEmployeeProjectsEmpId ON
    EmployeeProjects (employeeId);

EXPLAIN ANALYZE
WITH EligibleEmployees AS ( /* ...as above... */ ),
EmployeeProjectCounts AS ( /* ...as above... */ )
SELECT firstName, lastName, jobTitle, departmentName, hireDate,
    projectCount
FROM EmployeeProjectCounts
ORDER BY departmentName ASC, projectCount DESC, hireDate DESC;
```

   Conceptual Plan Description:

   - Subquery for department IDs: Fast Index Scan on `Departments.departmentName`.
   - `EligibleEmployees` CTE: Likely a BitmapAnd/BitmapOr operation combining results from multiple index scans on `Employees` (on `status`, `departmentId`, `hireDate`, `performanceScore`), followed by a Bitmap Heap Scan.
   - `EmployeeProjectCounts` CTE: Join to `Departments`, Left Join to `EmployeeProjects` (using `idxEmployeeProjectsEmpId`), then HashAggregate.
   - Final SELECT: A Sort operation.

   The indexes significantly reduce rows scanned from `Employees`, making filtering efficient. The index on `EmployeeProjects.employeeId` speeds up counting projects.

1.5 Solution for Exercise IS-5

1. Query:

```sql
SELECT projectId, projectName, startDate FROM Projects WHERE
    startDate > '2023-01-01';
```

2. Observe performance (before BRIN index):

```sql
EXPLAIN ANALYZE SELECT projectId, projectName, startDate FROM Projects WHERE startDate > '2023-01-01';
-- Expected: Seq Scan on Projects, as no specific index on startDate exists in the initial setup.
```

3. Create BRIN index:

```sql
CREATE INDEX idxProjectsStartDateBrin ON Projects USING BRIN (startDate);
```

4. Re-run and observe improvement:

```sql
EXPLAIN ANALYZE SELECT projectId, projectName, startDate FROM Projects WHERE startDate > '2023-01-01';
-- Expected: Bitmap Heap Scan on Projects with Bitmap Index Scan on idxProjectsStartDateBrin.
```

Description: The plan shifts from a Seq Scan to a Bitmap Heap Scan using the BRIN index. BRIN (Block Range Index) is ideal for large tables with naturally ordered columns like `startDate`, which tends to be sequential due to project creation over time. BRIN stores summaries of data ranges (min/max values per block), allowing the database to skip irrelevant blocks. This reduces I/O significantly compared to scanning the entire table, especially for range queries on large datasets. BRIN's small size and low maintenance overhead make it advantageous over B-tree indexes for such scenarios, though it's less precise and best for sequential data.

1.6 Solution for Exercise IS-6

1. Query:

```sql
SELECT employeeId, firstName, lastName, email FROM Employees WHERE email = 'user100@example.com';
```

2. Observe performance (with B-tree index):

```sql
EXPLAIN ANALYZE SELECT employeeId, firstName, lastName, email FROM Employees WHERE email = 'user100@example.com';
-- Expected: Index Scan using the UNIQUE B-tree index on email (from UNIQUE constraint).
```

3. Drop UNIQUE constraint, create Hash index, and restore constraint:

```sql
-- Drop the UNIQUE constraint (which drops its B-tree index)
ALTER TABLE Employees DROP CONSTRAINT employees_email_key;

-- Create Hash index CONCURRENTLY
CREATE INDEX CONCURRENTLY idxEmployeesEmailHash ON Employees USING HASH (email);

-- Restore UNIQUE constraint (creates a new B-tree index for constraint enforcement)
ALTER TABLE Employees ADD CONSTRAINT employees_email_key UNIQUE (email);
```

4. Re-run and analyze:

```sql
EXPLAIN ANALYZE SELECT employeeId, firstName, lastName, email FROM
    Employees WHERE email = 'user100@example.com';
-- Expected: Likely Index Scan using the UNIQUE B-tree index (from
    restored constraint), not the Hash index.
```

Explanation: The query planner may prefer the B-tree index created by the `UNIQUE` constraint over the Hash index, as B-trees are versatile and reliable for equality lookups, and the constraint ensures uniqueness. However, Hash indexes are optimized specifically for equality comparisons (`=`), storing hashed values for faster lookups with a smaller footprint than B-trees. They lack support for range queries or sorting, but for exact matches, they can be slightly faster due to simpler structure. Using `CONCURRENTLY` is critical in production, as it allows index creation without locking the table, preventing downtime during writes. The planner's choice depends on cost estimates, but Hash indexes shine in high-read, equality-heavy workloads.

1.7 Solution for Exercise IS-7

1. Add generated `tsvector` column:

```sql
ALTER TABLE Projects
ADD COLUMN descriptionTsv TSVECTOR
GENERATED ALWAYS AS (to_tsvector('english', projectDescription))
    STORED;
```

2. Create GIN index with INCLUDE:

```sql
CREATE INDEX idxProjectsDescriptionTsvGin ON Projects USING GIN (
    descriptionTsv) INCLUDE (projectName, startDate);
```

3. Query for full-text search:

```sql
SELECT projectName, startDate FROM Projects
WHERE descriptionTsv @@ to_tsquery('english', 'agile & release');
```

4. Analyze performance:

```sql
EXPLAIN ANALYZE SELECT projectName, startDate FROM Projects
WHERE descriptionTsv @@ to_tsquery('english', 'agile & release');
-- Expected: Index Only Scan using idxProjectsDescriptionTsvGin.
```

Explanation: The plan uses an `Index Only Scan`, retrieving `projectName` and `startDate` directly from the GIN index without accessing the table heap. The stored `tsvector` column precomputes the lexemes from `projectDescription`, avoiding runtime computation of `to_tsvector`, which speeds up query execution. The GIN index efficiently handles full-text search by mapping lexemes to rows, and the `INCLUDE` clause adds `projectName` and `startDate` to the index, enabling a covering index. This eliminates heap access, reducing I/O and boosting performance compared to a regular GIN index on `to_tsvector('english', projectDescription)`, which would require table access for non-indexed columns.

2 Solutions for EXPLAIN Plans

2.1 Solution for Exercise EP-1

1. Query:

```sql
SELECT
    e.firstName, e.lastName, e.jobTitle, d.departmentName
FROM
    Employees e
JOIN
    Departments d ON e.departmentId = d.departmentId
WHERE
    d.departmentName = 'Sales';
```

2. `EXPLAIN` output analysis:

```sql
EXPLAIN
SELECT
    e.firstName, e.lastName, e.jobTitle, d.departmentName
FROM
    Employees e
JOIN
    Departments d ON e.departmentId = d.departmentId
WHERE
    d.departmentName = 'Sales';
```

   Identify:

   - Scan type on `Employees`: Likely Index Scan on `idxEmployeesDepartmentId`.
   - Scan type on `Departments`: Index Scan on `departmentName` (UNIQUE index).
   - Join type: Likely Hash Join or Nested Loop.

3. "cost", "rows", "width" explanation:

   - `cost=X..Y`: Estimated startup (X) and total (Y) cost in arbitrary units.
   - `rows=N`: Estimated number of rows output by the node.
   - `width=W`: Estimated average row width in bytes.

2.2 Solution for Exercise EP-2

1. Query: `SELECT * FROM Employees WHERE salary > 150000;`

2. `EXPLAIN` before insert:

```sql
EXPLAIN SELECT * FROM Employees WHERE salary > 150000;
```

3. Insert a high earner:

```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');
```

4. Re-run EXPLAIN:

```sql
EXPLAIN SELECT * FROM Employees WHERE salary > 150000;
```

Estimated `rows` won't change significantly without ANALYZE, as statistics are stale. This shows EXPLAIN's reliance on outdated stats.

5. Run EXPLAIN ANALYZE:

```sql
EXPLAIN ANALYZE SELECT * FROM Employees WHERE salary > 150000;
```

Comparison: `actual time` and `actual rows` reflect real execution, revealing discrepancies due to stale stats. ANALYZE provides actual performance data.

2.3 Solution for Exercise EP-3

1. Correlated Subquery:

```sql
EXPLAIN ANALYZE
SELECT
    e.employeeId,
    e.firstName,
    e.lastName,
    (SELECT p.projectName
     FROM Projects p
     JOIN EmployeeProjects epFind ON p.projectId = epFind.projectId
     WHERE epFind.employeeId = e.employeeId AND p.projectName = '
   Project Alpha 1'
     LIMIT 1) AS projectAlphaName
FROM
    Employees e
WHERE EXISTS (
    SELECT 1 FROM EmployeeProjects epChk
    JOIN Projects pChk ON epChk.projectId = pChk.projectId
    WHERE epChk.employeeId = e.employeeId AND pChk.projectName = '
   Project Alpha 1'
);
```

2. Analyze: The subquery runs per row, causing high loop counts and slow performance.

3. Rewrite using JOIN:

```sql
EXPLAIN ANALYZE
SELECT
    e.employeeId, e.firstName, e.lastName, p.projectName
FROM
    Employees e
JOIN
    EmployeeProjects ep ON e.employeeId = ep.employeeId
JOIN
    Projects p ON ep.projectId = p.projectId
WHERE
    p.projectName = 'Project Alpha 1';
```

4. Comparison: The JOIN version uses efficient set-based operations (e.g., Hash Join), scanning tables once, making it faster than row-by-row subquery execution.

2.4 Solution for Exercise EP-4

```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)`:

   ```sql
   EXPLAIN (ANALYZE, BUFFERS) SELECT /* ... as above ... */;
   ```

2. Time-consuming operations: Likely Seq Scan on `Employees` or Sort for `ORDER BY`.

3. Discrepancies: Large differences between estimated and actual rows suggest stale statistics.

4. High `shared read`: Indicates heavy disk I/O, likely from Seq Scan.

5. Improvements:

   - Composite index: `CREATE INDEX idxEmpJobHireDept ON Employees (jobTitle, hireDate, departmentId) INCLUDE (salary);`
   - Increase `work_mem`: `SET LOCAL work_mem = '128MB';` to reduce disk spills.

3 Solutions for Optimizing Window Functions and Aggregates

3.1 Solution for Exercise OWA-1

1. Query:

   ```sql
   SELECT
       st.transactionId,
       st.customerId,
       st.transactionDate,
       st.totalAmount,
       AVG(st.totalAmount) OVER (PARTITION BY st.customerId) AS avgCustomerSalesAmount
   FROM
       SalesTransactions st
   ORDER BY
       st.customerId, st.transactionDate
   LIMIT 100;
   ```

2. Advantage: Window functions are concise, optimized for single-pass processing, and preserve row details unlike `GROUP BY`.

3.2 Solution for Exercise OWA-2

1. Query:

```sql
EXPLAIN ANALYZE
SELECT
    transactionId,
    totalAmount,
    RANK() OVER (ORDER BY totalAmount DESC) AS overallRank
FROM
    SalesTransactions
LIMIT 100;
```

2. Disadvantage: Sorting 1.5M rows is resource-intensive, risking disk spills.

3. With `PARTITION BY productId`: Smaller partitions reduce sort overhead and enable parallelism.

3.3 Solution for Exercise OWA-3

1. Inefficient Sketch: Correlated subquery per customer-month is slow due to repeated scans.

2. Optimized Query:

```sql
WITH CustomerMonthlySales2022 AS (
    SELECT
        st.customerId,
        DATE_TRUNC('month', st.transactionDate) AS saleMonth,
        SUM(st.totalAmount) AS monthlySales
    FROM
        SalesTransactions st
    WHERE
        st.transactionDate >= '2022-01-01' AND st.transactionDate < '2023-01-01'
    GROUP BY
        st.customerId, DATE_TRUNC('month', st.transactionDate)
)
SELECT
    cms.customerId,
    c.customerName,
    TO_CHAR(cms.saleMonth, 'YYYY-MM') AS saleMonthFormatted,
    cms.monthlySales,
    SUM(cms.monthlySales) OVER (PARTITION BY cms.customerId ORDER BY cms.saleMonth ASC
                                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS runningTotalSales
FROM
    CustomerMonthlySales2022 cms
JOIN
    Customers c ON cms.customerId = c.customerId
ORDER BY
    cms.customerId, cms.saleMonth
LIMIT 200;
```

3. Indexes: `CREATE INDEX idxSalesTransDateIncl ON SalesTransactions (transactionDate) INCLUDE (customerId, totalAmount);`

3.4 Solution for Exercise OWA-4

```sql
WITH Sales2022 AS (
    SELECT
        st.productId,
        st.customerId,
        st.totalAmount,
        p.category,
        r.regionName
    FROM
        SalesTransactions st
    JOIN
        Products p ON st.productId = p.productId
    JOIN
        Customers c ON st.customerId = c.customerId
    JOIN
        Regions r ON c.regionId = r.regionId
    WHERE
        st.transactionDate >= '2022-01-01' AND st.transactionDate < '
    2023-01-01'
),
CategoryRegionSales AS (
    SELECT
        category,
        regionName,
        SUM(totalAmount) AS categoryRegionTotalSales
    FROM
        Sales2022
    GROUP BY
        category, regionName
),
RankedAndOverallSales AS (
    SELECT
        category,
        regionName,
        categoryRegionTotalSales,
        RANK() OVER (ORDER BY categoryRegionTotalSales DESC) AS
    overallRank,
        SUM(categoryRegionTotalSales) OVER (PARTITION BY regionName) AS
     totalRegionSales,
        SUM(categoryRegionTotalSales) OVER (PARTITION BY category) AS
    totalCategorySales
    FROM
        CategoryRegionSales
)
SELECT
    category,
    regionName,
    categoryRegionTotalSales,
    overallRank,
    ROUND((categoryRegionTotalSales * 100.0 / totalRegionSales)::
    NUMERIC, 2) AS pctOfRegionSales,
    ROUND((categoryRegionTotalSales * 100.0 / totalCategorySales)::
    NUMERIC, 2) AS pctOfCategorySales
```

```sql
FROM
    RankedAndOverallSales
WHERE
    categoryRegionTotalSales > 10000
ORDER BY
    overallRank;
```