Query Optimization and Performance with PostgreSQL

Sequential SQL

May 23, 2025

Contents

1 Indexing Strategies: Speeding Up Your Data Dash!

An index in a database is like a well-organized library catalog; instead of rummaging through every book, you find exactly what you need in seconds. Indexes are your database's secret weapon for lightning-fast queries, and PostgreSQL offers a rich set of tools to make your data retrieval soar. Let's dive into the world of indexes and unlock their power for high-performance databases!

1.1 What Are They? (Meanings & Values)

1.1.1 Core Meaning

Indexing Strategies are the art and science of creating, using, and managing indexes to optimize database performance. An **index** is a specialized data structure (like a B-tree, GIN, or BRIN) that stores a subset of a table's columns in an optimized format, allowing the database to find rows quickly without scanning the entire table. Think of it as a shortcut that turns a treasure hunt into a quick map lookup!

Indexes enhance the speed of `SELECT` queries, `JOIN`s, `ORDER BY`, and other operations by reducing the amount of data the database needs to process. PostgreSQL's indexes are versatile, supporting everything from simple numeric lookups to complex full-text searches and geometric queries.

1.1.2 Typical Values or Outputs

The primary payoff of indexing is **blazing-fast query performance**. Here's what you gain:

- **Faster Data Retrieval:** Queries that once took ages now return in milliseconds.
- **Reduced I/O Operations:** Fewer disk reads mean less strain on your server.
- **Optimized Query Plans:** Indexes influence the database's execution plan (visible via `EXPLAIN`), often making complex queries simpler and faster.
- **Support for Constraints:** Indexes enforce `UNIQUE` and `PRIMARY KEY` constraints efficiently.

With indexes, your database becomes a sleek race car, not a sluggish cart!

1.1.3 Types of Indexes (PostgreSQL Focus)

PostgreSQL offers a variety of index types, each tailored to specific data and query patterns. Here's the lineup, drawn from PostgreSQL's robust toolkit:

- **B-tree Indexes:** The go-to choice for most scenarios. Perfect for equality (`=`) and range queries (`<`, `>`, `BETWEEN`) on columns with high cardinality (many unique values). B-trees keep data sorted, making them ideal for `INT`, `DECIMAL`, `DATE`, and similar types.
- **GIN (Generalized Inverted Index):** Designed for composite or complex data, like arrays, JSONB, or full-text search. GIN excels at answering "which rows contain this element or keyword?" questions, making it a star for full-text search and JSON queries.

- **GiST (Generalized Search Tree):** A flexible framework for complex data types, such as geometric shapes (with PostGIS) or advanced full-text search. GiST supports "nearest neighbor" searches and custom operator classes.
- **BRIN (Block Range Index):** Lightweight and space-efficient, ideal for very large tables with naturally ordered data (e.g., time-series data like timestamps). BRIN stores summaries of data blocks, making it great for sequential scans on massive datasets.
- **Hash Indexes:** Optimized for equality comparisons (`=`). Smaller and faster than B-trees for simple lookups but less versatile (no range queries). Note: Hash indexes are fully reliable in PostgreSQL 10+.
- **SP-GiST (Space-Partitioned GiST):** Specialized for non-balanced data structures, like quad-trees or k-d trees, useful for spatial data or hierarchical structures.

Each index type is a tool in your performance toolbox—choose wisely based on your data and queries![1]

1.2 Relations: How They Play with Others

Indexes don't work alone—they team up with your SQL queries to deliver results faster than a speeding bullet.

1.2.1 Relations within Indexing Itself

- **Choosing the Right Type:** The data type and query pattern dictate the best index. B-trees love simple comparisons, GIN shines for text search, and BRIN thrives on large, ordered datasets.
- **Single vs. Composite Indexes:** Index one column or multiple (composite). For composite indexes, column order is paramount—see Section 1.3.2.
- **Covering Indexes:** Indexes that include all columns needed for a query (via indexed columns or `INCLUDE`) enable `Index Only Scans`, skipping table access for maximum speed.
- **Partial Indexes:** Indexes on a subset of rows (e.g., `WHERE status = 'Active'`) save space and boost performance for specific queries.

1.2.2 Relations with SQL Concepts

Indexes supercharge these operations:

- **Conditionals: `WHERE` Clause**
  - Indexes on `WHERE` columns (e.g., `WHERE employeeId = 100` or `WHERE salary BETWEEN 50000 AND 70000`) slash query times.

[1] For a deep dive into PostgreSQL index types, see https://www.postgresql.org/docs/current/indexes-types.html and https://www.postgresql.org/docs/current/indexes.html.

  - B-trees work for `=`, `<`, `>`, `BETWEEN`, `IN`, and `LIKE 'prefix%'`. For `LIKE '%text%'`, GIN or GiST with full-text search is your friend.

- **Conditionals: `ORDER BY` Clause**
  - An index matching the `ORDER BY` column and direction (ASC/DESC) can eliminate sorting, especially with `LIMIT`.

- **Joins: `INNER JOIN`, `LEFT JOIN`, etc.**
  - Indexing join columns (e.g., `ON tableA.foreignKey = tableB.primaryKey`) is critical. Primary keys are auto-indexed, but foreign keys often need explicit indexes.

- **Aggregators: `GROUP BY`**
  - Indexes on `GROUP BY` columns can speed up grouping by providing pre-sorted data.

- **Uniqueness and Primary Keys**
  - `PRIMARY KEY` and `UNIQUE` constraints automatically create B-tree indexes.

- **Data Types**
  - Data types influence index choice. `INT` and `DATE` love B-trees, while `TEXT` or `JSONB` may prefer GIN/GiST.

- **Subqueries in `WHERE` (e.g., `IN`, `EXISTS`)**
  - Indexes on subquery columns improve performance, especially for correlated subqueries.

1.3 How to Use Them: Structures & Syntax

Creating an index is like giving your database a GPS for your data. Let's explore the syntax and options, including advanced features from PostgreSQL.

1.3.1 Basic B-tree Index

The workhorse for most queries.

```sql
-- Syntax: CREATE INDEX [indexName] ON tableName (columnName);
CREATE INDEX idxEmployeeLastName ON Employees (lastName);

-- Composite index (order matters!)
CREATE INDEX idxEmployeeDeptJob ON Employees (departmentId, jobTitle);
```

Listing 1: Creating a simple B-tree index

This index helps queries filtering on `departmentId` alone or both `departmentId` and `jobTitle`.

1.3.2 Composite Index Column Order: The Secret Sauce

For composite B-tree indexes (indexes on multiple columns), the order of columns in the index definition is critical for performance. It's not arbitrary!

- **Equality First, Highest Selectivity Up Front:** Columns used in `WHERE` clauses with equality predicates (`=`, `IN`) should come first. Among these, the column that is most selective (filters out the largest proportion of rows, often the one with highest cardinality) should be the very first. This allows the database to narrow down the search space most significantly.
- **Range Predicates Follow Equality:** Columns used with range predicates (`>`, `<`, `BETWEEN`, `LIKE 'prefix%'`) should come after all columns matched with equality. The index can efficiently scan a range on the first column not fixed by an equality predicate.
- **One Range is Key:** An index can effectively use its structure for range conditions on only one column after any equality conditions. Subsequent range filters on other columns in the index are applied as post-filters to the rows retrieved.
- **Sorting/Grouping Columns Last:** If a query filters on leading columns of the index and then sorts or groups by subsequent columns in the same order (and direction) as the index, a separate sort step might be avoided.

Think of it like a phone book: ordered by `(LastName, FirstName)`. You first look up "Smith" (high selectivity for last name), then "John" within the Smiths. The reverse (`(FirstName, LastName)`) would be far less efficient for finding "John Smith."

1.3.3 GIN Index for Full-Text Search

Perfect for searching text-heavy data, like document descriptions.

```sql
-- Index for full-text search using to_tsvector
CREATE INDEX idxProjectDescriptionFTS ON Projects USING GIN (
    to_tsvector('english', projectDescription));

-- Query example
SELECT projectName FROM Projects
WHERE to_tsvector('english', projectDescription) @@ to_tsquery('english
    ', 'innovation & strategy');
```

Listing 2: Creating a GIN index for full-text search

GIN is faster for frequent updates in full-text search compared to GiST, but GiST may be better for complex queries like nearest-neighbor searches.[2]

1.3.4 GiST Index for Complex Data

Great for geometric or advanced text searches (requires extensions like PostGIS for spatial data).

[2] See https://www.postgresql.org/docs/current/textsearch-indexes.html for GIN vs. GiST details.

```sql
-- Requires PostGIS extension
-- CREATE INDEX idxLocationsSpatial ON PointsOfInterest USING GiST (
    coordinates); -- Example
SELECT 'GiST example requires spatial data and PostGIS, placeholder
    shown.';
```

Listing 3: Creating a GiST index for geometric data

GiST is also used for full-text search but is slower for updates than GIN.

1.3.5 BRIN Index for Large, Ordered Data

Ideal for time-series or sequentially inserted data.

```sql
CREATE INDEX idxSalesTimestamp ON SalesTransactions USING BRIN (
    transactionTimestamp);
-- Example for BRIN with storage parameter and partial condition
CREATE INDEX idxOldProjectsStartDateBRIN ON Projects USING BRIN(
    startDate)
WITH (pages_per_range = 64)
WHERE startDate < DATE '2020-01-01';
```

Listing 4: Creating a BRIN index for time-series data

BRIN is lightweight, using minimal storage for massive tables.

1.3.6 Hash Index for Equality Lookups

Fast for simple = queries but limited in scope. (Reliable in PostgreSQL 10+)

```sql
CREATE INDEX idxEmployeeEmailHash ON Employees USING HASH (email);
```

Listing 5: Creating a Hash index

1.3.7 SP-GiST Index for Spatial Data

For specialized data like hierarchical or spatial structures.

```sql
-- CREATE INDEX idxSpatialPoints ON Points USING SPGIST (pointColumn);
    -- Example
SELECT 'SP-GiST example requires specific data structures, placeholder
    shown.';
```

Listing 6: Creating an SP-GiST index

1.3.8 Unique Index

Ensures no duplicates.

```sql
CREATE UNIQUE INDEX idxEmployeeEmail ON Employees (email);
```

Listing 7: Creating a unique index

1.3.9 Partial Index

Indexes a subset of rows to save space and boost speed.

```
CREATE INDEX idxActiveEmployees ON Employees (employeeId)
WHERE status = 'Active';

-- Partial composite index for specific job title and hire date range
CREATE INDEX idx_employees_swe_recent
ON query_optimizations_and_performance.Employees (departmentId) -- B-
    tree on departmentId
INCLUDE(salary) -- Store salary for index-only scans
WHERE jobTitle = 'Software Engineer'
  AND hireDate > '2018-01-01'; -- Only for recent software engineers
```

Listing 8: Creating a partial index

1.3.10 Index on Expressions

Indexes computed values for case-insensitive searches or transformations.

```
CREATE INDEX idxEmployeeEmailLower ON Employees (LOWER(email));
```

Listing 9: Creating an index on an expression

1.3.11 Advanced Index Options

PostgreSQL offers fine-tuned control:

- **CONCURRENTLY:** Build an index without locking the table, allowing writes during creation (slower but non-disruptive).

```
CREATE INDEX CONCURRENTLY idxEmployeeSalary ON Employees (salary);
```

Listing 10: Creating an index concurrently

- **INCLUDE:** Add non-indexed columns to a B-tree index for covering queries.

```
CREATE INDEX idxEmployeeDeptCovering ON Employees (departmentId)
    INCLUDE (salary, hireDate);
```

Listing 11: Creating a covering index with INCLUDE

- **WHERE for Partial Indexes:** As shown above, limit the indexed rows.
- **UNIQUE with NULLS NOT DISTINCT:** Treat NULLs as equal for uniqueness (PostgreSQL 15+).

```
-- CREATE UNIQUE INDEX idxEmployeeCodeNNF ON Employees (
    employeeCode) NULLS NOT DISTINCT;
SELECT 'NULLS NOT DISTINCT requires PostgreSQL 15+, placeholder
    shown.';
```

Listing 12: Unique index with NULLS NOT DISTINCT

- **FILLFACTOR:** Adjust how full index pages are packed, can help with updates.

```
CREATE INDEX idxEmployeeLastNameFill ON Employees (lastName) WITH (
    FILLFACTOR = 70);
```

Listing 13: Index with custom FILLFACTOR

1.3.12 Full-Text Search Indexes with Generated Columns

Full-text search in PostgreSQL uses `tsvector` and `tsquery` with GIN or GiST indexes to search text efficiently. Storing the `tsvector` in a generated column avoids recomputing it, boosting performance.

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

CREATE INDEX idxProjectsTSV ON Projects USING GIN (
    projectDescriptionTSV);

-- Query example
SELECT projectName, projectDescription
FROM Projects
WHERE projectDescriptionTSV @@ to_tsquery('english', 'peace & harmony')
    ;
```

Listing 14: Creating a table with a generated tsvector column for full-text search

Storing the `tsvector` *in a column avoids recomputing it, boosting performance.*[3]

1.3.13 Viewing Indexes

Check your indexes to ensure they're working as expected:

```sql
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'employees'; -- Lowercase table name common in
    pg_catalog
```

Listing 15: Listing indexes for a table

1.4 Why Use Them? (Advantages)

Indexes are your database's superpower:

- **Query Speed-Up:** Turn slow queries into instant results.
- **Reduced Server Load:** Less CPU, I/O, and memory usage.
- **Faster Joins and Sorting:** Critical for complex queries.
- **Constraint Enforcement:** `UNIQUE` and `PRIMARY KEY` indexes ensure data integrity.
- **Specialized Query Support:** GIN, GiST, and SP-GiST handle arrays, JSONB, full-text, and spatial data.
- **Scalability:** Indexes keep performance high as data grows.

[3]See https://www.postgresql.org/docs/current/textsearch-tables.html.

1.5 Watch Out! (Disadvantages)

Indexes aren't free magic:

- **Write Overhead:** `INSERT`, `UPDATE`, and `DELETE` operations slow down as indexes must be updated.
- **Storage Space:** Indexes can consume significant disk space, especially on large tables.
- **Maintenance Needs:** Indexes require periodic maintenance (e.g., `REINDEX`) to stay efficient.
- **Optimizer Pitfalls:** The query planner may not use an index if statistics are stale or the query isn't selective.
- **Not Always Useful:**
  - Small tables often don't benefit from indexes.
  - Low-cardinality columns (e.g., boolean) may not gain much from B-tree indexes.
  - Queries fetching most rows may prefer a sequential scan.

1.6 Index Maintenance and Monitoring

To keep your indexes in top shape, regular care is essential:

- **Rebuilding Indexes:** Over time, indexes can become fragmented. Use `REINDEX` to rebuild them.

```sql
REINDEX INDEX idxEmployeeLastName;
-- Rebuild CONCURRENTLY (PostgreSQL 12+)
-- REINDEX INDEX CONCURRENTLY idxEmployeeLastName;
```

Listing 16: Rebuilding an index

- **Clustering Tables:** Reorganize a table based on an index to improve sequential access. Use with caution as it locks the table.

```sql
-- CLUSTER Employees USING idxEmployeeLastName;
SELECT 'CLUSTER locks the table; use with caution.';
```

Listing 17: Clustering a table

- **Dropping Unused Indexes:** Remove indexes that aren't used to save space and reduce write overhead.

```sql
DROP INDEX IF EXISTS idxEmployeeLastName;
```

Listing 18: Dropping an index

- **Monitoring Index Usage:** Check if indexes are being used with `pg_stat_user_indexes`.

```sql
SELECT schemaname, relname AS tablename, indexrelname, idx_scan,
    idx_tup_read, idx_tup_fetch
FROM pg_stat_user_indexes
WHERE schemaname = 'public' -- Or your specific schema
ORDER BY idx_scan ASC; -- Find unused or rarely used indexes
```

Listing 19: Monitoring index usage

- **Updating Statistics:** Run `ANALYZE` to ensure the planner has accurate data distribution info. PostgreSQL's autovacuum daemon usually handles this, but manual runs can be useful. (See Section 3 for more details).

Regular maintenance keeps your indexes lean and mean, like a well-tuned engine.[4]

2 EXPLAIN Plans: Your Query's GPS!

Ever wonder what your database is *really* doing when you run a query? `EXPLAIN` is like a backstage pass, showing you the step-by-step execution plan. No more flying blind!

2.1 What Are They? (Meanings & Values)

2.1.1 Core Meaning

An **EXPLAIN Plan** is the database's roadmap for executing your SQL query. In PostgreSQL, the `EXPLAIN` command reveals this plan, showing operations like scans, joins, and sorts. It's like your query whispering, "Here's how I'm getting those results!" The plan is a tree of nodes, where each node represents an operation.

2.1.2 Typical Values or Outputs

The plan output includes:

- **Operation Type:** The specific action like `Seq Scan`, `Index Scan`, `Hash Join`, `Sort`, `Aggregate`.
- **Estimated Costs:**
  - `cost=startup_cost..total_cost`: Arbitrary units representing computational effort. `startup_cost` is before the first row is returned; `total_cost` is for all rows.
- **Estimated Rows (`rows`):** The planner's guess of how many rows this node will output. A large discrepancy between this and `actual rows` (from `EXPLAIN ANALYZE`) often indicates stale or insufficient statistics (see Section 3).
- **Estimated Width (`width`):** The planner's guess of the average size (in bytes) of rows output by this node.
- **With `ANALYZE`:**

[4] See https://www.postgresql.org/docs/current/routine-reindex.html and https://www.postgresql.org/docs/current/routine-vacuuming.html for maintenance details.

  - `actual time=startup_ms..total_ms`: Actual time taken (in milliseconds). The difference between startup and total time isn't a direct indicator of stale statistics, but rather reflects the nature of the operation (e.g., a scan returning many rows will naturally have a large difference).
  - `actual rows`: Actual number of rows output.
  - `loops`: Number of times the node was executed.

- **With `BUFFERS` (and `ANALYZE`):**
  - `shared hit/read/dirtied/written`: Shared buffer cache activity.
  - `local hit/read/dirtied/written`: Local/temporary buffer activity.
  - `temp read/written`: Temporary file I/O (e.g., for disk-based sorts/hashes). This is a key indicator of spilling.

Always use `EXPLAIN ANALYZE` for real-world performance insights, as estimates can be misleading.[5]

2.2 Relations: How They Play with Others

2.2.1 Relations within Query Optimization

- **Index Effectiveness Check:** `EXPLAIN` shows if an `Index Scan`, `Index Only Scan`, or `Bitmap Heap Scan` is used, or if the planner resorts to a `Seq Scan`, indicating a potential missing or ineffective index.

- **Join Strategy Verification:** Reveals which join type (`Nested Loop`, `Hash Join`, `Merge Join`) was chosen, helping to understand if join columns are properly indexed or if statistics are misleading the planner.

- **Sort and Aggregate Costs:** Highlights the cost of `Sort` operations (for `ORDER BY`, `Merge Join`, some `GROUP BY` or window functions) and `Aggregate` nodes (`HashAggregate`, `GroupAggregate`). High costs here might point to insufficient `work_mem` or missing indexes.

- **Subquery Behavior:** Shows how subqueries (correlated or uncorrelated) are being executed, often as sub-plans or joined into the main query.

- **Statistics Impact:** Discrepancies between estimated and actual row counts are strong indicators of stale or inadequate statistics, prompting an `ANALYZE` or review of statistics targets (see Section 3).

2.2.2 Relations with SQL Concepts

`EXPLAIN` provides a direct view of how PostgreSQL interprets and executes your SQL constructs:

- **`WHERE` Clause Filters:** Shows how predicates are applied and their estimated vs. actual selectivity.

[5] See the official documentation: https://www.postgresql.org/docs/current/sql-explain.html.

- **JOIN Conditions:** Illustrates the join order and method chosen by the planner.
- **GROUP BY and HAVING:** Details the aggregation strategy and how filtering is applied post-aggregation.
- **ORDER BY:** Shows if sorting is done via an explicit `Sort` node or by leveraging an index.
- **Window Functions (OVER(...)):** Displays `WindowAgg` nodes and any associated sorting.

2.3 How to Use Them: Structures & Syntax

Prefix your query with `EXPLAIN` or, more usefully, `EXPLAIN ANALYZE`.

```sql
EXPLAIN ANALYZE
SELECT e.firstName, d.departmentName
FROM Employees e
JOIN Departments d ON e.departmentId = d.departmentId
WHERE e.salary > 60000
ORDER BY d.departmentName, e.salary DESC;
```

Listing 20: Basic EXPLAIN ANALYZE

2.3.1 Common Options

- **ANALYZE:** Executes the query and shows actual run times and row counts. Essential for real tuning.
- **VERBOSE:** Provides more detailed output, like schema-qualifying table names and output column lists for each node.
- **BUFFERS:** (Requires `ANALYZE`) Shows buffer usage (cache hits, disk reads), crucial for diagnosing I/O issues.
- **COSTS:** Includes estimated costs (default is true). Can be set to false to simplify output if costs aren't the focus.
- **TIMING:** (Requires `ANALYZE`) Includes actual node timing (default is true). Can be set to false if per-node timing is too noisy.
- **SETTINGS:** (PostgreSQL 13+) Includes information about configuration parameters that affect query planning if they are non-default (e.g., `work_mem`).
- **SUMMARY:** Includes a summary line after the plan (default is true when `ANALYZE` is used).
- **FORMAT format_name:** Specifies output format. Options: `TEXT` (default), `JSON`, `XML`, `YAML`. `JSON` is very useful for feeding into plan visualization tools.

```sql
EXPLAIN (ANALYZE, BUFFERS, VERBOSE, SETTINGS, FORMAT JSON)
SELECT e.jobTitle, AVG(e.salary) as avg_salary
FROM Employees e
WHERE e.hireDate > DATE '2020-01-01'
GROUP BY e.jobTitle
HAVING COUNT(*) > 2
ORDER BY avg_salary DESC;
```

Listing 21: EXPLAIN with Multiple Options for Detailed Analysis

The JSON output can be pasted into tools like https://explain.dalibo.com/ or https://explain.depesz.com/ for a visual representation.

2.3.2 Key Plan Nodes (and their efficiency properties)

- **Scans:**
  - `Seq Scan`: Reads the whole table. Efficient for small tables or when fetching most rows; otherwise slow.
  - `Index Scan`: Uses an index to find row TIDs, then fetches rows from the table. Efficient for selective queries.
  - `Index Only Scan`: Gets all data from the index; very fast if applicable (covering index, visibility map hit).
  - `Bitmap Heap Scan` (with `Bitmap Index Scan`): Good for combining multiple index conditions or moderately selective queries. Index(es) build a bitmap of TIDs, then heap is scanned more sequentially.
- **Joins:**
  - `Nested Loop Join`: Good for small outer tables with an indexed inner table. Can start returning rows fast.
  - `Hash Join`: Builds a hash table on one (smaller) table, probes with the other. Efficient for large equi-joins if hash table fits in `work_mem`.
  - `Merge Join`: Sorts both inputs on join keys, then merges. Good if inputs are pre-sorted or for non-equi-joins on sorted data. Sorts can be expensive.
- **Others:**
  - `Sort`: Orders data. Can be costly, especially if spilling to disk (check `work_mem`).
  - `Aggregate` (`HashAggregate`, `GroupAggregate`, `MixedAggregate`): For `GROUP BY`. HashAggregate is common; GroupAggregate if input is sorted.
  - `WindowAgg`: For window functions. Often involves sorting.
  - `Limit`: Stops after N rows; can significantly change plans if combined with `ORDER BY` and suitable indexes.
  - `Materialize`: Stores the result of a subplan in memory for reuse. Can be good if reused often, but adds overhead if not.
  - `CTE Scan`: Reads from a Common Table Expression. Look at the CTE's subplan for its costs.

2.3.3 Identifying Performance Bottlenecks in EXPLAIN Output

When reviewing `EXPLAIN (ANALYZE, BUFFERS)` output, look for these common red flags that suggest areas for query enhancement:

- **High `actual time` on a Node:** This is the most direct indicator. The node taking the longest cumulative time (considering its loops) is your primary bottleneck.
- **`Seq Scan on <large_table_name>`:**
  - **Why it's a problem:** Reads the entire table, very inefficient if only a small subset of rows is needed.
  - **Suggestion:** Add an index on columns in the `WHERE` clause. Ensure predicates are SARGable. Check statistics.
- **Large Discrepancy between Estimated and Actual Rows (`rows=E actual rows=A`):**
  - **Why it's a problem:** Planner made a poor choice based on bad estimates.
  - **Suggestion:** Run `ANALYZE <table_name>`. Consider increasing `default_statistics_targe` for problematic columns or using extended statistics.
- **Sort Spilling to Disk: `Sort Method: external merge Disk: <size>kB` (where `<size> > 0`):**
  - **Why it's a problem:** Disk I/O for sorting is orders of magnitude slower than in-memory.
  - **Suggestion:** Increase `work_mem`. See if an index can provide pre-sorted data to avoid the sort.
- **Hash Operation Spilling to Disk: `Batches: N` (where `N > 1`) in `Hash Join` or `HashAggregate` nodes (or their underlying `Hash` sub-node):**
  - **Why it's a problem:** Similar to sorts, spilling hash tables to disk is very slow.
  - **Suggestion:** Increase `work_mem`. Filter data earlier to reduce the size of the hash table.
- **High `Rows Removed by Filter: N` in a `Filter` node, or `Rows Removed by Index Recheck: N` in a scan node:**
  - **Why it's a problem:** Many rows were read and processed, only to be discarded later. For index rechecks, it indicates a "lossy" index scan.
  - **Suggestion:** Try to push filter conditions earlier in the plan. For rechecks, `VACUUM` the table or check index selectivity.
- **High `Heap Fetches: N` (where `N > 0`) in an `Index Only Scan`:**
  - **Why it's a problem:** The query could almost be satisfied by the index alone, but still required table lookups.

  - **Suggestion:** `VACUUM` the table. Ensure all selected columns are in the index key or `INCLUDE` clause.

- **Nested Loop Join with High `loops` on Inner Side, especially if Inner Side is Slow:**
  - **Why it's a problem:** The inner plan is executed many times. If it's slow each time, the total is very slow.
  - **Suggestion:** Ensure an index on the join key of the inner table. Consider if a Hash or Merge Join would be better (often due to statistics).

- **High Buffer Reads: `Buffers: shared read=N` or `temp read=N` (where `N` is large):**
  - **Why it's a problem:** High disk I/O. Shared reads mean data wasn't in cache. Temp reads mean disk spilling.
  - **Suggestion:** For shared reads, improve indexing to read fewer blocks or increase `shared_buffers`. For temp reads, increase `work_mem`.

- **Expensive Function Calls:** If a `Function Scan` or a function within another node shows high `actual time`.
  - **Why it's a problem:** The function logic itself is slow.
  - **Suggestion:** Optimize the function code. Mark functions `STABLE` or `IMMUTABLE` if appropriate.

- **Slow CTE Materialization: `CTE Scan` with a slow underlying `Materialize` node for the CTE.**
  - **Why it's a problem:** The CTE itself is a bottleneck.
  - **Suggestion:** Optimize the query within the CTE. Consider `NOT MATERIALIZED` hint if appropriate (PostgreSQL 12+).

Identifying these patterns will guide you to the most impactful optimizations.

2.4 Why Use Them? (Advantages)

- **Identify Performance Bottlenecks:** Pinpoint slow operations (e.g., full table scans on large tables, inefficient joins, costly sorts).
- **Verify Index Usage:** Confirm if your carefully crafted indexes are actually being used by the planner.
- **Understand Query Execution:** Gain insight into how the database processes your SQL, leading to better query writing.
- **Compare Alternatives:** Objectively evaluate the performance implications of different query structures or indexing strategies.
- **Aid in Schema Design:** Inform decisions about table structures and data types based on how queries perform.

2.5 Watch Out! (Disadvantages)

- **Complexity:** Interpreting complex plans requires experience and understanding of PostgreSQL internals.
- **`EXPLAIN ANALYZE` Runs the Query:** Be cautious with DML (`INSERT`, `UPDATE`, `DELETE`) as it will execute the changes. Use within a transaction that you `ROLLBACK` for testing DML.
- **Estimates vs. Actuals:** `EXPLAIN` (without `ANALYZE`) relies on estimates from table statistics. If statistics are stale or inaccurate, the estimated plan might differ significantly from the actual execution.
- **Environment Dependent:** Plans can change based on PostgreSQL version, configuration settings (e.g., `work_mem`, planner cost constants), data volume, and data distribution. A plan from a development environment might not be the same as in production.
- **Overhead of `ANALYZE`:** While necessary for true insights, `EXPLAIN ANALYZE` adds some overhead, especially the timing instrumentation. For extremely fast queries, this overhead can be a significant portion of the reported time.

3 Statistics Management: Fueling the Query Planner

The PostgreSQL query planner is a sophisticated piece of software, but it's not a mind reader. To make intelligent decisions about how to execute your queries (e.g., which indexes to use, what join types are best), it relies heavily on statistical information about your data. If these statistics are inaccurate or insufficient, the planner can make poor choices, leading to suboptimal performance. Managing these statistics is therefore a cornerstone of query optimization.

3.1 What Are They? (Meanings & Values)

3.1.1 Core Meaning

Statistics Management encompasses the processes and configurations used to collect, store, and update data about the content and distribution of values within your tables and columns. PostgreSQL uses these statistics to estimate the costs of different execution plan alternatives and choose the one it believes will be the fastest.

3.1.2 Typical Values or Outputs

Well-managed statistics contribute to:

- **Accurate Cost Estimation by the Planner:** Leading to better choices of scan methods, join algorithms, and join orders.
- **Effective Index Utilization:** The planner is more likely to use appropriate indexes when it has a good understanding of data selectivity.

- **Stable and Predictable Query Performance:** Queries behave as expected because the planner is working with reliable information.
- **Reduced Need for Manual Query Tuning Hints (which PostgreSQL largely avoids anyway):** The planner can often find the best path if given good stats.

The primary "output" of statistics management is the set of populated system catalog tables like `pg_class` (for table-level stats like row counts) and `pg_stats` (for column-level stats).

3.2 Relations: How They Play with Others

3.2.1 Relations within Query Optimization

- **Foundation for `EXPLAIN`:** The "estimated rows" and "costs" in an `EXPLAIN` plan are derived directly from these statistics. Large discrepancies between estimated and actual rows in `EXPLAIN ANALYZE` are a tell-tale sign of stale or inadequate statistics.
- **Influences Index Choice:** The planner decides whether to use an index based on its estimated cost, which depends on how selective the index is believed to be (an assessment derived from statistics).
- **Impacts Join Method Selection:** For example, the decision to use a Hash Join vs. a Nested Loop Join depends heavily on the estimated sizes of the input relations, which come from statistics.

3.2.2 Relations with SQL Concepts

- **`ANALYZE` Command:** The primary SQL command used to manually trigger statistics collection for a table, schema, or entire database.
- **Autovacuum Daemon:** PostgreSQL's background process that, among other tasks, automatically runs `ANALYZE` on tables that have undergone a significant number of changes.
- **System Catalogs:** Statistics are stored in system tables like `pg_class`, `pg_statistic`, and viewed through `pg_stats`.
- **Configuration Parameters:** Settings like `autovacuum_analyze_threshold`, `autovacuum_analy` and `default_statistics_target` control how and when statistics are collected.

3.3 How to Use Them: Structures & Syntax

3.3.1 Keeping Statistics Updated

1. **Rely on and Tune Autovacuum (Autoanalyze):** This is PostgreSQL's first line of defense.
    - Ensure `autovacuum = on` in `postgresql.conf` (default).

- **Tune Thresholds:** For large tables, the default `autovacuum_analyze_scale_factor` (e.g., 0.10 for 10%) might mean millions of row changes before an analyze. Consider lowering it per table if statistics go stale too quickly:

```
ALTER TABLE query_optimizations_and_performance.Employees
SET (autovacuum_analyze_scale_factor = 0.02); -- Analyze after
    2% change + threshold
```

Listing 22: Adjusting autovacuum analyze scale factor for a table

- The `autovacuum_analyze_threshold` (default 50) is the minimum number of changes added to the scaled calculation.

2. **Manual `ANALYZE`:** Use when necessary.
   - After large data loads or bulk modifications.
   - Before critical performance tests or batch jobs.
   - When troubleshooting a query that `EXPLAIN ANALYZE` shows has poor estimates.
   - After creating new indexes that might significantly change data access patterns.

```
            ANALYZE VERBOSE query_optimizations_and_performance
    .Employees;
            -- Analyze all tables in the current database:
            -- ANALYZE VERBOSE;

```

Listing 23: Running ANALYZE manually

3.3.2 Understanding Column Statistics with `pg_stats`

The `pg_stats` view provides a wealth of information about each column.

```
SELECT
    attname AS column_name,
    n_distinct,         -- Estimated number of distinct values. >0 is
    direct count; <0 is fraction of total rows.
    null_frac,          -- Fraction of NULL values.
    avg_width,          -- Average width in bytes.
    most_common_vals, -- Array of most common values.
    most_common_freqs,-- Frequencies of most_common_vals.
    histogram_bounds  -- Array dividing non-MCV values into equal
    population buckets.
FROM
    pg_stats
WHERE
    tablename = 'employees' AND schemaname = '
    query_optimizations_and_performance'
ORDER BY
    attname;
```

Listing 24: Querying pg_stats for column statistics

Interpreting `n_distinct` for Cardinality:

- If `n_distinct > 0`: This is the direct estimate of unique values. Higher value = higher cardinality.

- If `n_distinct < 0`: The column is (nearly) unique. The absolute value is a fraction of total rows (e.g., -0.9 means 90% of rows are unique). Closer to -1 means higher cardinality.

A refined query to estimate distinct values, useful for prioritizing index columns:

```sql
WITH TableStats AS (
    SELECT reltuples AS total_rows
    FROM pg_class c JOIN pg_namespace n ON n.oid = c.relnamespace
    WHERE c.relname = 'employees' AND n.nspname = '
   query_optimizations_and_performance'
)
SELECT
    s.attname AS column_name, s.n_distinct,
    CASE
        WHEN s.n_distinct > 0 THEN s.n_distinct
        WHEN s.n_distinct < 0 THEN ROUND(-1 * s.n_distinct * ts.
   total_rows)
        ELSE 0
    END AS estimated_distinct_values,
    s.null_frac, s.avg_width
FROM pg_stats s, TableStats ts
WHERE s.tablename = 'employees' AND s.schemaname = '
   query_optimizations_and_performance'
ORDER BY estimated_distinct_values DESC;
```

Listing 25: Estimating distinct values for cardinality ranking

3.3.3 Controlling Statistics Detail: `default_statistics_target`

This parameter (default 100) controls the number of entries in `most_common_vals` and buckets in `histogram_bounds`. For columns with skewed distributions or many important distinct values, increasing this can help.

```sql
ALTER TABLE query_optimizations_and_performance.Employees
ALTER COLUMN jobTitle SET STATISTICS 500;
ANALYZE query_optimizations_and_performance.Employees; -- Re-analyze to
     apply
```

Listing 26: Increasing statistics target for a specific column

3.3.4 Extended Statistics: Beyond Single Columns

Sometimes, columns are correlated (e.g., city and state). Standard per-column statistics don't capture this. Extended statistics can help.

- **N-Distinct Counts for Combinations:** Estimates distinct pairs/tuples.
- **Functional Dependencies:** If colA determines colB (e.g., zip code determines city).
- **MCV Lists for Combinations:** Most common combinations of values.

```sql
-- Create statistics object for (firstName, lastName) combination
CREATE STATISTICS query_optimizations_and_performance.
   st_employee_fullname (ndistinct)
```

```sql
ON firstName, lastName
FROM query_optimizations_and_performance.employees;

ANALYZE query_optimizations_and_performance.employees; -- Populate the new statistics

-- Querying extended statistics (requires appropriate permissions, e.g., pg_read_all_stats)
SELECT
    s.stxname, s.stxkeys, s.stxkind, d.stxdndistinct
FROM pg_statistic_ext s
JOIN pg_statistic_ext_data d ON s.oid = d.stxoid
WHERE s.stxname = 'st_employee_fullname'
  AND s.stxrelid = 'query_optimizations_and_performance.employees'::regclass;
```

Listing 27: Creating and viewing N-Distinct extended statistics

To grant access for non-superusers to view extended statistics data, a superuser can execute: ***`GRANT pg_read_all_stats TO your_user;`****. If this fails with "must have admin option", the* ***`GRANT`*** *command itself must be run by a superuser.*

```sql
DROP STATISTICS IF EXISTS query_optimizations_and_performance.st_employee_fullname;
```

Listing 28: Dropping extended statistics

3.4 Why Use Them? (Advantages)

- **Accurate Query Plans:** The single most important factor for the planner to choose efficient execution paths.
- **Optimal Index Usage:** Prevents the planner from ignoring good indexes or choosing bad ones.
- **Predictable Performance:** Reduces query time variability caused by poor estimations.
- **Automated Maintenance Foundation:** Autovacuum relies on these mechanisms to keep things running smoothly.

3.5 Watch Out! (Disadvantages)

- **Staleness Impact:** Out-of-date statistics are a primary cause of sudden query performance degradation.
- **`ANALYZE` Overhead:** Running `ANALYZE` consumes resources (I/O, CPU). While usually lightweight, on very large tables or during peak hours, frequent manual runs can be noticeable. Autovacuum is designed to mitigate this.
- **Default Target Limitations:** The default statistics target (100) might not be sufficient for columns with complex distributions, requiring manual tuning.
- **Extended Statistics Complexity:** Defining and managing extended statistics adds a layer of administration, but can be crucial for complex correlations.

- **Sampling Imperfections:** `ANALYZE` works by sampling data. For very large tables or extremely skewed data, the sample might not always perfectly represent the whole dataset.

4 Optimizing Window Functions and Aggregates: Making Analytics Fly!

Window functions (`RANK()`, `LAG()`, `SUM() OVER (...)`) and aggregate functions (`COUNT()`, `AVG()`, typically with `GROUP BY`) are SQL's analytical superstars. They allow complex calculations across sets of rows. However, their power comes with potential performance costs if not tuned carefully.

4.1 What Are They? (Meanings & Values)

4.1.1 Core Meaning

Optimizing Window Functions and Aggregates involves structuring queries and leveraging database features (primarily indexes and appropriate data filtering) to ensure that these analytical operations run efficiently, especially on large datasets. The goal is to minimize computation, I/O, and memory usage.

4.1.2 Typical Values or Outputs

Effective optimization leads to:

- **Faster Analytical Queries:** Significant reduction in query execution time for reports and data analysis.
- **Lower Resource Consumption:** Reduced CPU, memory (especially `work_mem` for sorts/hashes), and I/O operations.
- **Improved Scalability:** Queries maintain good performance even as data volume grows.
- **More Responsive Applications:** Dashboards and analytical tools fed by these queries become snappier.

4.2 Relations: How They Play with Others

4.2.1 Relations within Optimization

- **Indexes are Key:**
  - For **Aggregates (`GROUP BY`)**: Indexes on grouping columns can allow PostgreSQL to use a more efficient `GroupAggregate` strategy (if data is already sorted by grouping keys) or speed up `HashAggregate` by providing faster access. Covering indexes that include aggregated columns can enable index-only scans.

  - For **Window Functions**: Indexes on columns in `PARTITION BY` and `ORDER BY` clauses within the `OVER()` definition are crucial. They can eliminate costly sort operations that `WindowAgg` nodes often require.

- **`EXPLAIN` Plan Analysis:** Essential for identifying bottlenecks. Look for:
  - Expensive `Sort` nodes preceding `WindowAgg` or `GroupAggregate`.
  - High costs or row counts in `WindowAgg` or `Aggregate` (e.g., `HashAggregate`) nodes.
  - Spilling to disk (indicated by ‘temp read/written‘ in ‘BUFFERS‘ output, or ‘Disk: XkB‘ in Sort nodes, or ‘Batches: ¿1‘ in HashAggregate/WindowAgg nodes) for sorts or hash aggregation. (See Section 2.3.3).

- **Memory Configuration (`work_mem`):** Sufficient `work_mem` allows sorts and hash tables (used in `HashAggregate` and some window function evaluations) to be done in memory, which is much faster than spilling to disk.

4.2.2 Relations with SQL Concepts

- **Filtering (`WHERE` Clause):** Applying filters *before* aggregation or window function computation drastically reduces the amount of data processed, leading to significant speedups.

- **Common Table Expressions (CTEs - `WITH` Clause):** Useful for breaking down complex queries. Can be used to pre-filter or pre-aggregate data before applying window functions. Be aware of CTE materialization behavior.

- **Subqueries:** Similar to CTEs, can be used for pre-processing, but their optimization can sometimes be less predictable than CTEs or direct joins.

- **Join Operations:** If aggregates or window functions are applied after joins, ensuring efficient join performance is paramount.

- **Data Types:** Operations on smaller, fixed-width data types are generally faster.

4.3 How to Use Them: Structures & Syntax

4.3.1 Techniques for Optimization

1. **Filter Early and Aggressively:** Use the `WHERE` clause to reduce the dataset as much as possible *before* any `GROUP BY` or window function is applied.

```sql
-- Less efficient: Aggregates then filters
SELECT departmentId, AVG(salary)
FROM Employees
GROUP BY departmentId
HAVING departmentId IN (1, 2); -- Filters after aggregation

-- More efficient: Filters then aggregates
SELECT departmentId, AVG(salary)
FROM Employees
WHERE departmentId IN (1, 2) -- Filters before aggregation
```

```sql
GROUP BY departmentId;
```

Listing 29: Filtering before aggregation

2. **Index for `GROUP BY`, `PARTITION BY`, and `ORDER BY`:**

```sql
-- For GROUP BY
CREATE INDEX idx_employees_dept_job ON Employees (departmentId,
    jobTitle);
SELECT departmentId, jobTitle, COUNT(*)
FROM Employees
GROUP BY departmentId, jobTitle;

-- For Window Functions
CREATE INDEX idx_employees_dept_salary ON Employees (departmentId,
    salary DESC);
SELECT
    firstName, lastName, departmentId, salary,
    RANK() OVER (PARTITION BY departmentId ORDER BY salary DESC) as
    dept_salary_rank
FROM Employees;
```

Listing 30: Indexing for GROUP BY and Window Functions

3. **Pre-aggregate Data using CTEs or Subqueries:** If you need to perform window functions on aggregated data, do the aggregation first.

```sql
WITH DepartmentAvgSalary AS (
    SELECT departmentId, AVG(salary) as avg_dept_salary
    FROM Employees
    GROUP BY departmentId
)
SELECT
    d.departmentName, das.avg_dept_salary,
    RANK() OVER (ORDER BY das.avg_dept_salary DESC) as
    global_avg_salary_rank
FROM Departments d
JOIN DepartmentAvgSalary das ON d.departmentId = das.departmentId;
```

Listing 31: Pre-aggregating with a CTE

4. **Use the `WINDOW` Clause for Readability and Potential Reuse:** Define named windows if you use the same window specification multiple times.

```sql
SELECT
    salary,
    AVG(salary) OVER w_dept as avg_dept_salary,
    SUM(salary) OVER w_dept as total_dept_salary
FROM Employees
WINDOW w_dept AS (PARTITION BY departmentId);
```

Listing 32: Using the WINDOW clause

5. **Consider `FILTER` Clause for Conditional Aggregation (PostgreSQL 9.4+):** Can be more efficient and readable than ‘CASE‘ statements inside aggregates for some scenarios.

```sql
SELECT
    departmentId,
    COUNT(*) FILTER (WHERE salary > 80000) as high_earners_count,
    COUNT(*) FILTER (WHERE jobTitle = 'Manager') as manager_count
FROM Employees
GROUP BY departmentId;
```

Listing 33: Conditional aggregation with FILTER

6. **Minimize Columns in `GROUP BY` and `PARTITION BY`:** Only include necessary columns, as more columns can increase complexity and reduce index effectiveness.
7. **Understand Cost of Different Window Functions:** Some window functions (like those requiring full partition sorts) are more expensive than others (like simple row numbering on an already sorted partition).

4.4 Why Use Them? (Advantages)

Optimization leads to:

- **Dramatically Faster Analytics:** Transform slow, resource-intensive analytical queries into responsive ones.
- **Reduced Server Strain:** Less CPU, memory, and I/O pressure, allowing the server to handle more concurrent workload.
- **Scalable Solutions:** Ensures that analytical capabilities can grow with the data volume without performance degradation.
- **Enable Complex Insights:** Makes it feasible to perform sophisticated data analysis that would otherwise be too slow.
- **Happier Users:** Faster reports and dashboards lead to increased user satisfaction and productivity.

4.5 Watch Out! (Disadvantages)

Ignoring optimization or misapplying techniques can lead to:

- **Extremely Slow Queries:** Unoptimized window functions or aggregates on large tables can run for hours or even appear to hang.
- **High Resource Consumption:** Can exhaust server memory (`work_mem` leading to disk spills) or CPU, impacting other operations.
- **Incorrect Results (Logical Errors):** While not a direct performance issue, complex analytical queries are prone to logical errors if not carefully constructed and tested. Optimization efforts should not compromise correctness.
- **Over-Indexing:** While indexes are crucial, adding too many, or the wrong ones, can slow down write operations (`INSERT`, `UPDATE`, `DELETE`) and consume disk space.
- **Premature Optimization:** Focusing too much on micro-optimizations before identifying the main bottlenecks (via `EXPLAIN ANALYZE`) can lead to overly complex and less maintainable code for minimal gain.

5 Query Rewriting and Refactoring: The Art of SQL Makeovers!

Sometimes, the best way to speed up a query isn't just adding an index, but fundamentally changing how the query is written. Query rewriting is about transforming your SQL into a more efficient form that the database can execute faster.

5.1 What Are They? (Meanings & Values)

5.1.1 Core Meaning

Query Rewriting and Refactoring refers to the process of modifying the structure and logic of an SQL query to achieve the same results more efficiently, often by guiding the database planner towards a better execution plan. This can involve simplifying complex expressions, changing join types, replacing subqueries, or using different SQL constructs.

5.1.2 Typical Values or Outputs

The goal is to achieve:

- **Improved Performance:** Faster query execution times.
- **Reduced Resource Usage:** Less CPU, memory, and I/O.
- **Better Readability and Maintainability:** Simpler queries are often easier to understand and modify.
- **Enhanced Scalability:** Queries that perform well even as data grows.

5.2 Relations: How They Play with Others

5.2.1 Relations within Query Optimization

- **Interplay with Indexes:** Rewriting might enable better use of existing indexes or highlight the need for new ones.
- **`EXPLAIN` as a Guide:** Use `EXPLAIN ANALYZE` to compare the plans of the original and rewritten queries to verify improvements.
- **Understanding Planner Behavior:** Effective rewriting often comes from understanding how the PostgreSQL query planner tends to optimize certain patterns.

5.2.2 Relations with SQL Concepts

Rewriting touches almost every SQL concept:

- **Joins:** Converting subqueries to joins, changing `LEFT` to `INNER` if logically equivalent, optimizing join order (though the planner often does this).
- **Subqueries:** Replacing correlated subqueries in `SELECT` or `WHERE` with joins or CTEs.

- **`WHERE` Clause:** Simplifying predicates, making them SARGable (Search ARGument Able - allowing index use).
- **Aggregates & Window Functions:** Restructuring to filter early or pre-aggregate.
- **CTEs (`WITH` Clause):** Using CTEs to break down complexity or materialize intermediate results (though PostgreSQL might inline them).
- **`UNION` vs. `OR`:** Sometimes rewriting `OR` conditions as a `UNION ALL` (if sets are disjoint) can lead to better plans.
- **`INTERSECT` and `EXCEPT`:** `INTERSECT` can be an alternative to complex AND conditions across subqueries if they select the same columns. `EXCEPT` for "A and NOT B" logic.

5.3 How to Use Them: Structures & Syntax

5.3.1 Common Rewriting Techniques

1. **Replace Correlated Subqueries with Joins:** Correlated subqueries in the `SELECT` list or `WHERE` clause often execute once per outer row and can be very slow.

```sql
-- Potentially slow (subquery per employee)
SELECT e.firstName, e.lastName,
    (SELECT d.departmentName FROM Departments d WHERE d.departmentId = e.departmentId) as deptName
FROM Employees e;

-- Rewritten with LEFT JOIN (generally faster)
SELECT e.firstName, e.lastName, d.departmentName
FROM Employees e
LEFT JOIN Departments d ON e.departmentId = d.departmentId;
```

Listing 34: Rewriting a correlated subquery in SELECT to a JOIN

2. **Convert `IN (subquery)` to `EXISTS (subquery)` or Join:** Sometimes `EXISTS` or a direct join is optimized better.

```sql
-- Using IN
SELECT * FROM Employees e
WHERE e.departmentId IN (SELECT d.departmentId FROM Departments d WHERE d.location = 'Building A');

-- Using EXISTS (often similar plan, but can differ)
SELECT * FROM Employees e
WHERE EXISTS (
    SELECT 1 FROM Departments d
    WHERE d.departmentId = e.departmentId AND d.location = 'Building A'
);
-- Or using JOIN (often very efficient)
SELECT e.*
FROM Employees e
JOIN Departments d ON e.departmentId = d.departmentId
WHERE d.location = 'Building A';
```

Listing 35: Rewriting IN with a subquery to EXISTS

3. **Simplify `WHERE` Clause Predicates:** Avoid functions on indexed columns if possible.

```sql
-- Non-SARGable (index on hireDate might not be used effectively)
SELECT * FROM Employees WHERE EXTRACT(YEAR FROM hireDate) = 2020;

-- SARGable (allows range scan on hireDate index)
SELECT * FROM Employees WHERE hireDate >= DATE '2020-01-01' AND
    hireDate < DATE '2021-01-01';
```

Listing 36: Making a WHERE clause SARGable

4. **Use `UNION ALL` instead of `OR` for Disparate Conditions:** If conditions target very different data subsets that could use different indexes, `UNION ALL` might be better (ensure no duplicates are introduced if `UNION` was intended, or use logic to make them disjoint).

```sql
-- Using OR (might struggle with index selection if jobTitle and
    departmentId are indexed separately)
SELECT * FROM Employees
WHERE jobTitle = 'Software Engineer' OR departmentId = 5;

-- Using UNION ALL (if conditions target distinct enough sets for
    different index uses)
SELECT * FROM Employees WHERE jobTitle = 'Software Engineer'
UNION ALL
SELECT * FROM Employees WHERE departmentId = 5 AND jobTitle <> '
    Software Engineer'; -- Make sets disjoint
```

Listing 37: OR vs. UNION ALL

5. **Break Down Complex Queries with CTEs:** Improves readability and can sometimes help the planner by isolating parts of the logic.

```sql
WITH ActiveEngineers AS (
    SELECT * FROM Employees
    WHERE status = 'Active' AND jobTitle LIKE '%Engineer%'
)
SELECT ae.firstName, ae.salary, d.departmentName
FROM ActiveEngineers ae
JOIN Departments d ON ae.departmentId = d.departmentId;
```

Listing 38: Using a CTE to simplify logic

6. **Minimize Data Processed at Each Step:** Filter early, select only necessary columns.

5.4 Why Use Them? (Advantages)

- **Unlock Performance Gains:** Can lead to order-of-magnitude speedups where indexing alone is insufficient.
- **Overcome Planner Limitations:** Helps guide the planner when it struggles with complex or unusual query patterns.
- **Improved Code Clarity:** Often, a more efficient query is also a more straightforward and understandable one.

- **Targeted Optimization:** Allows fine-tuning of specific problematic parts of a larger query.

5.5 Watch Out! (Disadvantages)

- **Complexity:** Requires a good understanding of SQL and how the database executes queries.
- **Risk of Introducing Errors:** Rewriting logic can inadvertently change the query's meaning or results if not done carefully. Always test thoroughly.
- **Planner Changes:** A rewrite that is optimal today might become suboptimal if future PostgreSQL versions change planner behavior. (Less common for fundamental rewrites).
- **Over-Optimization:** Making queries overly complex in an attempt to outsmart the planner can reduce readability for marginal gains. Sometimes the planner knows best.
- **Not a Silver Bullet:** Rewriting won't fix fundamental issues like missing indexes on critical columns or massively inefficient schemas.

6 Advanced Indexing Features: Supercharging Your Shortcuts!

Beyond basic B-trees, PostgreSQL offers a suite of advanced indexing features that provide fine-grained control and unlock performance for specialized use cases. These are the power tools in your indexing workshop!

Keep in mind the SQL queries to take the best decisions optimizing queries

```sql
SELECT
    schemaname,
    relname AS table_name,
    indexrelname AS index_name,
    idx_scan,        -- Number of times this index was scanned
    idx_tup_read,    -- Number of index entries returned by scans
    idx_tup_fetch    -- Number of live table rows fetched by simple
   index scans
FROM
    pg_stat_user_indexes
WHERE idx_scan = 0 -- Example: Find potentially unused indexes
ORDER BY
    schemaname, table_name, index_name;
```

Listing 39: The activity and usage of all indexes

```sql
SELECT attname AS column_name,
       n_distinct,
       null_frac,
       avg_width,
       most_common_vals,
       most_common_freqs,
       histogram_bounds
```

```sql
FROM pg_stats
WHERE tablename = 'employees'
  AND schemaname = 'query_optimizations_and_performance'
ORDER BY attname;
```

Listing 40: The more relevant statistical information about a table of a schema

This query gives incredible insights about the cardinality, common values and all necessary information to design queries and prioritize indexes (see Section 3 for interpretation).

6.1 What Are They? (Meanings & Values)

6.1.1 Core Meaning

Advanced Indexing Features encompass PostgreSQL-specific capabilities like partial indexes, expression indexes, covering indexes (`INCLUDE`), concurrent index creation, generated columns with indexes, and index storage parameters (`FILLFACTOR`, BRIN's `pages_per_range`). These features allow for more tailored and efficient indexing strategies.

6.1.2 Typical Values or Outputs

Using these features can lead to:

- **Smaller, More Targeted Indexes:** Reduced storage and faster scans.
- **Index-Only Scans:** Avoiding table heap access for maximum speed.
- **Non-Disruptive Index Creation:** Adding indexes without blocking application writes.
- **Optimized Performance for Specific Query Patterns:** Indexing computed values or specific data subsets.
- **Reduced Index Bloat and Maintenance:** Through better storage parameter settings.

6.2 Relations: How They Play with Others

6.2.1 Relations within Indexing Itself

- **Complement Basic Indexing:** These features enhance standard index types (B-tree, GIN, etc.), they don't replace them.
- **Influence Planner Choices:** A well-crafted partial or covering index can dramatically alter the query plan chosen by `EXPLAIN`.
- **Trade-offs:** E.g., `CONCURRENTLY` is slower to build but less disruptive. Partial indexes are smaller but only serve specific queries.

6.2.2 Relations with SQL Concepts

- **`WHERE` Clause:** Directly used by partial indexes.
- **Expressions in `SELECT`/`WHERE`/`ORDER BY`:** Can be indexed using expression indexes (e.g., `LOWER(column)`, `date_trunc('month', column)`).
- **Data Retrieval (`SELECT` list):** Covering indexes aim to satisfy all selected columns.
- **DDL (`CREATE INDEX`):** These features are options within the `CREATE INDEX` command.
- **Full-Text Search:** Indexing generated `TSVECTOR` columns is an advanced technique.

6.3 How to Use Them: Structures & Syntax

6.3.1 Key Advanced Features

1. **Partial Indexes (`WHERE` clause):** Index only a subset of rows.

```sql
CREATE INDEX idx_orders_active_high_priority
ON Orders (orderDate)
WHERE status = 'Active' AND priority > 5;
```

Listing 41: Partial index for active, high-priority orders

2. **Indexes on Expressions:** Index the result of a function or expression.

```sql
CREATE INDEX idx_employees_email_lower ON Employees (LOWER(email));
CREATE INDEX idx_events_month ON Events (date_trunc('month',
    eventTimestamp));
```

Listing 42: Index on a lowercased email and date truncation

3. **Covering Indexes (`INCLUDE` clause for B-tree):** Store extra non-key columns in the index to allow index-only scans.

```sql
CREATE INDEX idx_employees_dept_covering
ON Employees (departmentId)
INCLUDE (firstName, salary);
-- Useful for: SELECT firstName, salary FROM Employees WHERE
    departmentId = X;
```

Listing 43: Covering index including non-key columns

4. **Concurrent Index Creation (`CONCURRENTLY`):** Build indexes without blocking writes to the table (takes longer, more resource-intensive).

```sql
CREATE INDEX CONCURRENTLY idx_employees_hire_date
ON Employees (hireDate);
```

Listing 44: Creating an index without locking writes

5. **Generated Columns with Indexes:** Define columns whose values are automatically computed, then index them. Especially useful for `TSVECTOR`.

```sql
ALTER TABLE Documents ADD COLUMN tsv TSVECTOR
GENERATED ALWAYS AS (to_tsvector('english', body)) STORED;
CREATE INDEX idx_documents_tsv ON Documents USING GIN (tsv);
```

Listing 45: Indexing a generated TSVECTOR column

6. **Index Storage Parameters (`WITH (...)`):**
   - `FILLFACTOR` (B-tree, Hash): Percentage of page space to fill, leaving room for updates to reduce page splits.
   - `pages_per_range` (BRIN): Number of table blocks summarized by one BRIN index entry.

```sql
CREATE INDEX idx_products_name ON Products (productName) WITH (
    FILLFACTOR = 80);
CREATE INDEX idx_logs_event_time_brin ON Logs USING BRIN (eventTime
    ) WITH (pages\_per\_range = 32);
```

Listing 46: Index with FILLFACTOR and BRIN with pages_per_range

7. **`NULLS NOT DISTINCT` for Unique Indexes (PostgreSQL 15+):** Treat multiple NULLs as non-distinct for unique constraints.

```sql
-- CREATE UNIQUE INDEX idx_optional_code ON Items (optional_code)
    NULLS NOT DISTINCT;
SELECT 'NULLS NOT DISTINCT requires PostgreSQL 15+, placeholder.';
```

Listing 47: Unique index where multiple NULLs are allowed but not distinct

6.4 Why Use Them? (Advantages)

- **Precision Targeting:** Optimize for very specific query patterns that general indexes might not cover efficiently.
- **Reduced Storage and I/O:** Smaller indexes (partial, BRIN) mean less disk space and faster reads.
- **Maximized Index-Only Scans:** Covering indexes can eliminate table heap access, a huge performance win.
- **Operational Flexibility:** `CONCURRENTLY` allows index creation on busy production systems.
- **Improved Write Performance (with `FILLFACTOR`):** Can reduce page splits and fragmentation on heavily updated tables.

6.5 Watch Out! (Disadvantages)

- **Specificity:** Partial indexes and expression indexes are only useful if queries precisely match their definition.
- **Maintenance Overhead:** More indexes, even advanced ones, still add overhead to write operations.

- **Complexity:** Understanding when and how to use these features effectively requires deeper knowledge.
- **CONCURRENTLY Risks:** If it fails, it can leave an invalid index that needs to be dropped. It also takes longer and uses more resources.
- **Generated Columns Costs:** While convenient, generated columns add computation overhead on insert/update (though 'STORED' pre-computes it).
- **Storage Parameter Tuning:** Incorrectly tuning FILLFACTOR or pages_per_range can sometimes harm performance.

7 Subquery Optimization: Taming Queries-Within-Queries!

Subqueries, or inner queries, are powerful SQL tools for breaking down complex logic. However, if not handled well, they can become performance nightmares. Optimizing them is key to efficient SQL.

7.1 What Are They? (Meanings & Values)

7.1.1 Core Meaning

Subquery Optimization involves techniques to ensure that queries embedded within other queries execute efficiently. This can mean rewriting the subquery, ensuring it's properly indexed, or understanding how the database planner transforms or executes it. Subqueries can appear in SELECT lists, FROM clauses (derived tables), and WHERE clauses (e.g., with IN, EXISTS, comparisons).

7.1.2 Typical Values or Outputs

Effective subquery optimization leads to:

- **Significant Query Speedup:** Especially for correlated subqueries.
- **Reduced Repetitive Computation:** Avoiding re-execution of subqueries for each outer row.
- **Clearer Execution Plans:** Making it easier to see how the subquery integrates with the main query.
- **Improved Resource Utilization.**

7.2 Relations: How They Play with Others

7.2.1 Relations within Query Optimization

- **Often Rewritten to Joins:** The PostgreSQL planner is quite good at transforming many types of subqueries (especially uncorrelated ones or those in IN/EXISTS) into equivalent JOIN operations, which are often more efficient.

- **Indexing is Crucial:**
  - For correlated subqueries, indexes on the columns used in the subquery's `WHERE` clause that link to the outer query are vital.
  - For subqueries in `FROM`, if they produce a large intermediate result, further operations on that result benefit from thinking about its structure.
- **`EXPLAIN` Reveals Execution Strategy:** Shows if a subquery is executed as a "SubPlan," "InitPlan" (executed once), or "flattened" into a join.

7.2.2 Relations with SQL Concepts

- **Correlated vs. Uncorrelated Subqueries:**
  - **Uncorrelated:** Can be executed once independently of the outer query. Generally easier to optimize.
  - **Correlated:** References columns from the outer query, potentially executing once for each row of the outer query if not optimized well. These are prime candidates for rewriting.
- **Scalar Subqueries:** Return a single value. Often found in `SELECT` lists or `WHERE` clauses.
- **Subqueries in `FROM` (Derived Tables):** The result set of the subquery is treated like another table.
- **`IN`, `ANY`, `ALL`, `EXISTS`:** Common operators used with subqueries in the `WHERE` clause. PostgreSQL often has specialized optimization strategies for these.
- **CTEs (`WITH` Clause):** Can often replace subqueries, especially derived tables, for better readability and sometimes better planning (though PostgreSQL might "inline" CTEs similarly to subqueries).
- **Lateral Joins (`LATERAL`):** A powerful alternative to certain types of correlated subqueries, allowing a subquery in the `FROM` clause to reference columns from preceding `FROM` items.

7.3 How to Use Them: Structures & Syntax

7.3.1 Optimization Techniques

1. **Rewrite Correlated Subqueries in `SELECT` to Joins:** This is a very common and effective optimization.

```sql
-- Potentially slow if many employees
SELECT e.employeeId, e.firstName,
       (SELECT d.departmentName FROM Departments d WHERE d.departmentId = e.departmentId) AS deptName
FROM Employees e;

-- Rewritten (usually much faster)
SELECT e.employeeId, e.firstName, d.departmentName AS deptName
FROM Employees e
```

```sql
LEFT JOIN Departments d ON e.departmentId = d.departmentId;
```

Listing 48: Correlated scalar subquery to LEFT JOIN

2. **Convert `IN (subquery)` to `EXISTS` or Join:** While PostgreSQL often optimizes these similarly, sometimes one form gives a better plan. Joins are often preferred if the subquery returns many distinct values.

```sql
-- Using IN
SELECT p.projectName
FROM Projects p
WHERE p.projectId IN (SELECT ep.projectId FROM EmployeeProjects ep JOIN Employees e ON ep.employeeId = e.employeeId WHERE e.status = 'Active');

-- Rewritten using JOIN (often clearer and well-optimized)
SELECT DISTINCT p.projectName -- DISTINCT might be needed if one project has multiple active employees
FROM Projects p
JOIN EmployeeProjects ep ON p.projectId = ep.projectId
JOIN Employees e ON ep.employeeId = e.employeeId
WHERE e.status = 'Active';
```

Listing 49: IN subquery to JOIN

3. **Use `LATERAL` Joins for Complex Per-Row Computations:** When a subquery in the `FROM` clause needs to reference columns from a preceding table (like a more flexible correlated subquery).

```sql
-- Get each department and its two highest-paid employees
SELECT d.departmentName, emp_lat.firstName, emp_lat.salary
FROM Departments d
LEFT JOIN LATERAL (
    SELECT e.firstName, e.salary
    FROM Employees e
    WHERE e.departmentId = d.departmentId
    ORDER BY e.salary DESC
    LIMIT 2
) emp_lat ON true;
```

Listing 50: Using LATERAL JOIN to get top N per group

4. **Ensure Subquery Filters are Efficient:** If a subquery has a `WHERE` clause, make sure it can use indexes.

5. **Limit Subquery Output:** If a subquery returns many rows but only a few are needed (e.g., with `EXISTS` or scalar subquery using `LIMIT 1`), ensure it can stop early.

6. **Avoid Subqueries in `ORDER BY` or `GROUP BY` if Possible:** These can be hard for the planner to optimize. Try to move such logic into the main query or a CTE. A scalar subquery can be used in an `ORDER BY` clause. For `GROUP BY`, it's less common and often clearer to use a derived column from the `SELECT` list (which itself could be a scalar subquery) or join to a derived table/CTE.

7. **Use CTEs for Clarity and Potential Reuse:**

```sql
-- FROM subquery (derived table)
SELECT summary.status, COUNT(*)
FROM (SELECT employeeId, status FROM Employees WHERE hireDate > '2022-01-01') AS summary
GROUP BY summary.status;

-- Using a CTE
WITH RecentEmployees AS (
    SELECT employeeId, status FROM Employees WHERE hireDate > '2022-01-01'
)
SELECT status, COUNT(*)
FROM RecentEmployees
GROUP BY status;
```

Listing 51: Replacing a FROM subquery with a CTE

7.4 Why Use Them? (Advantages)

- **Performance Boost:** Transforms slow, complex queries into faster ones, especially critical for OLTP systems.
- **Reduced Database Load:** Efficient subqueries consume fewer resources, benefiting overall system health.
- **Enables Complex Logic:** Subqueries are essential for many SQL patterns; optimizing them makes these patterns viable.
- **Improved Readability (when rewritten well):** Converting obscure subqueries to clear joins or CTEs can make SQL easier to maintain.

7.5 Watch Out! (Disadvantages)

- **Planner Can Be Tricky:** While PostgreSQL's planner is smart, some subquery patterns can still confuse it or lead to suboptimal plans if not written carefully.
- **Correlated Subqueries are Often a Red Flag:** Especially scalar correlated subqueries in the `SELECT` list, if not optimizable by the planner into a join, can be performance killers.
- **Rewriting Risks:** Modifying complex subquery logic can introduce errors if the equivalence to the original query is not maintained. Rigorous testing is essential.
- **'NOT IN (subquery)' Pitfalls with NULLs:** If the subquery can return 'NULL', 'NOT IN' might not behave as expected (it won't match anything). Use 'NOT EXISTS' or ensure the subquery filters out 'NULL's.
- **Over-reliance on Planner Magic:** While the planner is good, don't assume it will always perfectly optimize every subquery. `EXPLAIN ANALYZE` is your friend.

8 Conclusion: Mastering the Performance Puzzle

Optimizing PostgreSQL queries is a multifaceted discipline that blends understanding database internals, smart SQL craftsmanship, and diligent analysis. By mastering indexing strategies (including composite index design and advanced features), effectively using `EXPLAIN` plans (and understanding their bottleneck indicators, see Section 2.3.3), meticulously managing database statistics, tuning analytical functions, and strategically rewriting queries and subqueries, you can transform sluggish database operations into high-speed data delivery engines. Remember that optimization is an iterative process: query, analyze, refine, and repeat. With these tools and techniques, you're well-equipped to make your PostgreSQL databases perform at their peak!