PostgreSQL Sort Methods – Quick Overview

PostgreSQL automatically chooses the **best sort method** based on data size, available memory (work_mem), and statistics.

Sort Method	When Used	Description
Quicksort	**Default & most common** (in-memory)	Fastest. Used when data fits in work_mem. when rows × row_width ≤ work_mem.
Top-N Heapsort	LIMIT queries	Builds a heap of size LIMIT, then replaces — very fast for small result sets.
External Merge Sort	Large data > work_mem	Spills to **temporary files** on disk. Multi-phase merge (like tapesort).
Hash-based (rare)	Not for ORDER BY	Only for DISTINCT, GROUP BY, etc.
Incremental Sort	ORDER BY with multiple columns. Left column(s) are indexed Here only unsorted Right column(s) are sorted	reuses already-sorted input from a previous step (like an index scan)

Join Alogorithms used by PostgreSQL

Core Join Algorithms (Same in v16 & v17)

Join Type	Algorithm	When Used	Notes
Nested Loop	Nested Loop	Small tables, indexed joins	Fastest with index on inner side
Hash Join	Hash	Large tables, equality (=)	Builds hash on smaller table, Efficient even without Index
Merge Join	Merge	Sorted inputs, equality or range	Needs pre-sorted data (index or sort). Use Incremental Sort Algorithm

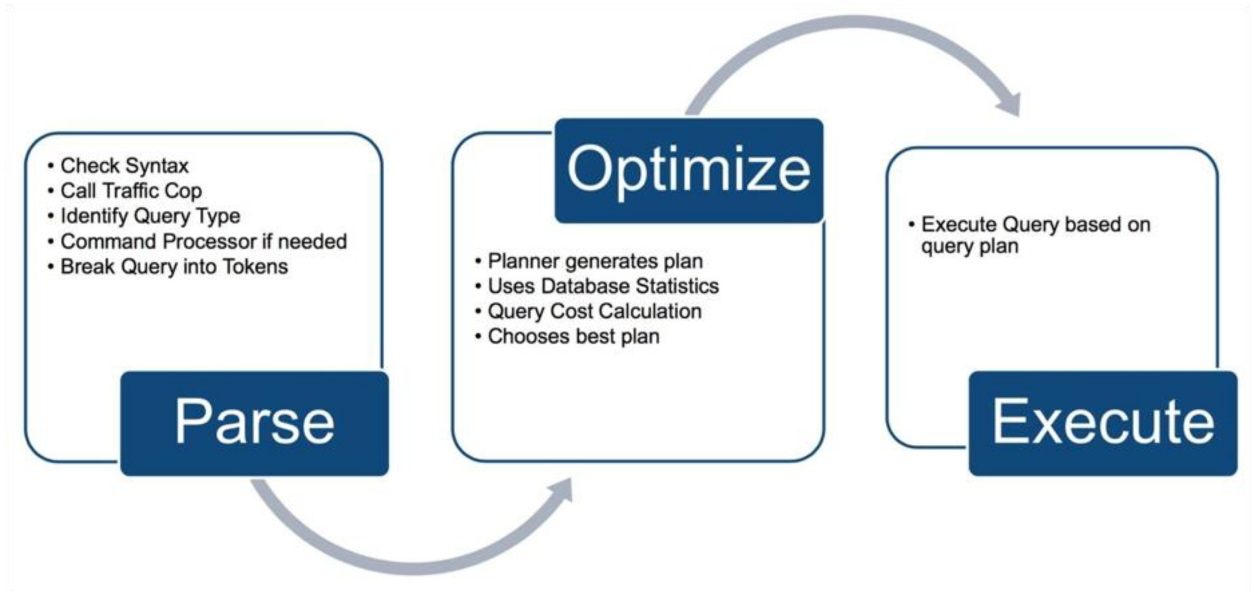
Scenarios when type of Join is efficient

Join Method	Most Efficient When	Why	Typical Cost
Nested Loop	Small inner table + **index on join key**	1 probe per outer row → **O(n)** with index	**Fastest** for small/medium
Hash Join	Large tables + **equality (=)** + one side fits in memory	Build hash → probe → **O(n + m)**	**Best for big equality joins**
Merge Join	Both sides **pre-sorted** (via index) + equality or range	Merge like tapes → **O(n log n)** if sort needed	**Best when data is already sorted**

Query Tuning

Query Tuning Basics

- Statement Processing
- Common Query Performance Issues
- SQL Tuning Goals
- SQL Tuning Steps
  - Identify slow queries
  - Review the query execution plan
  - Optimizer statistics and behavior
  - Restructure SQL statements
  - Indexes
  - Review Final Execution Plan



Common Query Performance Issues

- Full tables scans

- Bad SQL
- Sorts using disk
- Old or missing statistics
- I/O issues
- Bad connection management

SQL Tuning Goals

- Identify bad or slow SQL
- Find the possible performance issue in a query
- Reduce total execution time
- Reduce the resource usage of a query
- Determine most efficient execution plan
- Balance or parallelize the workload

Six Step Query Tuning

1. Identify Slow Queries
2. Review the Query Execution Plan
3. Optimizer Statistics and Behavior
4. Restructure SQL Statements
5. Add / Remove Indexes
6. Review Final Execution Plan

STEP 1: Identify Slow Queries

`log_min_duration_statement`

- This statement sets a minimum statement execution time (in milliseconds) that causes a statement to be logged.

- All SQL statements that run for the time specified or longer will be logged with their duration.
- Enabling this option can be useful in tracking down non-optimized queries in your applications.
- Not all queries shown will be problematic. Some large queries that are already optimized will also be logged. For example, reporting queries, large batch jobs, etc.

STEP 2: Review the Query Execution Plan

- An execution plan shows the detailed steps necessary to execute a SQL statement
- Planner is responsible for generating the execution plan
- The Optimizer determines the most efficient execution plan
- Optimization is cost-based, cost is estimated resource usage for a plan
- Cost estimates rely on accurate table statistics, gathered with **`ANALYZE`**
- Costs also rely on **`seq_page_cost`**, **`random_page_cost`**, and others
- The **`EXPLAIN`** command is used to view a query plan
- **`EXPLAIN ANALYZE`** is used to run the query to get actual runtime stats

Optimizer Statistics

- The PostgreSQL Optimizer and Planner use table statistics for generating query plans
- Choice of query plans are only as good as table stats
- Table statistics
  - Stored in catalog tables like **`pg_class`**, **`pg_stats`** etc.
  - Stores row sampling information such as number of rows, distribution frequency, etc.
- Maintained by **`Autovacuum`**, or can be refreshed manually with **`ANALYZE`**
- Run **`ANALYZE`** after bulk data changes to ensure good plans

Explain Example

```
postgres=# EXPLAIN SELECT * FROM emp;
```

```
                    QUERY PLAN
------------------------------------------------------
Seq Scan on emp     (cost=0.00..1.14 rows=14 width=135)
```

The numbers that are quoted by **EXPLAIN** are:

Cost: **0.00** Estimated startup cost

1.14 Estimated total cost (This is the number you should aim to minimize)

Rows: **14** Estimated number of rows output by this plan node

Width: **135** Estimated average width (in bytes) of rows output by this plan node

Seq Scan on emp: tells that sequential scan method is used.

If query uses multiple tables then the above information is provided for each table separately.

Reviewing Explain Plans

- Examine the various costs at different levels in the explain plan
- Look for sequential scans on large tables
- All sequential scans are not inefficient
- Check whether a join type is appropriate for the number of rows returned
- Check for indexes used in the explain plan
- Examine the cost of sorting and aggregations
- Review whether views are used efficiently

STEP 3: Optimizer Statistics and Behavior

Table Statistics

- Absolutely critical to have accurate statistics to ensure optimal query plans
- Are not updated in real time, so should be manually updated after bulk operations
- Can be updated using **ANALYZE** command or OS command **vacuumdb** with **-Z** option
- Stored in **pg_class** and **pg_statistics**

- You can run the **`ANALYZE`** command from psql on specific tables and just specific columns
- **`Autovacuum`** will run **`ANALYZE`** as configured

```
Syntax for ANALYZE
– ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

- Planner relies on Statistics to estimate the rows retrieved by query
- Absolutely necessary to have accurate statistics
- **`pg_class`** tables **`relpages & reltuples`** provides estimates of total entries
- Information provides by **`relpages & reltuples`** is slightly outdated as they are not calculated on fly
- Most queries retrieves only a fraction of total rows due to where clause
- For queries with where clause row estimation depends on **`pg_statistics`**
- The **`pg_statistics`** stores statistics on per columns basis
- Sampling depends on **`default_statistics_target`** configuration parameter (default is 100)
- Can be changed by

```
– ALTER TABLE tablename ALTER COLUMN columnname SET STATISTICS N;
Where N is a number between 100 and 10000
```

Example:

```
ALTER TABLE address ALTER COLUMN phone SET STATISTICS 500;
```

In Above example for phone columns of address table sampling is set to 500.

Extended Statistics

- Planner estimates gets wrong for queries with multiple columns in where clause
- Planner assumes that all columns used in where clause are Independent
- More often columns are correlated
- In such cases we have to use multivariate statistics to help planner
- multivariate stats are not computed automatically but needs to create statistics object

- First identify the most frequent queries with multi column where clause or order by clause
- Then create statistics object for such columns

`CREATE STATISTICS` creates the statics objects

```
– CREATE STATISTICS statistics_name
    [ ( statistics_kind ) ]
    ON column_name, column_name [, ...]
    FROM table_name
```

Statistics kind is either `dependencies` or `ndistinct`

- Once the object is created `ANALYZE / VACUUM` computes the stats
- `pg_statistic_ext` table stores the Extended statistics

Functional Dependencies

- Functional Dependencies exists on Primary Key or Superkey
- However not all databases are 100% Normalized
- Without knowledge of functional dependencies Query planner treats both columns independently which results in inaccurate estimates
- Wherever if such functionally dependent columns are used in query's where clause it is better to create statistics object
- CREATE STATISTICS creates the statics objects

```
– CREATE STATISTICS statistics_name
    (dependencies)
    ON column_name, column_name [, ...]
    FROM table_name
```

Example:

```
CREATE STATISTICS mystat1(dependencies) ON city, country_id FROM city;
```

Run `ANALYZE` immediately

- Functional dependencies are not used for range / like operators

ndistinct

- Its Normal for order by or Group by queries to use multiple columns
- Planner may not get accurate statistics because it examines all involved columns individually
- Such queries need to n-distinct statistics for group of columns

- CREATE STATISTICS creates the statics objects

```
CREATE STATISTICS statistics_name  (ndistinct)  ON column_name, column_name [, ...]
FROM table_name;
```

Example:

```
CREATE STATISTICS mystat2(ndistinct)ON first_name, last_name FROM actor;
```

Run **ANALYZE** immediately

Cost Estimation

- Query Cost Depends many parameters
- Primarily query cost depends on Total rows and total pages in table that can found from `pg_class`

```
select relpages,reltuples from pg_class where relname='city';
```

- Also query cost depends on following parameters
- Also query cost depends on following parameters

```
Name                |Setting  | Description
--------------------+---------+------------------------------------
random_page_cost    | 4       | Cost of reading a page using index
seq_page_cost       | 1       | Cost of reading a page sequentially
cpu_index_tuple_cost| 0.005   | Cost of reading a row using Index
cpu_operator_cost   | 0.0025  |Cost of reading a row with WHERE clause
cpu_tuple_cost      | 0.01    | Cost of processing each row
parallel_setup_cost | 1000    | Cost of setting up of parallel worker
parallel_tuple_cost | 0.1     | Cost of processing a row parallelly
```

Example1:

```
explain select * from city;
                     QUERY PLAN
--------------------------------------------------------
 Seq Scan on city  (cost=0.00..11.00 rows=600 width=23)
(1 row)

It is sequential Scan Cost estimate is as follows
select relpages, reltuples from pg_class where relname='city';
 relpages | reltuples
```

```
----------+-----------
        5 |       600
Cost =  (relpages * seq_page_cost) + (Estimate Rows * cpu_tuple_cost  )
==> ((5*1)+(600*0.01)) = 11.00
```

Example2:

```
explain select * from city where city like 's%';
                      QUERY PLAN
-----------------------------------------------------
 Seq Scan on city  (cost=0.00..12.50 rows=1 width=23)
   Filter: ((city)::text ~~ 's%'::text)

It is a sequential Scan Cost and estimate is as follows
select relpages, reltuples from pg_class where relname='city';
 relpages | reltuples
----------+-----------
        5 |       600
Cost =  (relpages * seq_page_cost) + (Estimate Rows * cpu_tuple_cost  ) + (Estimated
Rows * cpu_operator_cost)
==> ((5*1)+(600*0.01)+(600*0.0025)) = 12.50
```

Tweaking Cost Parameters

- Default values work optimally for most of the workloads
- If changed with same proportion then there is no impact of such change
- `random_page_cost` is always 4 times of `seq_page_cost` as CPU has to read INDEX as well
- However due to less pages overall cost decreases significantly
- Decreasing `random_page_cost` makes planner to consider use of indexes
- Increasing `random_page_cost` will make planner to think that the indexes are more expensive
- Run `ANALYZE` immediately
- If database is 100% cached then set `seq_page_cost` and `random_page_cost` to `1`
- If database is heavily cached then set `seq_page_cost` and `random_page_cost` nearer to respective CPU costs parameters
- Custom values for `seq_page_cost` and `random_page_cost` can be set for individual tablespaces

- If the storage is SSD then `random_page_cost` can be set nearer to `seq_page_cost`

Parallel Query Execution

- By default query use one process to execute
- Parallel queries uses more than one process to improve performance
- Planner does not use parallel plan for write queries or any query which locks data
- In Isolation level serializable planner ignore parallelism
- Number of processes limited by `max_parallel_workers_per_gather`

`max_parallel_workers_per_gather` is limited by `max_worker_processes` and `max_parallel_workers.`

- Also the availability of background workers.
- In such case `max_worker_processes` and `max_parallel_workers` cab be increased

STEP 4: Restructuring SQL Statements

- Rewriting inefficient SQL is often easier than repairing it
- Avoid implicit type conversion
- Avoid expressions as the optimizer may ignore the indexes on such columns
- Use equi-joins wherever possible to improve SQL efficiency
- Try changing the access path and join orders
- Avoid full table scans if using an index is more efficient
- The join order can have a significant effect on performance
- Use views or materialized views for complex queries

Restructure SQL Statements - Join Order

- Join Order is often determined by Planner
- Sometimes Planner may not select proper join order resulting in cartesian products
- Explicitly use JOIN clause to determine join Order

- Force planner to use join order by setting `join_collapse_limit` to 1

Restructure SQL Statements - Sub Queries

- Sub Queries are often merged with outer query where ever possible
- This results in better sub query performance
- Planner merges the sub queries with outer query if the resulting FROM clause has less than `from_collapse_limit`
- To Control the behavior of Planner, adjust these settings

STEP 5: Add / Remove Indexes

- Create indexes as and when needed
- Remove unused indexes
- Adding an index for one SQL can slow down other queries
- Verify index usage using the EXPLAIN command

STEP 6: Review Final Execution Plan

- Check again for missing indexes
- Check table statistics are showing correct estimates
- Check large table sequential scans
- Compare the cost of first and final plans

Vacuum Cost

DBA can control vacuum behavior by setting various vacuum related parameters. These parameters fine tune vacuum process to avoid any conflicts.

`vacuum_cost_page_hit` : Cost if the page to be vacuumed found in **`shared_buffer`**

`vacuum_cost_page_miss` : Cost if the page to be vacuumed has to read from Disk

`vacuum_cost_page_dirty` : Cost of actually vacuuming the page

`vacuum_cost_limit` : When this much cost reached while vacuuming, vacuum process seeps for a time
`vacuum_cost_delay` duration

vacuum_cost_delay : Vacuum sleep duration in case it has reached **vcacuum_cost_limit**

Autovacuum Optimization

Autovacuum performs 3 different maintenance task namely

1. **Vacuum** (Marking dead tuples as free space)
2. **Analyzing** (Recalculate Optimizer Statistics)
3. **Freezing** (Freezing transaction IDs to avoid Wrap-Around problem)

Autovacuum is very important maintenance activity within PostgreSQL and it needs to be optimized to give better performance and avoid any of performance bottlenecks which may arise due the maintenance activities it is performing.

To control the behavior of autovaccum following thresholds are set and can be modified by DBAs

Controlling Autovacuum

autovacuum : ON / OFF enables or disables the auto vacuum. Default is on

autovacuum_naptime : Time delay to run auto vacuum. default is 60s. After every nap time auto vacuum makes a round in each database identifying the eligible table for vacuuming.

autovacuum_max_workers : Sets the number of worker process which in turn performs actual vacuum or other maintenance activity.

log_autovacuum_min_duration : Specifies the a time duration. If the auto vacuum is running this much time or more causes it write in log. A DBA can analyze and identify issue of long running Autovacuum actions.

autovacuum_work_mem : This is the amount of memory allocated for each Autovacuum worker process for performing vacuuming. by default it is set to same as **Maintenance_work_mem (64MB).**

autovacuum_vacuum_cost_limit : if the total cost of performing various activities reaches this much Autovacuum sleeps and resumes the operations after sleep.

autovacuum_vacuum_cost_delay : If the total cost of permitting various activities reached **autovacuum_vacuum_cost_limit** then auto vacuum sleeps for this much time specified in milliseconds.

Controlling Auto vacuum at table level

```
ALTER TABLE your_table_name SET (autovacuum_enabled = off);
ALTER TABLE your_table_name SET (autovacuum_enabled = on);
```

Check if Currently disabled or enabled

```
SELECT schemaname, tablename,  reloptions, pg_table_size(relid)
FROM pg_tables t JOIN pg_class c ON c.relname = t.tablename
WHERE tablename = 'your_table_name';
```

Set thresholds

```
ALTER TABLE your_table_name SET (
    autovacuum_vacuum_threshold = 100000,
    autovacuum_vacuum_scale_factor = 0.5
);
```

Throttling Autovaccum Vacuum process :

`autovacuum_vacuum_threshold` : Minimum No of updates and deletes required on the table for auto vacuum.

`autovacuum_vacuum_scale_factor` : Specifies the percentage of table to added `autovacuum_cacuum_threshold` to trigger auto vacuum on a table.

Throttling Autovacuum Analyze process :

`autovacuum_analyze_threshold` : Minimum No of updates and deletes required on the table for auto analyze.

`autovacuum_analyze_scale_factor` : Specifies the percentage of table added `autovacuum_cacuum_threshold` to trigger auto analyze on a table.

Throttling Autovacuum Freezing process :

`autovacuum_freeze_max_age` : Perform freeze operations after this many transactions has passed on a row.

PostgreSQL performs freezing operations using auto vacuum even if the auto vacuum is disabled.

pg_hint_plan

In PostgreSQL, planner (optimizer) cost estimations for query plans are based on statistics. Plan are generated when the query is executed by the session and these plans are not available for other session.
Optimizer selects the query execution plan which is lowest in cost. Most of the time the execution plan selected by the optimizer is the best one but not perfect
Pg_hint_plan allows developers to pass on hints to the planner using special comments to force certain indexes, join order etc.

Installation Pre Requisites

- gcc compiler and make tool

```
dbadmin@dbserver:~$ sudo apt install gcc
dbadmin@dbserver:~$ sudo apt install make
```

- Postgresql development package

```
dbadmin@dbserver:~$ sudo apt install postgresql-
server-dev-14
```

Download pg_hints_plan software

```
dbadmin@dbserver:~$ wget https://github.com/ossc-
db/pg_hint_plan/archive/refs/heads/PG14.zip
```

Installation

```
dbadmin@dbserver:~$  unzip  PG14.zip
dbadmin@dbserver:~$ cd  pg_hint_plan-PG14
dbadmin@dbserver:~$ make
dbadmin@dbserver:~$ sudo su
dbadmin@dbserver:~$ make install
```

Usage

Pg_hint_plan uses special comment to pass the hints to the planner. These comments are used before query.

```
/*+
    hints
*/  Query
```

Scan method hints

There are 2 Scan Method Hints

```
            IndexScan(tablealias  [indexname])
            SeqScan(tablealias)
```
Example:
```
            /*+
                 IndexScan(e)

            */
            explain select * from emp e where ename = 'SMITH';
```

Join Method hints

There are 2 join method hints.
```
            NestLoop(tablealias tablealias)
            MergeJoin(tablealias tablealias tablealias)
```

Joining order hints

There is one join order hint
```
            Leading(tablealias tablealias tablealias)
```

Row number correction hints

Due to incorrect statistics planner's row estimates may be incorrect a result set. This hint allows us to correct it by specifying

1. Specify exact Number of rows
2. By adding
3. By Subtracting
4. By Multiplying

Example:
```
postgres=# /*+ Rows(a b #10) */ SELECT... ; Sets rows of join
result to 10
postgres=# /*+ Rows(a b +10) */ SELECT... ; Increments row
number by 10
postgres=# /*+ Rows(a b -10) */ SELECT... ; Subtracts 10 from
the row number.
postgres=# /*+ Rows(a b *10) */ SELECT... ; Makes the number 10
times larger.
```

Grand Unified Configuration (GUC)

Sets GUC parameter values during the target statement is under planning. GUC variables are PostgreSQL Configuration parameters like seq_page_cost etc.
Example:
```
postgres=# /*+
postgres*#     Set(random_page_cost 2.0)
postgres*#  */
postgres-# SELECT * FROM emp e WHERE ename = 'SMITH';
```

JIT in PostgreSQL

Just-In-Time Compilation – compiles SQL expressions (WHERE, projections, aggregates) to machine code at runtime for faster execution. Enabled By Default (since PG 11)

When JIT Helps

Scenario	**Benefit**
Complex WHERE clauses	Faster filtering
Heavy aggregations (SUM, AVG)	Faster math
Large SELECT with expressions	Faster row processing
CPU-bound queries (> 1 sec)	Biggest win

Parameters

Parameter	**Default**	**Purpose**
jit	on	Enable/disable JIT
jit_above_cost	100000	Only use JIT if query cost > this
jit_inline_above_cost	500000	Inline small expressions
jit_optimize_above_cost	500000	Optimize LLVM code

Pros & Cons

Pros	**Cons**
2x–10x faster on CPU-heavy queries	**Slight overhead** on small queries
LLVM-based	Compilation delay (~1–5 ms)
Auto-tuned	Uses more memory