PostgreSQL Query Execution Process

CATs
Lauro Ojeda
Paula Berenguel

PostgreSQL Planner

Understanding Query Execution Process

Foundation of Optimization

Understanding how the database optimizer chooses access paths and uses indexes.

Effective Tuning Strategies

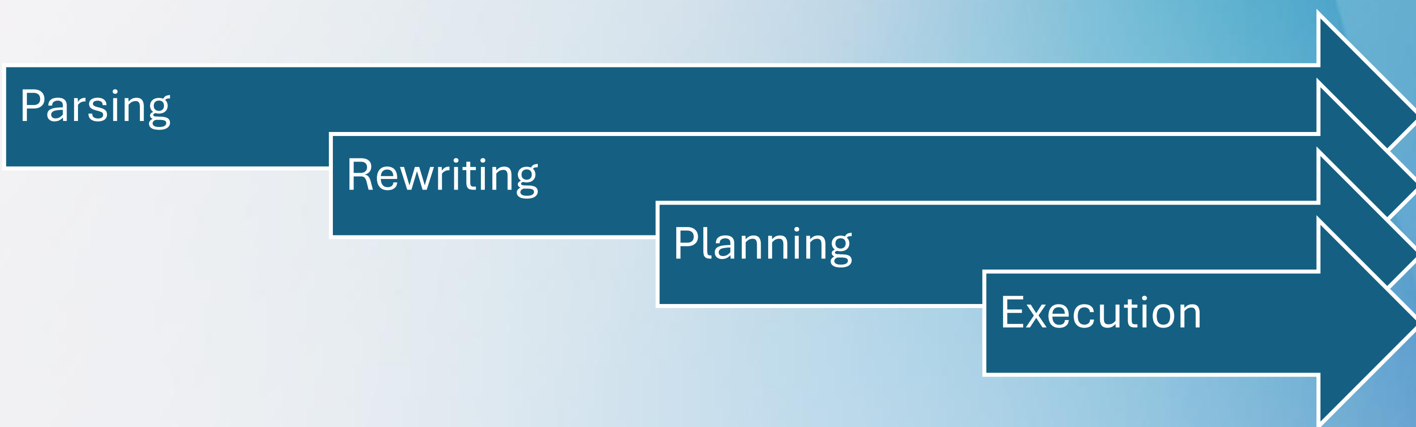
Applying precise optimizations based on the execution process rather than guesswork.

Identifying Bottlenecks

Gaining insights into data processing to pinpoint performance issues.

Query Execution in a nutshell

- Client sends SQL query
- The planner **works out the fastest path** to fetch data
- Executor **pulls data** from shared buffers or disk
- Results streamed back to client



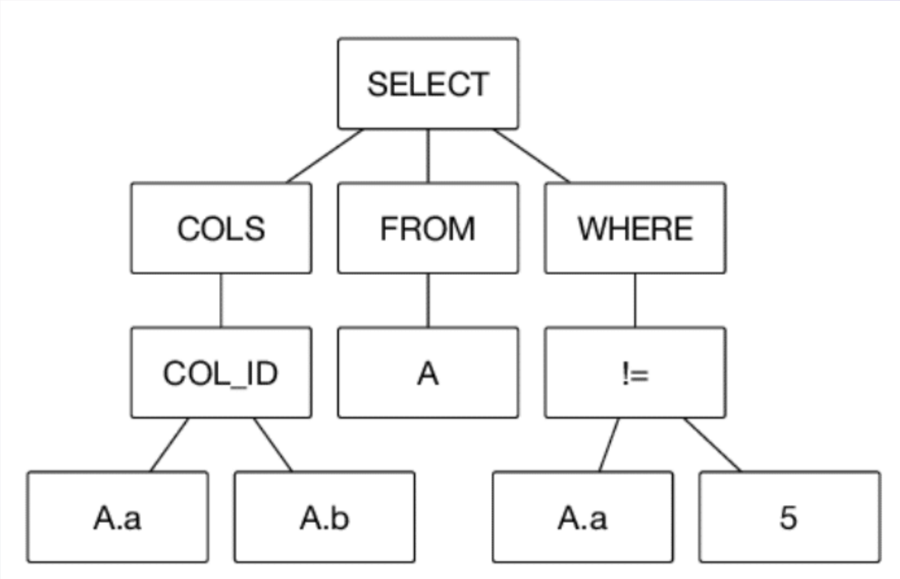
Step 1: Client Issues Query

- Query arrives via PostgreSQL wire protocol (libpq)
- Backend process is assigned to the connection
- Query enters "frontend/backend" communication pipeline

Step 2: Parsing & Syntax Checking

Microsoft

- Parser validates SQL grammar and structure
- SQL is transformed into an initial parse tree



- Semantic checks: objects, types, privileges
- Errors at this stage: syntax errors, unknown identifiers

Step 3: Query Rewrite Phase

- Rule system rewrites parse tree (views, rules)
- Produces logical query tree prior to planning
- Ensures logical correctness before planning

Step 4: Query Planner — Overview

Definition: Cost-based component that chooses the cheapest valid execution strategy for a query using statistics and path exploration

- Converts query tree into an efficient execution plan
- Uses a cost-based optimization approach
- Generates multiple possible strategies ("paths")
  - For each path, estimates CPU cost, I/O cost, row counts
  - Identifies the JOIN strategy possibilities and estimate their cost

Planner Mechanics: Cost-Based Optimization

- Cost model estimates the cost for:
  - Seq scans
  - Index scans
  - Joins (nested loop, hash join, merge join)
  - Sorts and aggregation
- Based on statistics (pg_stats): histograms, n_distinct, correlation
- Uses sampling-based planning for large tables

Planner: Plan Generation & Path Selection

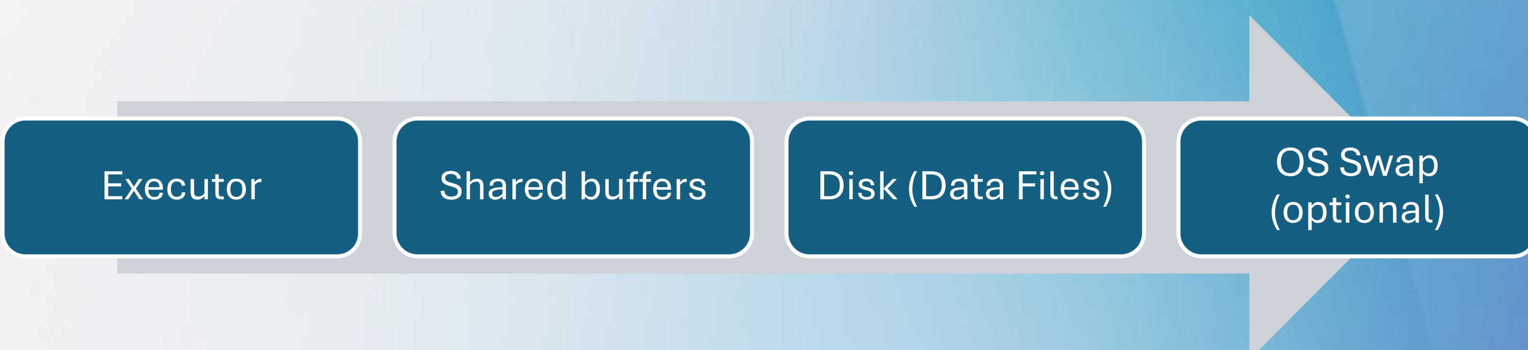
- Evaluates multiple possible join orders
- Uses dynamic programming for join order optimization
- Chooses cheapest path based on total estimated cost
- Final output: Plan Tree, including scan nodes, join nodes, filters

Step 5: The Executor — Overview

- Executor receives the plan tree from the planner
- Walks the tree node by node (bottom to the top)
- Executes each node: scan → filter → join → aggregate
- Returns tuples back up through parent nodes

How PostgreSQL Fetches Data (Disk & Memory)

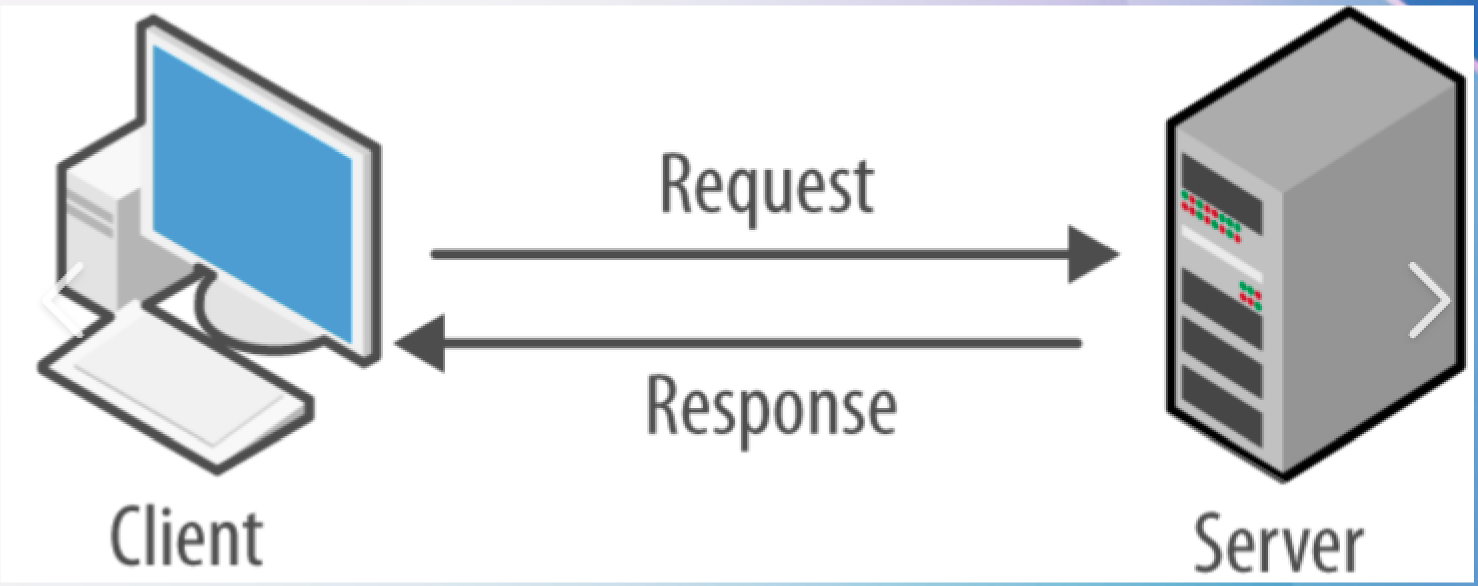
- Executor requests data
  - Buffer Manager checks Shared Buffers if data is in memory
  - If miss: brings data from disk into buffers
- If memory pressure: OS may swap least-used pages



Buffer Manager & Shared Buffers

- Shared Buffers = main PostgreSQL memory cache
- All data reads/writes pass through shared buffers
- Background writer flushes dirty pages
- Checkpoint ensures data persistence to disk
- Eviction policies manage cache under memory pressure

Data which is ready, goes back to the client



Interpreting the Execution Plan

What’s an Execution Plan?

An execution plan describes how the database runs a SQL query, including the order of operations, access methods, and join algorithms, and understanding it helps identify performance issues such as inefficient joins, incorrect row estimates, or missing indexes so you can optimize the query or schema to improve speed and resource usage.

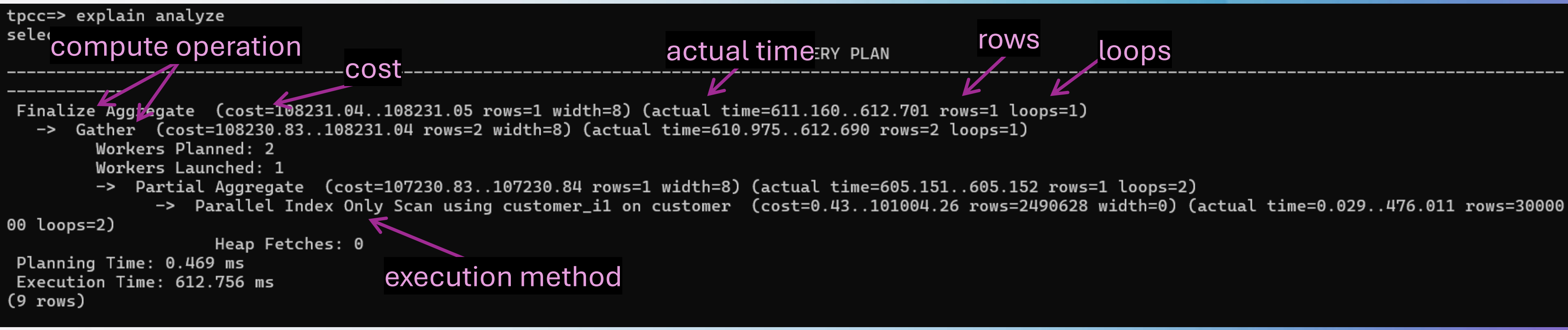
Execution Plan components 1

- PostgreSQL plan is a hierarchical tree of nodes
- Each node produces tuples for its parent node
- Each node can be identified by the right arrow in the EP
- Types of nodes:
  - Compute operations: Gather, Sort, Aggregate, Hash
  - Execution methods: Sequential Scan, Index Scan, Index Only Scan, Bitmap Heap/Index Scans,
  - Join Nodes

```
tpcc=> explain analyze
select count(*) from customer;
                                                                               QUERY PLAN
---------------------------------------------------------------------------------------------------------
------------
 Finalize Aggregate  (cost=108231.04..108231.05 rows=1 width=8) (actual time=611.160..612.701 rows=1 l
   ->  Gather  (cost=108230.83..108231.04 rows=2 width=8) (actual time=610.975..612.690 rows=2 loops=1
         Workers Planned: 2
         Workers Launched: 1
         ->  Partial Aggregate  (cost=107230.83..107230.84 rows=1 width=8) (actual time=605.151..605.1
               ->  Parallel Index Only Scan using customer_i1 on customer  (cost=0.43..101004.26 rows=
00 loops=2)
                     Heap Fetches: 0
 Planning Time: 0.469 ms
 Execution Time: 612.756 ms
(9 rows)
```

Execution Plan components 2

- **compute operation**: type of compute work necessary to merge, aggregate, filter, sort, etc
- **execution method**: how Postgres will fetch the data in the relation
- **cost**: statistics computed cost of each operation (startup..total cost)
- **actual time**: the time spent by the operation (startup..total time)
- **rows**: amount of rows returned by the node to the parent node
- **loops**: how many iterations the operation made to fetch rows=? rows



Execution Plan components 3

Operations summary

- **Planning time**: time spent by the planner to workout multiple execution plans until deciding which of the evaluated plans is considered optimal
- **Execution time**: The actual total time spent in both planning and executing the query.
- **Transfer time**: not listed in the EP as dependant to the network settings. It is the time the data is returned to the client

```
tpcc=> explain analyze
select count(*) from customer;
                                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
------------
 Finalize Aggregate  (cost=108231.04..108231.05 rows=1 width=8) (actual time=611.160..612.701 rows=1 loops=1)
   ->  Gather  (cost=108230.83..108231.04 rows=2 width=8) (actual time=610.975..612.690 rows=2 loops=1)
         Workers Planned: 2
         Workers Launched: 1
         ->  Partial Aggregate  (cost=107230.83..107230.84 rows=1 width=8) (actual time=605.151..605.152 rows=1 loops=2)
               ->  Parallel Index Only Scan using customer_i1 on customer  (cost=0.43..101004.26 rows=2490628 width=0) (actual time=0.029..476.011 rows=30000
00 loops=2)
                     Heap Fetches: 0
 Planning Time: 0.469 ms
 Execution Time: 612.756 ms
(9 rows)
```

Microsoft

How to Interpret EXPLAIN / EXPLAIN ANALYZE

- **EXPLAIN** shows the chosen execution plan
- **EXPLAIN ANALYZE** shows real execution stats
  - It will execute the query but will suppress data output
  - Displays the planning and execution time
- **EXPLAIN (ANALYZE, BUFFERS)** shows the above PLUS memory and disk buffers reads
- Real vs estimated analysis reveals planning accuracy
- Evaluate the time spent per each node; start working towards reducing the actual time spent per node
  - Usually, I go straight to the node which takes the longest
- **Tip:** multiply the actual time per node with the loops

Common relations operations

- **Sequential Scan** → reads entire table
- **Index Scan** → uses B-tree index for targeted reads
- **Bitmap Scan** → efficient for many scattered hits
- **Nested Loop** → good for small input sets
- **Hash Join** → best for large joins, requires memory
- **Merge Join** → sorted inputs, efficient for ordered data

Examples

-- sequential scan

```
explain analyze
select * from customer;
```

-- aggregation

```
explain analyze
select count(*) from customer;
```

-- index scan

```
explain analyze
select * from customer where c_id=1000;
```

-- join

```
explain (buffers, analyze)
select c.c_first||' '||c.c_last customer, c.c_balance, h.h_date, h.h_amount, h.h_data
from customer c join history h on (c.c_id=h.h_c_id and c.c_d_id=h.h_c_d_id and
c.c_w_id=h.h_c_w_id)
where c.c_id=1000 and c.c_d_id=6 and c.c_w_id=66;
```

Example – tuning in action 1

The common mistake. Query executing 1000's times per min.

```
-- EP execution time ~ 250ms
explain (buffers, analyze)
select c.c_first||' '||c.c_last customer, c.c_balance, h.h_date, h.h_amount, h.h_data
from customer c join history h on (c.c_id=h.h_c_id and c.c_d_id=h.h_c_d_id and
c.c_w_id=h.h_c_w_id)
where c.c_id=1000 and c.c_d_id=6 and c.c_w_id=66;
```

Check the amount of data returned by the query: if too much data, reduce the amount of data returned

Look into table's definitions to see if there's something missing

What's the solution?

```
create index history_i1 on history (h_c_w_id, h_c_d_id, h_c_id);
```

Example – tuning in action 2

Disk spillage issue.

```
-- EP execution time ~ 900ms
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT c.C_ID, c.C_LAST, ol.OL_I_ID, ol.OL_AMOUNT, ol.OL_DIST_INFO, c.C_DATA
FROM
  ORDER_LINE ol
    JOIN CUSTOMER c ON (ol.OL_W_ID = c.C_W_ID AND ol.OL_D_ID = c.C_D_ID)
WHERE
  ol.OL_W_ID = 1 AND ol.OL_D_ID = 1 AND c.C_ID <= 5
ORDER BY c.C_LAST desc;
```

Look at sorting node…

What's the solution?

Increase work_mem

Disk is getting better – disks are almost as fast as memory

Example – tuning in action 3

The column type conversion

```
-- EP execution time ~ 180ms
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT o_id, o_c_id, o_d_id, o_w_id, o_entry_d
FROM orders
WHERE o_c_id = 3000
  AND o_w_id::text = '1'
  AND o_d_id = 1
ORDER BY o_id;
```

Look at tables definition…

Why isn’t it using the index?

What’s the solution?

Casting is not needed, and its avoiding the index to be used.

TIPS!

Microsoft

- Good STATISTICS are essential for generating good plans
- In rare cases one may need to influence the optimizer by using query hints (unusual)
- The page explain.dalibo.com is excellent for visual explain plan!
- Focus in reducing time on the most expensive query nodes first
- Duplicated indexes can create a real performance nightmare
- Index is GREAT, but DON'T OVER-INDEX!

Summary & Key Takeaways

- Multi-phase pipeline: parse → rewrite → plan → execute
- Planner is statistics-driven and cost-based
- Executor accesses data via shared buffers, heap, indexes and than disk
- Use EXPLAIN/ANALYZE to diagnose and tune performance
- Understanding execution plans is essential for performance tuning

THANK YOU!