

Query Execution Planning and Execution



Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.com

Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Common Approaches to Optimization

Simple Protocol and Query Processing Phases

Extended Protocol

More on planning

Parameter Configuration

- Tuning for the current workload
- Global impact on the entire system
- monitoring

Query performance tuning

- Reducing workload
- Localized impact (a single query or multiple queries)
- Profiling

This course focuses on query optimization. Generally speaking, optimization is a broad concept; it's important to consider it during the system design and architecture selection stage. We will only discuss the tasks executed during the operation of an existing application.

Two main approaches can be identified. The first approach is about monitoring the system's state and making sure it can handle the existing workload. To achieve this, you can tune DBMS parameters (the key ones covered in the DBA2 course and partially in this course) and also configure the operating system. If the settings don't help, the only remaining option with this approach is to upgrade hardware (which doesn't always work).

Another approach, which we will mainly discuss next, involves not adapting to the workload but rather reducing it. The "productive" workload consists of queries. If a bottleneck can be identified, you can attempt to influence the query's execution in some way to achieve the same result with fewer resources. This approach is more targeted (affecting specific queries or a group of queries), but reducing the workload positively impacts the entire system's performance.

We'll begin by exploring query execution mechanisms in depth, then move on to discussing how to detect inefficient operations and practical approaches for optimization.

Phases of Query Execution



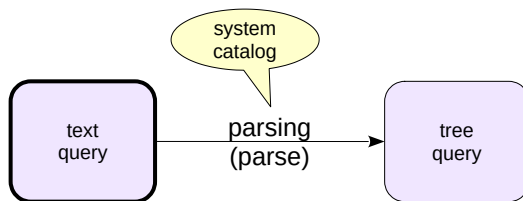
Parsing

Query Rewriting (Transformation)

Query Planning (Optimization)

Execution

We'll start by examining how a query is executed in a straightforward scenario—for instance, when you issue a `SELECT` command in `psql`.



The processing of a regular query is carried out in multiple stages.

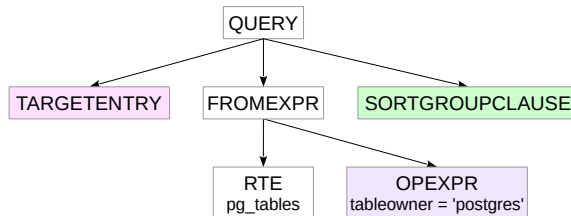
First, the query is parsed (parse).

The first step is syntax analysis, where the query text is converted into a tree structure — this makes it easier to work with.

Next, semantic analysis (parse analysis) occurs, during which the query's referenced database objects are identified, and the user's access to them is verified (the parser consults the system catalog for this).

Syntax Parsing

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



Let's take a look at a simple example: the query shown on the slide.

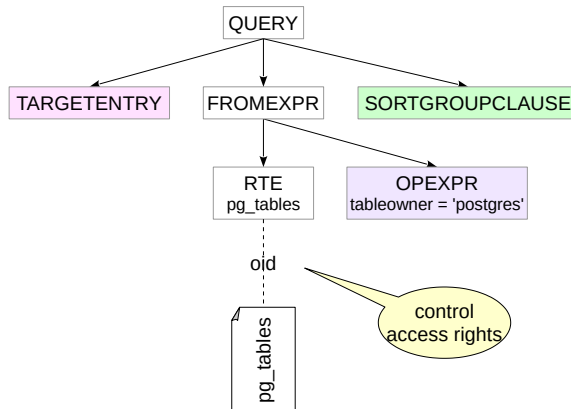
During the parsing phase, a tree is constructed in the backend process's memory, as illustrated in the simplified diagram below the query. Color indicates the approximate mapping between parts of the query text and tree nodes.

RTE is a non-obvious abbreviation for Range Table Entry. In PostgreSQL, this term refers to tables, subqueries, and join results — essentially sets of rows that SQL can manipulate.)

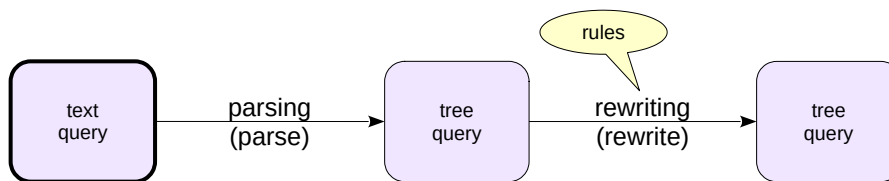
For those curious, the actual parse tree can be examined by setting the `debug_print_parse` parameter and checking the server log. There's no practical value in this (unless, of course, you're a PostgreSQL kernel developer).

Semantic Parsing

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



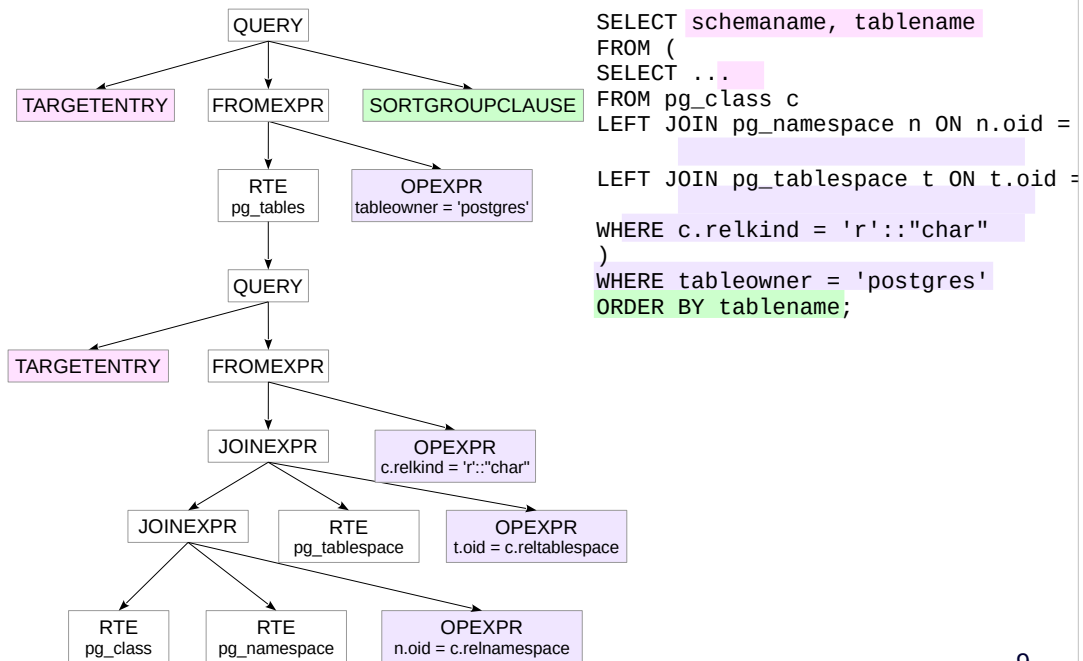
During semantic analysis, the parser consults the system catalog to associate the name "pg_tables" with a view that has a specific object identifier (OID) within the system catalog. Access permissions for this view will be verified as well.



Secondly, the query is rewritten or transformed (rewrite) according to the rules>

An important special case of rewriting is substituting the query text in place of the view name. Keep in mind that the view text must be parsed again, so we simplify by stating that the first two stages follow each other.

Rewriting

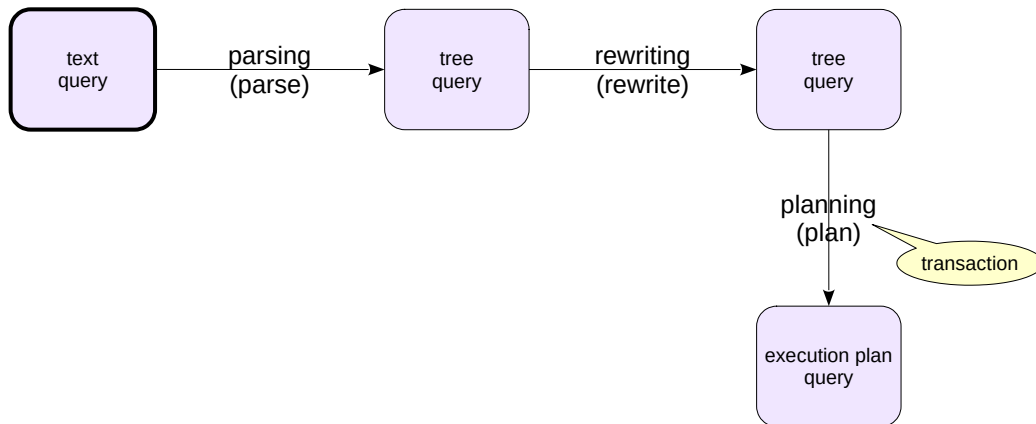


9

The slide presents a query with the view's text included (this is an abstraction: the query in this form does not actually exist — all rewriting is performed on the query tree).

The parent node of the subtree associated with the subquery is the node that references this view. The figure clearly shows the tree structure of the query within this subtree.

The rewritten query tree can be seen in the server log by enabling the `debug_print_rewritten` parameter.



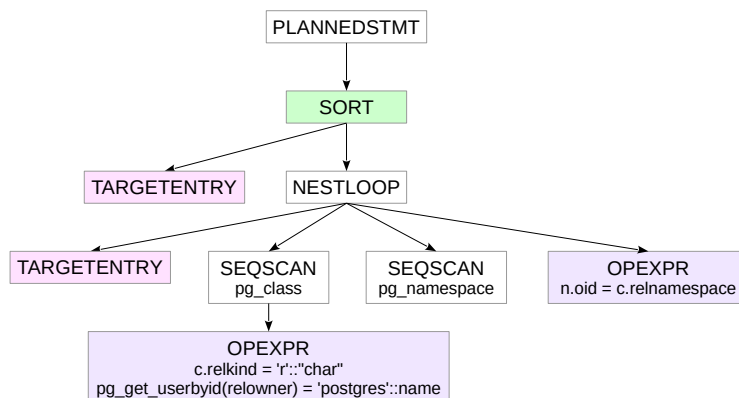
Thirdly, the query is optimized (plan).

SQL is a declarative language, so a single query can be executed in various ways. The planner (also known as the optimizer) considers different execution approaches and evaluates them. The evaluation is based on a mathematical model that uses statistics about the data being processed.

The execution approach with the lowest estimated cost is represented as an execution plan.

Planning

```
Sort (cost=19.59..19.59 rows=1 width=128)
Sort Key: c.relname
-> Nested Loop Left Join (cost=0.00..19.58 rows=1 width=128)
Join Filter: (n.oid = c.relnamespace)
-> Seq Scan on pg_class c (cost=0.00..18.44 rows=1 width=72)
Filter: ((relkind = 'r'::"char") AND
(pg_get_userbyid(reowner) = 'postgres'::name))
-> Seq Scan on pg_namespace n (cost=0.00..1.06 rows=6 width=68)
```



11

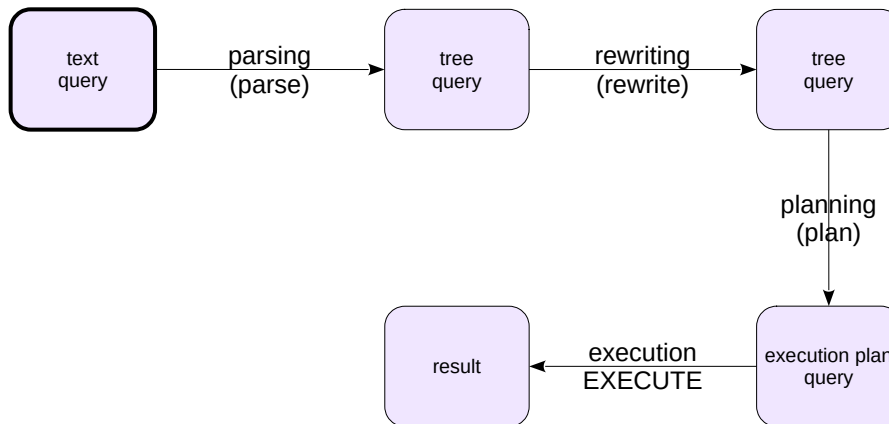
The slide shows an example of an execution plan, demonstrating how the query is executed.

Here, the Seq Scan steps involve reading the relevant tables, while the Nested Loop represents the method for joining two tables. The slide displays the execution plan in the format shown by the EXPLAIN command. We will discuss data access methods, join methods, and the EXPLAIN command in more detail in the next sections.

For now, it's worth noting two key points:

- Out of the three tables, only two remain: the planner determined that one table wasn't necessary for the result and could be safely removed from the execution plan.
- Each node in the tree includes information about the estimated number of rows (rows) and the cost (cost).

For those interested, the actual execution plan can be viewed by setting the debug_print_plan parameter.



Fourthly, the query is executed (execute) according to the selected plan, with the result returned to the client.

Setting the `log_parser_stats`, `log_planner_stats`, and `log_executor_stats` parameters to 'on' enables detailed statistics for each stage to be logged. But in practice, it's usually not required.

<https://postgrespro.com/docs/postgresql/16/query-path>

Simple Protocol

The Simple Query Protocol is used when a command is sent to the server and, The Simple Query Protocol is used when a command is sent to the server, and if it's a SELECT or a command with the RETURNING clause, we expect to receive all result rows. If it's a SELECT or a command with the RETURNING clause, we expect to receive all result rows. For example:

```
=> SELECT model FROM aircrafts WHERE aircraft_code = '773';
```

```
model
-----
Бойнг 777-300
(1 row)
```

The standard way to obtain an execution plan is the EXPLAIN command (cost output is currently disabled):

```
=> EXPLAIN (buffers, costs off)
SELECT * FROM airports;
```

```
QUERY PLAN
-----
Seq Scan on airports_data ml
Planning:
  Buffers: shared hit=54 read=6 dirtied=1
(3 rows)
```

If the buffers parameter is specified, the Planning Buffers field will display the number of buffer cache pages read while building the plan.

If you add the analyze parameter, the server will execute the query and show the execution plan with detailed information. Exercise caution with data-modifying queries!

```
=> EXPLAIN (analyze, buffers, costs off)
SELECT * FROM airports;
```

```
QUERY PLAN
-----
Seq Scan on airports_data ml (actual time=0.136..1.868 rows=104 loops=1)
  Buffers: shared read=3
Planning Time: 0.125 ms
Execution Time: 1.904 ms
(4 rows)
```

The Buffers field displays the cache type (shared – buffer cache in shared memory) and the number of pages:

- hit – found in the buffer cache;
- read – pages not found in the cache and read from the operating system;
- written – pages written to files;
- dirtied – pages that have become dirty (i.e., modified for the first time in the cache)

Now, let's create a temporary table and insert the rows from the airports table into it.

```
=> CREATE TEMP TABLE temp_airports (LIKE airports);
```

```
CREATE TABLE
```

```
=> EXPLAIN (analyze, buffers, costs off)
INSERT INTO temp_airports SELECT * FROM airports;
```

```
QUERY PLAN
-----
Insert on temp_airports (actual time=3.862..3.862 rows=0 loops=1)
  Buffers: shared hit=3, local hit=104 dirtied=3 written=5
  -> Seq Scan on airports_data ml (actual time=0.860..1.706 rows=104 loops=1)
    Buffers: shared hit=3
Planning Time: 0.091 ms
Execution Time: 3.889 ms
(6 rows)
```

The temporary table is cached in the session's local memory (Buffers ... local).

Warning: displaying the execution time for each step, as shown in this example, may significantly impact query performance on certain platforms. If this information isn't needed, it's recommended to use timing off.

Pipeline

Tree traversal starting at the root

Data is transmitted upwards—either as it arrives or all at once.

Data Access

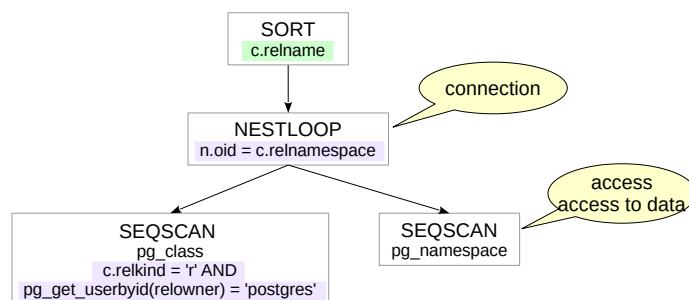
table and index reads

Joins

always in pairs

Order matters

Other operations



14

During the execution phase, the execution plan tree (as shown on the slide, it's simplified to highlight the essentials) functions like a conveyor belt.

Execution starts at the root node. The root node (in our case, the **SORT** operation) retrieves data from the child node; once it receives the data, it performs the sorting operation and sends the results upward (i.e., to the client).

Some nodes (such as the **NESTLOOP** node) join data from different sources. Here, the node accesses two child nodes in sequence (the join is always performed pairwise) and, upon receiving rows from them, joins them and sends them upward to the sort node.

The two lower nodes represent table access for data retrieval. They read rows from the relevant tables and pass them up to the join node.

Some nodes can only return a result once they have received all data from their child nodes. Such nodes include sorting — it cannot process an incomplete sample. Other nodes can return data as it comes in. For example, table scan data access can return data as it is read (enabling quick retrieval of the initial portion of the result — such as for paginated display on a web page).

To get a handle on execution plans, you need to understand the available data access methods, the techniques for joining data, and examine some other operations.

Compiling parts of queries into source code

- Evaluation of expressions in the WHERE clause
- Evaluation of expressions in the SELECT clause
- Aggregates and Projections

Transformation of Tuple Versions

- Moving tuple versions from disk to an unrolled representation in memory

JIT (just-in-time, "exactly at the right time") dynamic compilation is used to compile code or its parts during program execution. This technology enables faster execution of interpreted code and is used in many systems.

In PostgreSQL, JIT compilation can compile parts of the code executed when processing SQL queries. PostgreSQL needs to be compiled with support for LLVM.

JIT compilation is more suitable for long-running, CPU-intensive analytical queries. For short OLTP queries, JIT compilation overhead can exceed the queries' execution time.

JIT compilation can be influenced by configuration parameters. There are several JIT-related optimizations that are enabled only when the query cost exceeds the threshold value set in the relevant configuration parameters.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

JIT compilation Just-In-Time Compilation

By default, JIT compilation is enabled, though it's disabled in the demo database:

```
=> SELECT setconfig
FROM pg_db_role_setting
WHERE setdatabase = (SELECT oid FROM pg_database WHERE datname='demo');
```

```
              setconfig
-----
{"search_path=bookings, public",bookings.lang=ru,jit=off}
(1 row)
```

```
=> SELECT name, setting, boot_val
FROM pg_settings
WHERE name='jit';
```

```
 name | setting | boot_val
-----+-----+-----
  jit |  off   |    on
(1 row)
```

Let's run a query that computes the value of π . The query makes extensive use of calculations that JIT can optimize:

```
=> SET jit = on;
```

SET

```
=> WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10_000_000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

```
 val
-----
3.1421784
(1 row)
```

The EXPLAIN ANALYZE command will display detailed information on which JIT optimizations were applied:

```
=> EXPLAIN (analyze, timing off)
WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10_000_000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

QUERY PLAN

```
-----
Aggregate  (cost=525000.00..525000.02 rows=1 width=8) (actual rows=1 loops=1)
  CTE pi
    -> Function Scan on generate_series  (cost=0.00..150000.00 rows=10000000 width=16)
(actual rows=10000000 loops=1)
    -> CTE Scan on pi  (cost=0.00..200000.00 rows=10000000 width=16) (actual
rows=10000000 loops=1)
Planning Time: 0.267 ms
JIT:
  Functions: 6
  Options: Inlining true, Optimization true, Expressions true, Deforming true
Execution Time: 11707.291 ms
(9 rows)
```

Disable JIT compilation and rerun the query:

```
=> SET jit = off;
```

SET

```
=> EXPLAIN (analyze, timing off)
WITH pi AS (
  SELECT random() x, random() y
  FROM generate_series(1,10_000_000)
)
SELECT 4*sum(1-floor(x*x+y*y))/count(*) val FROM pi;
```

QUERY PLAN

```
-----
Aggregate  (cost=525000.00..525000.02 rows=1 width=8) (actual rows=1 loops=1)
  CTE pi
    -> Function Scan on generate_series  (cost=0.00..150000.00 rows=10000000 width=16)
(actual rows=10000000 loops=1)
    -> CTE Scan on pi  (cost=0.00..200000.00 rows=10000000 width=16) (actual
rows=10000000 loops=1)
  Planning Time: 0.094 ms
  Execution Time: 12060.671 ms
(6 rows)
```

The query execution time is likely to increase slightly.

Throughout the course, we'll skip the details of JIT optimizations. To avoid cluttering query plans with JIT compilation messages, JIT is disabled in the demo database.

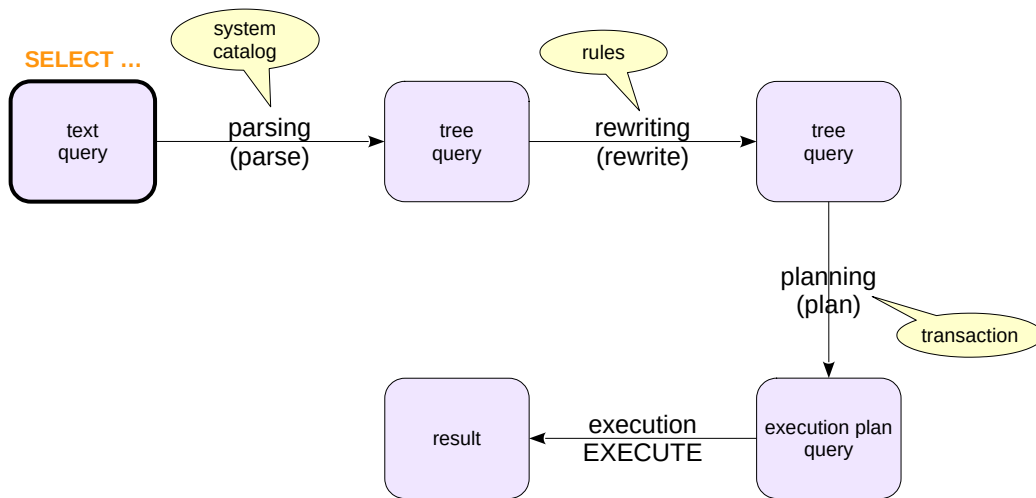
Refining the Query Processing Schema

Prepared Statements

Cursors

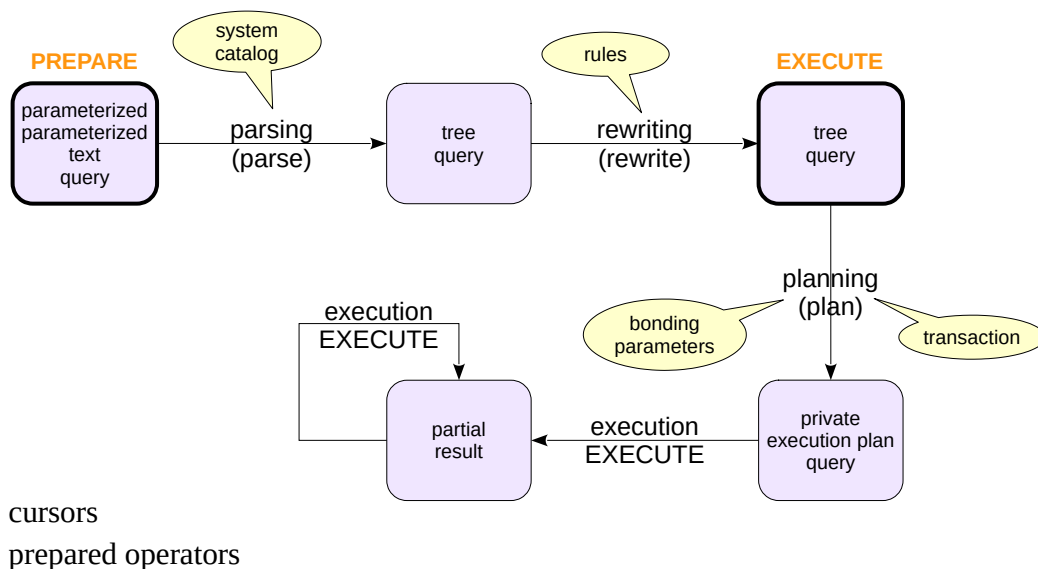
PostgreSQL also supports the Extended Query Protocol. In practice, this allows for the use of prepared statements and cursors.

Simple Query Protocol



The slide revisits the complete query processing workflow via the simple protocol, which we have already discussed.

Extended Protocol



The extended protocol provides finer-grained control over query processing. First, the query can be prepared. To do this, the client sends a query to the server (possibly in a parameterized form), and the server parses and rewrites it, storing the prepared query plan in the backend process's local memory.

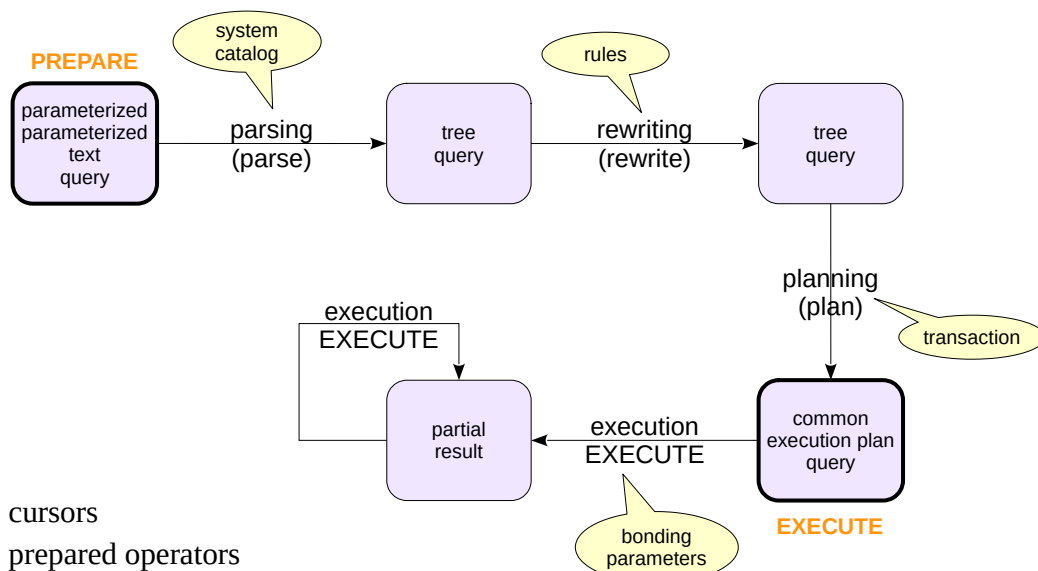
To execute a prepared query, the client identifies it by name and provides specific parameter values. The server constructs a private query plan, considering the parameter values, and executes it.

Preparation helps avoid the need for repeated parsing and rewriting of the same query when executed multiple times within a single session.

Second, you can use cursors. The cursor mechanism allows retrieving query results row by row instead of all at once. The information about the open cursor is also kept in the backend process's local memory.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

Extended Protocol



If a query has no parameters, there's no need for the server to re-plan it every time it's executed. In this case, it immediately caches a generic execution plan. This helps save even more resources.

If the query includes parameters, the server can use a generic plan if it determines that, on average, it performs as well as or better than specific plans. For more information on when the switch occurs, see the "Basic Statistics" section.

Another reason to use prepared statements is to prevent SQL injection when the query's input data originates from an untrusted source, such as input fields on a web form.

Prepared Statements

Let's create a prepared statement for a parameterized query:

```
=> PREPARE model(varchar) AS
    SELECT model FROM aircrafts WHERE aircraft_code = $1;
```

PREPARE

Now we can invoke the statement by name:

```
=> EXECUTE model('773');
```

```
      model
-----
Бойнг 777-300
(1 row)
```

```
=> EXECUTE model('763');
```

```
      model
-----
Бойнг 767-300
(1 row)
```

All prepared statements are available in the view:

```
=> SELECT * FROM pg_prepared_statements \gx
```

```
-[ RECORD 1 ]-----+-----
name          | model
statement     | PREPARE model(varchar) AS
               |     SELECT model FROM aircrafts WHERE aircraft_code = $1;
prepare_time  | 2025-10-18 22:25:25.349355+03
parameter_types | {"character varying"}
result_types   | {text}
from_sql      | t
generic_plans  | 0
custom_plans  | 2
```

- name — the name of the prepared statement
- statement — the command that generated it
- prepare_time — creation time
- parameter_types — parameter types (array),
- result_types — result column types (array),
- from_sql — indicates that the statement was created using the SQL PREPARE command (not via a driver function call),
- generic_plans — number of times the generic plan was used
- custom_plans — the number of times custom plans were created and used

If a prepared statement is no longer required, you can remove it using the DEALLOCATE command. However, it will be automatically dropped when the session ends.

```
=> \c
```

You are now connected to database "demo" as user "postgres".

```
=> SELECT * FROM pg_prepared_statements \gx
```

```
(0 rows)
```

PREPARE, EXECUTE, and DEALLOCATE are SQL commands. Clients developed in other programming languages will use the operations defined in their respective drivers. However, any driver uses the same protocol for communication with the server.

When executing a parameterized query in psql, the extended query protocol is used. In this case, you need to set the parameters in advance:

```
=> \bind '773'
```

A prepared statement is created implicitly.

```
=> SELECT model FROM aircrafts WHERE aircraft_code = $1;
```

```
      model
-----
Бойнг 777-300
(1 row)
```

After executing the query, the prepared statement is automatically dropped:

```
=> SELECT * FROM pg_prepared_statements \gx
(0 rows)
```

Parameter values are cleared after execution, so they must be set before each query:

```
=> \bind '763'
=> EXPLAIN SELECT model FROM aircrafts WHERE aircraft_code = $1;

              QUERY PLAN
-----
Seq Scan on aircrafts_data ml  (cost=0.00..1.36 rows=1 width=32)
  Filter: (aircraft_code = '763'::bpchar)
(2 rows)
```

This approach can be used to debug query execution using the extended protocol.

Cursors

Cursors allow for row-by-row processing of results. They are commonly used in applications and include optimization features.

In SQL, cursors can be demonstrated as follows: Declare a cursor (which is opened immediately) and fetch the first row:

```
=> BEGIN;
BEGIN
=> DECLARE c CURSOR FOR SELECT * FROM aircrafts;
DECLARE CURSOR
=> FETCH c;

 aircraft_code |      model      | range
-----+-----+-----
       773      | Бойнг 777-300   | 11100
(1 row)
```

Read the second result row and close the open cursor (the cursor will close automatically at the end of the transaction):

```
=> FETCH c;

 aircraft_code |      model      | range
-----+-----+-----
       763      | Бойнг 767-300   | 7900
(1 row)

=> CLOSE c;
CLOSE CURSOR
=> COMMIT;
COMMIT
```


Planning Process

Cardinality Estimation

Cost estimation

Selecting the Optimal Plan

Query planning — a critical and complex process, so we'll explore it in more detail.

Statistics

table size and data distribution statistics

Cardinality Estimation

Condition selectivity — the proportion of rows selected.

Cardinality refers to the total number of rows

Calculation requires statistics

Cost estimation

Mainly determined by the node type and the number of rows processed

Optimizer's plan evaluation

the plan with the lowest cost is chosen

The optimizer evaluates all possible execution plans and selects the one with the lowest cost.

When evaluating the cost of a plan node, the optimizer takes into account the node's type (notably, the cost of reading data directly from a table differs from that of using an index) and the amount of data processed by the node. Other factors are regarded as less important.

Two key concepts are crucial for assessing data volume:

- cardinality — the total number of rows;
- <Selectivity> refers to the proportion of rows that meet the conditions (predicates).

To evaluate selectivity and cardinality, you need to have information about the data, such as table sizes, value distribution in columns and other factors.

Ultimately, everything relies on statistics — data gathered and maintained through the auto-analyze process or the ANALYZE command.

If cardinality is correctly estimated, the cost is usually quite accurate. The optimizer's primary issues stem from inaccurate cardinality estimation. This can occur due to insufficient statistics, the inability to use them, or flaws in the models underlying the optimizer. We'll go into more detail about this.

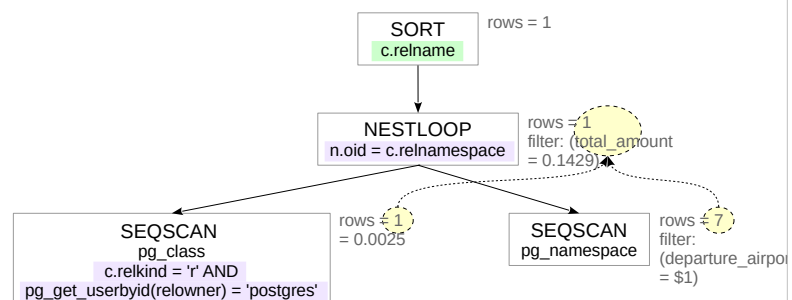
Cardinality Estimation

Access Method Cardinality

$$rows_{A \text{ where } cond} = rows_A \cdot sel_{cond}$$

Join Cardinality

$$rows_{A \text{ join } B \text{ on } cond} = rows_A \cdot rows_B \cdot sel_{cond}$$



24

It is convenient to view cardinality estimation as a recursive process. To estimate the cardinality of a node, you first need to estimate the cardinalities of its child nodes, and then calculate its cardinality based on those values once you know the node's type.

Therefore, the first step is to calculate the cardinalities of leaf nodes containing data access methods. To do this, we need to know the table's size and the selectivity of the conditions applied to it. We'll discuss how exactly this is done later.

We can note for now that it's enough to estimate the selectivity of simple conditions, as the selectivity of conditions constructed with logical operations can be calculated with simple formulas:

$$sel_{x \text{ and } y} = sel_x \cdot sel_y; sel_{x \text{ or } y} = 1 - (1 - sel_x) (1 - sel_y)$$

Keep in mind that these formulas assume independence of the predicates. If they correlate, this estimate will be inaccurate. In the case of correlated predicates, this estimate will be inaccurate (it can be improved with extended statistics).

Next, you can calculate the join cardinalities. We already know the cardinalities of the joined data sets, but we still need to estimate the selectivity of the join conditions. Let's just assume it's possible for now.

Similarly, this approach can be applied to other nodes, such as sortings or aggregations.

Note that a cardinality calculation mistake in a lower node will propagate upward, resulting in inaccurate cost estimation and, ultimately, a sub-optimal plan.

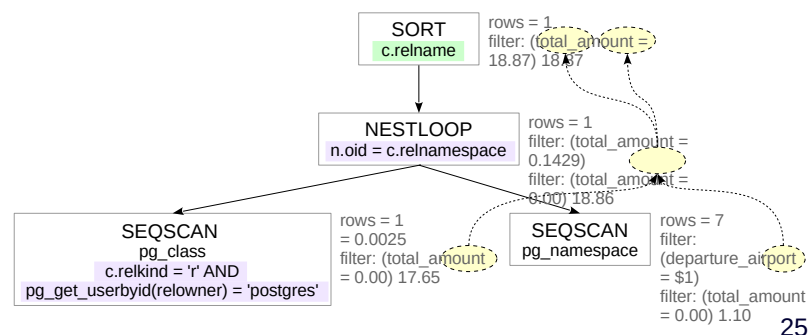
Cost estimation

Calculated using mathematical models

$$\text{cost} = \text{costA} + \sum \text{costA's child nodes}$$

Two elements

Preparation .. retrieving all rows



Now, let's look at the overall process of cost estimation. It is also inherently recursive To calculate the cost of a subtree, you first need to compute the costs of the child nodes and sum them, then add the node's own cost.

The cost of a node's operation is calculated using the mathematical model built into the planner, taking into account the estimated number of rows processed (which has already been computed).

The cost consists of two components, each evaluated separately. The first component is the initial cost (initial cost), while the second represents the cost of retrieving all the rows in the result set (total cost).

Some operations don't require any preparation; their initial cost is zero.

Conversely, other operations require preliminary steps. For example, the sorting operation in the example needs to first retrieve all data from the child node before it can begin. For these nodes, the initial cost will not be zero – this cost must be incurred regardless of how many result rows are required.

It's important to recognize that the cost represents the planner's estimate and might not align with the actual execution time. It's often viewed as the cost being measured in hypothetical "units" that don't carry any inherent meaning on their own. Cost is only used to allow the planner to compare different plans for the same query.

Exploring Plans

- Join order, join methods, access methods

- A full scan where possible; with numerous options, the search space is reduced.

Simple Queries and Prepared Statements

- Optimizing retrieval of all rows

- Minimum total cost

Cursors

- Optimizing retrieval of a portion of the first rows

- Minimum cost for `cursor_tuple_fraction` rows

The optimizer tries to evaluate all possible query execution plans to choose the best one.

A dynamic programming algorithm is used to reduce the search space, but when the number of options is high—especially due to the number of tables being joined—finding an exact solution to the optimization problem becomes impractical within a reasonable time. In such cases, the planner reduces the number of plans it evaluates by either not considering all pairwise join options or switching to a genetic optimization algorithm (GEQO - Genetic Query Optimization). This could result in the optimizer choosing a suboptimal plan not because of evaluation errors, but merely because the best plan wasn't considered.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

What defines the "best plan"?

For typical queries, the optimal plan minimizes the resources needed to retrieve all rows, meaning it has the lowest total cost.

However, when using cursors, it's important to retrieve the first rows as soon as possible. Therefore, there's a parameter `cursor_tuple_fraction` (default 0.1) that determines the fraction of rows to be retrieved as quickly as possible. The lower the parameter value, the more the initial cost influences the plan selection instead of the total cost.

Query processing involves multiple stages: parsing and rewriting, planning, and execution.

There are two protocols for executing queries.

- Simple — direct execution and immediate result retrieval

- Extended: prepared statements and cursors

Runtime depends on the quality of planning.

The optimizer constructs the plan based on cost.

1. The Effect of Preparing a Long Statement on Its Execution
Compute the average cost of a flight and determine the average run time of this query.
Prepare the statement for this query and then recalculate the average run time.
How many times faster did the execution become?
2. The Impact of Preparing Short Statements on Their Execution
Run the query for booking data with ID 0824C5 multiple times; then calculate the average execution time.
Prepare the statement for this query and then recalculate the average run time.
How many times faster did the execution become in this case?

The execution time of the same query can vary significantly, especially during the first execution. To reduce the variation, average the execution time by running the query multiple times. It's convenient to use PL/pgSQL, keeping in mind that:

- A dynamic query executed via the PL/pgSQL EXECUTE command (note: not to be confused with the SQL EXECUTE command) goes through all stages every time;
- An SQL query embedded in PL/pgSQL code is executed using prepared statements.

Example of the syntax for a regular operator command:

```
DO $$  
BEGIN  
FOR i IN 1..10 LOOP  
EXECUTE 'SELECT ... FROM ...';  
END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

For the prepared statement (in this case, SELECT is replaced with PERFORM because the result is not needed):

```
DO $$  
BEGIN  
FOR i IN 1..10 LOOP  
PERFORM ... FROM ...;  
END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

1. A Long Query

```
=> \timing on
```

Timing is on.

Standard Statement:

```
=> DO $$  
BEGIN  
  FOR i IN 1..10 LOOP  
    EXECUTE 'SELECT avg(amount) FROM ticket_flights';  
  END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 40066.207 ms (00:40.066)

Prepared Statement:

```
=> DO $$  
BEGIN  
  FOR i IN 1..10 LOOP  
    PERFORM avg(amount) FROM ticket_flights;  
  END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 39736.029 ms (00:39.736)

The time change is slight — most of the time is spent on query execution.

2. A Fast Query

Standard Statement:

```
=> DO $$  
BEGIN  
  FOR i IN 1..100_000 LOOP  
    EXECUTE 'SELECT * FROM bookings WHERE book_ref = ''0824C5''';  
  END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 6469.245 ms (00:06.469)

Prepared Statement:

```
=> DO $$  
BEGIN  
  FOR i IN 1..100_000 LOOP  
    PERFORM * FROM bookings WHERE book_ref = '0824C5';  
  END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

DO

Time: 1211.851 ms (00:01.212)

The time decreased significantly — parsing and planning account for most of the total time.