Query execution stages

Case of PostgreSQL

Source: https://postgrespro.com

Parsing

First, the query text is *parsed*, so that the server understands exactly what needs to be done.

Lexer and parser. The *lexer* is responsible for recognizing *lexemes* in the query string (such as SQL keywords, string and numeric literals, etc.), and

the *parser* makes sure that the resulting set of lexemes is grammatically valid. The parser and lexer are implemented using the standard tools Bison and Flex.

The parsed query is represented as an abstract syntax tree.

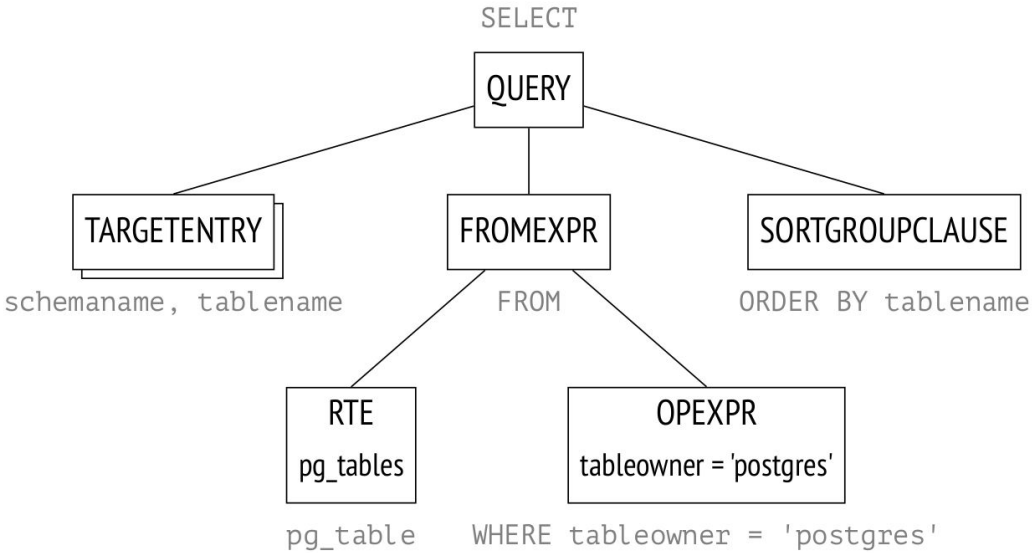
```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

A tree will be built in backend memory. The figure below shows the tree in a highly simplified form. The nodes of the tree are labeled with the corresponding parts of the query.

RTE istands for "Range Table Entry." The name "range table" in the PostgreSQL source code refers to tables, subqueries, results of joins—in other words, any record sets that SQL statements operate on.

Semantic analyzer. The semantic analyzer determines whether there are tables and other objects in the database that the query refers to by name, and whether the user has the right to access these objects. All the information required for semantic analysis is stored in the system catalog.

The semantic analyzer receives the parse tree from the parser and rebuilds it, supplementing it with references to specific database objects, data type information, etc.



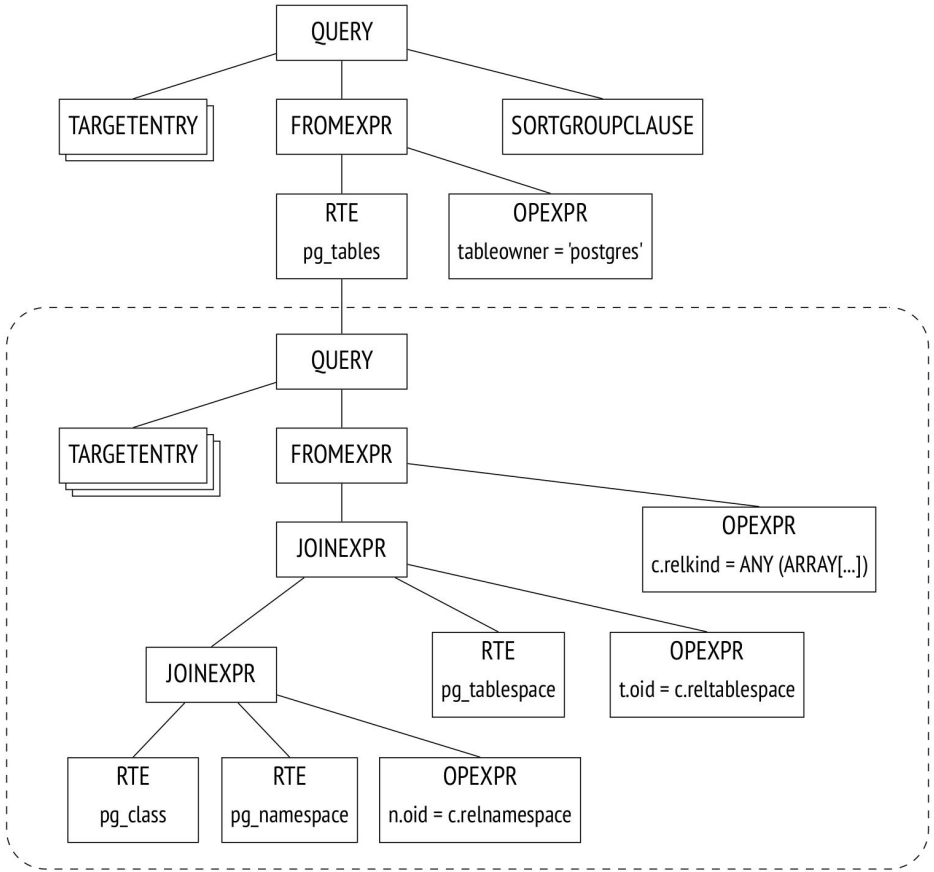
Transformation

Next, the query can be *transformed (rewritten)*.

Transformations are used by the system core for several purposes. One of them is to replace the name of a view in the parse tree with a subtree corresponding to the query of this view.

`pg_tables` from the example above is a view

```
SELECT schemaname, tablename
FROM (
    -- pg_tables
    SELECT n.nspname AS schemaname,
      c.relname AS tablename,
      pg_get_userbyid(c.relowner) AS tableowner,
      ...
    FROM pg_class c
      LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
      LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
    WHERE c.relkind = ANY (ARRAY['r'::char, 'p'::char])
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```



Planning

SQL is a declarative language: a query specifies *what* to retrieve, but not *how* to retrieve it.

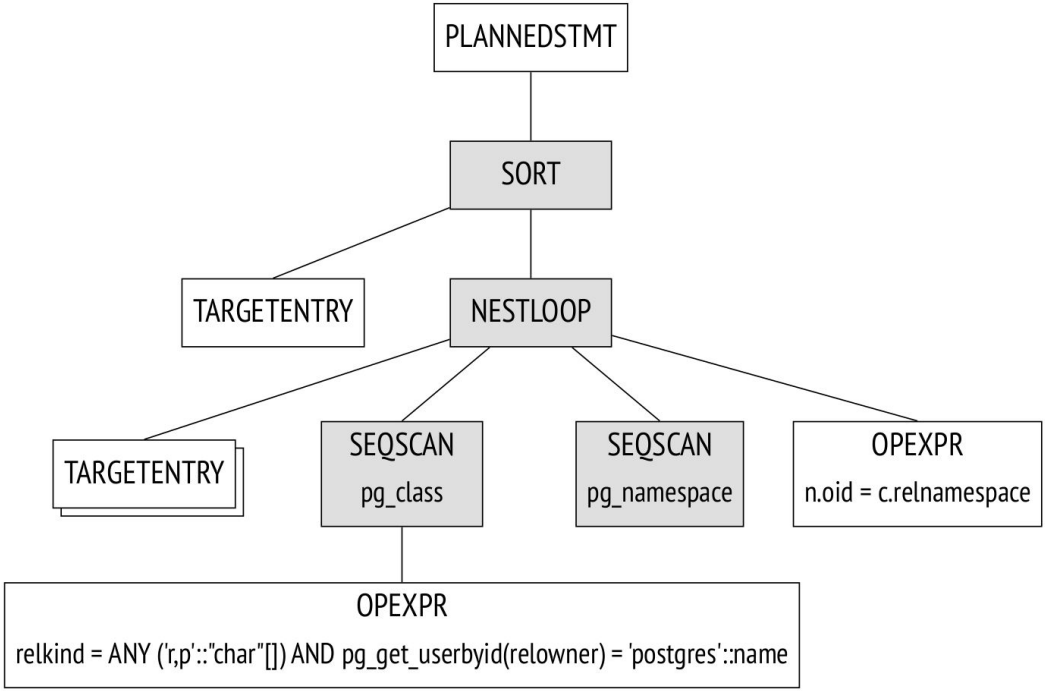
Any query can be executed in a number of ways. Each operation in the parse tree has multiple execution options.

> For example, you can retrieve specific records from a table by reading the whole table and discarding rows you don't need, or you can use indexes to find the rows that match your query.
>
> Data sets are always joined in pairs. Variations in the order of joins result in a multitude of execution options.
>
> Then there are various ways to join two sets of rows together. For example, you could go through the rows in the first set one by one and look for matching rows in the other set, or you could sort both sets first, and then merge them together. Different approaches perform better in some cases and worse in others.

The optimal plan may execute faster than a non-optimal one by several orders of magnitude. This is why the *planner*, which *optimizes* the parsed query, is one of the most complex elements of the system.

Plan tree. The execution plan can also be presented as a tree, but with its nodes as physical rather than logical operations on data.

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




```
                                QUERY PLAN
-----------------------------------------------------------
 Sort  (cost=21.03..21.04 rows=1 width=128)
   Sort Key: c.relname
   -> Nested Loop Left Join  (cost=0.00..21.02 rows=1 width=128)
        Join Filter: (n.oid = c.relnamespace)
        -> Seq Scan on pg_class c  (cost=0.00..19.93 rows=1 width=72)
             Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
        -> Seq Scan on pg_namespace n  (cost=0.00..1.04 rows=4 wid...
(7 rows)
```

The image shows the main nodes of the tree. The same nodes are marked with arrows in the EXPLAIN output.

The Seq Scan node represents the table read operation, while the Nested Loop node represents the join operation. There are two interesting points to take note of here:

- One of the initial tables is gone from the plan tree because the planner figured out that it's not required to process the query and removed it.
- There is an estimated number of rows to process and the cost of processing next to each node.

Plan search. To find the optimal plan, PostgreSQL utilizes the *cost-based query optimizer*. The optimizer goes through various available execution plans and estimates the required amounts of resources, such as I/O operations and CPU cycles. This calculated estimate, converted into arbitrary units, is known as the *plan cost*. The plan with the lowest resulting cost is selected for execution.

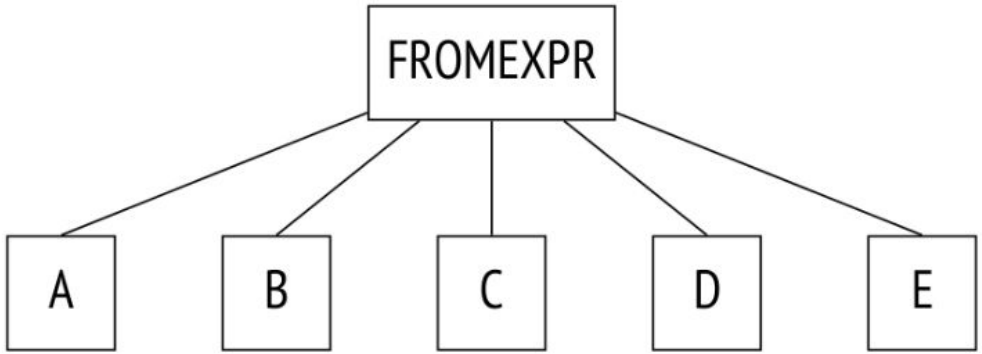
The trouble is, the number of possible plans grows exponentially as the number of joins increases, and sifting through all the plans one by one is impossible even for relatively simple queries. Therefore, dynamic programming and heuristics are used to limit the scope of search. This allows to precisely solve the problem for a greater number of tables in a query within reasonable time, but the selected plan is not guaranteed to be *truly* optimal because the planner utilizes simplified mathematical models and may use imprecise initial data.

Ordering joins. A query can be structured in specific ways to significantly reduce the search scope (at a risk of missing the opportunity to find the optimal plan):

- Common table expressions are usually optimized separately from the main query.
- Queries from non-SQL functions are optimized separately from the main query. (SQL functions can be inlined into the main query in some cases.)
- In PostgreSQL there are some patamets that quide the optimizer in generating the execution plan:
  a. The *join_collapse_limit* parameter together with explicit JOIN clauses,
  b. as well as the *from_collapse_limit* parameter together with sub-queries
- may define the order of some joins, depending on the query syntax.

The query below calls several tables within a FROM clause with no explicit joins:

```
SELECT ...
FROM a, b, c, d, e
WHERE ...
```



The parse tree for this query:

In this query, the planner will consider all possible join orders.

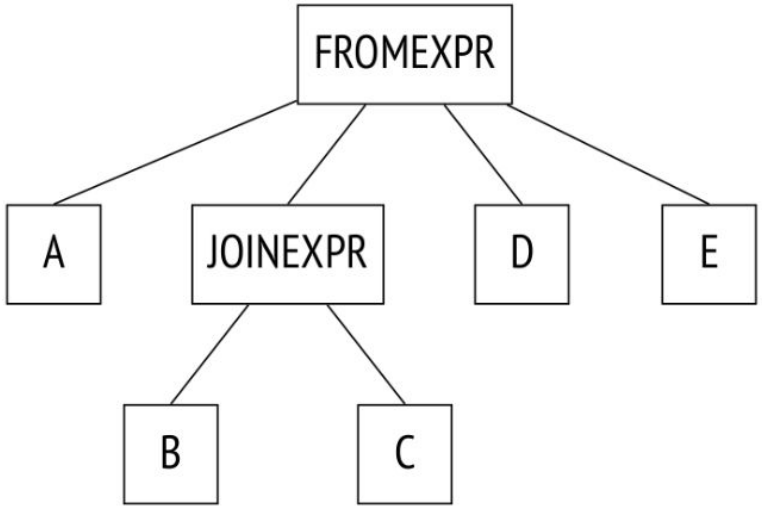
In the next example, some joins are explicitly defined by the JOIN clause:

```
SELECT ...
FROM a, b JOIN c ON ..., d, e
WHERE ...
```

The planner collapses the join tree, effectively transforming it into the tree from the previous example. The algorithm recursively traverses the tree and replaces each JOINEXPR node with a flat list of its components.

Selecting the best plan. In general, the plan must optimize the retrieval of all rows that match the query.

The parse tree reflects this:



Selecting the best plan.

PostgreSQL addresses this by calculating two cost components. They are displayed in the query plan output after the word "cost":

Sort (cost=21.03..21.04 rows=1 width=128)

The first component, startup cost, is the cost to prepare for the execution of node of the query (node: part of the query that contain join, selection or sorting) ; the second component, total cost, represents the total node execution cost.

The planner selects the plan with the least total cost.

Cost calculation process.

To estimate a plan cost, each of its nodes has to be individually estimated. A node cost depends on the node type (reading from a table costs much less than sorting the table) and the amount of data processed (in general, the more data, the higher the cost). While the node type is known right away, to assess the amount of data we first need to estimate the node's cardinality (the amount of input rows) and selectivity (the fraction of rows left over for output). To do that, we need data statistics: table sizes, data distribution across columns.

Execution

An optimized query is executed in accordance with the plan.

Execution starts at the root node. The root node (the sorting node SORT in the example) requests data from the child node. When it receives all requested data, it performs the sorting operation and then delivers the data upward, to the client.

Some nodes (such as the NESTLOOP node) join data from different sources. This node requests data from two child nodes. Upon receiving two rows that match the join condition, the node immediately passes the resulting row to the parent node (unlike with sorting, which must receive all rows before processing them). The node then stops until its parent node requests another row. Because of that, if only a partial result is required (as set by LIMIT, for example), the operation will not be executed fully.