Chapter 3

Query Processing (Part 1)

As described in the official document, PostgreSQL supports a very large number of features required by the SQL standard of 2011. Query processing is the most complicated subsystem in PostgreSQL, and it efficiently processes the supported SQL. This chapter outlines this query processing, in particular, it focuses on query optimization.

This chapter comprises the following three parts:

Part 1: Section 3.1.
This section provides an overview of query processing in PostgreSQL.

Part 2: Sections 3.2. — 3.4.
This part explains the steps followed to obtain the optimal plan of a single-table query. In Sections 3.2 and 3.3, the processes of estimating the cost and creating the plan tree are explained, respectively. Section 3.4 briefly describes the operation of the executor.

Part 3: Sections 3.5. — 3.6.
This part explains the process of obtaining the optimal plan of a multiple-table query. In Section 3.5, three join methods are described: nested loop, merge, and hash join. Section 3.6 explains the process of creating the plan tree of a multiple-table query.

PostgreSQL supports three technically interesting and practical features: Foreign Data Wrappers (FDW), Parallel Query and JIT compilation which is supported from version 11. The first two of them will be described in Chapter 4. The JIT compilation is out of scope of this document; see the official document in details.

Go to Part 2, Part 3 ➔

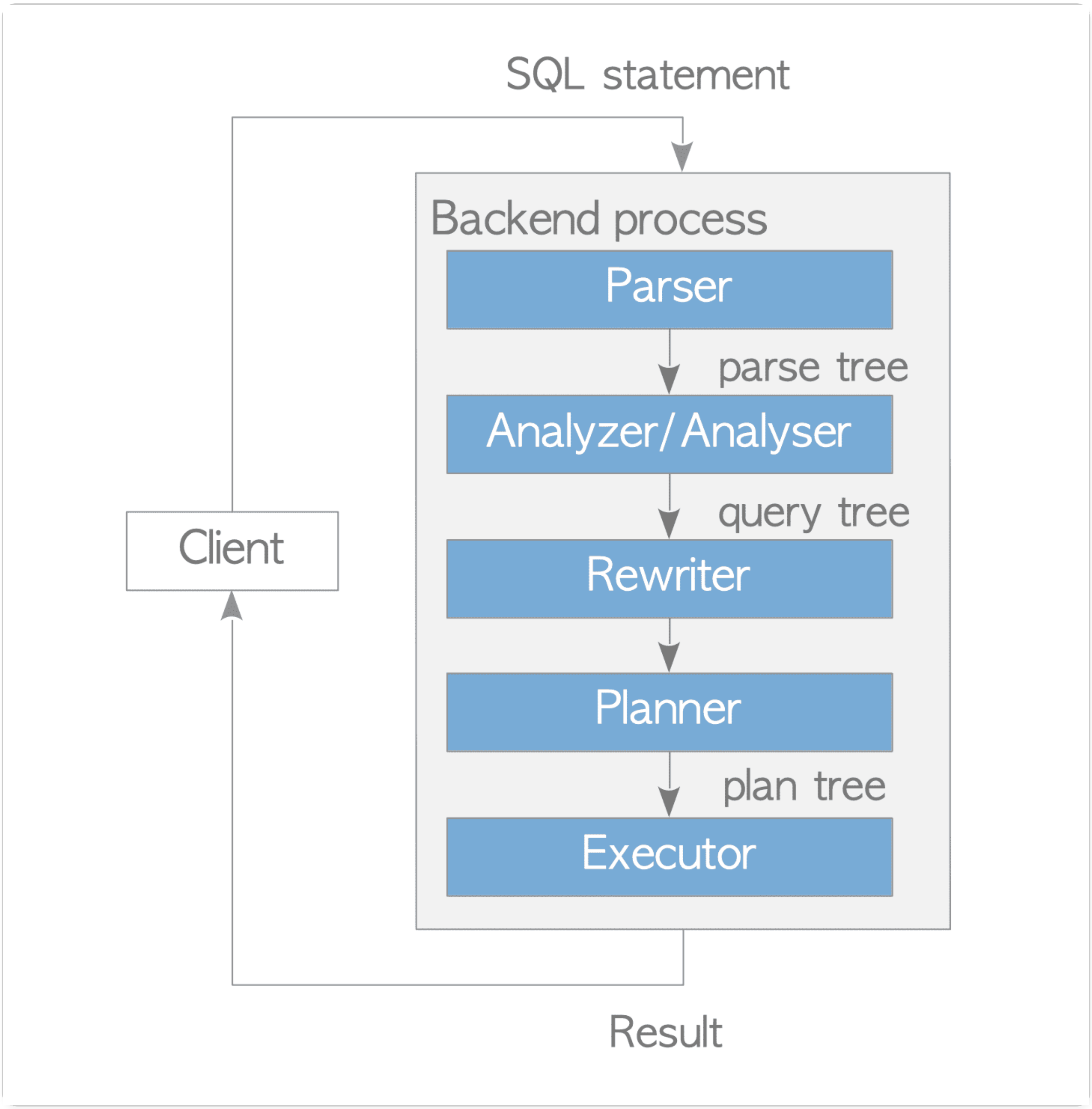
3.1. Overview

In PostgreSQL, although the parallel query implemented in version 9.6 uses multiple background worker processes, a backend process basically handles all queries issued by the connected client. This backend consists of five subsystems:

1. Parser
   The parser generates a parse tree from an SQL statement in plain text.
2. Analyzer/Analyser
   The analyzer/analyser carries out a semantic analysis of a parse tree and generates a query tree.
3. Rewriter
   The rewriter transforms a query tree using the rules stored in the rule system if such rules exist.

4. Planner
   The planner generates the plan tree that can most effectively be executed from the query tree.
5. Executor
   The executor executes the query by accessing the tables and indexes in the order that was created by the plan tree.

Fig. 3.1. Query Processing.



In this section, an overview of these subsystems is provided. Due to the fact that the planner and the executor are very complicated, a detailed explanation for these functions will be provided in the following sections.

PostgreSQL's query processing is described in the official document in detail.

3.1.1. Parser

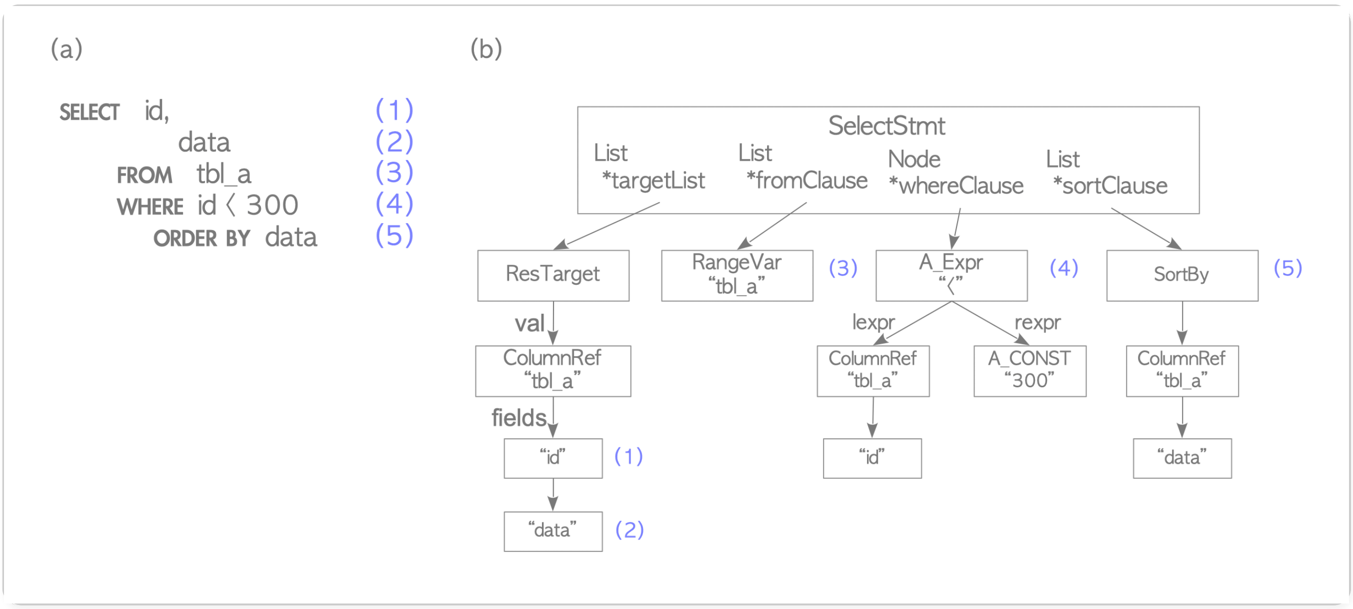
The parser generates a parse tree that can be read by subsequent subsystems from an SQL statement in plain text. Here is a specific example, without a detailed description.

Let us consider the query shown below.

```
testdb=# SELECT id, data FROM tbl_a WHERE id < 300 ORDER BY data;
```

A parse tree is a tree whose root node is the SelectStmt structure defined in parsenodes.h. Figure 3.2(b) illustrates the parse tree of the query shown in Fig. 3.2(a).

Fig. 3.2. An example of a parse tree.



The elements of the SELECT query and the corresponding elements of the parse tree are numbered the same. For example, (1) is an item of the first target list, and it is the column 'id' of the table; (4) is a WHERE clause; and so on.

The parser only checks the syntax of an input when generating a parse tree. Therefore, it only returns an error if there is a syntax error in the query.

The parser does not check the semantics of an input query. For example, even if the query contains a table name that does not exist, the parser does not return an error. Semantic checks are done by the analyzer/analyser.

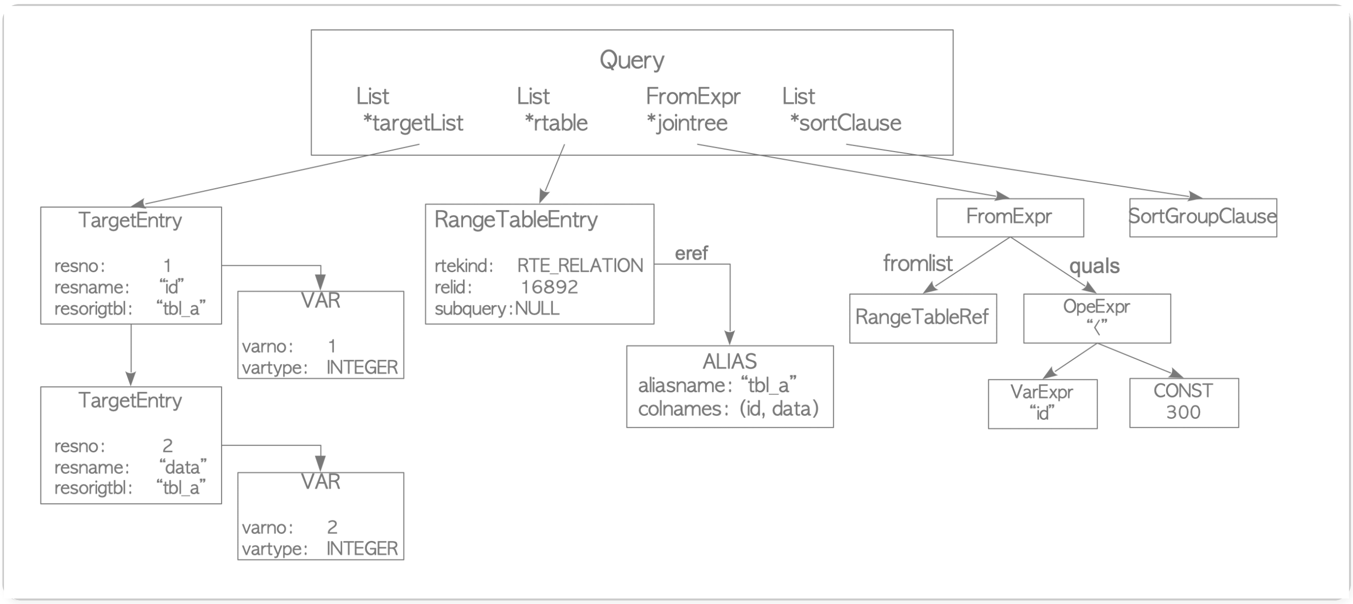
3.1.2. Analyzer/Analyser

The analyzer/analyser runs a semantic analysis of a parse tree generated by the parser and generates a query tree.

The root of a query tree is the Query structure defined in parsenodes.h. This structure contains metadata of its corresponding query, such as the type of the command (SELECT, INSERT, or others), and several leaves. Each leaf forms a list or a tree and holds data for the individual particular clause.

Figure 3.3 illustrates the query tree of the query shown in Fig. 3.2(a) in the previous subsection.

Fig. 3.3. An example of a query tree.



The above query tree is briefly described as follows:

- The targetlist is a list of columns that are the result of this query. In this example, the list is composed of two columns: *'id'* and *'data'*. If the input query tree uses '∗' (asterisk), the analyzer/analyser will explicitly replace it with all of the columns.
- The range table is a list of relations that are used in this query. In this example, the list holds the information of the table *'tbl_a'*, such as the *OID* of the table and the name of the table.
- The join tree stores the FROM clause and the WHERE clauses.
- The sort clause is a list of SortGroupClause.

The details of the query tree are described in the official document.

3.1.3. Rewriter

The rewriter is the system that realizes the rule system. It transforms a query tree according to the rules stored in the pg_rules system catalog, if necessary. The rule system is an interesting system in itself, but the descriptions of the rule system and the rewriter have been omitted to prevent this chapter from becoming too long.

ℹ View

Views in PostgreSQL are implemented by using the rule system. When a view is defined by the CREATE VIEW command, the corresponding rule is automatically generated and stored in the catalog.

Assume that the following view is already defined and the corresponding rule is stored in the *pg_rules* system catalog:

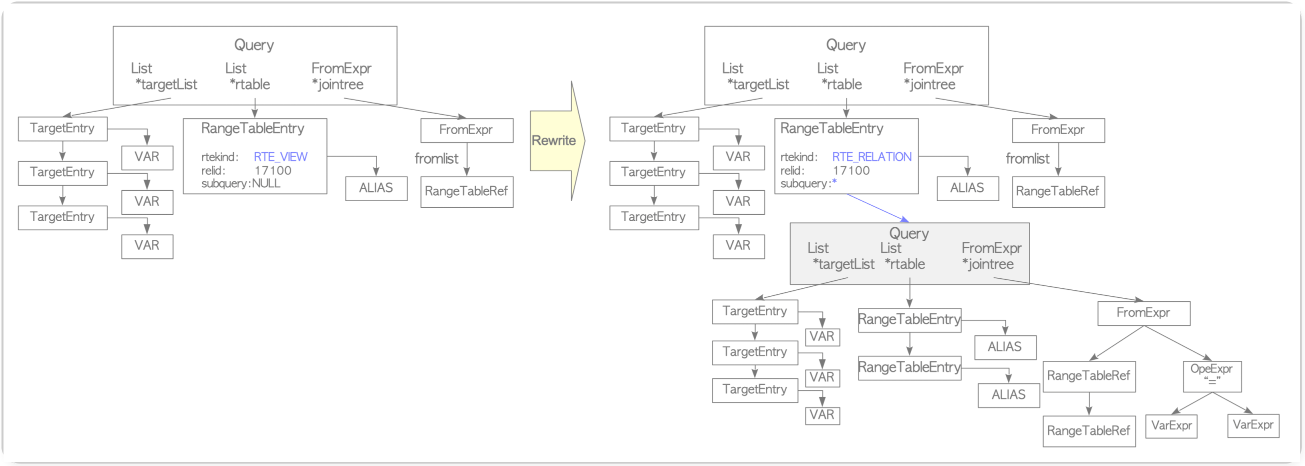
```
sampledb=# CREATE VIEW employees_list
sampledb-#      AS SELECT e.id, e.name, d.name AS department
sampledb-#            FROM employees AS e, departments AS d WHERE e.department_id = d.id;
```

When a query that contains a view shown below is issued, the parser creates the parse tree as shown in Fig. 3.4(a).

```
sampledb=# SELECT * FROM employees_list;
```

At this stage, the rewriter processes the range table node to a parse tree of the subquery, which is the corresponding view, stored in *pg_rules*.

Fig. 3.4. An example of the rewriter stage.



Since PostgreSQL realizes views using such a mechanism, views could not be updated until version 9.2. However, views can be updated from version 9.3 onwards; nonetheless, there are many limitations in updating the view. These details are described in the official document.

3.1.4. Planner and Executor

The planner receives a query tree from the rewriter and generates a (query) plan tree that can be processed by the executor most effectively.

The planner in PostgreSQL is based on pure cost-based optimization. It does not support rule-based optimization or hints. This planner is the most complex subsystem in PostgreSQL. Therefore, an overview of the planner will be provided in the subsequent sections of this chapter.

pg_hint_plan

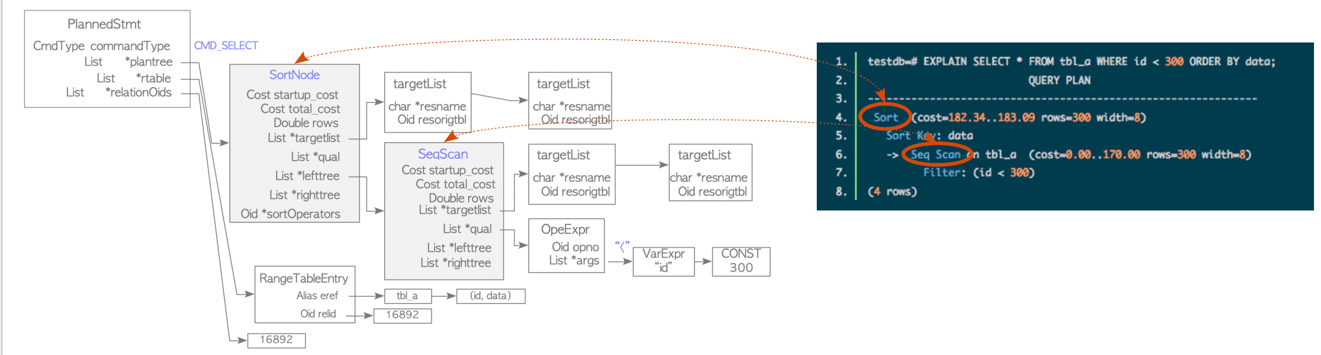
PostgreSQL does not support planner hints in SQL, and it will not be supported forever. If you want to use hints in your queries, the extension referred to *pg_hint_plan* will be worth considering. Refer to the official site in detail.

As in other RDBMS, the EXPLAIN command in PostgreSQL displays the plan tree itself. A specific example is shown below:

```
testdb=# EXPLAIN SELECT * FROM tbl_a WHERE id < 300 ORDER BY data;
                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=182.34..183.09 rows=300 width=8)
   Sort Key: data
   ->  Seq Scan on tbl_a  (cost=0.00..170.00 rows=300 width=8)
         Filter: (id < 300)
(4 rows)
```

This result shows the plan tree shown in Fig. 3.5.

Fig. 3.5. A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command.



```
testdb=# EXPLAIN SELECT * FROM tbl_a WHERE id < 300 ORDER BY data;
                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=182.34..183.09 rows=300 width=8)
   Sort Key: data
   ->  Seq Scan on tbl_a  (cost=0.00..170.00 rows=300 width=8)
         Filter: (id < 300)
(4 rows)
```

A plan tree is composed of elements called *plan nodes*, and it is connected to the plantree list of the *PlannedStmt* structure. These elements are defined in plannodes.h. Details will be explained in Section 3.3.3 (and Section 3.5.4.2).

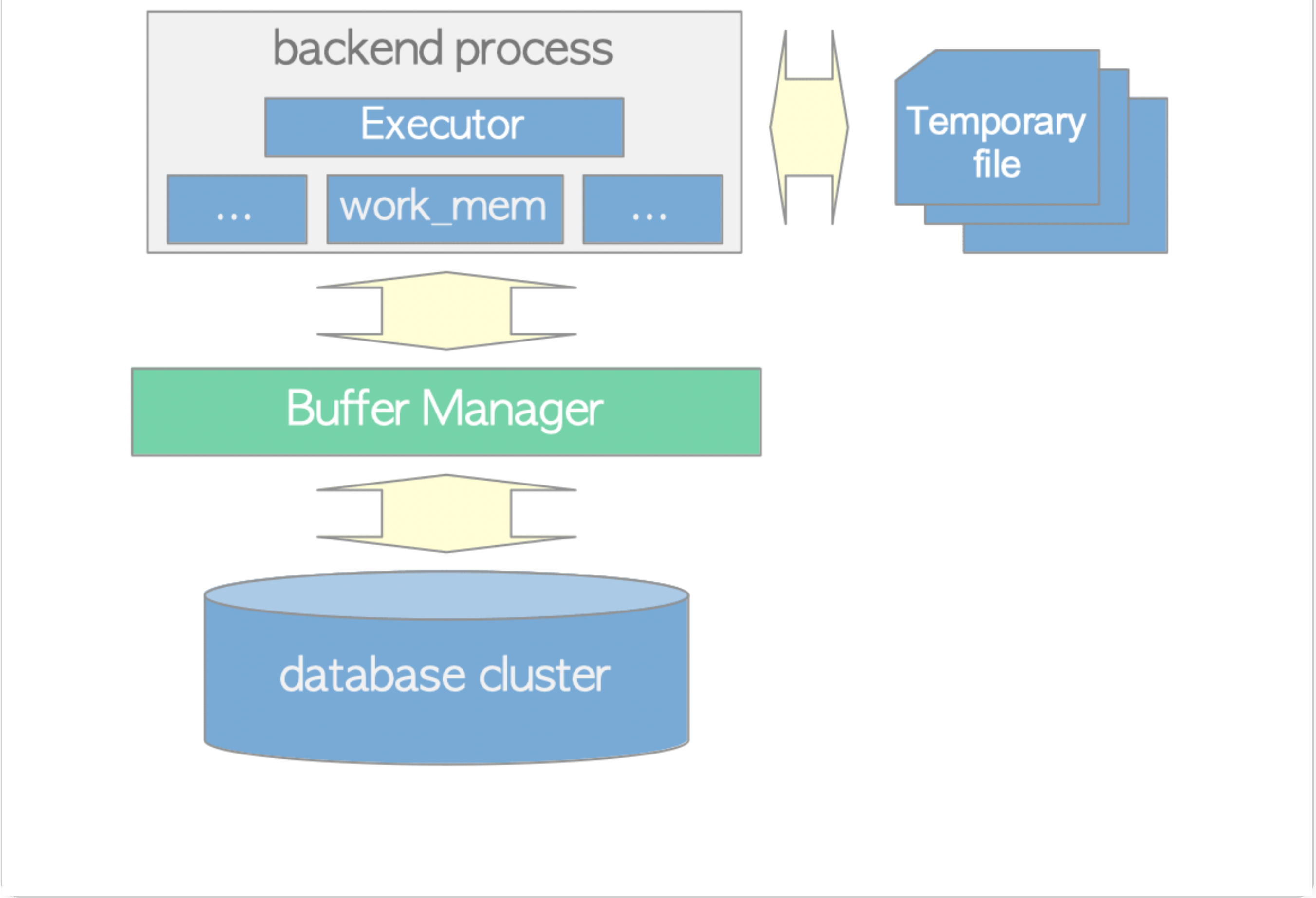
Each plan node has information that the executor requires for processing. In the case of a single-table query, the executor processes from the end of the plan tree to the root.

For example, the plan tree shown in Fig. 3.5 is a list of a sort node and a sequential scan node. Therefore, the executor scans the table *tbl_a* by a sequential scan and then sorts the obtained result.

The executor reads and writes tables and indexes in the database cluster via the buffer manager described in Chapter 8. When processing a query, the executor uses some memory areas, such as temp_buffers and work_mem, allocated in advance and creates temporary files if necessary.

In addition, when accessing tuples, PostgreSQL uses the concurrency control mechanism to maintain consistency and isolation of the running transactions. The concurrency control mechanism is described in Chapter 5.

Fig. 3.6. The relationship among the executor, buffer manager and temporary files.



3.2. Cost Estimation in Single-Table Query

Go to Section 3.2.

Go to Part 2, Part 3 ➔