Chapter 3

Query Processing (Part 2)

← Back to Part 1 Go to Part 3 →

3.2. Cost Estimation in Single-Table Query

PostgreSQL's query optimization is based on cost. Costs are dimensionless values, and they are not absolute performance indicators, but rather indicators to compare the relative performance of operations.

Costs are estimated by the functions defined in costsize.c. All operations executed by the executor have corresponding cost functions. For example, the costs of sequential scans and index scans are estimated by cost_seqscan() and cost_index(), respectively.

In PostgreSQL, there are three kinds of costs: **start-up**, **run** and **total**. The total cost is the sum of the start-up and run costs, so only the start-up and run costs are independently estimated.

- The **start-up** cost is the cost expended before the first tuple is fetched. For example, the start-up cost of the index scan node is the cost of reading index pages to access the first tuple in the target table.
- The **run** cost is the cost of fetching all tuples.
- The **total** cost is the sum of the costs of both start-up and run costs.

The EXPLAIN command shows both of start-up and total costs in each operation. The simplest example is shown below:

```
1. testdb=# EXPLAIN SELECT * FROM tbl;
2.                         QUERY PLAN
3. -------------------------------------------------------------
4.  Seq Scan on tbl  (cost=0.00..145.00 rows=10000 width=8)
5. (1 row)
```

In Line 4, the command shows information about the sequential scan. In the cost section, there are two values; 0.00 and 145.00. In this case, the start-up and total costs are 0.00 and 145.00, respectively.

In this section, we will explore how to estimate the sequential scan, index scan, and sort operation in detail.

In the following explanations, we will use a specific table and an index that are shown below:

```
testdb=# CREATE TABLE tbl (id int PRIMARY KEY, data int);
testdb=# CREATE INDEX tbl_data_idx ON tbl (data);
testdb=# INSERT INTO tbl SELECT generate_series(1,10000),generate_series(1,10000);
testdb=# ANALYZE;
testdb=# \d tbl
      Table "public.tbl"
```

```
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
 data   | integer |
Indexes:
    "tbl_pkey" PRIMARY KEY, btree (id)
    "tbl_data_idx" btree (data)
```

3.2.1. Sequential Scan

The cost of the sequential scan is estimated by the cost_seqscan() function. In this subsection, we will explore how to estimate the sequential scan cost of the following query:

```
testdb=# SELECT * FROM tbl WHERE id < 8000;
```

In the sequential scan, the start-up cost is equal to 0, and the run cost is defined by the following equation:

$$\begin{aligned}\texttt{'run cost'} &= \texttt{'cpu run cost'} + \texttt{'disk run cost'} \\ &= (\texttt{cpu\_tuple\_cost} + \texttt{cpu\_operator\_cost}) \times N_{tuple} + \texttt{seq\_page\_cost} \times N_{page},\end{aligned}$$

where seq_page_cost, cpu_tuple_cost and cpu_operator_cost are set in the postgresql.conf file, and the default values are *1.0*, *0.01*, and *0.0025*, respectively. N_{tuple} and N_{page} are the numbers of all tuples and all pages of this table, respectively. These numbers can be shown using the following query:

```
testdb=# SELECT relpages, reltuples FROM pg_class WHERE relname = 'tbl';
 relpages | reltuples
----------+-----------
       45 |     10000
(1 row)
```

$$N_{tuple} = 10000, \tag{1}$$
$$N_{page} = 45. \tag{2}$$

Therefore,

$$\texttt{'run cost'} = (0.01 + 0.0025) \times 10000 + 1.0 \times 45 = 170.0.$$

Finally,

$$\texttt{'total cost'} = 0.0 + 170.0 = 170.$$

For confirmation, the result of the EXPLAIN command of the above query is shown below:

```
1. testdb=# EXPLAIN SELECT * FROM tbl WHERE id < 8000;
2.                         QUERY PLAN
3. -------------------------------------------------------
4.  Seq Scan on tbl  (cost=0.00..170.00 rows=8000 width=8)
5.    Filter: (id < 8000)
6. (2 rows)
```

In Line 4, we can see that the start-up and total costs are 0.00 and 170.00, respectively. It is also estimated that 8000 rows (tuples) will be selected by scanning all rows.

In line 5, a filter 'Filter:(id < 8000)' of the sequential scan is shown. More precisely, it is called a *table level filter predicate*. Note that this type of filter is used when reading all the tuples in the

table, and it does not narrow the scanned range of table pages.

As understood from the run-cost estimation, PostgreSQL assumes that all pages will be read from storage. In other words, PostgreSQL does not consider whether the scanned page is in the shared buffers or not.

3.2.2. Index Scan

Although PostgreSQL supports some index methods, such as BTree, GiST, GIN and BRIN, the cost of the index scan is estimated using the common cost function cost_index().

In this subsection, we explore how to estimate the index scan cost of the following query:

```
testdb=# SELECT id, data FROM tbl WHERE data < 240;
```

Before estimating the cost, the numbers of the index pages and index tuples, $N_{index,page}$ and $N_{index,tuple}$, are shown below:

```
testdb=# SELECT relpages, reltuples FROM pg_class WHERE relname = 'tbl_data_idx';
 relpages | reltuples
----------+-----------
       30 |     10000
(1 row)
```

$$N_{index,tuple} = 10000, \tag{3}$$
$$N_{index,page} = 30. \tag{4}$$

3.2.2.1. Start-Up Cost

The start-up cost of the index scan is the cost of reading the index pages to access the first tuple in the target table. It is defined by the following equation:

$$\texttt{'start-up cost'} = \{\text{ceil}(\log_2(N_{index,tuple})) + (H_{index} + 1) \times 50\} \times \texttt{cpu\_operator\_cost},$$

where H_{index} is the height of the index tree.

In this case, according to (3), $N_{index,tuple}$ is 10000, H_{index} is 1; `cpu_operator_cost` is 0.0025 (by default). Therefore,

$$\texttt{'start-up cost'} = \{\text{ceil}(\log_2(10000)) + (1 + 1) \times 50\} \times 0.0025 = 0.285. \tag{5}$$

3.2.2.2. Run Cost

The run cost of the index scan is the sum of the CPU costs and the I/O (input/output) costs of both the table and the index:

$$\texttt{'run cost'} = (\texttt{'index cpu cost'} + \texttt{'table cpu cost'}) + (\texttt{'index IO cost'} + \texttt{'table IO cost'}).$$

If the Index-Only Scans, which is described in Section 7.2, can be applied, `'table cpu cost'` and `'table IO cost'` are not estimated.

The first three costs (i.e., index CPU cost, table CPU cost, and index I/O cost) are shown below:

$$\begin{aligned}
\texttt{'index cpu cost'} &= \texttt{Selectivity} \times N_{index,tuple} \times (\texttt{cpu_index_tuple_cost} + \texttt{qual_op_cost}),\\
\texttt{'table cpu cost'} &= \texttt{Selectivity} \times N_{tuple} \times \texttt{cpu_tuple_cost},\\
\texttt{'index IO cost'} &= \text{ceil}(\texttt{Selectivity} \times N_{index,page}) \times \texttt{random_page_cost},
\end{aligned}$$

where

- cpu_index_tuple_cost and random_page_cost are set in the postgresql.conf file. The defaults are 0.005 and 4.0, respectively.
- `qual_op_cost` is, roughly speaking, the cost of evaluating the index predicate. The default is 0.0025.
- `Selectivity` is the proportion of the search range of the index that satisfies the WHERE clause, it is a floating-point number from 0 to 1. ($\texttt{Selectivity} \times N_{tuple}$) means *the number of the table tuples to be read*, ($\texttt{Selectivity} \times N_{index,page}$) means *the number of the index pages to be read*.

Selectivity is described in detail in ℹ below.

ℹ Selectivity

The selectivity of query predicates is estimated using either the histogram_bounds or the MCV (Most Common Value), both of which are stored in the statistics information in the pg_stats. Here, the calculation of the selectivity is briefly described using specific examples. More details are provided in the official document.

The MCV of each column of a table is stored in the pg_stats view as a pair of columns named *most_common_vals* and *most_common_freqs*:

- *most_common_vals* is a list of the MCVs in the column.
- *most_common_freqs* is a list of the frequencies of the MCVs.

Here is a simple example: The table "countries" has two columns:

- a column 'country': This column stores the country name.
- a column 'continent': This column stores the continent name to which the country belongs.

```
testdb=# \d countries
   Table "public.countries"
  Column   | Type | Modifiers
-----------+------+-----------
 country   | text |
 continent | text |
Indexes:
    "continent_idx" btree (continent)

testdb=# SELECT continent, count(*) AS "number of countries",
testdb-#     (count(*)/(SELECT count(*) FROM countries)::real) AS "number of countries / all countries"
testdb-#       FROM countries GROUP BY continent ORDER BY "number of countries" DESC;
   continent   | number of countries | number of countries / all countries
---------------+---------------------+-------------------------------------
 Africa        |                  53 |                  0.274611398963731
 Europe        |                  47 |                  0.243523316062176
 Asia          |                  44 |                  0.227979274611399
 North America |                  23 |                  0.119170984455959
 Oceania       |                  14 |                 0.0725388601036269
 South America |                  12 |                 0.0621761658031088
(6 rows)
```

Consider the following query, which has a WHERE clause, 'continent = 'Asia'':

```
testdb=# SELECT * FROM countries WHERE continent = 'Asia';
```

In this case, the planner estimates the index scan cost using the MCV of the 'continent' column. The `most_common_vals` and `most_common_freqs` of this column are shown below:

```
testdb=# \x
Expanded display is on.
testdb=# SELECT most_common_vals, most_common_freqs FROM pg_stats
testdb-#                 WHERE tablename = 'countries' AND attname='continent';
-[ RECORD 1 ]-----+-----------------------------------------------------------
most_common_vals  | {Africa,Europe,Asia,"North America",Oceania,"South America"}
most_common_freqs | {0.274611,0.243523,0.227979,0.119171,0.0725389,0.0621762}
```

The value of most_common_freqs corresponding to 'Asia' of the most_common_vals is 0.227979. Therefore, 0.227979 is used as the selectivity in this estimation.

If the MCV cannot be used, e.g., the target column type is integer or double precision, then the value of the *histogram_bounds* of the target column is used to estimate the cost.

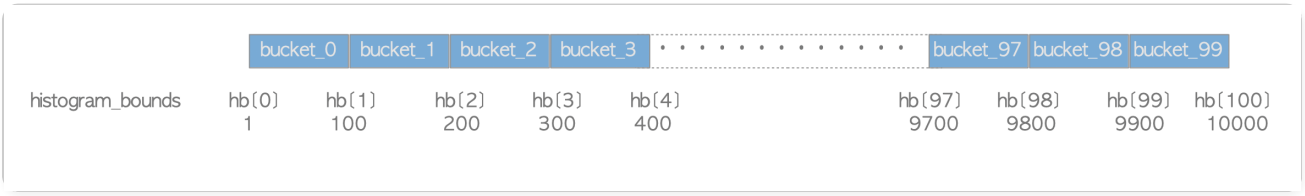
- **histogram_bounds** is a list of values that divide the column's values into groups of approximately equal population.

A specific example is shown. This is the value of the histogram_bounds of the column 'data' in the table 'tbl':

```
testdb=# SELECT histogram_bounds FROM pg_stats WHERE tablename = 'tbl' AND attname = 'data';
                                                histogram_bounds
------------------------------------------------------------------------------------------------------------
 {1,100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600,1700,1800,1900,2000,2100,
2200,2300,2400,2500,2600,2700,2800,2900,3000,3100,3200,3300,3400,3500,3600,3700,3800,3900,4000,4100,
4200,4300,4400,4500,4600,4700,4800,4900,5000,5100,5200,5300,5400,5500,5600,5700,5800,5900,6000,6100,
6200,6300,6400,6500,6600,6700,6800,6900,7000,7100,7200,7300,7400,7500,7600,7700,7800,7900,8000,8100,
8200,8300,8400,8500,8600,8700,8800,8900,9000,9100,9200,9300,9400,9500,9600,9700,9800,9900,10000}
(1 row)
```

By default, the histogram_bounds is divided into 100 buckets. Figure 3.7 illustrates the buckets and the corresponding histogram_bounds in this example. Buckets are numbered starting from 0, and every bucket stores (approximately) the same number of tuples. The values of histogram_bounds are the bounds of the corresponding buckets. For example, the 0th value of the histogram bounds is 1, which means that it is the minimum value of the tuples stored in bucket 0. The 1st value is 100 and this is the minimum value of the tuples stored in bucket 1, and so on.

Fig. 3.7. Buckets and histogram_bounds.



Next, the calculation of the selectivity will be shown using the example in this subsection. The query has a WHERE clause 'data < 240' and the value '240' is in the second bucket. In this case, the selectivity can be derived by applying linear interpolation. Thus, the selectivity of the column 'data' in this query is calculated using the following equation:

$$\texttt{Selectivity} = \frac{2 + (240 - \texttt{hb[2]})/(\texttt{hb[3]} - \texttt{hb[2]})}{100} = \frac{2 + (240 - 200)/(300 - 200)}{100} = \frac{2 + 40/100}{100} = 0.024. \quad (6)$$

Thus, according to (1),(3),(4) and (6),

$$\texttt{'index cpu cost'} = 0.024 \times 10000 \times (0.005 + 0.0025) = 1.8, \tag{7}$$
$$\texttt{'table cpu cost'} = 0.024 \times 10000 \times 0.01 = 2.4, \tag{8}$$
$$\texttt{'index IO cost'} = \text{ceil}(0.024 \times 30) \times 4.0 = 4.0. \tag{9}$$

'table IO cost' is defined by the following equation:

$$\texttt{'table IO cost'} = \texttt{max\_IO\_cost} + \texttt{indexCorrelation}^2 \times (\texttt{min\_IO\_cost} - \texttt{max\_IO\_cost}).$$

`max_IO_cost` is the worst case of the IO cost, that is, the cost of randomly scanning all table pages; this cost is defined by the following equation:

$$\texttt{max\_IO\_cost} = N_{page} \times \texttt{random\_page\_cost}.$$

In this case, according to (2), $N_{page} = 45$, and thus

$$\texttt{max\_IO\_cost} = 45 \times 4.0 = 180.0. \tag{10}$$

`min_IO_cost` is the best case of the IO cost, that is, the cost of sequentially scanning the selected table pages; this cost is defined by the following equation:

$$\texttt{min\_IO\_cost} = 1 \times \texttt{random\_page\_cost} + (\text{ceil}(\texttt{Selectivity} \times N_{page}) - 1) \times \texttt{seq\_page\_cost}.$$

In this case,

$$\texttt{min\_IO\_cost} = 1 \times 4.0 + (\text{ceil}(0.024 \times 45)) - 1) \times 1.0 = 5.0. \tag{11}$$

`indexCorrelation` is described in detail in ℹ below, and in this example,

$$\texttt{indexCorrelation} = 1.0. \tag{12}$$

Thus, according to (10),(11) and (12),

$$\texttt{'table IO cost'} = 180.0 + 1.0^2 \times (5.0 - 180.0) = 5.0. \tag{13}$$

Finally, according to (7),(8),(9) and (13),

$$\texttt{'run cost'} = (1.8 + 2.4) + (4.0 + 5.0) = 13.2. \tag{14}$$

ℹ Index Correlation

Index correlation is a statistical correlation between the physical row ordering and the logical ordering of the column values (cited from the official document). This ranges from -1 to $+1$. To understand the relation between the index scan and the index correlation, a specific example is shown below.

The table *tbl_corr* has five columns: two columns are text type and three columns are integer type. The three integer columns store numbers from 1 to 12. Physically, tbl_corr is composed of three pages, and each page has four tuples. Each integer type column has an index with a name such as index_col_asc and so on.

```
testdb=# \d tbl_corr
    Table "public.tbl_corr"
  Column  |  Type   | Modifiers
----------+---------+-----------
 col      | text    |
 col_asc  | integer |
 col_desc | integer |
 col_rand | integer |
 data     | text    |
Indexes:
    "tbl_corr_asc_idx" btree (col_asc)
    "tbl_corr_desc_idx" btree (col_desc)
```

```
    "tbl_corr_rand_idx" btree (col_rand)
```

```
testdb=# SELECT col,col_asc,col_desc,col_rand
testdb-#                         FROM tbl_corr;
   col    | col_asc | col_desc | col_rand
----------+---------+----------+----------
 Tuple_1  |       1 |       12 |        3
 Tuple_2  |       2 |       11 |        8
 Tuple_3  |       3 |       10 |        5
 Tuple_4  |       4 |        9 |        9
 Tuple_5  |       5 |        8 |        7
 Tuple_6  |       6 |        7 |        2
 Tuple_7  |       7 |        6 |       10
 Tuple_8  |       8 |        5 |       11
 Tuple_9  |       9 |        4 |        4
 Tuple_10 |      10 |        3 |        1
 Tuple_11 |      11 |        2 |       12
 Tuple_12 |      12 |        1 |        6
(12 rows)
```

The index correlations of these columns are shown below:

```
testdb=# SELECT tablename,attname, correlation FROM pg_stats WHERE tablename = 'tbl_corr';
 tablename | attname  | correlation
-----------+----------+-------------
 tbl_corr  | col_asc  |           1
 tbl_corr  | col_desc |          -1
 tbl_corr  | col_rand |    0.125874
(3 rows)
```

When the following query is executed, PostgreSQL reads only the first page because all of the target tuples are stored in the first page. See Fig. 3.8(a).

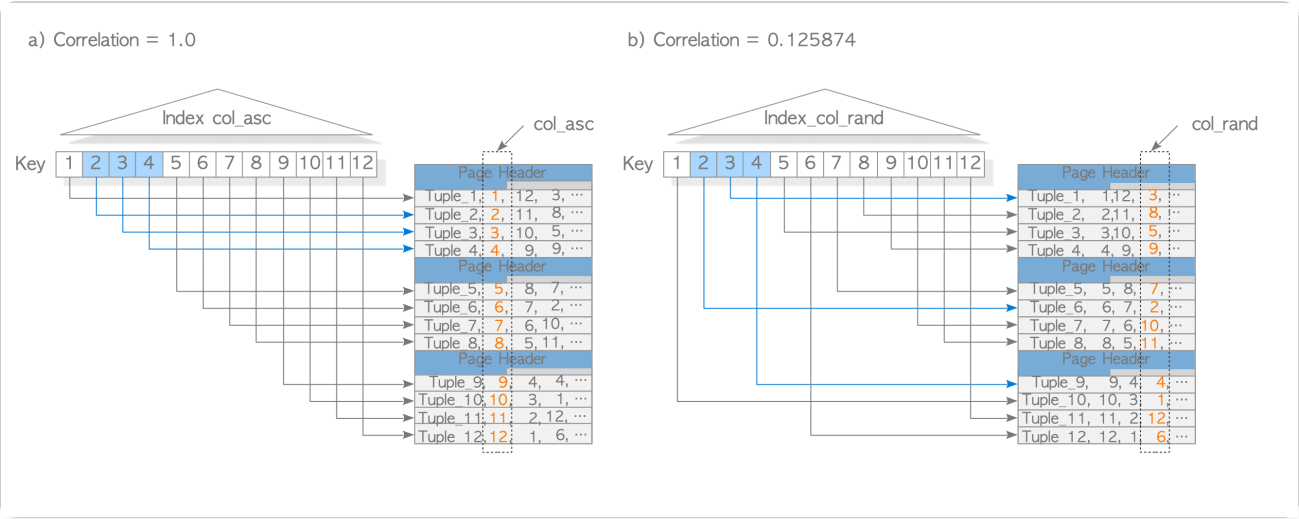
```
testdb=# SELECT * FROM tbl_corr WHERE col_asc BETWEEN 2 AND 4;
```

On the other hand, when the following query is executed, PostgreSQL has to read all pages. See Fig. 3.8(b).

```
testdb=# SELECT * FROM tbl_corr WHERE col_rand BETWEEN 2 AND 4;
```

In this way, the index correlation is a statistical correlation that reflects the impact of random access caused by the discrepancy between the index ordering and the physical tuple ordering in the table when estimating the index scan cost.

Fig. 3.8. Index correlation.



3.2.2.3. Total Cost

According to (3) and (14),

$$\texttt{'total cost'} = 0.285 + 13.2 = 13.485. \tag{15}$$

For confirmation, the result of the EXPLAIN command of the above SELECT query is shown below:

```
1. testdb=# EXPLAIN SELECT id, data FROM tbl WHERE data < 240;
2.                                 QUERY PLAN
3. ------------------------------------------------------------------------------
4.  Index Scan using tbl_data_idx on tbl  (cost=0.29..13.49 rows=240 width=8)
5.    Index Cond: (data < 240)
6. (2 rows)
```

In Line 4, we can find that the start-up and total costs are 0.29 and 13.49, respectively, and it is estimated that 240 rows (tuples) will be scanned.

In Line 5, an index condition 'Index Cond:(data < 240)' of the index scan is shown. More precisely, this condition is called an *access predicate*, and it expresses the start and stop conditions of the index scan.

According to this post, EXPLAIN command in PostgreSQL does not distinguish between the access predicate and index filter predicate. Therefore, if you analyze the output of EXPLAIN, pay attention not only to the index conditions but also to the estimated value of rows.

Indexes Internals

This document does not explain indexes in details. To understand them, I recommend to read the valuable posts shown below:

- Indexes in PostgreSQL — 1
- Indexes in PostgreSQL — 2
- Indexes in PostgreSQL — 3 (Hash)
- Indexes in PostgreSQL — 4 (Btree)
- Indexes in PostgreSQL — 5 (GiST)
- Indexes in PostgreSQL — 6 (SP-GiST)
- Indexes in PostgreSQL — 7 (GIN)
- Indexes in PostgreSQL — 9 (BRIN)

seq_page_cost and random_page_cost

The default values of seq_page_cost and random_page_cost are 1.0 and 4.0, respectively. This means that PostgeSQL assumes that the random scan is four times slower than the sequential scan. In other words, the default value of PostgreSQL is based on using HDDs.

On the other hand, in recent days, the default value of random_page_cost is too large because SSDs are mostly used. If the default value of random_page_cost is used despite using an SSD, the planner may select ineffective plans. Therefore, when using an SSD, it is better to change the value of random_page_cost to 1.0.

This blog reported the problem when using the default value of random_page_cost.

3.2.3. Sort

The sort path is used for sorting operations, such as ORDER BY, the preprocessing of merge join operations, and other functions. The cost of sorting is estimated using the cost_sort() function.

In the sorting operation, if all tuples to be sorted can be stored in work_mem, the quicksort algorithm is used. Otherwise, a temporary file is created and the file merge sort algorithm is used.

The start-up cost of the sort path is the cost of sorting the target tuples. Therefore, the cost is $O(N_{sort} \times \log_2(N_{sort}))$, where N_{sort} is the number of the tuples to be sorted. The run cost of the sort path is the cost of reading the sorted tuples. Therefore the cost is $O(N_{sort})$.

In this subsection, we explore how to estimate the sorting cost of the following query. Assume that this query will be sorted in work_mem, without using temporary files.

```
testdb=# SELECT id, data FROM tbl WHERE data < 240 ORDER BY id;
```

In this case, the start-up cost is defined in the following equation:

$$\texttt{'start-up cost'} = C + \texttt{comparison\_cost} \times N_{sort} \times \log_2(N_{sort}),$$

where

- C is the total cost of the last scan, that is, the total cost of the index scan; according to (15), it is 13.485.
- N_{sort} is the number of tuples to be sorted. In this case, it is 240.
- `comparison_cost` is defined in $2 \times$ `cpu_operator_cost`.

Therefore, the `start-up cost` is calculated as follows:

$$\texttt{'start-up cost'} = 13.485 + (2 \times 0.0025) \times 240.0 \times \log_2(240.0) = 22.973.$$

The `run cost` is the cost of reading sorted tuples in the memory. Therefore,

$$\texttt{'run cost'} = \texttt{cpu\_operator\_cost} \times N_{sort} = 0.0025 \times 240 = 0.6.$$

Finally,

$$\texttt{'total cost'} = 22.973 + 0.6 = 23.573.$$

For confirmation, the result of the EXPLAIN command of the above SELECT query is shown below:

```
1.  testdb=# EXPLAIN SELECT id, data FROM tbl WHERE data < 240 ORDER BY id;
2.                                 QUERY PLAN
3.  ------------------------------------------------------------------------------
4.   Sort  (cost=22.97..23.57 rows=240 width=8)
5.     Sort Key: id
6.     ->  Index Scan using tbl_data_idx on tbl  (cost=0.29..13.49 rows=240 width=
    8)
```

```
7.            Index Cond: (data < 240)
8.  (4 rows)
```

In line 4, we can find that the start-up cost and total cost are 22.97 and 23.57, respectively.

3.3. Creating the Plan Tree of a Single-Table Query

As the processing of the planner is very complicated, this section describes the simplest process, namely, how a plan tree of a single-table query is created. More complex processing, namely, how a plan tree of a multi-table query is created, is described in Section 3.6.

The planner in PostgreSQL performs three steps, as shown below:

1. Carry out preprocessing.
2. Get the cheapest access path by estimating the costs of all possible access paths.
3. Create the plan tree from the cheapest path.

An access path is a unit of processing for estimating the cost. For example, the sequential scan, index scan, sort, and various join operations have their corresponding paths. Access paths are used only inside the planner to create the plan tree. The most fundamental data structure of access paths is the Path structure defined in pathnodes.h, and it corresponds to the sequential scan. All other access paths are based on it. Details will be described in the following explanations.

```
typedef struct PathKey
{
        pg_node_attr(no_read, no_query_jumble)
        NodeTag                 type;

        /* the value that is ordered */
        EquivalenceClass *pk_eclass pg_node_attr(copy_as_scalar, equal_as_scalar);
        Oid             pk_opfamily;    /* btree opfamily defining the ordering */
        int             pk_strategy;    /* sort direction (ASC or DESC) */
        bool            pk_nulls_first; /* do NULLs come before normal values? */
} PathKey;

typedef struct Path
{
        pg_node_attr(no_copy_equal, no_read, no_query_jumble)

        NodeTag                 type;

        /* tag identifying scan/join method */
        NodeTag                 pathtype;

        /*
         * the relation this path can build
         *
         * We do NOT print the parent, else we'd be in infinite recursion.  We can
         * print the parent's relids for identification purposes, though.
         */
        RelOptInfo *parent pg_node_attr(write_only_relids);

        /*
         * list of Vars/Exprs, cost, width
         *
         * We print the pathtarget only if it's not the default one for the rel.
```

```
         */
        PathTarget *pathtarget pg_node_attr(write_only_nondefault_pathtarget);

        /*
         * parameterization info, or NULL if none
         *
         * We do not print the whole of param_info, since it's printed via
         * RelOptInfo; it's sufficient and less cluttering to print just the
         * required outer relids.
         */
        ParamPathInfo *param_info pg_node_attr(write_only_req_outer);

        /* engage parallel-aware logic? */
        bool            parallel_aware;
        /* OK to use as part of parallel plan? */
        bool            parallel_safe;
        /* desired # of workers; 0 = not parallel */
        int                     parallel_workers;

        /* estimated size/costs for path (see costsize.c for more info) */
        Cardinality rows;                       /* estimated number of result tuples
*/
        Cost            startup_cost;   /* cost expended before fetching any tuples
*/
        Cost            total_cost;             /* total cost (assuming all tuples fe
tched) */

        /* sort ordering of path's output; a List of PathKey nodes; see above */
        List       *pathkeys;
} Path;
```

To process the above steps, the planner internally creates a PlannerInfo structure, and holds the query tree, the information about the relations contained in the query, the access paths, and so on.

```
/*----------
 * PlannerInfo
 *              Per-query information for planning/optimization
 *
 * This struct is conventionally called "root" in all the planner routines.
 * It holds links to all of the planner's working state, in addition to the
 * original Query.  Note that at present the planner extensively modifies
 * the passed-in Query data structure; someday that should stop.
 *
 * For reasons explained in optimizer/optimizer.h, we define the typedef
 * either here or in that header, whichever is read first.
 *
 * Not all fields are printed.  (In some cases, there is no print support for
 * the field type; in others, doing so would lead to infinite recursion or
 * bloat dump output more than seems useful.)
 *----------
 */
#ifndef HAVE_PLANNERINFO_TYPEDEF
typedef struct PlannerInfo PlannerInfo;
#define HAVE_PLANNERINFO_TYPEDEF 1
#endif

struct PlannerInfo
{
        pg_node_attr(no_copy_equal, no_read, no_query_jumble)

        NodeTag                 type;
```

```
        /* the Query being planned */
        Query     *parse;

        /* global info for current planner run */
        PlannerGlobal *glob;

        /* 1 at the outermost Query */
        Index          query_level;

        /* NULL at outermost Query */
        PlannerInfo *parent_root pg_node_attr(read_write_ignore);

        /*
         * plan_params contains the expressions that this query level needs to
         * make available to a lower query level that is currently being planned.
         * outer_params contains the paramIds of PARAM_EXEC Params that outer
         * query levels will make available to this query level.
         */
        /* list of PlannerParamItems, see below */
        List      *plan_params;
        Bitmapset  *outer_params;

        /*
         * simple_rel_array holds pointers to "base rels" and "other rels" (see
         * comments for RelOptInfo for more info).  It is indexed by rangetable
         * index (so entry 0 is always wasted).  Entries can be NULL when an RTE
         * does not correspond to a base relation, such as a join RTE or an
         * unreferenced view RTE; or if the RelOptInfo hasn't been made yet.
         */
        struct RelOptInfo **simple_rel_array pg_node_attr(array_size(simple_rel_ar
ray_size));
        /* allocated size of array */
        int                    simple_rel_array_size;

        /*
         * simple_rte_array is the same length as simple_rel_array and holds
         * pointers to the associated rangetable entries.  Using this is a shade
         * faster than using rt_fetch(), mostly due to fewer indirections.  (Not
         * printed because it'd be redundant with parse->rtable.)
         */
        RangeTblEntry **simple_rte_array pg_node_attr(read_write_ignore);

        /*
         * append_rel_array is the same length as the above arrays, and holds
         * pointers to the corresponding AppendRelInfo entry indexed by
         * child_relid, or NULL if the rel is not an appendrel child.  The array
         * itself is not allocated if append_rel_list is empty.  (Not printed
         * because it'd be redundant with append_rel_list.)
         */
        struct AppendRelInfo **append_rel_array pg_node_attr(read_write_ignore);

        /*
         * all_baserels is a Relids set of all base relids (but not joins or
         * "other" rels) in the query.  This is computed in deconstruct_jointree.
         */
        Relids         all_baserels;

        /*
         * outer_join_rels is a Relids set of all outer-join relids in the query.
         * This is computed in deconstruct_jointree.
```

```
     */
    Relids          outer_join_rels;

    /*
     * all_query_rels is a Relids set of all base relids and outer join relids
     * (but not "other" relids) in the query.  This is the Relids identifier
     * of the final join we need to form.  This is computed in
     * deconstruct_jointree.
     */
    Relids          all_query_rels;

    /*
     * join_rel_list is a list of all join-relation RelOptInfos we have
     * considered in this planning run.  For small problems we just scan the
     * list to do lookups, but when there are many join relations we build a
     * hash table for faster lookups.  The hash table is present and valid
     * when join_rel_hash is not NULL.  Note that we still maintain the list
     * even when using the hash table for lookups; this simplifies life for
     * GEQO.
     */
    List       *join_rel_list;
    struct HTAB *join_rel_hash pg_node_attr(read_write_ignore);

    /*
     * When doing a dynamic-programming-style join search, join_rel_level[k]
     * is a list of all join-relation RelOptInfos of level k, and
     * join_cur_level is the current level.  New join-relation RelOptInfos are
     * automatically added to the join_rel_level[join_cur_level] list.
     * join_rel_level is NULL if not in use.
     *
     * Note: we've already printed all baserel and joinrel RelOptInfos above,
     * so we don't dump join_rel_level or other lists of RelOptInfos.
     */
    /* lists of join-relation RelOptInfos */
    List      **join_rel_level pg_node_attr(read_write_ignore);
    /* index of list being extended */
    int                     join_cur_level;

    /* init SubPlans for query */
    List       *init_plans;

    /*
     * per-CTE-item list of subplan IDs (or -1 if no subplan was made for that
     * CTE)
     */
    List       *cte_plan_ids;

    /* List of Lists of Params for MULTIEXPR subquery outputs */
    List       *multiexpr_params;

    /* list of JoinDomains used in the query (higher ones first) */
    List       *join_domains;

    /* list of active EquivalenceClasses */
    List       *eq_classes;

    /* set true once ECs are canonical */
    bool            ec_merging_done;

    /* list of "canonical" PathKeys */
    List       *canon_pathkeys;
```

```
        /*
         * list of OuterJoinClauseInfos for mergejoinable outer join clauses
         * w/nonnullable var on left
         */
        List      *left_join_clauses;

        /*
         * list of OuterJoinClauseInfos for mergejoinable outer join clauses
         * w/nonnullable var on right
         */
        List      *right_join_clauses;

        /*
         * list of OuterJoinClauseInfos for mergejoinable full join clauses
         */
        List      *full_join_clauses;

        /* list of SpecialJoinInfos */
        List      *join_info_list;

        /* counter for assigning RestrictInfo serial numbers */
        int                   last_rinfo_serial;

        /*
         * all_result_relids is empty for SELECT, otherwise it contains at least
         * parse->resultRelation.  For UPDATE/DELETE/MERGE across an inheritance
         * or partitioning tree, the result rel's child relids are added.  When
         * using multi-level partitioning, intermediate partitioned rels are
         * included. leaf_result_relids is similar except that only actual result
         * tables, not partitioned tables, are included in it.
         */
        /* set of all result relids */
        Relids         all_result_relids;
        /* set of all leaf relids */
        Relids         leaf_result_relids;

        /*
         * list of AppendRelInfos
         *
         * Note: for AppendRelInfos describing partitions of a partitioned table,
         * we guarantee that partitions that come earlier in the partitioned
         * table's PartitionDesc will appear earlier in append_rel_list.
         */
        List      *append_rel_list;

        /* list of RowIdentityVarInfos */
        List      *row_identity_vars;

        /* list of PlanRowMarks */
        List      *rowMarks;

        /* list of PlaceHolderInfos */
        List      *placeholder_list;

        /* array of PlaceHolderInfos indexed by phid */
        struct PlaceHolderInfo **placeholder_array pg_node_attr(read_write_ignore,
array_size(placeholder_array_size));
        /* allocated size of array */
        int                   placeholder_array_size pg_node_attr(read_write_ignor
e);
```

```
        /* list of ForeignKeyOptInfos */
        List       *fkey_list;

        /* desired pathkeys for query_planner() */
        List       *query_pathkeys;

        /* groupClause pathkeys, if any */
        List       *group_pathkeys;

        /*
         * The number of elements in the group_pathkeys list which belong to the
         * GROUP BY clause.  Additional ones belong to ORDER BY / DISTINCT
         * aggregates.
         */
        int                     num_groupby_pathkeys;

        /* pathkeys of bottom window, if any */
        List       *window_pathkeys;
        /* distinctClause pathkeys, if any */
        List       *distinct_pathkeys;
        /* sortClause pathkeys, if any */
        List       *sort_pathkeys;

        /* Canonicalised partition schemes used in the query. */
        List       *part_schemes pg_node_attr(read_write_ignore);

        /* RelOptInfos we are now trying to join */
        List       *initial_rels pg_node_attr(read_write_ignore);

        /*
         * Upper-rel RelOptInfos. Use fetch_upper_rel() to get any particular
         * upper rel.
         */
        List       *upper_rels[UPPERREL_FINAL + 1] pg_node_attr(read_write_ignore);

        /* Result tlists chosen by grouping_planner for upper-stage processing */
        struct PathTarget *upper_targets[UPPERREL_FINAL + 1] pg_node_attr(read_wri
te_ignore);

        /*
         * The fully-processed groupClause is kept here.  It differs from
         * parse->groupClause in that we remove any items that we can prove
         * redundant, so that only the columns named here actually need to be
         * compared to determine grouping.  Note that it's possible for *all* the
         * items to be proven redundant, implying that there is only one group
         * containing all the query's rows.  Hence, if you want to check whether
         * GROUP BY was specified, test for nonempty parse->groupClause, not for
         * nonempty processed_groupClause.
         *
         * Currently, when grouping sets are specified we do not attempt to
         * optimize the groupClause, so that processed_groupClause will be
         * identical to parse->groupClause.
         */
        List       *processed_groupClause;

        /*
         * The fully-processed distinctClause is kept here.  It differs from
         * parse->distinctClause in that we remove any items that we can prove
         * redundant, so that only the columns named here actually need to be
         * compared to determine uniqueness.  Note that it's possible for *all*
```

```
     * the items to be proven redundant, implying that there should be only
     * one output row.  Hence, if you want to check whether DISTINCT was
     * specified, test for nonempty parse->distinctClause, not for nonempty
     * processed_distinctClause.
     */
    List       *processed_distinctClause;

    /*
     * The fully-processed targetlist is kept here.  It differs from
     * parse->targetList in that (for INSERT) it's been reordered to match the
     * target table, and defaults have been filled in.  Also, additional
     * resjunk targets may be present.  preprocess_targetlist() does most of
     * that work, but note that more resjunk targets can get added during
     * appendrel expansion.  (Hence, upper_targets mustn't get set up till
     * after that.)
     */
    List       *processed_tlist;

    /*
     * For UPDATE, this list contains the target table's attribute numbers to
     * which the first N entries of processed_tlist are to be assigned.  (Any
     * additional entries in processed_tlist must be resjunk.)  DO NOT use the
     * resnos in processed_tlist to identify the UPDATE target columns.
     */
    List       *update_colnos;

    /*
     * Fields filled during create_plan() for use in setrefs.c
     */
    /* for GroupingFunc fixup (can't print: array length not known here) */
    AttrNumber *grouping_map pg_node_attr(read_write_ignore);
    /* List of MinMaxAggInfos */
    List       *minmax_aggs;

    /* context holding PlannerInfo */
    MemoryContext planner_cxt pg_node_attr(read_write_ignore);

    /* # of pages in all non-dummy tables of query */
    Cardinality total_table_pages;

    /* tuple_fraction passed to query_planner */
    Selectivity tuple_fraction;
    /* limit_tuples passed to query_planner */
    Cardinality limit_tuples;

    /*
     * Minimum security_level for quals. Note: qual_security_level is zero if
     * there are no securityQuals.
     */
    Index          qual_security_level;

    /* true if any RTEs are RTE_JOIN kind */
    bool           hasJoinRTEs;
    /* true if any RTEs are marked LATERAL */
    bool           hasLateralRTEs;
    /* true if havingQual was non-null */
    bool           hasHavingQual;
    /* true if any RestrictInfo has pseudoconstant = true */
    bool           hasPseudoConstantQuals;
    /* true if we've made any of those */
    bool           hasAlternativeSubPlans;
```

```
        /* true once we're no longer allowed to add PlaceHolderInfos */
        bool            placeholdersFrozen;
        /* true if planning a recursive WITH item */
        bool            hasRecursion;

        /*
         * Information about aggregates. Filled by preprocess_aggrefs().
         */
        /* AggInfo structs */
        List       *agginfos;
        /* AggTransInfo structs */
        List       *aggtransinfos;
        /* number of aggs with DISTINCT/ORDER BY/WITHIN GROUP */
        int                     numOrderedAggs;
        /* does any agg not support partial mode? */
        bool            hasNonPartialAggs;
        /* is any partial agg non-serializable? */
        bool            hasNonSerialAggs;

        /*
         * These fields are used only when hasRecursion is true:
         */
        /* PARAM_EXEC ID for the work table */
        int                     wt_param_id;
        /* a path for non-recursive term */
        struct Path *non_recursive_path;

        /*
         * These fields are workspace for createplan.c
         */
        /* outer rels above current node */
        Relids          curOuterRels;
        /* not-yet-assigned NestLoopParams */
        List       *curOuterParams;

        /*
         * These fields are workspace for setrefs.c.  Each is an array
         * corresponding to glob->subplans.  (We could probably teach
         * gen_node_support.pl how to determine the array length, but it doesn't
         * seem worth the trouble, so just mark them read_write_ignore.)
         */
        bool       *isAltSubplan pg_node_attr(read_write_ignore);
        bool       *isUsedSubplan pg_node_attr(read_write_ignore);

        /* optional private data for join_search_hook, e.g., GEQO */
        void       *join_search_private pg_node_attr(read_write_ignore);

        /* Does this query modify any partition key columns? */
        bool            partColsUpdated;
};
```

In this section, how plan trees are created from query trees is described using specific examples.

3.3.1. Preprocessing

Before creating a plan tree, the planner carries out some preprocessing of the query tree stored in the PlannerInfo structure.

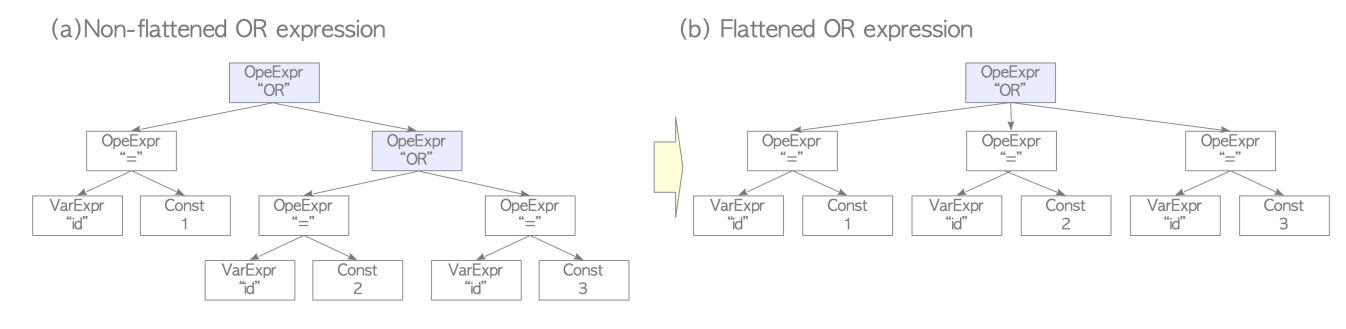
Although preprocessing involves many steps, we only discuss the main preprocessing for the single-table query in this subsection. The other preprocessing operations are described in Section

3.6.

The preprocessing steps include:

1. Simplifying target lists, limit clauses, and so on.
   For example, the eval_const_expressions() function defined in clauses.c rewrites '2 + 2' to '4'.
2. Normalizing Boolean expressions.
   For example, 'NOT (NOT a)' is rewritten to 'a'.
3. Flattening AND/OR expressions.
   AND and OR in the SQL standard are binary operators, but in PostgreSQL internals, they are n-ary operators and the planner always assumes that all nested AND and OR expressions are to be flattened.
   A specific example is shown. Consider a Boolean expression '(id = 1) OR (id = 2) OR (id = 3)'. Figure 3.9(a) shows part of the query tree when using the binary operator. The planner simplified this tree by flattening using a ternary operator. See Fig. 3.9(b).

Fig. 3.9. An example of flattening AND/OR expressions.



3.3.2. Getting the Cheapest Access Path

To get the cheapest access path, the planner estimates the costs of all possible access paths and chooses the cheapest one. More specifically, the planner performs the following operations:

1. Create a RelOptInfo structure to store the access paths and the corresponding costs.
   A RelOptInfo structure is created by the make_one_rel() function and is stored in the *simple_rel_array* of the PlannerInfo structure. See Fig. 3.10. In its initial state, the RelOptInfo holds the *baserestrictinfo* and the *indexlist* if related indexes exist. The baserestrictinfo stores the WHERE clauses of the query, and the indexlist stores the related indexes of the target table.

```
typedef enum RelOptKind
{
        RELOPT_BASEREL,
        RELOPT_JOINREL,
        RELOPT_OTHER_MEMBER_REL,
        RELOPT_OTHER_JOINREL,
        RELOPT_UPPER_REL,
        RELOPT_OTHER_UPPER_REL
} RelOptKind;

/*
 * Is the given relation a simple relation i.e a base or "other" member
 * relation?
 */
#define IS_SIMPLE_REL(rel) \
        ((rel)->reloptkind == RELOPT_BASEREL || \
```

```
         (rel)->reloptkind == RELOPT_OTHER_MEMBER_REL)

/* Is the given relation a join relation? */
#define IS_JOIN_REL(rel)        \
        ((rel)->reloptkind == RELOPT_JOINREL || \
         (rel)->reloptkind == RELOPT_OTHER_JOINREL)

/* Is the given relation an upper relation? */
#define IS_UPPER_REL(rel)       \
        ((rel)->reloptkind == RELOPT_UPPER_REL || \
         (rel)->reloptkind == RELOPT_OTHER_UPPER_REL)

/* Is the given relation an "other" relation? */
#define IS_OTHER_REL(rel) \
        ((rel)->reloptkind == RELOPT_OTHER_MEMBER_REL || \
         (rel)->reloptkind == RELOPT_OTHER_JOINREL || \
         (rel)->reloptkind == RELOPT_OTHER_UPPER_REL)

typedef struct RelOptInfo
{
        pg_node_attr(no_copy_equal, no_read, no_query_jumble)

        NodeTag                 type;

        RelOptKind      reloptkind;

        /*
         * all relations included in this RelOptInfo; set of base + OJ relids
         * (rangetable indexes)
         */
        Relids          relids;

        /*
         * size estimates generated by planner
         */
        /* estimated number of result tuples */
        Cardinality rows;

        /*
         * per-relation planner control flags
         */
        /* keep cheap-startup-cost paths? */
        bool            consider_startup;
        /* ditto, for parameterized paths? */
        bool            consider_param_startup;
        /* consider parallel paths? */
        bool            consider_parallel;

        /*
         * default result targetlist for Paths scanning this relation; list of
         * Vars/Exprs, cost, width
         */
        struct PathTarget *reltarget;

        /*
         * materialization information
         */
        List       *pathlist;           /* Path structures */
        List       *ppilist;            /* ParamPathInfos used in pathlist */
        List       *partial_pathlist;   /* partial Paths */
        struct Path *cheapest_startup_path;
```

```
        struct Path *cheapest_total_path;
        struct Path *cheapest_unique_path;
        List      *cheapest_parameterized_paths;

        /*
         * parameterization information needed for both base rels and join rels
         * (see also lateral_vars and lateral_referencers)
         */
        /* rels directly laterally referenced */
        Relids          direct_lateral_relids;
        /* minimum parameterization of rel */
        Relids          lateral_relids;

        /*
         * information about a base rel (not set for join rels!)
         */
        Index           relid;
        /* containing tablespace */
        Oid                     reltablespace;
        /* RELATION, SUBQUERY, FUNCTION, etc */
        RTEKind                 rtekind;
        /* smallest attrno of rel (often <0) */
        AttrNumber      min_attr;
        /* largest attrno of rel */
        AttrNumber      max_attr;
        /* array indexed [min_attr .. max_attr] */
        Relids     *attr_needed pg_node_attr(read_write_ignore);
        /* array indexed [min_attr .. max_attr] */
        int32      *attr_widths pg_node_attr(read_write_ignore);
        /* relids of outer joins that can null this baserel */
        Relids          nulling_relids;
        /* LATERAL Vars and PHVs referenced by rel */
        List      *lateral_vars;
        /* rels that reference this baserel laterally */
        Relids          lateral_referencers;
        /* list of IndexOptInfo */
        List      *indexlist;
        /* list of StatisticExtInfo */
        List      *statlist;
        /* size estimates derived from pg_class */
        BlockNumber pages;
        Cardinality tuples;
        double          allvisfrac;
        /* indexes in PlannerInfo's eq_classes list of ECs that mention this rel
*/
        Bitmapset  *eclass_indexes;
        PlannerInfo *subroot;           /* if subquery */
        List      *subplan_params; /* if subquery */
        /* wanted number of parallel workers */
        int                     rel_parallel_workers;
        /* Bitmask of optional features supported by the table AM */
        uint32          amflags;

        /*
         * Information about foreign tables and foreign joins
         */
        /* identifies server for the table or join */
        Oid                     serverid;
        /* identifies user to check access as; 0 means to check as current user
*/
        Oid                     userid;
```

```
        /* join is only valid for current user */
        bool            useridiscurrent;
        /* use "struct FdwRoutine" to avoid including fdwapi.h here */
        struct FdwRoutine *fdwroutine pg_node_attr(read_write_ignore);
        void       *fdw_private pg_node_attr(read_write_ignore);

        /*
         * cache space for remembering if we have proven this relation unique
         */
        /* known unique for these other relid set(s) */
        List       *unique_for_rels;
        /* known not unique for these set(s) */
        List       *non_unique_for_rels;

        /*
         * used by various scans and joins:
         */
        /* RestrictInfo structures (if base rel) */
        List       *baserestrictinfo;
        /* cost of evaluating the above */
        QualCost        baserestrictcost;
        /* min security_level found in baserestrictinfo */
        Index           baserestrict_min_security;
        /* RestrictInfo structures for join clauses involving this rel */
        List       *joininfo;
        /* T means joininfo is incomplete */
        bool            has_eclass_joins;

        /*
         * used by partitionwise joins:
         */
        /* consider partitionwise join paths? (if partitioned rel) */
        bool            consider_partitionwise_join;

        /*
         * inheritance links, if this is an otherrel (otherwise NULL):
         */
        /* Immediate parent relation (dumping it would be too verbose) */
        struct RelOptInfo *parent pg_node_attr(read_write_ignore);
        /* Topmost parent relation (dumping it would be too verbose) */
        struct RelOptInfo *top_parent pg_node_attr(read_write_ignore);
        /* Relids of topmost parent (redundant, but handy) */
        Relids          top_parent_relids;

        /*
         * used for partitioned relations:
         */
        /* Partitioning scheme */
        PartitionScheme part_scheme pg_node_attr(read_write_ignore);

        /*
         * Number of partitions; -1 if not yet set; in case of a join relation 0
         * means it's considered unpartitioned
         */
        int                     nparts;
        /* Partition bounds */
        struct PartitionBoundInfoData *boundinfo pg_node_attr(read_write_igno
re);
        /* True if partition bounds were created by partition_bounds_merge() */
        bool            partbounds_merged;
        /* Partition constraint, if not the root */
```

```
        List      *partition_qual;

        /*
         * Array of RelOptInfos of partitions, stored in the same order as bounds
         * (don't print, too bulky and duplicative)
         */
        struct RelOptInfo **part_rels pg_node_attr(read_write_ignore);

        /*
         * Bitmap with members acting as indexes into the part_rels[] array to
         * indicate which partitions survived partition pruning.
         */
        Bitmapset  *live_parts;
        /* Relids set of all partition relids */
        Relids          all_partrels;

        /*
         * These arrays are of length partkey->partnatts, which we don't have at
         * hand, so don't try to print
         */

        /* Non-nullable partition key expressions */
        List      **partexprs pg_node_attr(read_write_ignore);
        /* Nullable partition key expressions */
        List      **nullable_partexprs pg_node_attr(read_write_ignore);
} RelOptInfo;
```

2. Estimate the costs of all possible access paths, and add the access paths to the RelOptInfo structure.
   Details of this processing are as follows:
   1. A path is created, the cost of the sequential scan is estimated, and the estimated costs are written to the path. Then, the path is added to the pathlist of the RelOptInfo structure.
   2. If indexes related to the target table exist, index access paths are created, all index scan costs are estimated, and the estimated costs are written to the path. Then, the index paths are added to the pathlist.
   3. If the bitmap scan can be done, bitmap scan paths are created, all bitmap scan costs are estimated, and the estimated costs are written to the path. Then, the bitmap scan paths are added to the pathlist.
3. Get the cheapest access path in the pathlist of the RelOptInfo structure.
4. Estimate LIMIT, ORDER BY and ARREGISFDD costs if necessary.

To understand how the planner performs clearly, two specific examples are shown below.

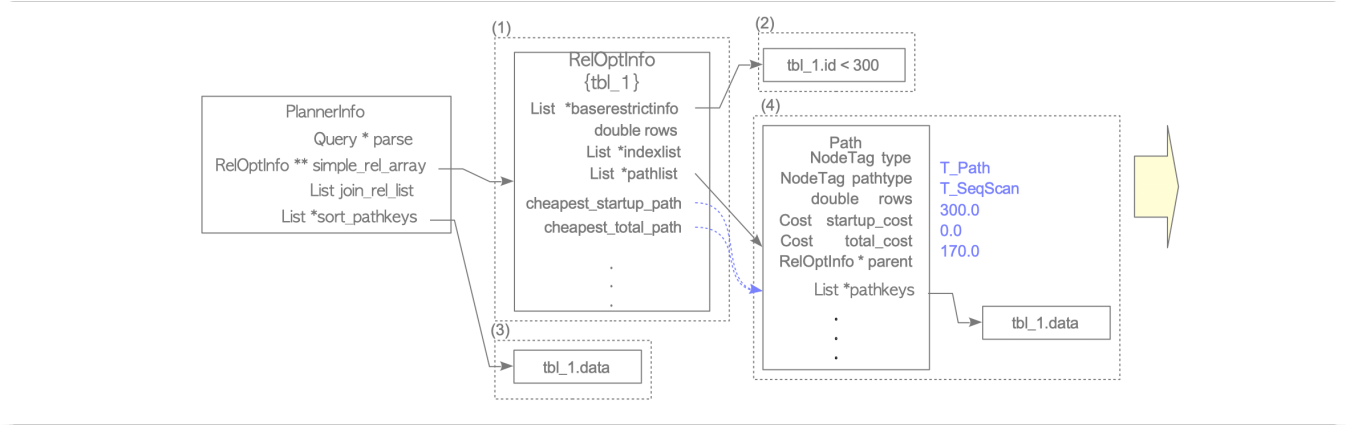
3.3.2.1. Example 1

First, we explore a simple-single table query without indexes; this query contains both WHERE and ORDER BY clauses.

```
testdb=# \d tbl_1
     Table "public.tbl_1"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 data   | integer |

testdb=# SELECT * FROM tbl_1 WHERE id < 300 ORDER BY data;
```

Figures 3.10 and 3.11 depict how the planner performs in this example.

Fig. 3.10. How to get the cheapest path of Example 1.



(1) Create a RelOptInfo structure and store it in the simple_rel_array of the PlannerInfo.

(2) Add a WHERE clause to the baserestrictinfo of the RelOptInfo.
A WHERE clause *'id < 300'* is added to the baserestrictinfo by the distribute_restrictinfo_to_rels() function defined in initsplan.c. In addition, the indexlist of the RelOptInfo is NULL because there are no related indexes of the target table.

(3) Add the pathkey for sorting to the sort_pathkeys of the PlannerInfo by the standard_qp_callback() function defined in planner.c.
Pathkey is a data structure representing the sort ordering for the path. In this example, the column "data" is added to the sort_pathkeys as a pathkey because this query contains an ORDER BY clause and its column is 'data'.

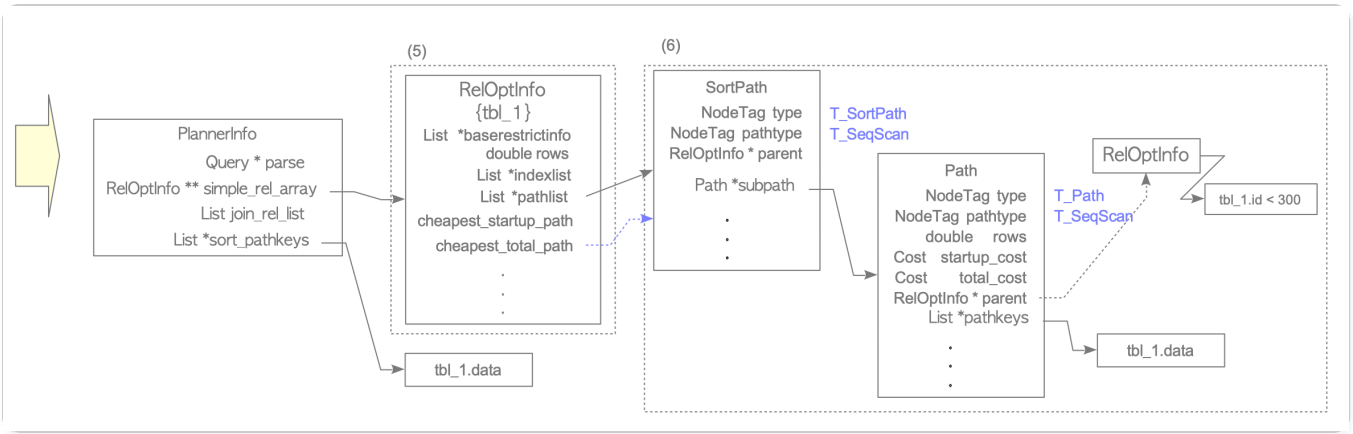
(4) Create a path structure and estimate the cost of the sequential scan using the cost_seqscan function and write the estimated costs into the path. Then, add the path to the RelOptInfo by the add_path() function defined in pathnode.c.
As mentioned before, the Path structure contains both of the start-up and the total costs which are estimated by the cost_seqscan function, and so on.

```
typedef struct PathKey
{
        pg_node_attr(no_read, no_query_jumble)
        NodeTag                 type;

        /* the value that is ordered */
        EquivalenceClass *pk_eclass pg_node_attr(copy_as_scalar, equal_as_scalar);
        Oid             pk_opfamily;    /* btree opfamily defining the ordering */
        int             pk_strategy;    /* sort direction (ASC or DESC) */
        bool            pk_nulls_first; /* do NULLs come before normal values? */
} PathKey;

typedef struct Path
{
        pg_node_attr(no_copy_equal, no_read, no_query_jumble)

        NodeTag                 type;

        /* tag identifying scan/join method */
        NodeTag                 pathtype;

        /*
         * the relation this path can build
         *
```

```
         * We do NOT print the parent, else we'd be in infinite recursion.  We can
         * print the parent's relids for identification purposes, though.
         */
        RelOptInfo *parent pg_node_attr(write_only_relids);

        /*
         * list of Vars/Exprs, cost, width
         *
         * We print the pathtarget only if it's not the default one for the rel.
         */
        PathTarget *pathtarget pg_node_attr(write_only_nondefault_pathtarget);

        /*
         * parameterization info, or NULL if none
         *
         * We do not print the whole of param_info, since it's printed via
         * RelOptInfo; it's sufficient and less cluttering to print just the
         * required outer relids.
         */
        ParamPathInfo *param_info pg_node_attr(write_only_req_outer);

        /* engage parallel-aware logic? */
        bool            parallel_aware;
        /* OK to use as part of parallel plan? */
        bool            parallel_safe;
        /* desired # of workers; 0 = not parallel */
        int                     parallel_workers;

        /* estimated size/costs for path (see costsize.c for more info) */
        Cardinality rows;                       /* estimated number of result tuples
*/
        Cost            startup_cost;   /* cost expended before fetching any tuples
*/
        Cost            total_cost;             /* total cost (assuming all tuples fe
tched) */

        /* sort ordering of path's output; a List of PathKey nodes; see above */
        List       *pathkeys;
} Path;
```

In this example, the planner only estimates the sequential scan cost because there are no indexes of the target table. Therefore, the cheapest access path is automatically determined.

Fig. 3.11. How to get the cheapest path of Example 1 (continued from Fig. 3.10).



(5) Create a new RelOptInfo structure to process the ORDER BY procedure.

Note that the new RelOptInfo does not have the baserestrictinfo, that is, the information of the WHERE clause.

(6) Create a sort path and add it to the new RelOptInfo; then, link the sequential scan path to the subpath of the sort path.
The SortPath structure is composed of two path structures: path and subpath; the path stores information about the sort operation itself, and the subpath stores the cheapest path.
Note that the item 'parent' of the sequential scan path holds the link to the old RelOptInfo which stores the WHERE clause in its baserestrictinfo. Therefore, in the next stage, that is, creating a plan tree, the planner can create a sequential scan node that contains the WHERE clause as the 'Filter', even though the new RelOptInfo does not have the baserestrictinfo.

```
typedef struct SortPath
{
        Path    path;
        Path    *subpath;               /* path representing input source */
} SortPath;
```

Based on the cheapest access path obtained here, a plan tree is generated. Details are described in Section 3.3.3.

3.3.2.2. Example 2

Next, we explore another single-table query with two indexes; this query contains a WHERE clause.

```
testdb=# \d tbl_2
     Table "public.tbl_2"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
 data   | integer |
Indexes:
    "tbl_2_pkey" PRIMARY KEY, btree (id)
    "tbl_2_data_idx" btree (data)

testdb=# SELECT * FROM tbl_2 WHERE id < 240;
```

Figures 3.12 to 3.14 depict how the planner performs in this example.

(1) Create a RelOptInfo structure.

(2) Add the WHERE clause to the baserestrictinfo, and add the indexes of the target table to the indexlist.

In this example, a WHERE clause 'id < 240' is added to the baserestrictinfo, and two indexes, *tbl_2_pkey* and *tbl_2_data_idx*, are added to the indexlist of the RelOptInfo.

(3) Create a path, estimate the cost of the sequential scan, and add the path to the pathlist of the RelOptInfo.

Fig. 3.12. How to get the cheapest path of Example 2.

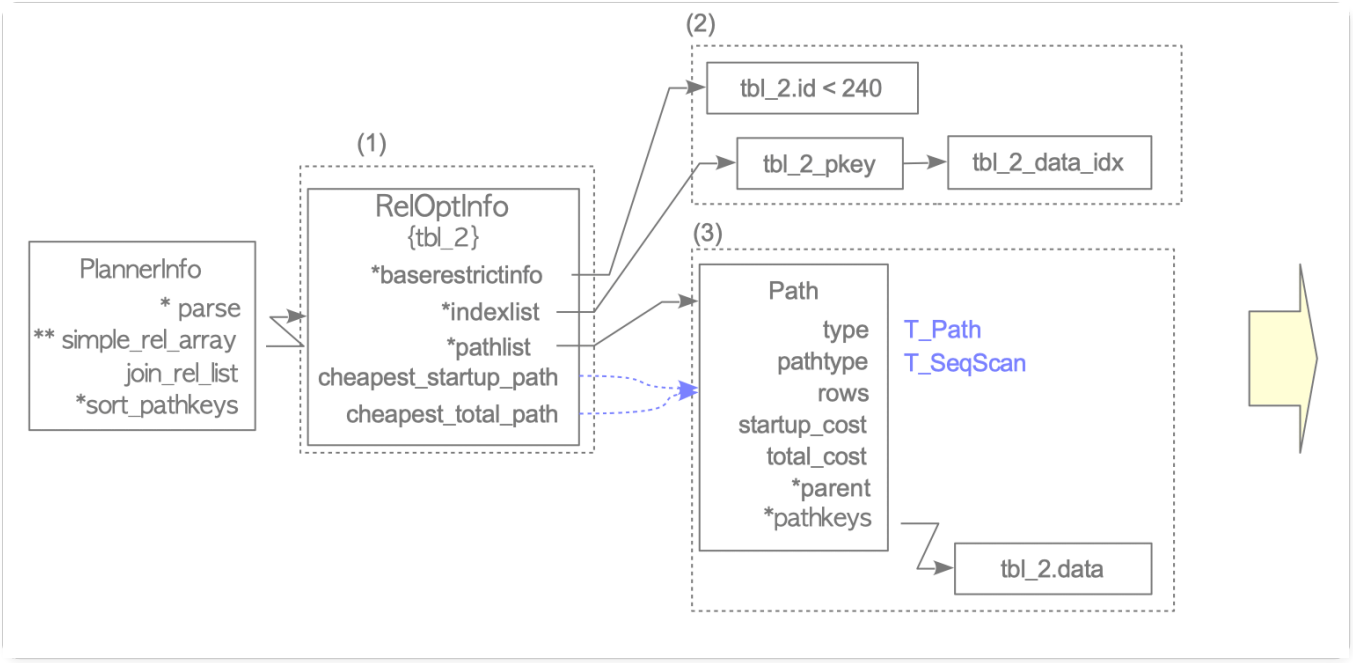
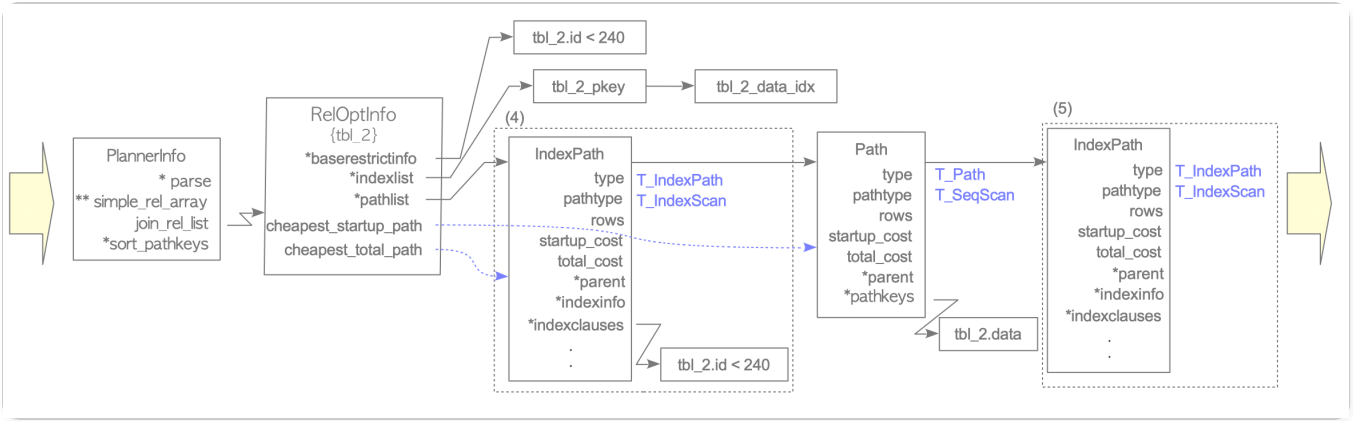


Fig. 3.13. How to get the cheapest path of Example 2 (continued from Fig. 3.12).



(4) Create an IndexPath, estimate the cost of the index scan, and add the IndexPath to the pathlist of the RelOptInfo using the add_path() function.

In this example, as there are two indexes, *tbl_2_pkey* and *tbl_2_data_idx*, these indexes are processed in order. *tbl_2_pkey* is processed first.

An IndexPath is created for *tbl_2_pkey*, and both the start-up and the total costs are estimated. In this example, *tbl_2_pkey* is the index related to the column 'id', and the WHERE clause contains the column 'id'; therefore, the WHERE clause is stored in the *indexclauses* of the IndexPath.

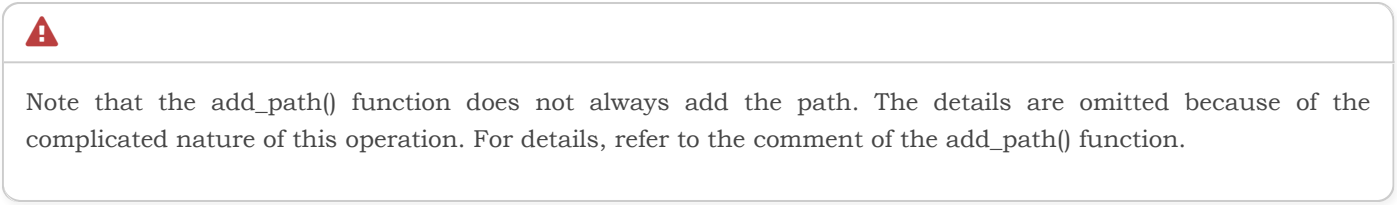
Note that when adding access paths to the pathlist, the add_path() function adds paths in the sort order of the total cost. In this example, the total cost of this index scan is smaller than the sequential total cost; thus, this index path is inserted before the sequential scan path.

```
typedef struct IndexPath
{
        Path          path;
        IndexOptInfo *indexinfo;
        List      *indexclauses;
        List      *indexorderbys;
        List      *indexorderbycols;
        ScanDirection indexscandir;
        Cost          indextotalcost;
        Selectivity indexselectivity;
} IndexPath;

/*
 * IndexOptInfo
 *              Per-index information for planning/optimization
 *
 *              indexkeys[], indexcollations[] each have ncolumns entries.
 *              opfamily[], and opcintype[]     each have nkeycolumns entries. Th
ey do
 *              not contain any information about included attributes.
 *
 *              sortopfamily[], reverse_sort[], and nulls_first[] have
 *              nkeycolumns entries, if the index is ordered; but if it is unorde
red,
 *              those pointers are NULL.
 *
 *              Zeroes in the indexkeys[] array indicate index columns that are
 *              expressions; there is one element in indexprs for each such colum
n.
 *
 *              For an ordered index, reverse_sort[] and nulls_first[] describe t
he
 *              sort ordering of a forward indexscan; we can also consider a back
ward
 *              indexscan, which will generate the reverse ordering.
 *
 *              The indexprs and indpred expressions have been run through
 *              prepqual.c and eval_const_expressions() for ease of matching to
 *              WHERE clauses.  indpred is in implicit-AND form.
 *
 *              indextlist is a TargetEntry list representing the index columns.
 *              It provides an equivalent base-relation Var for each simple colum
n,
 *              and links to the matching indexprs element for each expression co
lumn.
 *
 *              While most of these fields are filled when the IndexOptInfo is cr
eated
 *              (by plancat.c), indrestrictinfo and predOK are set later, in
 *              check_index_predicates().
 */
#ifndef HAVE_INDEXOPTINFO_TYPEDEF
typedef struct IndexOptInfo IndexOptInfo;
#define HAVE_INDEXOPTINFO_TYPEDEF 1
#endif

struct IndexOptInfo
{
        pg_node_attr(no_copy_equal, no_read, no_query_jumble)

        NodeTag               type;
```

```
        /* OID of the index relation */
        Oid                     indexoid;
        /* tablespace of index (not table) */
        Oid                     reltablespace;
        /* back-link to index's table; don't print, else infinite recursion */
        RelOptInfo *rel pg_node_attr(read_write_ignore);

        /*
         * index-size statistics (from pg_class and elsewhere)
         */
        /* number of disk pages in index */
        BlockNumber pages;
        /* number of index tuples in index */
        Cardinality tuples;
        /* index tree height, or -1 if unknown */
        int                     tree_height;

        /*
         * index descriptor information
         */
        /* number of columns in index */
        int                     ncolumns;
        /* number of key columns in index */
        int                     nkeycolumns;

        /*
         * table column numbers of index's columns (both key and included
         * columns), or 0 for expression columns
         */
        int                *indexkeys pg_node_attr(array_size(ncolumns));
        /* OIDs of collations of index columns */
        Oid                *indexcollations pg_node_attr(array_size(nkeycolumn
s));
        /* OIDs of operator families for columns */
        Oid                *opfamily pg_node_attr(array_size(nkeycolumns));
        /* OIDs of opclass declared input data types */
        Oid                *opcintype pg_node_attr(array_size(nkeycolumns));
        /* OIDs of btree opfamilies, if orderable.  NULL if partitioned index */
        Oid                *sortopfamily pg_node_attr(array_size(nkeycolumns));
        /* is sort order descending? or NULL if partitioned index */
        bool       *reverse_sort pg_node_attr(array_size(nkeycolumns));
        /* do NULLs come first in the sort order? or NULL if partitioned index */
        bool       *nulls_first pg_node_attr(array_size(nkeycolumns));
        /* opclass-specific options for columns */
        bytea     **opclassoptions pg_node_attr(read_write_ignore);
        /* which index cols can be returned in an index-only scan? */
        bool       *canreturn pg_node_attr(array_size(ncolumns));
        /* OID of the access method (in pg_am) */
        Oid                     relam;

        /*
         * expressions for non-simple index columns; redundant to print since we
         * print indextlist
         */
        List       *indexprs pg_node_attr(read_write_ignore);
        /* predicate if a partial index, else NIL */
        List       *indpred;

        /* targetlist representing index columns */
        List       *indextlist;
```

```
        /*
         * parent relation's baserestrictinfo list, less any conditions implied b
y
         * the index's predicate (unless it's a target rel, see comments in
         * check_index_predicates())
         */
        List       *indrestrictinfo;

        /* true if index predicate matches query */
        bool            predOK;
        /* true if a unique index */
        bool            unique;
        /* is uniqueness enforced immediately? */
        bool            immediate;
        /* true if index doesn't really exist */
        bool            hypothetical;

        /*
         * Remaining fields are copied from the index AM's API struct
         * (IndexAmRoutine).  These fields are not set for partitioned indexes.
         */
        bool            amcanorderbyop;
        bool            amoptionalkey;
        bool            amsearcharray;
        bool            amsearchnulls;
        /* does AM have amgettuple interface? */
        bool            amhasgettuple;
        /* does AM have amgetbitmap interface? */
        bool            amhasgetbitmap;
        bool            amcanparallel;
        /* does AM have ammarkpos interface? */
        bool            amcanmarkpos;
        /* AM's cost estimator */
        /* Rather than include amapi.h here, we declare amcostestimate like this
*/
        void            (*amcostestimate) () pg_node_attr(read_write_ignore);
};
```

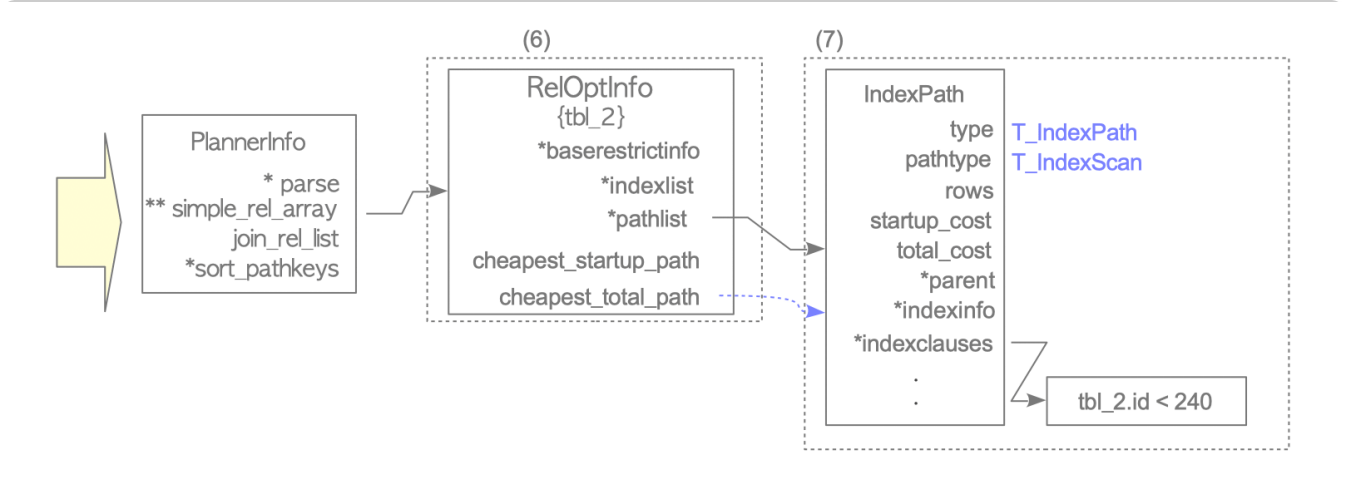
(5) Create another IndexPath, estimate the cost of other index scans, and add the index path to the pathlist of the RelOptInfo.
Next, an IndexPath is created for *tbl_2_data_idx*, the costs are estimated, and this IndexPath is added to the pathlist. In this example, there is no WHERE clause related to the *tbl_2_data_idx* index; therefore, the index clauses are NULL.



Note that the add_path() function does not always add the path. The details are omitted because of the complicated nature of this operation. For details, refer to the comment of the add_path() function.

(6) Create a new RelOptInfo structure.
(7) Add the cheapest path to the pathlist of the new RelOptInfo.
In this example, the cheapest path is the index path using the index *tbl_2_pkey*; thus, its path is added to the pathlist of the new RelOptInfo.

Fig. 3.14. How to get the cheapest path of Example 2 (continued from Fig. 3.13).



3.3.3. Creating a Plan Tree

At the last stage, the planner generates a plan tree from the cheapest path.

The root of the plan tree is a PlannedStmt structure defined in plannodes.h. It contains nineteen fields, but here are four representative fields:

- **commandType** stores a type of operation, such as SELECT, UPDATE or INSERT.
- **rtable** stores rangeTable entries.
- **relationOids** stores oids of the related tables for this query.
- **plantree** stores a plan tree that is composed of plan nodes, where each node corresponds to a specific operation, such as sequential scan, sort and index scan.

```
typedef struct PlannedStmt
{
        pg_node_attr(no_equal, no_query_jumble)

        NodeTag                 type;

        CmdType                 commandType;    /* select|insert|update|delete|merge|
utility */

        uint64          queryId;                /* query identifier (copied from Quer
y) */

        bool            hasReturning;   /* is it insert|update|delete RETURNING? */

        bool            hasModifyingCTE;        /* has insert|update|delete in WITH?
*/

        bool            canSetTag;              /* do I set the command result tag?
*/

        bool            transientPlan;  /* redo plan when TransactionXmin changes? */

        bool            dependsOnRole;  /* is plan specific to current role? */

        bool            parallelModeNeeded; /* parallel mode required to execute? */

        int                     jitFlags;               /* which forms of JIT should
be performed */
```

```
        struct Plan *planTree;          /* tree of Plan nodes */

        List       *rtable;                     /* list of RangeTblEntry nodes */

        List       *permInfos;          /* list of RTEPermissionInfo nodes for rtable
                                                                * entries needing on
e */

        /* rtable indexes of target relations for INSERT/UPDATE/DELETE/MERGE */
        List       *resultRelations;    /* integer list of RT indexes, or NIL */

        List       *appendRelations;    /* list of AppendRelInfo nodes */

        List       *subplans;           /* Plan trees for SubPlan expressions; note
                                                                * that some could be
NULL */

        Bitmapset  *rewindPlanIDs;      /* indices of subplans that require REWIND */

        List       *rowMarks;           /* a list of PlanRowMark's */

        List       *relationOids;       /* OIDs of relations the plan depends on */

        List       *invalItems;         /* other dependencies, as PlanInvalItems */

        List       *paramExecTypes; /* type OIDs for PARAM_EXEC Params */

        Node       *utilityStmt;        /* non-null if this is utility stmt */

        /* statement location in source string (copied from Query) */
        int                     stmt_location;  /* start location, or -1 if unknown
*/
        int                     stmt_len;               /* length in bytes; 0 means
"rest of string" */
} PlannedStmt;
```

As mentioned above, a plan tree is composed of various plan nodes. The PlanNode structure is the base node, and other nodes always contain it. For example, SeqScanNode, which is for sequential scanning, is composed of a PlanNode and an integer variable 'scanrelid'. A PlanNode contains fourteen fields. The following are seven representative fields.

- **start-up cost** and **total_cost** are the estimated costs of the operation corresponding to this node.
- **rows** is the number of rows to be scanned, which is estimated by the planner.
- **targetlist** stores the target list items contained in the query tree.
- **qual** is a list that stores qual conditions.
- **lefttree** and **righttree** are the nodes for adding the children nodes.

```
/* ----------------
 *              Plan node
 *
 * All plan nodes "derive" from the Plan structure by having the
 * Plan structure as the first field.  This ensures that everything works
 * when nodes are cast to Plan's.  (node pointers are frequently cast to Plan*
 * when passed around generically in the executor)
 *
 * We never actually instantiate any Plan nodes; this is just the common
 * abstract superclass for all Plan-type nodes.
 * ----------------
```

```
 */
typedef struct Plan
{
        pg_node_attr(abstract, no_equal, no_query_jumble)

        NodeTag         type;

        /*
         * estimated execution costs for plan (see costsize.c for more info)
         */
        Cost            startup_cost;   /* cost expended before fetching any tuples
*/
        Cost            total_cost;             /* total cost (assuming all tuples fe
tched) */

        /*
         * planner's estimate of result size of this plan step
         */
        Cardinality plan_rows;                  /* number of rows plan is expected to
emit */
        int                     plan_width;             /* average row width in bytes
*/

        /*
         * information needed for parallel query
         */
        bool            parallel_aware; /* engage parallel-aware logic? */
        bool            parallel_safe;  /* OK to use as part of parallel plan? */

        /*
         * information needed for asynchronous execution
         */
        bool            async_capable;  /* engage asynchronous-capable logic? */

        /*
         * Common structural data for all Plan types.
         */
        int                     plan_node_id;   /* unique across entire final plan tr
ee */
        List       *targetlist;         /* target list to be computed at this node */
        List       *qual;                       /* implicitly-ANDed qual conditions
*/
        struct Plan *lefttree;          /* input plan tree(s) */
        struct Plan *righttree;
        List       *initPlan;           /* Init Plan nodes (un-correlated expr
                                                        * subselects) */

        /*
         * Information for management of parameter-change-driven rescanning
         *
         * extParam includes the paramIDs of all external PARAM_EXEC params
         * affecting this plan node or its children.  setParam params from the
         * node's initPlans are not included, but their extParams are.
         *
         * allParam includes all the extParam paramIDs, plus the IDs of local
         * params that affect the node (i.e., the setParams of its initplans).
         * These are _all_ the PARAM_EXEC params that affect this node.
         */
        Bitmapset  *extParam;
        Bitmapset  *allParam;
} Plan;
```

```
/*
 * ==========
 * Scan nodes
 *
 * Scan is an abstract type that all relation scan plan types inherit from.
 * ==========
 */
typedef struct Scan
{
        pg_node_attr(abstract)

        Plan           plan;
        Index          scanrelid;              /* relid is index into the range tabl
e */
} Scan;

/* ----------------
 *              sequential scan node
 * ----------------
 */
typedef struct SeqScan
{
        Scan           scan;
} SeqScan;
```

In the following, two plan trees, which will be generated from the cheapest paths shown in the examples in the previous subsection, are described.

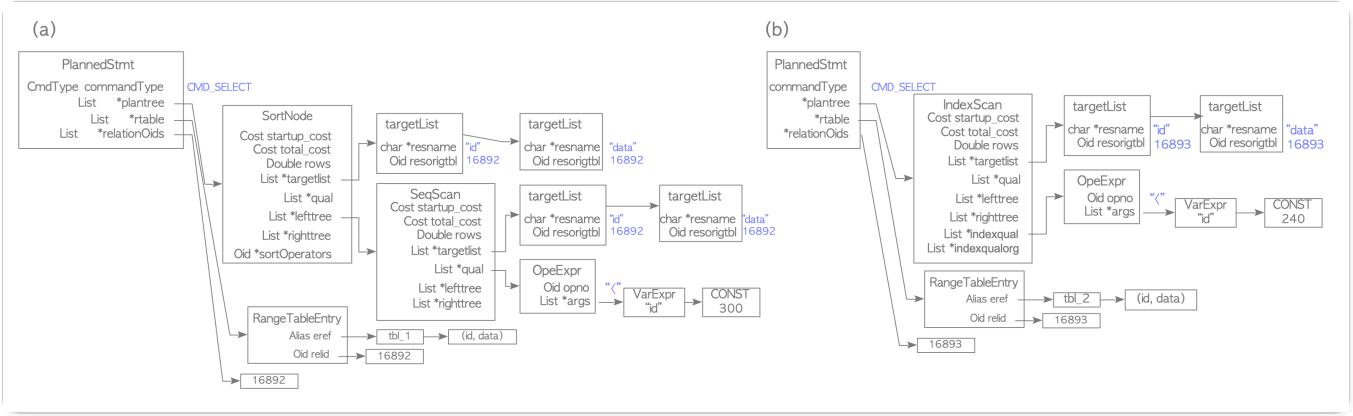
3.3.3.1. Example 1

The first example is the plan tree of the example in Section 3.3.2.1. The cheapest path shown in Figure 3.11 is a tree composed of a sort path and a sequential scan path. The root path is the sort path, and the child path is the sequential scan path. Although detailed explanations are omitted, it will be easy to understand that the plan tree can be almost trivially generated from the cheapest path. In this example, a SortNode is added to the plantree of the PlannedStmt structure, and a SeqScanNode is added to the lefttree of the SortNode. See Fig. 3.15(a).

```
/* ----------------
 *              sort node
 * ----------------
 */
typedef struct Sort
{
        Plan           plan;

        /* number of sort-key columns */
        int                     numCols;

        /* their indexes in the target list */
        AttrNumber *sortColIdx pg_node_attr(array_size(numCols));

        /* OIDs of operators to sort them by */
        Oid                *sortOperators pg_node_attr(array_size(numCols));

        /* OIDs of collations */
        Oid                *collations pg_node_attr(array_size(numCols));

        /* NULLS FIRST/LAST directions */
```

```
        bool       *nullsFirst pg_node_attr(array_size(numCols));
} Sort;
```

Fig. 3.15. Examples of plan trees.



In the SortNode, the lefttree points to the SeqScanNode.

In the SeqScanNode, the qual holds the WHERE clause 'id < 300'.

3.3.3.2. Example 2

The second example is the plan tree of the example in Section 3.3.2.2. The cheapest path shown in Fig. 3.14 is the index scan path, so the plan tree is composed of an IndexScanNode structure alone. See Fig. 3.15(b).

```
/* ----------------
 *              index scan node
 *
 * indexqualorig is an implicitly-ANDed list of index qual expressions, each
 * in the same form it appeared in the query WHERE condition.  Each should
 * be of the form (indexkey OP comparisonval) or (comparisonval OP indexkey).
 * The indexkey is a Var or expression referencing column(s) of the index's
 * base table.  The comparisonval might be any expression, but it won't use
 * any columns of the base table.  The expressions are ordered by index
 * column position (but items referencing the same index column can appear
 * in any order).  indexqualorig is used at runtime only if we have to recheck
 * a lossy indexqual.
 *
 * indexqual has the same form, but the expressions have been commuted if
 * necessary to put the indexkeys on the left, and the indexkeys are replaced
 * by Var nodes identifying the index columns (their varno is INDEX_VAR and
 * their varattno is the index column number).
 *
 * indexorderbyorig is similarly the original form of any ORDER BY expressions
 * that are being implemented by the index, while indexorderby is modified to
 * have index column Vars on the left-hand side.  Here, multiple expressions
 * must appear in exactly the ORDER BY order, and this is not necessarily the
 * index column order.  Only the expressions are provided, not the auxiliary
 * sort-order information from the ORDER BY SortGroupClauses; it's assumed
 * that the sort ordering is fully determinable from the top-level operators.
 * indexorderbyorig is used at runtime to recheck the ordering, if the index
 * cannot calculate an accurate ordering.  It is also needed for EXPLAIN.
 *
 * indexorderbyops is a list of the OIDs of the operators used to sort the
 * ORDER BY expressions.  This is used together with indexorderbyorig to
 * recheck ordering at run time.  (Note that indexorderby, indexorderbyorig,
 * and indexorderbyops are used for amcanorderbyop cases, not amcanorder.)
```

```
 *
 * indexorderdir specifies the scan ordering, for indexscans on amcanorder
 * indexes (for other indexes it should be "don't care").
 * ----------------
 */
typedef struct Scan
{
        pg_node_attr(abstract)

        Plan           plan;
        Index          scanrelid;              /* relid is index into the range tabl
e */
} Scan;

typedef struct IndexScan
{
        Scan           scan;
        Oid                     indexid;                /* OID of index to scan */
        List       *indexqual;          /* list of index quals (usually OpExprs) */
        List       *indexqualorig;      /* the same in original form */
        List       *indexorderby;       /* list of index ORDER BY exprs */
        List       *indexorderbyorig;   /* the same in original form */
        List       *indexorderbyops;    /* OIDs of sort ops for ORDER BY exprs */
        ScanDirection indexorderdir;  /* forward or backward or don't care */
} IndexScan;
```

In this example, the WHERE clause 'id < 240' is an access predicate, so it is stored in the indexqual of the IndexScanNode.

3.4. How the Executor Performs

In single-table queries, the executor takes the plan nodes in an order from the end of the plan tree to the root and then invokes the functions that perform the processing of the corresponding nodes.

Each plan node has functions that are meant for executing the respective operation. These functions are located in the src/backend/executor/ directory. For example, the functions for executing the sequential scan (ScanScan) are defined in nodeSeqscan.c; the functions for executing the index scan (IndexScanNode) are defined in nodeIndexscan.c; the functions for sorting SortNode are defined in nodeSort.c, and so on.

Of course, the best way to understand how the executor performs is to read the output of the EXPLAIN command. PostgreSQL's EXPLAIN shows the plan tree almost as it is. It will be explained using Example 1 in Section 3.3.3.

```
1. testdb=# EXPLAIN SELECT * FROM tbl_1 WHERE id < 300 ORDER BY data;
2.                            QUERY PLAN
3. ------------------------------------------------------------------
4.  Sort  (cost=182.34..183.09 rows=300 width=8)
5.    Sort Key: data
6.    ->  Seq Scan on tbl_1  (cost=0.00..170.00 rows=300 width=8)
7.          Filter: (id < 300)
8. (4 rows)
```

Let's explore how the executor performs. Read the result of the EXPLAIN command from the bottom line to the top line.

Line 6: At first, the executor carries out a sequential scan operation using the functions defined in nodeSeqscan.c.

Line 4: Next, the executor sorts the result of the sequential scan using the functions defined in nodeSort.c.

Temporary Files

Although the executor uses the work_men and temp_buffers, which are allocated in the memory, for query processing, it uses temporary files if the processing cannot be performed within the memory alone.

Using the ANALYZE option, the EXPLAIN command actually executes the query and displays the true row counts, true run time, and the actual memory usage. A specific example is shown below:

```
1.  testdb=# EXPLAIN ANALYZE SELECT id, data FROM tbl_25m ORDER BY id;
2.                                                          QUERY PLAN
3.  ------------------------------------------------------------------------------------------------------------
    ----------------------
4.   Sort  (cost=3944070.01..3945895.01 rows=730000 width=4104) (actual time=885.648..1033.746 rows=73
    0000 loops=1)
5.     Sort Key: id
6.     Sort Method: external sort  Disk: 10000kB
7.     ->  Seq Scan on tbl_25m  (cost=0.00..10531.00 rows=730000 width=4104) (actual time=0.024..102.5
    48 rows=730000 loops=1)
8.   Planning time: 1.548 ms
9.   Execution time: 1109.571 ms
10.  (6 rows)
```

In Line 6, the EXPLAIN command shows that the executor has used a temporary file whose size is 10000kB.

Temporary files are created in the base/pg_tmp subdirectory temporarily, and the naming method is shown follows:

```
{"pgsql_tmp"} + {PID of the postgres process which creates the file} . {sequencial number from 0}
```

For example, the temporary file 'pgsql_tmp8903.5' is the 6th temporary file created by the postgres process with the pid of 8903.

```
$ ls -la /usr/local/pgsql/data/base/pgsql_tmp*
-rw-------  1 postgres  postgres  10240000 12  4 14:18 pgsql_tmp8903.5
```

3.5. Join Operations

Go to Section 3.5.

← Back to Part 1 Go to Part 3 →