Chapter 3

Query Processing (Part 3)

🡐 Back to Part 1, Part 2

3.5. Join Operations

PostgreSQL supports three join operations: nested loop join, merge join and hash join. The nested loop join and the merge join in PostgreSQL have several variations.

In the following, we assume that the reader is familiar with the basic behavior of these three joins. If you are unfamiliar with these terms, see [1, 2]. However, as there is not much explanation on the hybrid hash join with skew supported by PostgreSQL, it will be explained in more detail here.

Note that the three join methods supported by PostgreSQL can perform all join operations, not only INNER JOIN, but also LEFT/RIGHT OUTER JOIN, FULL OUTER JOIN, and so on. However, for simplicity, we focus on the NATURAL INNER JOIN in this chapter.

3.5.1. Nested Loop Join

The nested loop join is the most fundamental join operation, and it can be used in all join conditions. PostgreSQL supports the nested loop join and five variations of it.

3.5.1.1. Nested Loop Join

The nested loop join does not need any start-up operation, so the start-up cost is 0:

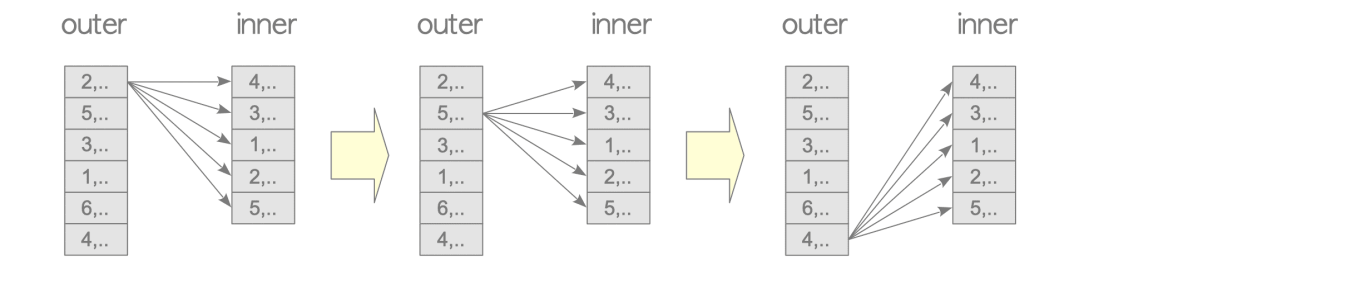
$$\texttt{'start-up cost'} = 0.$$

The run cost of the nested loop join is proportional to the product of the sizes of the outer and inner tables. In other words, the `'run cost'` is $O(N_{outer} \times N_{inner})$, where N_{outer} and N_{inner} are the numbers of tuples of the outer table and the inner table, respectively. More precisely, the run cost is defined by the following equation:

$$\texttt{'run cost'} = (\texttt{cpu\_operator\_cost} + \texttt{cpu\_tuple\_cost}) \times N_{outer} \times N_{inner} + C_{inner} \times N_{outer} + C_{outer}$$

where C_{outer} and C_{inner} are the scanning costs of the outer table and the inner table, respectively.

Fig. 3.16. Nested loop join.



The cost of the nested loop join is always estimated, but this join operation is rarely used because more efficient variations, which are described in the following, are usually used.

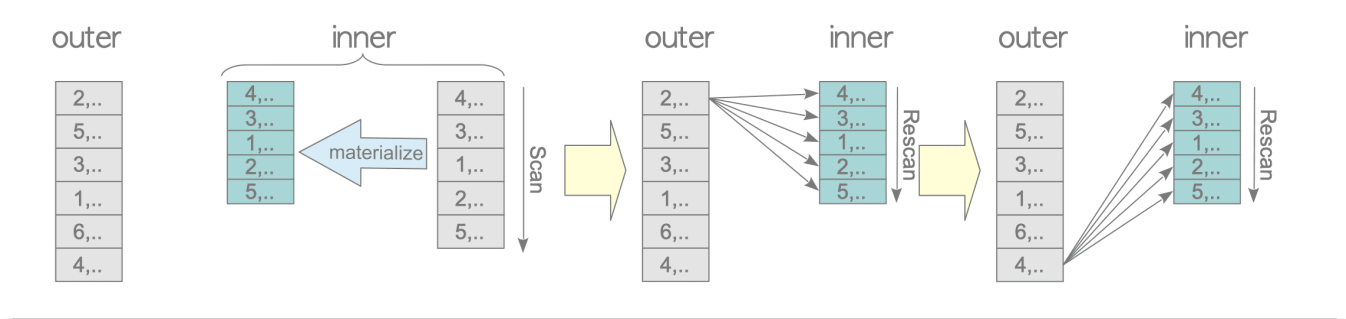
3.5.1.2. Materialized Nested Loop Join

The nested loop join described above has to scan all the tuples of the inner table whenever each tuple of the outer table is read. Since scanning the entire inner table for each outer table tuple is a costly process, PostgreSQL supports the *materialized nested loop join* to reduce the total scanning cost of the inner table.

Before running a nested loop join, the executor writes the inner table tuples to the work_mem or a temporary file by scanning the inner table once using the *temporary tuple storage* module described in ℹ below. This has the potential to process the inner table tuples more efficiently than using the buffer manager, especially if at least all the tuples are written to work_mem.

Figure 3.17 illustrates how the materialized nested loop join performs. Scanning materialized tuples is internally called **rescan**.

Fig. 3.17. Materialized nested loop join.



> ℹ Temporary Tuple Storage
>
> PostgreSQL internally provides a temporary tuple storage module for materializing tables, creating batches in hybrid hash join and so on. This module is composed of the functions defined in tuplestore.c, and they store and read a sequence of tuples to/from work_mem or temporary files. The decision of whether to use the work_mem or temporary files depends on the total size of the tuples to be stored.

We will explore how the executor processes the plan tree of the materialized nested loop join and how the cost is estimated using the specific example shown below.

```
1.  testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id;
2.                                QUERY PLAN
3.  ------------------------------------------------------------------------
```

```
4.  Nested Loop  (cost=0.00..750230.50 rows=5000 width=16)
5.    Join Filter: (a.id = b.id)
6.    ->  Seq Scan on tbl_a a  (cost=0.00..145.00 rows=10000 width=8)
7.    ->  Materialize  (cost=0.00..98.00 rows=5000 width=8)
8.          ->  Seq Scan on tbl_b b  (cost=0.00..73.00 rows=5000 width=8)
9.  (5 rows)
```

First, the operation of the executor is shown. The executor processes the displayed plan nodes as follows:

Line 7: The executor materializes the inner table tbl_b by sequential scanning (Line 8).
Line 4: The executor carries out the nested loop join operation; the outer table is tbl_a and the inner one is the materialized tbl_b.

In what follows, the costs of the 'Materialize' (Line 7) and 'Nested Loop' (Line 4) operations are estimated. Assume that the materialized inner tuples are stored in the work_mem.

Materialize:

There is no cost to start up, so the start-up cost is 0.

$$\texttt{'start-up cost'} = 0$$

The run cost is defined by the following equation:

$$\texttt{'run cost'} = 2 \times \texttt{cpu\_operator\_cost} \times N_{inner}$$

Therefore,

$$\texttt{'run cost'} = 2 \times 0.0025 \times 5000 = 25.0.$$

In addition, the total cost is the sum of the startup cost, the total cost of the sequential scan, and the run cost:

$$\texttt{'total cost'} = (\texttt{'start-up cost'} + \texttt{'total cost of seq scan'}) + \texttt{'run cost'}$$

Therefore,

$$\texttt{'total cost'} = (0.0 + 73.0) + 25.0 = 98.0.$$

(Materialized) Nested Loop:

There is no cost to start up, so the start-up cost is 0.

$$\texttt{'start-up cost'} = 0$$

Before estimating the run cost, we consider the *rescan cost*. This cost is defined by the following equation:

$$\texttt{'rescan cost'} = \texttt{cpu\_operator\_cost} \times N_{inner}$$

In this case,

$$\texttt{'rescan cost'} = (0.0025) \times 5000 = 12.5.$$

The run cost is defined by the following equation:

$$\texttt{'run cost'} = (\texttt{cpu\_operator\_cost} + \texttt{cpu\_tuple\_cost}) \times N_{inner} \times N_{outer} + \texttt{'rescan cost'} \times (N_{outer} - 1) + C^{total}_{outer,seqscan} + C^{total}_{materialize},$$

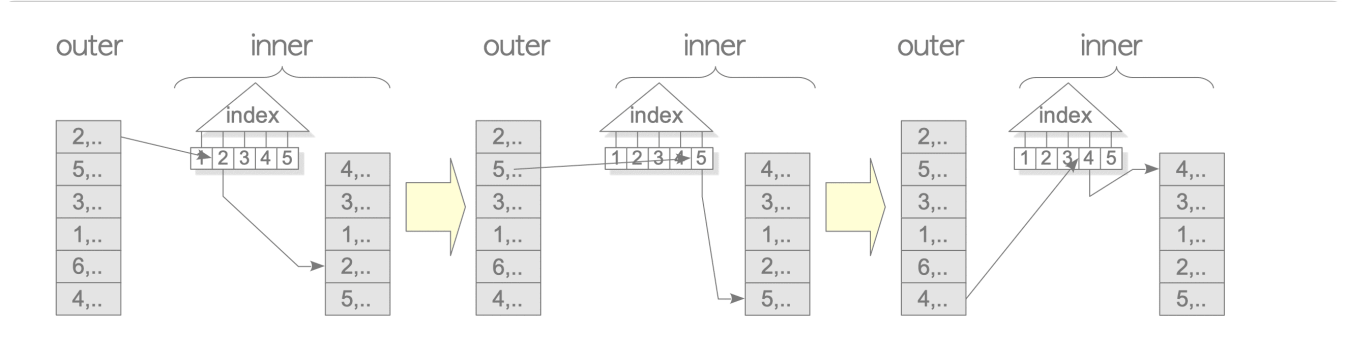
where $C^{total}_{outer,seqscan}$ is the total scan cost of the outer table and $C^{total}_{materialize}$ is the total cost of the materialized. Therefore,

$$\texttt{'run cost'} = (0.0025 + 0.01) \times 5000 \times 10000 + 12.5 \times (10000 - 1) + 145.0 + 98.0 = 750230.5.$$

3.5.1.3. Indexed Nested Loop Join

If there is an index on the inner table that can be used to look up the tuples satisfying the join condition for each tuple of the outer table, the planner will consider using this index for directly searching the inner table tuples instead of sequential scanning. This variation is called **indexed nested loop join**; see Fig. 3.18. Despite the name, this algorithm can process all the tuples of the outer table in a single loop, so it can perform the join operation efficiently.

Fig. 3.18. Indexed nested loop join.



A specific example of the indexed nested loop join is shown below.

```
1.  testdb=# EXPLAIN SELECT * FROM tbl_c AS c, tbl_b AS b WHERE c.id = b.id;
2.                                 QUERY PLAN
3.  ------------------------------------------------------------------------------
4.   Nested Loop  (cost=0.29..1935.50 rows=5000 width=16)
5.     ->  Seq Scan on tbl_b b (cost=0.00..73.00 rows=5000 width=8)
6.     ->  Index Scan using tbl_c_pkey on tbl_c c  (cost=0.29..0.36 rows=1 width=8)
7.           Index Cond: (id = b.id)
8.  (4 rows)
```

In Line 6, the cost of accessing a tuple of the inner table is displayed. This is the cost of looking up the inner table if the tuple satisfies the index condition (id = b.id), which is shown in Line 7.

In the index condition (id = b.id) in Line 7, 'b.id' is the value of the outer table's attribute used in the join condition. Whenever a tuple of the outer table is retrieved by sequential scanning, the index scan path in Line 6 looks up the inner tuples to be joined. In other words, whenever the outer table is passed as a parameter, this index scan path looks up the inner tuples that satisfy the join condition. Such an index path is called a **parameterized (index) path**. Details are described in README.

The start-up cost of this nested loop join is equal to the cost of the index scan in Line 6; thus,

$$\texttt{'start-up cost'} = 0.285.$$

The total cost of the indexed nested loop join is defined by the following equation:

$$\texttt{'total cost'} = (\texttt{cpu\_tuple\_cost} + C^{total}_{inner,parameterized}) \times N_{outer} + C^{run}_{outer,seqscan},$$

where $C^{total}_{inner,parameterized}$ is the total cost of the parameterized inner index scan.

In this case,

$$\texttt{'total cost'} = (0.01 + 0.3625) \times 5000 + 73.0 = 1935.5,$$

and the run cost is

$$\texttt{'run cost'} = 1935.5 - 0.285 = 1935.215.$$

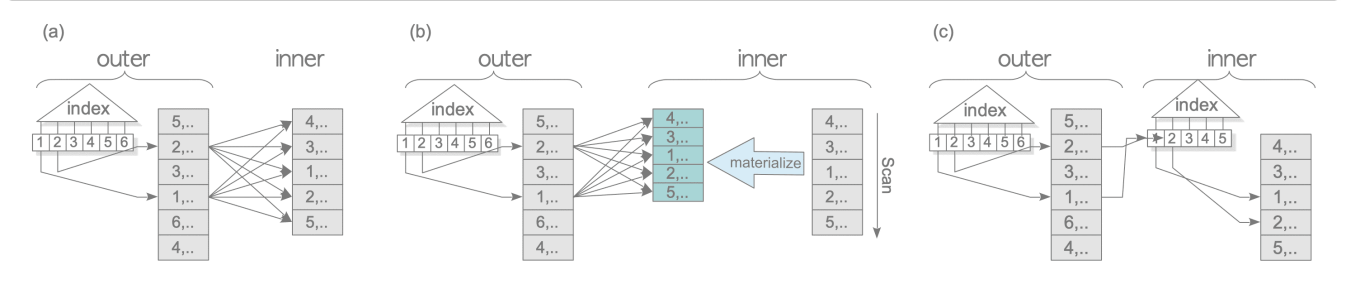
As shown above, the total cost of the indexed nested loop is $O(N_{outer})$.

3.5.1.4. Other Variations

If there is an index of the outer table and its attributes are involved in the join condition, it can be used for index scanning instead of the sequential scan of the outer table. In particular, if there is an index whose attribute can be used as an access predicate in the WHERE clause, the search range of the outer table is narrowed. This can drastically reduce the cost of the nested loop join.

PostgreSQL supports three variations of the nested loop join with an outer index scan. See Fig. 3.19.

Fig. 3.19. The three variations of the nested loop join with an outer index scan.



The results of these joins' EXPLAIN are shown here.

3.5.2. Merge Join

Unlike the nested loop join, the merge join can only be used in natural joins and equi-joins.

The cost of the merge join is estimated by the initial_cost_mergejoin() and final_cost_mergejoin() functions.

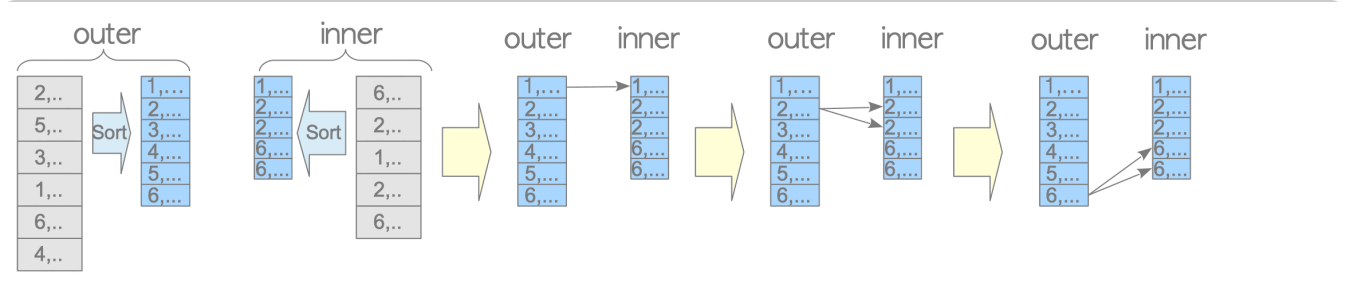
The exact cost estimation is complicated, so it is omitted here. Instead, we will only the runtime order of the merge join algorithm. The start-up cost of the merge join is the sum of sorting costs of both inner and outer tables. This means that the start-up cost is $O(N_{outer} \log_2(N_{outer}) + N_{inner} \log_2(N_{inner}))$, where N_{outer} and N_{inner} are the number of tuples of the outer and inner tables, respectively. The run cost is $O(N_{outer} + N_{inner})$.

Similar to the nested loop join, the merge join in PostgreSQL has four variations.

3.5.2.1. Merge Join

Figure 3.20 shows a conceptual illustration of a merge join.

Fig. 3.20. Merge join.



If all tuples can be stored in memory, the sorting operations will be able to be carried out in memory itself. Otherwise, temporary files will be used.

A specific example of the EXPLAIN command's result of the merge join is shown below.

```
1.  testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND b.id
    < 1000;
2.                                  QUERY PLAN
3.  ------------------------------------------------------------------------------
4.   Merge Join  (cost=944.71..984.71 rows=1000 width=16)
5.     Merge Cond: (a.id = b.id)
6.     ->  Sort  (cost=809.39..834.39 rows=10000 width=8)
7.           Sort Key: a.id
8.           ->  Seq Scan on tbl_a a  (cost=0.00..145.00 rows=10000 width=8)
9.     ->  Sort  (cost=135.33..137.83 rows=1000 width=8)
10.          Sort Key: b.id
11.          ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=1000 width=8)
12.                Filter: (id < 1000)
13. (9 rows)
```

Line 9: The executor sorts the inner table tbl_b using sequential scanning (Line 11).

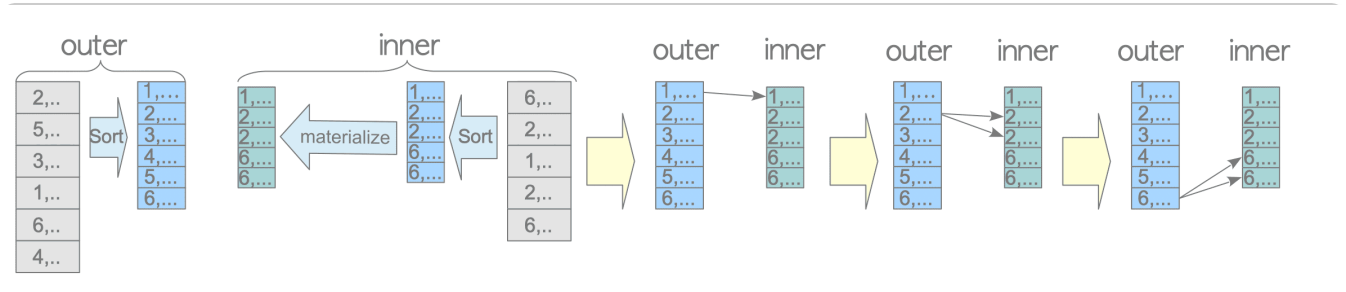
Line 6: The executor sorts the outer table tbl_a using sequential scanning (Line 8).

Line 4: The executor carries out a merge join operation; the outer table is the sorted tbl_a and the inner one is the sorted tbl_b.

3.5.2.2. Materialized Merge Join

Same as in the nested loop join, the merge join also supports the materialized merge join to materialize the inner table to make the inner table scan more efficient.

Fig. 3.21. Materialized merge join.



An example of the result of the materialized merge join is shown. It is easy to see that the difference from the merge join result above is Line 9: 'Materialize'.

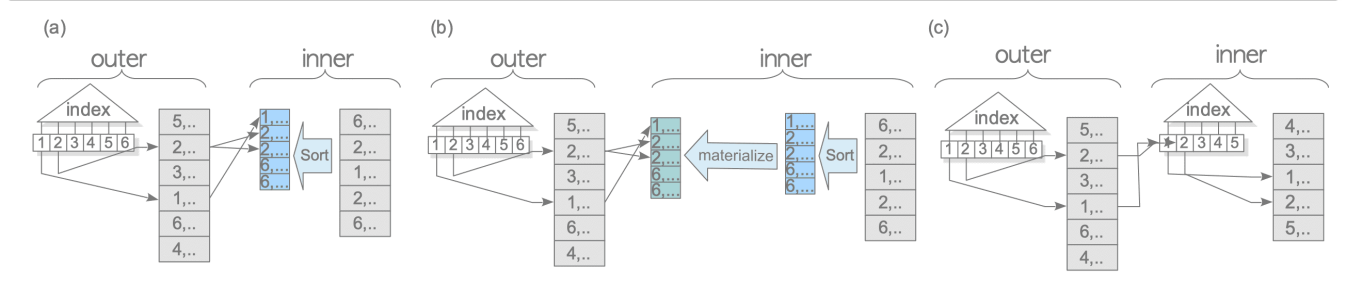
```
1.  testdb=# EXPLAIN SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id;
2.                                     QUERY PLAN
3.  ------------------------------------------------------------------------------
    --
4.   Merge Join  (cost=10466.08..10578.58 rows=5000 width=2064)
5.     Merge Cond: (a.id = b.id)
6.     ->  Sort  (cost=6708.39..6733.39 rows=10000 width=1032)
7.           Sort Key: a.id
8.           ->  Seq Scan on tbl_a a  (cost=0.00..1529.00 rows=10000 width=1032)
9.     ->  Materialize  (cost=3757.69..3782.69 rows=5000 width=1032)
10.          ->  Sort  (cost=3757.69..3770.19 rows=5000 width=1032)
11.                Sort Key: b.id
12.                ->  Seq Scan on tbl_b b  (cost=0.00..1193.00 rows=5000 width=103
    2)
13. (9 rows)
```

Line 10: The executor sorts the inner table tbl_b using sequential scanning (Line 12).
Line 9: The executor materializes the result of the sorted tbl_b.
Line 6: The executor sorts the outer table tbl_a using sequential scanning (Line 8).
Line 4: The executor carries out a merge join operation; the outer table is the sorted tbl_a and the inner one is the materialized sorted tbl_b.

3.5.2.3. Other Variations

Similar to the nested loop join, the merge join in PostgreSQL also has variations based on which the index scanning of the outer table can be carried out.

Fig. 3.22. The three variations of the merge join with an outer index scan.



The results of these joins' EXPLAIN are shown here.

3.5.3. Hash Join

Similar to the merge join, the hash join can be only used in natural joins and equi-joins.

The hash join in PostgreSQL behaves differently depending on the sizes of the tables. If the target table is small enough (more precisely, the size of the inner table is 25% or less of the work_mem), it will be a simple two-phase in-memory hash join. Otherwise, the hybrid hash join is used with the skew method.

In this subsection, the execution of both hash joins in PostgreSQL is described.

Discussion of the cost estimation has been omitted because it is complicated. Roughly speaking, the start-up and run costs are $O(N_{outer} + N_{inner})$ if assuming there is no conflict when searching and

inserting into a hash table.

3.5.3.1. In-Memory Hash Join

In this subsection, the in-memory hash join is described.

This in-memory hash join is processed in the work_mem, and the hash table area is called a **batch** in PostgreSQL. A batch has hash *slots*, internally called **buckets**, and the number of buckets is determined by the ExecChooseHashTableSize() function defined in nodeHash.c; the number of buckets is always 2^n, where n is an integer.

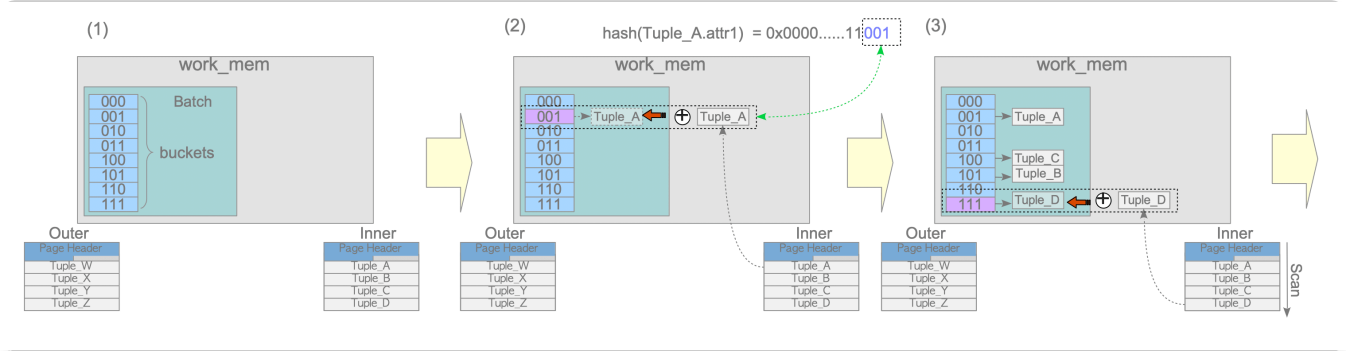
The in-memory hash join has two phases: the **build** and the **probe** phases. In the build phase, all tuples of the inner table are inserted into a batch; in the probe phase, each tuple of the outer table is compared with the inner tuples in the batch and joined if the join condition is satisfied.

A specific example is shown to clearly understand this operation. Assume that the query shown below is executed using a hash join.

```
testdb=# SELECT * FROM tbl_outer AS outer, tbl_inner AS inner WHERE inner.attr1 = outer.attr2;
```

In the following, the operation of a hash join is shown. Refer to Figs. 3.23 and 3.24.

Fig. 3.23. The build phase in the in-memory hash join.



(1) Create a batch on work_mem.
In this example, the batch has 8 buckets, which means the number of buckets is 2^3.

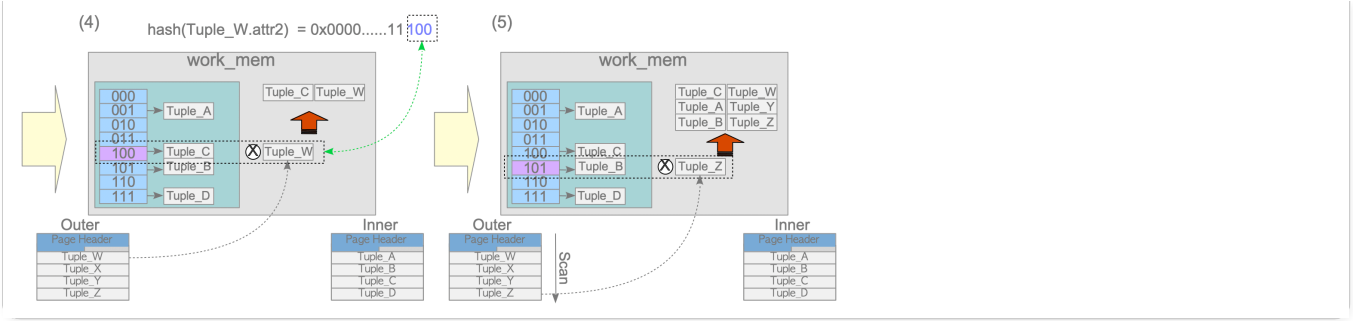
(2) Insert the first tuple of the inner table into the corresponding bucket of the batch.
The details are as follows:

1. Calculate the hash-key of the first tuple's attribute that is involved in the join condition.
   In this example, the hash-key of the attribute 'attr1' of the first tuple is calculated using the built-in hash function because the WHERE clause is 'inner.attr1 = outer.attr2'.
2. Insert the first tuple into the corresponding bucket.
   Assume that the hash-key of the first tuple is '0x000...001' by binary notatio, which means the last three bits are '001'. In this case, this tuple is inserted into the bucket whose key is '001'.

In this document, this insertion operation to build a batch is represented by this operator: ⊕

(3) Insert the remaining tuples of the inner table.

Fig. 3.24. The probe phase in the in-memory hash join.



(4) Probe the first tuple of the outer table.
The details are as follows:

1. Calculate the hash key of the first tuple's attribute that is involved in the join condition of the outer table.
   In this example, assume that the hash-key of the first tuple's attribute 'attr2' is '0x000...100'; that is, the last three bits are '100'.
2. Compare the first tuple of the outer table with the inner tuples in the batch and join tuples if the join condition is satisfied.
   Because the last three bits of the hash-key of the first tuple are '100', the executor retrieves the tuples belonging to the bucket whose key is '100' and compares both values of the respective attributes of the tables specified by the join condition (defined by the WHERE clause).
   If the join condition is satisfied, the first tuple of the outer table and the corresponding tuple of the inner table will be joined. Otherwise, the executor does nothing.
   In this example, the bucket whose key is '100' has Tuple_C. If the attr1 of Tuple_C is equal to the attr2 of the first tuple (Tuple_W), then Tuple_C and Tuple_W will be joined and saved to memory or a temporary file.

In this document, such operation to probe a batch is represented by the operator: ⊗

(5) Probe the remaining tuples of the outer table.

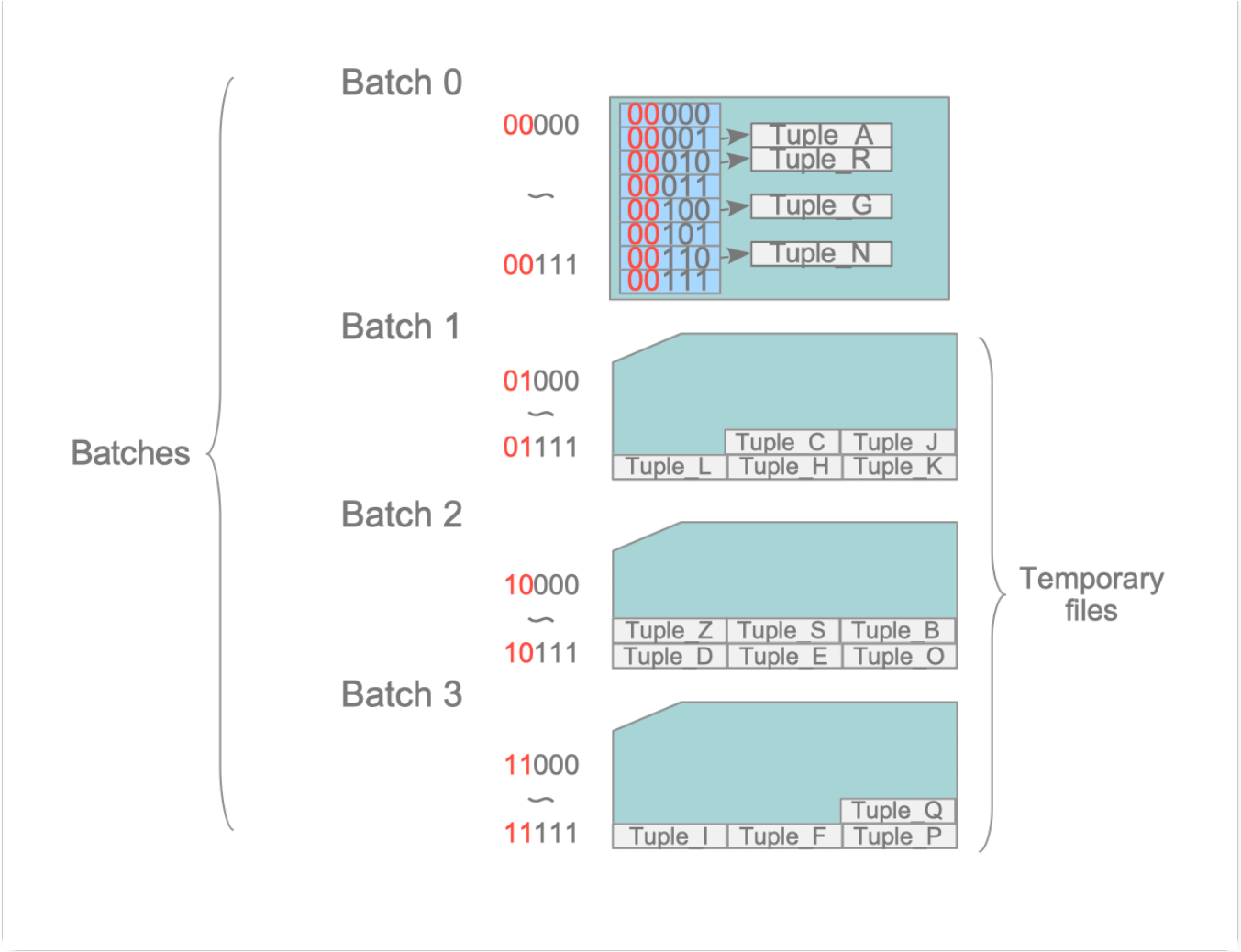
3.5.3.2. Hybrid Hash Join with Skew

When the tuples of the inner table cannot be stored in one batch in work_mem, PostgreSQL uses the hybrid hash join with the skew algorithm, which is a variation based on the hybrid hash join.

First, the basic concept of the hybrid hash join is described. In the first build and probe phases, PostgreSQL prepares multiple batches. The number of batches is the same as the number of buckets, determined by the ExecChooseHashTableSize() function. It is always 2^m, where m is an integer. At this stage, only one batch is allocated in work_mem, and the other batches are created as temporary files. The tuples belonging to these batches are written to the corresponding files and saved using the temporary tuple storage feature.

Figure 3.25 illustrates how tuples are stored in four (=2^2) batches. In this case, which batch stores each tuple is determined by the first two bits of the last 5 bits of the tuple's hash-key, because the sizes of the buckets and batches are 2^3 and 2^2, respectively. Batch_0 stores the tuples whose last 5 bits of the hash-key are between '00000' and '00111', Batch_1 stores the tuples whose last 5 bits of the hash-key are between '01000' and '01111' and so on.

Fig. 3.25. Multiple batches in hybrid hash join.



In the hybrid hash join, the build and probe phases are performed the same number of times as the number of batches, because the inner and outer tables are stored in the same number of batches. In the first round of the build and probe phases, not only is every batch created, but also the first batches of both the inner and the outer tables are processed. On the other hand, the processing of the second and subsequent rounds needs writing and reloading to/from the temporary files, so these are costly processes. Therefore, PostgreSQL also prepares a special batch called **skew** to process many tuples more efficiently in the first round.

The skew batch stores the inner table tuples that will be joined with the outer table tuples whose MCV values of the attribute involved in the join condition are relatively large. However, this explanation may not be easy to understand, so it will be explained using a specific example.

Assume that there are two tables: customers and purchase_history. The customers' table has two attributes: 'name' and 'address'; the purchase_history table has two attributes: 'customer_name' and 'purchased_item'. The customers' table has 10,000 rows, and the purchase_history table has 1,000,000 rows. The top 10% customers have purchased 70% of all items.

Under these assumptions, let us consider how the hybrid hash join with skew performs in the first round when the query shown below is executed.

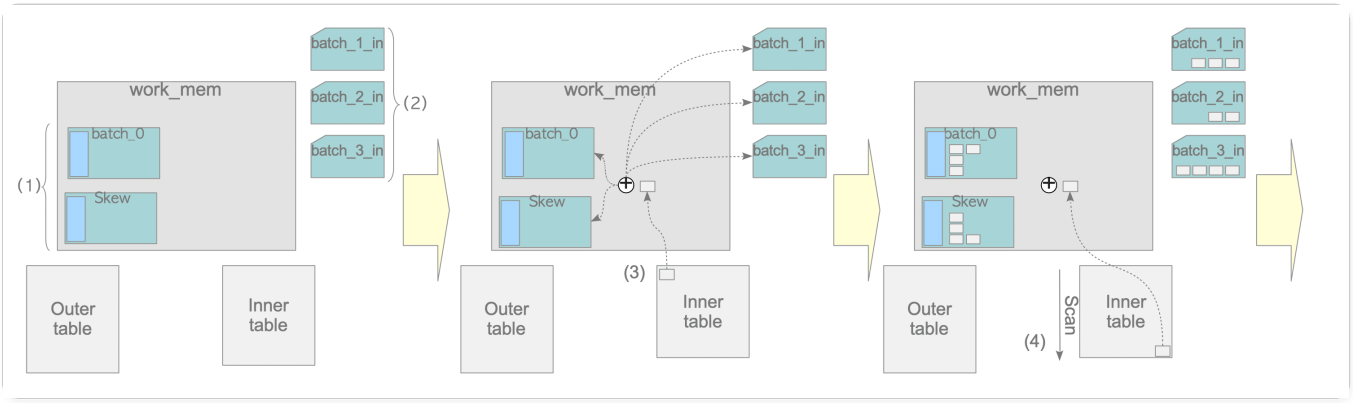
```
testdb=# SELECT * FROM customers AS c, purchase_history AS h WHERE c.name = h.customer_n
ame;
```

If the customers' table is inner and the purchase_history is outer, the top 10% of customers are stored in the skew batch using the MCV values of the purchase_history table. Note that the outer table's MCV values are referenced to insert the inner table tuples into the skew batch. In the probe

phase of the first round, 70% of the tuples of the outer table (purchase_history) will be joined with the tuples stored in the skew batch. This way, the more non-uniform the distribution of the outer table, the more tuples of the outer table can be processed in the first round.

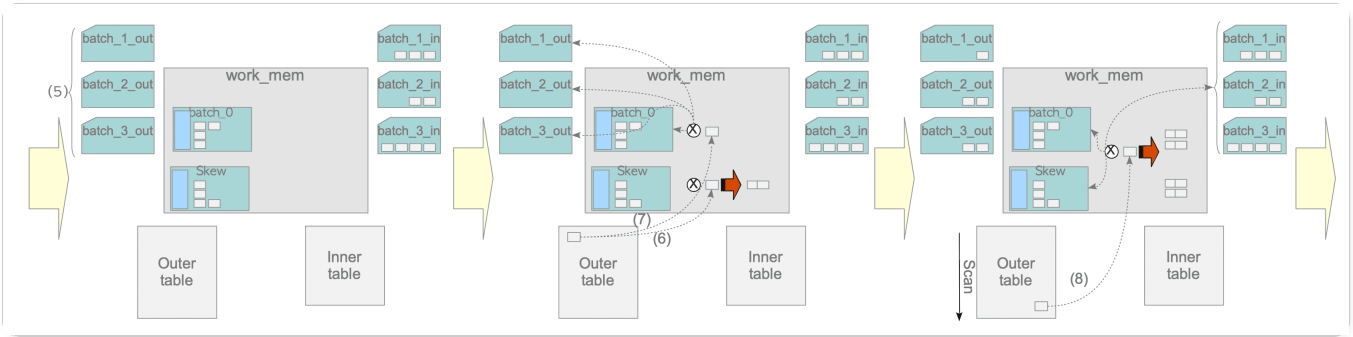
In the following, the working of the hybrid hash join with skew is shown. Refer to Figs. 3.26 to 3.29.

Fig. 3.26. The build phase of the hybrid hash join in the first round.



(1) Create a batch and a skew batch on work_mem.
(2) Create temporary batch files for storing the inner table tuples.
In this example, three batch files are created because the inner table will be divided into four batches.
(3) Perform the build operation for the first tuple of the inner table.
The detail are described below:
1. If the first tuple should be inserted into the skew batch, do so. Otherwise, proceed to 2.
In the example explained above, if the first tuple is one of the top 10% customers, it is inserted into the skew batch.
2. Calculate the hash key of the first tuple and then insert it into the corresponding batch.
(4) Perform the build operation for the remaining tuples of the inner table.

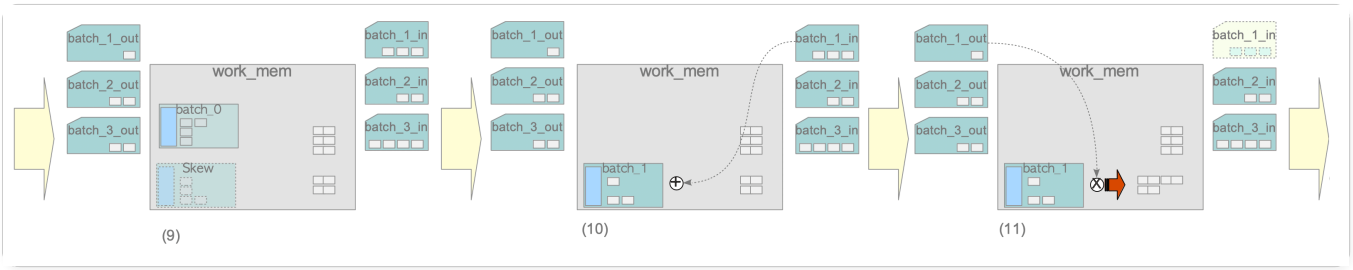
Fig. 3.27. The probe phase of the hybrid hash join in the first round.



(5) Create temporary batch files for storing the outer table tuples.
(6) If the MCV value of the first tuple is large, perform a probe operation with the skew batch. Otherwise, proceed to (7).
In the example explained above, if the first tuple is the purchase data of the top 10% customers, it is compared with the tuples in the skew batch.
(7) Perform the probe operation of the first tuple.
Depending on the hash-key value of the first tuple, the following process is performed:
If the first tuple belongs to Batch_0, perform the probe operation.
Otherwise, insert into the corresponding batch.

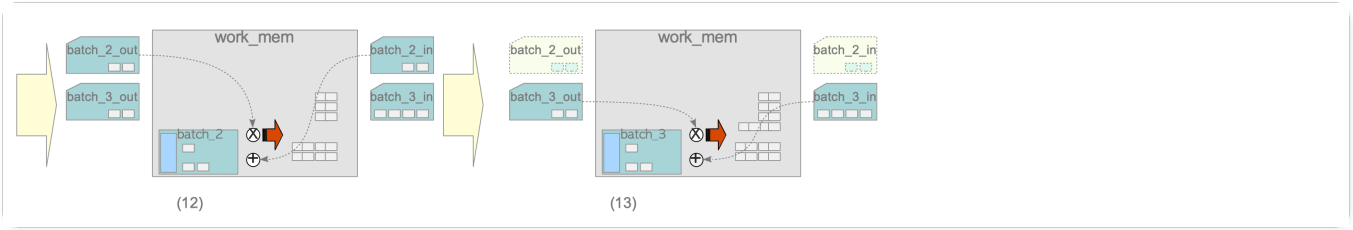
(8) Perform the probe operation from the remaining tuples of the outer table. Note that, in this example, 70% of the tuples of the outer table have been processed by the skew in the first round without writing and reading to/from temporary files.

Fig. 3.28. The build and probe phases in the second round.



(9) Remove the skew batch and clear Batch_0 to prepare the second round.
(10) Perform the build operation from the batch file 'batch_1_in'.
(11) Perform the probe operation for tuples which are stored in the batch file 'batch_1_out'.

Fig. 3.29. The build and probe phases in the third and the last rounds.



(12) Perform build and probe operations using batch files 'batch_2_in' and 'batch_2_out'.
(13) Perform build and probe operations using batch files 'batch_3_in' and 'batch_3_out'.

3.5.3.3. Index Scans in Hash Join

Hash join in PostgreSQL uses index scans if possible. A specific example is shown below.

```
1.  testdb=# EXPLAIN SELECT * FROM pgbench_accounts AS a, pgbench_branches AS b
2.  testdb-#                                            WHERE a.bid = b.bid AND a.a
    id BETWEEN 100 AND 1000;
3.                                                 QUERY PLAN
4.  ------------------------------------------------------------------------------
    -------------------------
5.   Hash Join  (cost=1.88..51.93 rows=865 width=461)
6.     Hash Cond: (a.bid = b.bid)
7.     ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a  (cost=0.4
    3..47.73 rows=865 width=97)
8.           Index Cond: ((aid >= 100) AND (aid <= 1000))
9.     ->  Hash  (cost=1.20..1.20 rows=20 width=364)
10.          ->  Seq Scan on pgbench_branches b  (cost=0.00..1.20 rows=20 width=364)
11. (6 rows)
```

Line 7: In the probe phase, PostgreSQL uses the index scan when scanning the pgbench_accounts table because there is a condition of the column 'aid' which has an index in the WHERE clause.

3.5.4. Join Access Paths and Join Nodes

3.5.4.1. Join Access Paths

The JoinPath structure is an access path for the nested loop join. Other join access paths, such as MergePath and HashPath, are based on it.

All join access paths are illustrated below, without explanation.

```
typedef JoinPath NestPath;

/*
 * JoinType -
 *        enums for types of relation joins
 *
 * JoinType determines the exact semantics of joining two relations using
 * a matching qualification.  For example, it tells what to do with a tuple
 * that has no match in the other relation.
 *
 * This is needed in both parsenodes.h and plannodes.h, so put it here...
 */
typedef enum JoinType
{
        /*
         * The canonical kinds of joins according to the SQL JOIN syntax. Only
         * these codes can appear in parser output (e.g., JoinExpr nodes).
         */
        JOIN_INNER,                                     /* matching tuple pairs only
*/
        JOIN_LEFT,                                      /* pairs + unmatched LHS tupl
es */
        JOIN_FULL,                                      /* pairs + unmatched LHS + un
matched RHS */
        JOIN_RIGHT,                                     /* pairs + unmatched RHS tupl
es */

        /*
         * Semijoins and anti-semijoins (as defined in relational theory) do not
         * appear in the SQL JOIN syntax, but there are standard idioms for
         * representing them (e.g., using EXISTS).  The planner recognizes these
         * cases and converts them to joins.  So the planner and executor must
         * support these codes.  NOTE: in JOIN_SEMI output, it is unspecified
         * which matching RHS row is joined to.  In JOIN_ANTI output, the row is
         * guaranteed to be null-extended.
         */
        JOIN_SEMI,                                      /* 1 copy of each LHS row tha
t has match(es) */
        JOIN_ANTI,                                      /* 1 copy of each LHS row tha
t has no match */
        JOIN_RIGHT_ANTI,                        /* 1 copy of each RHS row that has no
match */

        /*
         * These codes are used internally in the planner, but are not supported
         * by the executor (nor, indeed, by most of the planner).
         */
        JOIN_UNIQUE_OUTER,                      /* LHS path must be made unique */
        JOIN_UNIQUE_INNER                       /* RHS path must be made unique */
```

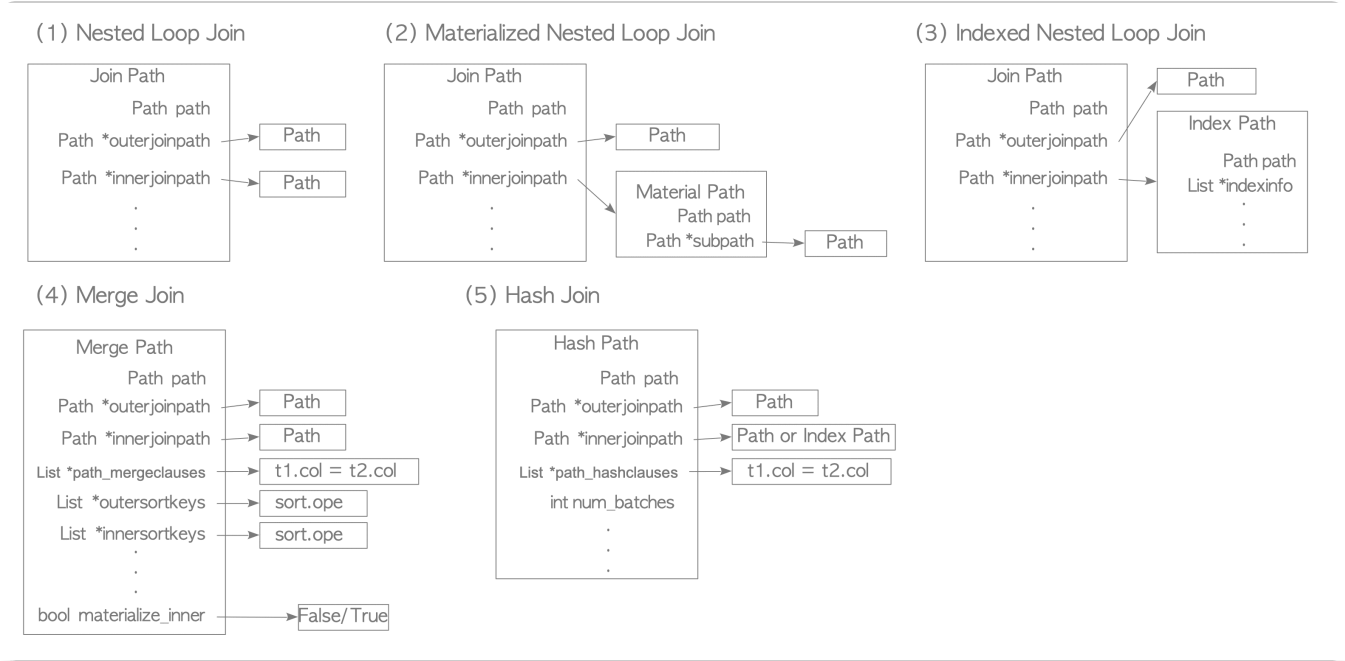
```
        /*
         * We might need additional join types someday.
         */
} JoinType;

/*
 * All join-type paths share these fields.
 */

typedef struct JoinPath
{
        pg_node_attr(abstract)

        Path           path;

        JoinType       jointype;

        bool           inner_unique;   /* each outer tuple provably matches no more
                                                                * than one inner tup
le */

        Path      *outerjoinpath;      /* path for the outer side of the join */
        Path      *innerjoinpath;      /* path for the inner side of the join */

        List      *joinrestrictinfo;   /* RestrictInfos to apply to join */

        /*
         * See the notes for RelOptInfo and ParamPathInfo to understand why
         * joinrestrictinfo is needed in JoinPath, and can't be merged into the
         * parent RelOptInfo.
         */
} JoinPath;
```

```
/*
 * A mergejoin path has these fields.
 *
 * Unlike other path types, a MergePath node doesn't represent just a single
 * run-time plan node: it can represent up to four.  Aside from the MergeJoin
 * node itself, there can be a Sort node for the outer input, a Sort node
 * for the inner input, and/or a Material node for the inner input.  We could
 * represent these nodes by separate path nodes, but considering how many
 * different merge paths are investigated during a complex join problem,
 * it seems better to avoid unnecessary palloc overhead.
 *
 * path_mergeclauses lists the clauses (in the form of RestrictInfos)
 * that will be used in the merge.
 *
 * Note that the mergeclauses are a subset of the parent relation's
 * restriction-clause list.  Any join clauses that are not mergejoinable
 * appear only in the parent's restrict list, and must be checked by a
 * qpqual at execution time.
 *
 * outersortkeys (resp. innersortkeys) is NIL if the outer path
 * (resp. inner path) is already ordered appropriately for the
 * mergejoin.  If it is not NIL then it is a PathKeys list describing
 * the ordering that must be created by an explicit Sort node.
 *
 * skip_mark_restore is true if the executor need not do mark/restore calls.
 * Mark/restore overhead is usually required, but can be skipped if we know
 * that the executor need find only one match per outer tuple, and that the
 * mergeclauses are sufficient to identify a match.  In such cases the
```

```
 * executor can immediately advance the outer relation after processing a
 * match, and therefore it need never back up the inner relation.
 *
 * materialize_inner is true if a Material node should be placed atop the
 * inner input.  This may appear with or without an inner Sort step.
 */

typedef struct MergePath
{
        JoinPath        jpath;
        List       *path_mergeclauses;  /* join clauses to be used for merge */
        List       *outersortkeys;      /* keys for explicit sort, if any */
        List       *innersortkeys;      /* keys for explicit sort, if any */
        bool            skip_mark_restore;      /* can executor skip mark/restore? */
        bool            materialize_inner;      /* add Materialize to inner? */
} MergePath;
```

```
/*
 * A hashjoin path has these fields.
 *
 * The remarks above for mergeclauses apply for hashclauses as well.
 *
 * Hashjoin does not care what order its inputs appear in, so we have
 * no need for sortkeys.
 */

typedef struct HashPath
{
        JoinPath        jpath;
        List           *path_hashclauses;      /* join clauses used for hashing */
        int             num_batches;            /* number of batches expected */
        Cardinality     inner_rows_total;       /* total inner rows expected */
} HashPath;
```

Fig. 3.30. Join access paths.



3.5.4.2. Join Nodes

This subsection shows the three join nodes, NestedLoopNode, MergeJoinNode and HashJoinNode, without explanation. They are all based on the JoinNode.

```
/* ----------------
 *              Join node
 *
 * jointype:    rule for joining tuples from left and right subtrees
 * inner_unique each outer tuple can match to no more than one inner tuple
 * joinqual:    qual conditions that came from JOIN/ON or JOIN/USING
 *                              (plan.qual contains conditions that came from WHERE)
 *
 * When jointype is INNER, joinqual and plan.qual are semantically
 * interchangeable.  For OUTER jointypes, the two are *not* interchangeable;
 * only joinqual is used to determine whether a match has been found for
 * the purpose of deciding whether to generate null-extended tuples.
 * (But plan.qual is still applied before actually returning a tuple.)
 * For an outer join, only joinquals are allowed to be used as the merge
 * or hash condition of a merge or hash join.
 *
 * inner_unique is set if the joinquals are such that no more than one inner
 * tuple could match any given outer tuple.  This allows the executor to
 * skip searching for additional matches.  (This must be provable from just
 * the joinquals, ignoring plan.qual, due to where the executor tests it.)
 * ----------------
 */
typedef struct Join
{
        pg_node_attr(abstract)

        Plan           plan;
        JoinType       jointype;
        bool           inner_unique;
        List      *joinqual;           /* JOIN quals (in addition to plan.qual) */
} Join;
```

```
/* ----------------
 *              nest loop join node
 *
 * The nestParams list identifies any executor Params that must be passed
 * into execution of the inner subplan carrying values from the current row
 * of the outer subplan.  Currently we restrict these values to be simple
 * Vars, but perhaps someday that'd be worth relaxing.  (Note: during plan
 * creation, the paramval can actually be a PlaceHolderVar expression; but it
 * must be a Var with varno OUTER_VAR by the time it gets to the executor.)
 * ----------------
 */
typedef struct NestLoop
{
        Join           join;
        List      *nestParams;         /* list of NestLoopParam nodes */
} NestLoop;

typedef struct NestLoopParam
{
        pg_node_attr(no_equal, no_query_jumble)

        NodeTag                type;
        int                    paramno;               /* number of the PARAM_EXEC P
aram to set */
        Var               *paramval;           /* outer-relation Var to assign to Pa
ram */
} NestLoopParam;
```

```
/* ----------------
 *              merge join node
```

```
 *
 * The expected ordering of each mergeable column is described by a btree
 * opfamily OID, a collation OID, a direction (BTLessStrategyNumber or
 * BTGreaterStrategyNumber) and a nulls-first flag.  Note that the two sides
 * of each mergeclause may be of different datatypes, but they are ordered the
 * same way according to the common opfamily and collation.  The operator in
 * each mergeclause must be an equality operator of the indicated opfamily.
 * ----------------
 */
typedef struct MergeJoin
{
        Join            join;

        /* Can we skip mark/restore calls? */
        bool            skip_mark_restore;

        /* mergeclauses as expression trees */
        List       *mergeclauses;

        /* these are arrays, but have the same length as the mergeclauses list: */

        /* per-clause OIDs of btree opfamilies */
        Oid                *mergeFamilies pg_node_attr(array_size(mergeclauses));

        /* per-clause OIDs of collations */
        Oid                *mergeCollations pg_node_attr(array_size(mergeclauses));

        /* per-clause ordering (ASC or DESC) */
        int                *mergeStrategies pg_node_attr(array_size(mergeclauses));

        /* per-clause nulls ordering */
        bool       *mergeNullsFirst pg_node_attr(array_size(mergeclauses));
} MergeJoin;
```

```
/* ----------------
 *              hash join node
 * ----------------
 */
typedef struct HashJoin
{
        Join            join;
        List       *hashclauses;
        List       *hashoperators;
        List       *hashcollations;

        /*
         * List of expressions to be hashed for tuples from the outer plan, to
         * perform lookups in the hashtable over the inner plan.
         */
        List       *hashkeys;
} HashJoin;
```

3.6. Creating the Plan Tree of Multiple-Table Query

In this section, the process of creating a plan tree of a multiple-table query is explained.

3.6.1. Preprocessing

The subquery_planner() function defined in planner.c invokes preprocessing. The preprocessing for single-table queries has already been described in Section 3.3.1. In this subsection, the preprocessing for a multiple-table query will be described. However, only some parts will be described, as there are many.

1. Planning and Converting CTE
   If there are WITH lists, the planner processes each WITH query by the SS_process_ctes() function.
2. Pulling Subqueries Up
   If the FROM clause has a subquery and it does not have GROUP BY, HAVING, ORDER BY, LIMIT or DISTINCT clauses, and also it does not use INTERSECT or EXCEPT, the planner converts to a join form by the pull_up_subqueries() function. For example, the query shown below which contains a subquery in the FROM clause can be converted to a natural join query. Needless to say, this conversion is done in the query tree.

```
testdb=# SELECT * FROM tbl_a AS a, (SELECT * FROM tbl_b) as b WHERE a.id = b.id;
                                  ↓
testdb=# SELECT * FROM tbl_a AS a, tbl_b as b WHERE a.id = b.id;
```

3. Transforming an Outer Join to an Inner Join
   The planner transforms an outer join query to an inner join query if possible.

3.6.2. Getting the Cheapest Path

To get the optimal plan tree, the planner has to consider all the combinations of indexes and join methods. This is a very expensive process and it will be infeasible if the number of tables exceeds a certain level due to a combinatorial explosion. Fortunately, if the number of tables is smaller than around 12, the planner can get the optimal plan by applying dynamic programming. Otherwise, the planner uses the *genetic algorithm*. Refer to the ℹ below.

ℹ Genetic Query Optimizer

When a query joining many tables is executed, a huge amount of time will be needed to optimize the query plan. To deal with this situation, PostgreSQL implements an interesting feature: the Genetic Query Optimizer. This is a kind of approximate algorithm to determine a reasonable plan within a reasonable time. Hence, in the query optimization stage, if the number of the joining tables is higher than the threshold specified by the parameter geqo_threshold (the default is 12), PostgreSQL generates a query plan using the genetic algorithm.

Determination of the optimal plan tree by dynamic programming can be explained by the following steps:

Level = 1
Get the cheapest path for each table. The cheapest path is stored in the respective RelOptInfo.

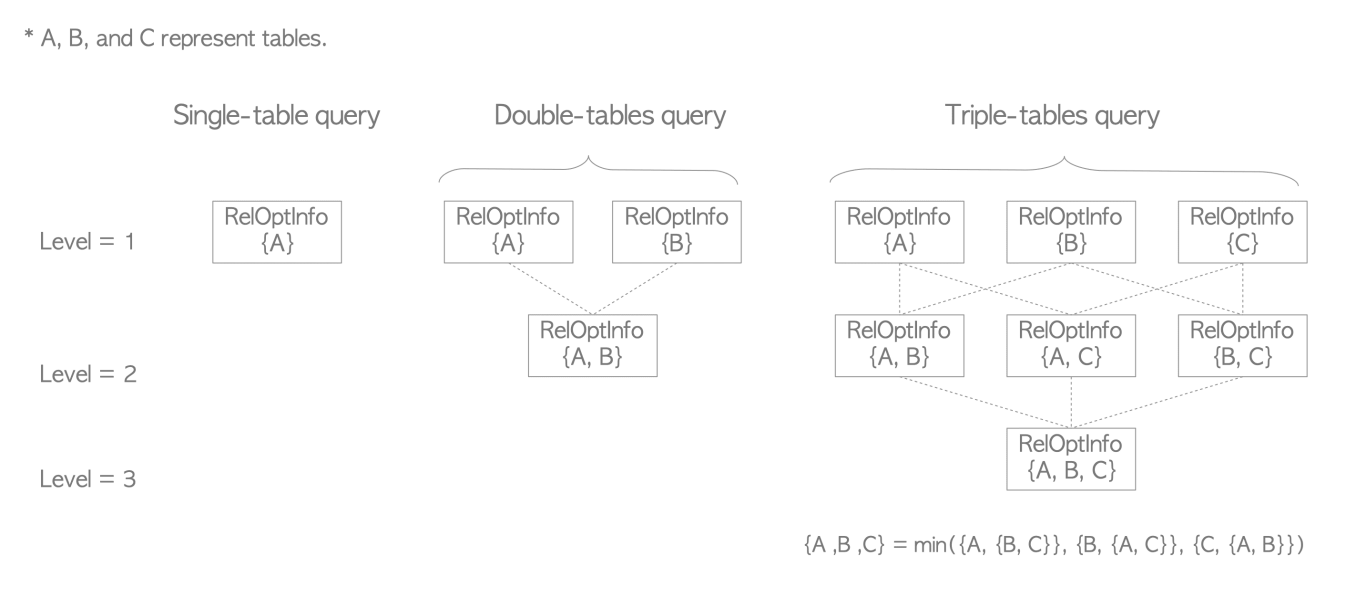
Level = 2
Get the cheapest path for each combination of two tables.
For example, if there are two tables, A and B, get the cheapest join path of tables A and B. This is the final answer.
In the following, the RelOptInfo of two tables is represented by {A, B}.
If there are three tables, get the cheapest path for each of {A, B}, {A, C}, and {B, C}.

Level = 3 and higher

Continue the same process until the level that equals the number of tables is reached.

This way, the cheapest paths of the partial problems are obtained at each level and are used to get the upper level's calculation. This makes it possible to calculate the cheapest plan tree efficiently.

Fig. 3.31. How to get the cheapest access path using dynamic programming.



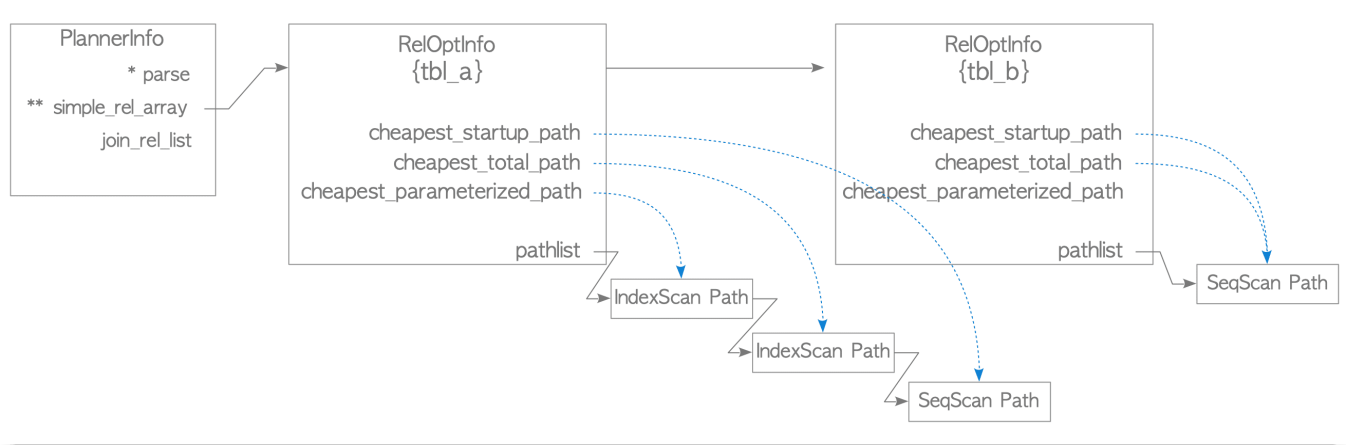
In the following, the process of how the planner gets the cheapest plan of the following query is described.

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

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

testdb=# SELECT * FROM tbl_a AS a, tbl_b AS b WHERE a.id = b.id AND b.data < 400;
```

3.6.2.1. Processing in Level 1

In Level 1, the planner creates a RelOptInfo structure and estimates the cheapest costs for each relation in the query. The RelOptInfo structures are added to the simple_rel_array of the PlannerInfo of this query.

Fig. 3.32. The PlannerInfo and RelOptInfo after processing in Level 1.



The RelOptInfo of tbl_a has three access paths, which are added to the pathlist of the RelOptInfo. Each access path is linked to a cheapest cost path: the *cheapest start-up (cost) path*, the *cheapest total (cost) path*, and the *cheapest parameterized (cost) path*. The cheapest start-up and total cost paths are obvious, so the cost of the cheapest parameterized index scan path will be described.

As described in Section 3.5.1.3, the planner considers the use of the parameterized path for the indexed nested loop join (and rarely the indexed merge join with an outer index scan). The cheapest parameterized cost is the cheapest cost of the estimated parameterized paths.

The RelOptInfo of tbl_b only has a sequential scan access path because tbl_b does not have a related index.

3.6.2.2. Processing in Level 2

In level 2, a RelOptInfo structure is created and added to the join_rel_list of the PlannerInfo. Then, the costs of all possible join paths are estimated, and the best access path, whose total cost is the cheapest, is selected. The RelOptInfo stores the best access path as the cheapest total cost path. See Fig. 3.33.

Fig. 3.33. The PlannerInfo and RelOptInfo after processing in Level 2.

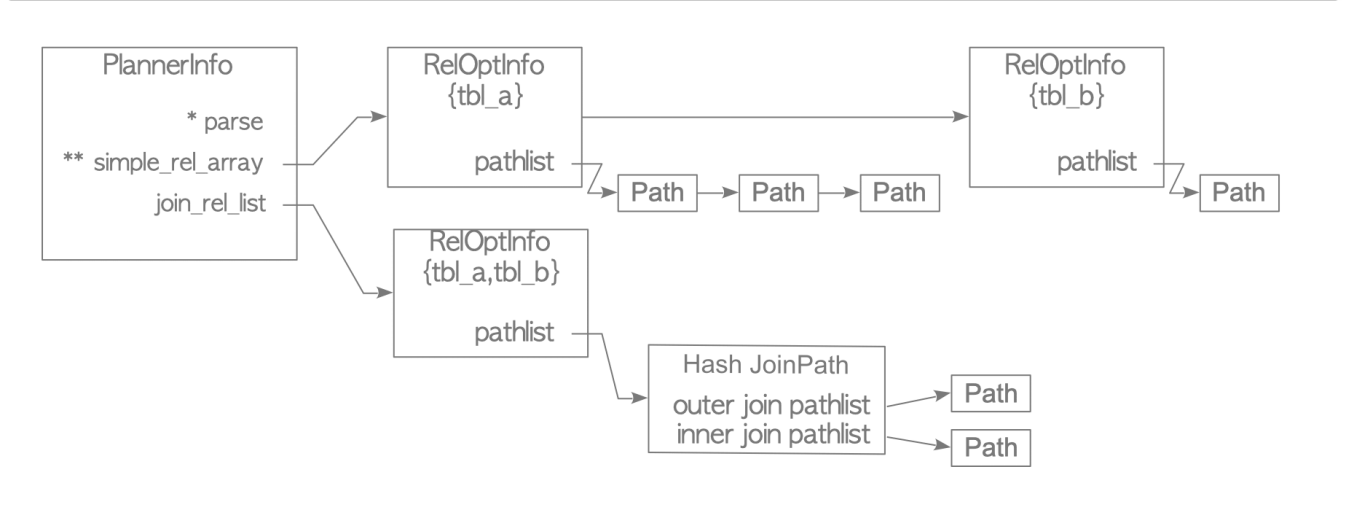


Table 3.1 shows all combinations of join access paths in this example. The query of this example is an equi-join type, so all three join methods are estimated. For convenience, some notations of access paths are introduced:

- *SeqScanPath(table)* means the sequential scan path of table.
- *Materialized->SeqScanPath(table)* means the materialized sequential scan path of a table.

- *IndexScanPath(table, attribute)* means the index scan path by the attribute of the a table.
- *ParameterizedIndexScanPath(table, attribute1, attribute2)* means the parameterized index path by the attribute1 of the table, and it is parameterized by attribute2 of the outer table.

Table 3.1: All combinations of join access paths in this example

	Outer Path	Inner Path	
Nested Loop Join			
1	SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2	SeqScanPath(tbl_a)	Materialized->SeqScanPath(tbl_b)	Materialized nested loop join
3	IndexScanPath(tbl_a,id)	SeqScanPath(tbl_b)	Nested loop join with outer index scan
4	IndexScanPath(tbl_a,id)	Materialized->SeqScanPath(tbl_b)	Materialized nested loop join with outer index scan
5	SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	
6	SeqScanPath(tbl_b)	Materialized->SeqScanPath(tbl_a)	Materialized nested loop join
7	SeqScanPath(tbl_b)	ParametalizedIndexScanPath(tbl_a, id, tbl_b.id)	Indexed nested loop join
Merge Join			
1	SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2	IndexScanPath(tbl_a,id)	SeqScanPath(tbl_b)	Merge join with outer index scan
3	SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	
Hash Join			
1	SeqScanPath(tbl_a)	SeqScanPath(tbl_b)	
2	SeqScanPath(tbl_b)	SeqScanPath(tbl_a)	

For example, in the nested loop join, seven join paths are estimated. The first one indicates that the outer and inner paths are the sequential scan paths of tbl_a and tbl_b, respectively. The second indicates that the outer path is the sequential scan path of tbl_a and the inner path is the materialized sequential scan path of tbl_b. And so on.

The planner finally selects the cheapest access path from the estimated join paths, and the cheapest path is added to the pathlist of the RelOptInfo {tbl_a,tbl_b}. See Fig. 3.33.

In this example, as shown in the result of EXPLAIN below, the planner selects the hash join whose inner and outer tables are tbl_b and tbl_c.

```
testdb=# EXPLAIN  SELECT * FROM tbl_b AS b, tbl_c AS c WHERE c.id = b.id AND b.da
ta < 400;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=90.50..277.00 rows=400 width=16)
   Hash Cond: (c.id = b.id)
   ->  Seq Scan on tbl_c c  (cost=0.00..145.00 rows=10000 width=8)
```

```
   ->  Hash  (cost=85.50..85.50 rows=400 width=8)
         ->  Seq Scan on tbl_b b  (cost=0.00..85.50 rows=400 width=8)
               Filter: (data < 400)
(6 rows)
```

3.6.3. Getting the Cheapest Path of a Triple-Table Query

Obtaining the cheapest path of a query involving three tables is as follows:

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

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

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

testdb=# SELECT * FROM tbl_a AS a, tbl_b AS b, tbl_c AS c
testdb-#                WHERE a.id = b.id AND b.id = c.id AND a.data < 40;
```

Level 1:

The planner estimates the cheapest paths of all tables and stores this information in the corresponding RelOptInfo objects: {tbl_a}, {tbl_b}, and {tbl_c}.

Level 2:

The planner picks all the combinations of pairs of the three tables and estimates the cheapest path for each combination. The planner then stores the information in the corresponding RelOptInfo objects: {tbl_a, tbl_b}, {tbl_b, tbl_c}, and {tbl_a, tbl_c}.

Level 3:

The planner finally gets the cheapest path using the already obtained RelOptInfo objects. More precisely, the planner considers three combinations of RelOptInfo objects: {tbl_a, {tbl_b, tbl_c}}, {tbl_b, {tbl_a, tbl_c}}, and {tbl_c, {tbl_a, tbl_b}}, because

$$\{\texttt{tbl\_a}, \texttt{tbl\_b}, \texttt{tbl\_c}\} = min(\{\texttt{tbl\_a}, \{\texttt{tbl\_b}, \texttt{tbl\_c}\}\}, \{\texttt{tbl\_b}, \{\texttt{tbl\_a}, \texttt{tbl\_c}\}\}, \{\texttt{tbl\_c}, \{\texttt{tbl\_a}, \texttt{tbl\_b}\}\}).$$

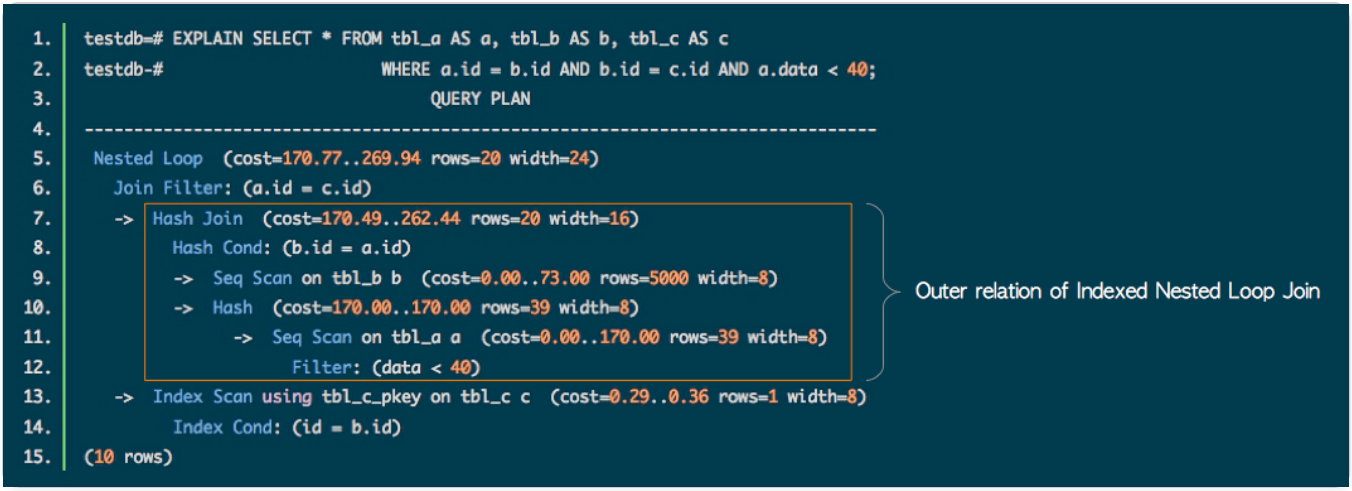
The planner then estimates the costs of all possible join paths in them.

In the RelOptInfo onject {tbl_c, {tbl_a, tbl_b}}, the planner estimates all the combinations of tbl_c and the cheapest path of {tbl_a, tbl_b}, which is the hash join whose inner and outer tables are

tbl_a and tbl_b, respectively, in this example. The estimated join paths will contain three kinds of join paths and their variations, such as those shown in the previous subsection, that is, the nested loop join and its variations, the merge join and its variations, and the hash join.

The planner processes the RelOptInfo objects {tbl_a, {tbl_b, tbl_c}} and {tbl_b, {tbl_a, tbl_c}} in the same way and finally selects the cheapest access path from all the estimated paths.

The result of the EXPLAIN command of this query is shown below:



The outermost join is the indexed nested loop join (Line 5). The inner parameterized index scan is shown in line 13, and the outer relation is the result of the hash join whose inner and outer tables are tbl_b and tbl_a, respectively (lines 7-12). Therefore, the executor first executes the hash join of tbl_a and tbl_b and then executes the indexed nested loop join.

References

[1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, "Database System Concepts", McGraw-Hill Education, ISBN-13: 978-0073523323

[2] Thomas M. Connolly, and Carolyn E. Begg, "Database Systems", Pearson, ISBN-13: 978-0321523068

← Back to Part 1, Part 2