EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

PostgreSQL Join Methods

Legal Notice: This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM®.

Confidential

CONTENTS

1. JOIN METHODS

If all the query planner had to do was decide between index scan types and how to combine them using its wide array of derived nodes, its life would be pretty easy. All the serious complexity in the planner and optimizer relates to joining tables together. Each time another table is added to a list that needs to be joined, the number of possible ways goes up dramatically. If there's, say, three tables to join, you can expect the query plan to consider every possible plan and select the optimal one. But if there are 20 tables to join, there's no possible way it can exhaustively search each join possibility. As there are a variety of techniques available to join each table pair, that further expands the possibilities. The universe of possible plans has to be pruned somehow.

Fundamentally, each way two tables can be joined together gives the same output. The only difference between them is how efficient the result is to execute. All joins consider an outer and inner table. These alternately may be called the left and the right, if considering a query plan as a tree or a graph, and that usage shows up in some of the PostgreSQL internal documentation.

1.1 NESTED LOOP

If you need every possible row combination joined together, the NESTED LOOP is what you want. In most other cases, it's probably not. The classic pseudo code description of a NESTED LOOP join looks like the following:

for each outer row:

for each inner row:

if join condition is true:

output combined row

Plan Example:

```
EXPLAIN ANALYZE SELECT * FROM orders,orderlines WHERE
orders.totalamount=329.78 AND orders.orderid=orderlines.orderid;
QUERY PLAN
----------
Nested Loop  (cost=0.00..265.41 rows=5 width=54) (actual time=0.108..12.886
rows=9 loops=1)
       ->  Seq Scan on orders  (cost=0.00..250.00 rows=1 width=36) (actual
time=0.073..12.787 rows=1 loops=1)
             Filter: (totalamount = 329.78)
       ->  Index Scan using ix_orderlines_orderid on orderlines
(cost=0.00..15.34 rows=5 width=18) (actual time=0.024..0.050 row
    s=9 loops=1)
             Index Cond: (orderlines.orderid = orders.orderid)
Total runtime: 12.999 ms
```

Both the inner and outer loops here could be executing against any of the scan types: sequential, indexed, bitmap, or even the output from another join. As you can see from the code, the amount of time this takes to run is proportional to the number of rows in the outer table multiplied by the rows in the inner. It is considering every possible way to join every row in each table with every other row.

Joining data using merges and hashes is normal for the real world that tends to be indexed or has a clear relationship between tables. You can see one if you just forget to put a WHERE condition on a join, though, which then evaluates the cross product and outputs a ton of rows.

Note that a NESTED LOOP is the only way to execute a CROSS JOIN, and it can potentially be the only way to compute complicated conditions that don't map into either a useful MERGE or a HASH JOIN instead.

The standard situation you'll see a NESTED LOOP in is one where the inner table is only returning back a limited number of rows. If an index exists on one of the two tables involved, the optimizer is going to use it to limit the number of rows substantially, and the result may then make the inner * outer rows runtime of the NESTED LOOP acceptable.

A NESTED LOOP can also show up when both the inner and outer scan use an index.

1.2 MERGE JOIN

The idea here is to use sorted lists to join the results. If both sides of the join are sorted, the system can just take the rows from the top and see if they match and return them. The main requirement here is that the lists are sorted.

The inner table can be rescanned more than once if the outer one has duplicate values. That's where the normally-forwards scan on it goes backwards, to consider the additional matching set of rows from the duplication. You can only see a MERGE JOIN when joining on an equality condition, not an inequality or a range.

Here is a sample plan:

Merge join

Sort table 1

Sequential scan table 1

Sort table 2

Sequential scan table 2

To join these two tables (table 1 and table 2), data has to be provided in a sorted order. In many cases, PostgreSQL will just sort the data. However, there are other options we can use to provide the join with sorted data. One way is to consult an index, as shown in the following example:

Merge join

Index scan table 1

Index scan table 2

Plan Example:

```
EXPLAIN ANALYZE SELECT C.customerid,sum(netamount) FROM customers C, orders
O WHERE C.customerid=O.customerid GROUP BY C.customerid;
QUERY PLAN
----------
GroupAggregate  (cost=0.05..2069.49 rows=12000 width=12) (actual
time=0.099..193.668 rows=8996 loops=1)
       ->  Merge Join  (cost=0.05..1859.49 rows=12000 width=12) (actual
time=0.071..146.272 rows=12000 loops=1)
             Merge Cond: (c.customerid = o.customerid)
             ->  Index Scan using customers_pkey on customers c
(cost=0.00..963.25 rows=20000 width=4) (actual time=0.031..37.242
rows=20000 loo
    ps=1)
             ->  Index Scan using ix_order_custid on orders o
(cost=0.00..696.24 rows=12000 width=12) (actual time=0.025..30.722
rows=12000 loop
    s=1)
Total runtime: 206.353 ms
```

One side of the join or both sides can use sorted data coming from lower levels of the plan. If the table is accessed directly, an index is the obvious choice for this, but only if the returned result set is significantly smaller than the entire table. Otherwise, we encounter almost double the overhead because we have to read the entire index and then the entire table. If the result set is a large portion of the table, a sequential scan is more efficient, especially if it is being accessed in the primary key order.

The beauty of a merge join is that it can handle a lot of data. The downside is that data has to be sorted or taken from an index at some point.

1.3 HASH JOIN

The primary alternative to a MERGE JOIN, a HASH JOIN doesn't sort its input. Instead, it creates a hash table from each row of the inner table, scanning for matching ones in the outer. The output will not necessarily be in any useful order.

Whether a HASH JOIN is better or worse than the other possibilities depends on things such as whether input is already sorted (in which case, a MERGE JOIN may be inexpensive) and how much memory is required to execute it. The hash tables built for the inner scan here require enough memory to store all the rows, which can be large.

The following listing shows how a hash join works:

Hash join

Sequential scan table 1

Sequential scan table 2

Plan Example:

```
EXPLAIN ANALYZE SELECT prod_id,title FROM products p WHERE EXISTS (SELECT 1
FROM orderlines ol WHERE ol.prod_id=p.prod_id);
QUERY PLAN
--------------------
Hash Join  (cost=1328.16..2270.16 rows=9724 width=19) (actual
time=249.783..293.588 rows=9973 loops=1)
       Hash Cond: (p.prod_id = ol.prod_id)
       ->  Seq Scan on products p  (cost=0.00..201.00 rows=10000 width=19)
(actual time=0.007..12.781 rows=10000 loops=1)
       ->  Hash  (cost=1206.62..1206.62 rows=9724 width=4) (actual
time=249.739..249.739 rows=9973 loops=1)
             Buckets: 1024  Batches: 1  Memory Usage: 234kB
             ->  HashAggregate  (cost=1109.38..1206.62 rows=9724 width=4)
(actual time=219.695..234.154 rows=9973 loops=1)
                   ->  Seq Scan on orderlines ol  (cost=0.00..958.50
rows=60350 width=4) (actual time=0.005..91.874 rows=60350 lo
    ops=1)
Total runtime: 306.523 ms
```

The basic version of the hash join algorithm includes two phases:

1. During the build phase, all tuples of table 1 are stored in buckets according to the values of the hash function.

2. In the probe phase, each row of table 2 is sent to an appropriate bucket. If matching rows of table 1 are in the bucket, output rows are produced.

The easiest way to find matching rows in the bucket is to use nested loops. PostgreSQL uses a better matching algorithm based on Bloom filtering. The two phases of the hash-based algorithm are shown as separate physical operations in the execution plan.

The basic hash join algorithm works if all buckets produced at the build phase can fit into main memory. Another variation, called hybrid hash join, joins tables that cannot fit into main memory. The hybrid hash join partitions both tables so that partitions of one table can fit and then executes a basic algorithm for each pair of corresponding partitions. The cost of a hybrid hash join is higher because partitions are stored temporarily on the hard disk and both tables are scanned twice. However, the cost is still proportional to the sum of the sizes, rather than the product.

A hash-based algorithm is significantly better than nested loops for large tables and a large number of different values of the join attribute. For example, if the join attribute is unique in one of the input tables, then the last term will be equal to just the size of the other table.

Looking up values in a hash table only works if the operator in the join condition is =, so you need at least one join condition with that operator.

2. JOIN TYPES

2.1 STANDART TYPES

Inner Join - join that returns only rows that satisfy the join condition. Inner joins are either equijoins or non-equijoins.

Left/Right Outer Joins - An outer join returns all rows from one table and only those rows from the joined table where the join condition is met.

Full Outer Join - A full outer join will join two tables from left-to-right and right-to-left. Records that join in both directions are output once to avoid duplication. The full outer join will return all the rows from both tables that match plus the rows that are unique to each table.

Semi or anti joins are kind of sub join types to the joining methods such as hash, merge, and nested loop, where the optimizer prefers to use them for EXISTS/IN or NOT EXISTS/NOT IN operators.

Semi join will return a single value for all the matching records from the other table. That is, if the second table has multiple matching entries for the first table's record, then it will return only one copy from the first table. However, a normal join it will return multiple copies from the first table.

Anti-join will return rows, when no matching records are found in the second table. It is quite opposite to the semi join, since it is returning records from the first table, when there is no match in the second table.

Cross Joins - The Cross Join creates a **cartesian product between two sets of data**. This type of join does not maintain any relationship between the sets; instead returns the result, which is the number of rows in the first table multiplied by the number of rows in the second table. For every possible combination of rows from T1 and T2 (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in T1 followed by all columns in T2. If the tables have N and M rows respectively, the joined table will have N * M rows.

2.2 LATERAL JOINS

Suppose you perform joins on two tables or subqueries; normally, the pair participating in the join are independent units and can’t read data from each other.

LATERAL lets you share data in columns across two tables in a FROM clause. However, it works only in one direction: the righthand side can draw from the left-hand side, but not vice versa.

Lateral is also helpful for using values from the left-hand side to limit the number of rows returned from the righthand side.

Although you can achieve the same results by using window functions, lateral joins yield faster results with more succinct syntax.

You can use multiple lateral joins in your SQL and even chain them in sequence as you would when joining more than two subqueries.

You can sometimes get away with omitting the LATERAL keyword; the query parser is smart enough to figure out a lateral join if you have a correlated expression. But we advise that you always include the keyword for the sake of clarity. Also, you'll get an error if you write your statement assuming the use of a lateral join but run the statement on a pre-lateral version PostgreSQL. Without the keyword, PostgreSQL might end up performing a join with unintended results.

3. JOIN ORDER

3.1 INTRODUCING JOIN PRUNING

PostgreSQL also provides an optimization called join pruning. The idea is to remove joins if they are not required by the query. This can come in handy if queries are generated by some middleware or ORM. If a join can be removed, it naturally speeds things up dramatically and leads to less overhead.

There are two reasons why this is actually possible and logically correct:

- No columns are selected from the right-hand side of the join; thus, looking those columns up doesn't buy us anything.
- The right-hand side is unique, which means that joining cannot increase the number of rows due to duplicates on the right-hand side.

If joins can be pruned automatically, then the queries may be a magnitude faster. The beauty here is that an increase in speed can be achieved by just removing columns that may not be required by the application in any case.

3.2 JOIN ORDER

During the planning process, PostgreSQL tries to check all the possible join orders. In many cases, this can be pretty expensive because there can be many permutations, which naturally slows down the planning process. So, the optimizer has chosen the best join order without any intervention from the SQL developer, but this isn't always the case.

Long queries are more likely in OLAP systems. A query is considered long when query selectivity is high for at least one of the large tables; that is, almost all rows contribute to the output, even when the output size is small. In other words, a long query is most likely an analytical report that most likely joins a number of tables. This number, as anyone who has worked with OLAP systems can attest, can be massive. When the number of tables involved in a query becomes too large, the optimizer no longer attempts to find the best possible join order. In PostgreSQL there is a system parameter called *join_collapse_limit*.

This parameter caps the number of tables in a join that will be still processed by the cost-based optimizer. The default value of this parameter is 8. This means that if the number of tables in a join is eight or fewer, the optimizer will perform a selection of candidate plans, compare plans, and choose the best one. But if the number of tables is nine or more, it will simply execute the joins in the order the tables are listed in the SELECT statement.

Setting it to 1 prevents any reordering of explicit JOINs (Implicit JOINs - all tables are listed in the FROM clause and are later connected in the WHERE clause. Explicit JOINs - tables are connected directly using an ON and JOIN). Thus, the explicit join order specified in the query will be the actual order in which the relations are joined. Because the query planner does not always choose the optimal join order, advanced users can elect to temporarily set this variable to 1, and then specify the join order they desire explicitly.

Why not set this parameter to the highest possible value? There is no official upper limit to this parameter, so it can be the maximum integer, which is 2147483647. However, the higher you set this parameter, the more time will be spent to choose the best plan. The number of possible plans to consider for a query joining n is n! Thus, when the value is 8, a maximum of 40,000 plans can be compared. If this value is increased to 10, the number of plans to consider will increase to three million, and the number rises predictably from there—when this parameter is set to 20, the total number of plans is already too big to fit the integer. The results were excruciating—not only did the execution stall but even the EXPLAIN command couldn't return a result.

4. CTE

4.1 TEMPORARY TEBLES

Sometimes the attempt of SQL developers to speed up a query execution may result in slowing it down. This often happens when they decide to use temporary tables.

Temporary tables are visible to the current session only and are dropped when the session disconnects if not dropped explicitly before that. Otherwise, they are as good as regular tables, they can be used in the queries with no limitations, and they can even be indexed. Temporary tables are often used to store intermediate results of the queries:

CREATE TEMP TABLE temp_a AS

SELECT…

It all works great if you use a temporary table to store results of your query for some analysis and then discard it when done. But often, when a SQL developer starts to use temporary tables to store the results of each step.

The chain of temporary tables can become quite long. Does it cause any problems? Yes, and there are many of them, including the following:

• *Indexes* – After selected data is stored in a temporary table, we can't use indexes that were created on the source table(s). We either need to continue without indexes or build new ones on temporary tables, which takes time.

• *Statistics* – Since we created a new table, the optimizer can't utilize statistical data on value distribution from the source table(s), so we need either to go without statistics or run the ANALYZE command on a temporary table.

• *Tempdb space* – Temporary tables are stored in tempdb, a tablespace, which is also used for joins, sorting, grouping, and so on, when intermediate results can't fit into available main memory. As unlikely as it may sound, we've observed situations where large queries were competing for space with temporary tables, resulting in queries being canceled.

• *Excessive I/O* – Temporary tables are still tables, which means they may be written to disk, and it takes extra time to write to and read from disk.

The most important negative implication of excessive use of temporary tables is that this practice blocks the optimizer from doing rewrites.

By saving the results of each join into a temporary table, you prevent the optimizer from choosing the optimal join order; you "lock" in the order in which you created the temporary tables.

4.2 CTES

Common table expressions, or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

For all versions below 12, a CTE was processed exactly like a temporary table. The results were materialized in main memory with possible disk failover. That means that there was no advantage to using a CTE instead of a temporary table.

To be fair, a CTE's intended purpose was different. The idea behind the usage of a CTE was that if you need to use some possibly complex sub-select more than once, you can define it as a CTE and reference it in a query multiple times. In this case, PostgreSQL will compute results just once and reuse it as many times as needed.

Because of this intended usage, the optimizer planned the CTE execution separately from the rest of the query and did not push any join conditions inside the CTE, providing a so-called optimization fence. This is especially important if WITH is used in INSERT/DELETE/UPDATE statements where there may be side effects or in recursive CTE calls. In addition, having the optimization fence means that the tables involved in the CTE are not counted against *join_collapse_limit*. Thus, we can effectively use PostgreSQL optimizer capabilities with queries that join a large number of tables.

PostgreSQL 12 brought a drastic change to CTE optimization. For SELECT statements with no recursion, if a CTE is used in a query only once, it will be inlined into the outer query (removing the optimization fence). If it is called more than once, the old behavior will be preserved.

What is more important, the behavior described earlier is a default, but it can be overwritten by using the keywords MATERIALIZED and NOT MATERIALIZED. The first one forces the old behavior, and the second one forces inlining, regardless of all other considerations.

We would still encourage SQL developers to be mindful not to force a suboptimal execution plan, but using a chain of CTEs is much better than using a sequence of temporary tables; in the latter case, the optimizer is helpless.

4.3 RECURSIVE CTES

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output.

The general form of a recursive WITH query is always a non-recursive term, then UNION (or UNION ALL), then a recursive term, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.

2. So long as the working table is not empty, repeat these steps:

    a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.

    b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

Recursive queries are typically used to deal with hierarchical or tree-structured data.

When computing a tree traversal using a recursive query, you might want to order the results in either depth-first or breadth-first order. This can be done by computing an ordering column alongside the other data columns and using that to sort the results at the end. Note that this does not actually control in which order the query evaluation visits the rows; that is as always in SQL implementation-dependent. This approach merely provides a convenient way to order the results afterwards.

When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using UNION instead of UNION ALL can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values.

4.4 DATA-MODIFYING CTES

You can use data-modifying statements (INSERT, UPDATE, or DELETE) in WITH. This allows you to perform several different operations in the same query.

Data-modifying statements in WITH usually have RETURNING clauses. It is the output of the RETURNING clause, not the target table of the data-modifying statement, that forms the temporary table that can be referred to by the rest of the query. If a data-modifying statement in WITH lacks a RETURNING clause, then it forms no temporary table and cannot be referred to in the rest of the query. Such a statement will be executed nonetheless.

Recursive self-references in data-modifying statements are not allowed. In some cases it is possible to work around this limitation by referring to the output of a recursive WITH

Data-modifying statements in WITH are executed exactly once, and always to completion, independently of whether the primary query reads all (or indeed any) of their output. Notice that this is different from the rule for SELECT in WITH: as stated in the previous section, execution of a SELECT is carried only as far as the primary query demands its output.

The sub-statements in WITH are executed concurrently with each other and with the main query. Therefore, when using data-modifying statements in WITH, the order in which the specified updates actually happen is unpredictable. All the statements are executed with the same snapshot, so they cannot "see" one another's effects on the target tables. This alleviates the effects of the unpredictability of the actual order of row updates, and means that RETURNING data is the only way to communicate changes between different WITH sub-statements and the main query.

Trying to update the same row twice in a single statement is not supported. Only one of the modifications takes place, but it is not easy (and sometimes not possible) to reliably predict which one. This also applies to deleting a row that was already updated in the same statement: only the update is performed. Therefore you should generally avoid trying to modify a single row twice in a single statement. In particular avoid writing WITH sub-statements that could affect the same rows changed by the main statement or a sibling sub-statement. The effects of such a statement will not be predictable.

At present, any table used as the target of a data-modifying statement in WITH must not have a conditional rule, nor an ALSO rule, nor an INSTEAD rule that expands to multiple statements.

5. SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries, Apress, 2021

3. Hans-Jürgen Schönig, Mastering PostgreSQL 12 Third Edition, Packt Publishing, 2019