EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

PostgreSQL Data Access and Query Optimizer

CONTENTS

1. THE PATH OF A QUERY

PostgreSQL query execution involves the following four key components:

• **Parser**: This performs syntax and semantic checking of the SQL string

• **Rewriter**: This changes the query is some cases; for example, if the query is against a view, the rewriter will modify the query so that it goes against the base tables instead of the view

• **Planner**: This key component comes up with a plan of execution

• **Executor**: This executes the plan generated by the planner

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and waits to receive the results sent back by the server.
2. The parser stage checks the query transmitted by the application program for correct syntax and creates a query tree.
3. The rewrite system takes the query tree created by the parser stage and looks for any rules (stored in the system catalogs) to apply to the query tree. It performs the transformations given in the rule bodies.
4. One application of the rewrite system is in the realization of views. Whenever a query against a view (i.e., a virtual table) is made, the rewrite system rewrites the user's query to a query that accesses the base tables given in the view definition instead.
5. The planner/optimizer takes the (rewritten) query tree and creates a query plan that will be the input to the executor.
6. It does so by first creating all possible paths leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each path is estimated and the cheapest path is chosen. The cheapest path is expanded into a complete plan that the executor can use.
7. The executor recursively steps through the plan tree and retrieves rows in the way represented by the plan. The executor makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the rows derived.

1.1 PLANNER/OPTIMIZER

The task of the *planner/optimizer* is to create an optimal execution plan. A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest.

The planner's search procedure actually works with data structures called *paths*, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged *plan tree* is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it.

The planner/optimizer starts by generating plans for scanning each individual relation (table) used in the query. The possible plans are determined by the available indexes on each relation. There is always the possibility of performing a sequential scan on a relation, so a sequential scan plan is always created.

If the query requires joining two or more relations, plans for joining relations are considered after all feasible plans have been found for scanning single relations.

When the query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs. The planner examines different possible join sequences to find the cheapest one.

The planner preferentially considers joins between any two relations for which there exist a corresponding join clause in the WHERE qualification (i.e., for which a restriction like where rel1.attr1=rel2.attr2 exists). Join pairs with no join clause are considered only when there is no other choice, that is, a particular relation has no available join clauses to any other relation. All possible plans are generated for every join pair considered by the planner, and the one that is (estimated to be) the cheapest is chosen.

The finished plan tree consists of **sequential** or **index scans** of the base relations, plus **nested-loop**, **merge**, or **hash join** nodes as needed, plus any auxiliary steps needed, such as sort nodes or aggregate-function calculation nodes. Most of these plan node types have the additional ability to do *selection* (discarding rows that do not meet a specified Boolean condition) and *projection* (computation of a derived column set based on given column values, that is, evaluation of scalar expressions where needed). One of the responsibilities of the planner is to attach selection conditions from the `WHERE` clause and computation of required output expressions to the most appropriate nodes of the plan tree.

1.2 STATISTICS

The query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. The system uses the statistics for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table *pg_class*, in the columns *reltuples* and *relpages*.

Statistics are collected for each column in every table in a database when ANALIZE is executed against the table. If you're running with autovaccum turned on, it will usually run often enough to keep accurate statistics available. Unlike a complete vacuum cleanup, which can take quite some time on large tables, analyzing a table should take only a few seconds at any table size. It doesn't take anything other than a read lock while running, either.

The statistics information collected for each table is easiest to see using the *pg_stats* view.

Most queries retrieve only a fraction of the rows in a table, due to WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the selectivity of WHERE clauses, that is, the fraction of rows that match each condition in the WHERE clause.

1.3 CACHING

If you execute a complex query that takes a while to run, subsequent runs are often much faster. Thank caching. If the same query executes in sequence, by the same user or different users, and no changes have been made to the underlying data, you should get back the same result. As long as there’s space in memory to cache the data, the planner can skip replanning or re-retrieving. Using common table expressions and immutable functions in your queries encourages caching.

The second time you run the query, you should notice at least a 10% performance speed increase. This represents hot cache behavior, meaning that the data needed for the query was already in either the database or OS caches. It was left behind in the cache from when the data was loaded in the first place. Whether your cache is hot, or cold (not in the cache), is another thing to be very careful of. If you run a query twice with two different approaches, the second will likely be much faster simply because of caching, regardless of whether the plan was better or worse.

The more onboard memory you have dedicated to the cache, the more room you'll have to cache data. You can set the amount of dedicated memory by changing the shared_buffers setting in postgresql.conf. Don't go overboard; raising shared_buffers too much will bloat your cache, leading to more time wasted scanning the cache.

2. ACCESS PATHS

Now, we know that the planner has quite a few decisions to make and that these decisions will ultimately decide the execution time as well as consumption of resources (CPU, memory, and disk I/O). How do we know what decision the planner will take? We can use the EXPLAIN command to find out the possible (note that it's possible) execution plan. To use EXPLAIN, we have to just prefix the EXPLAIN word to the SQL statement we want to execute

This command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in cost units that are arbitrary, but conventionally mean disk page fetches).

The ANALYZE option causes the statement to be actually executed, not only planned. Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality.

The VERBOSE option display additional information regarding the plan. Specifically, include the output column list for each node in the plan tree, schema-qualify table and function names, always label variables in expressions with their range table alias, and always print the name of each trigger for which statistics are displayed. This parameter defaults to FALSE.

In order to allow the PostgreSQL query planner to make reasonably informed decisions when optimizing queries, the pg_statistic data should be up-to-date for all tables used in the query. Normally the autovacuum daemon will take care of that automatically. But if a table has recently had substantial changes in its contents, you might need to do a manual ANALYZE rather than wait for autovacuum to catch up with the changes.

EXPLAIN can be also used for the UPDATE, DELETE, and INSERT statements. To avoid data changes when you do this, use the following command:

BEGIN;

EXPLAIN ANALYZE ...;

ROLLBACK;

The query plan should be read from most indented to least intended: from bottom to top. Each line with cost/rows/width is a node. The inner (child) nodes feed the outer (parent) nodes. In the preceding plan, there are no child nodes. Let's look at each word/number and see what it means.

A crucial decision the optimizer has to take is how to get the data that the user wants. Should it:

• Fetch the data from a table, or

• Go to an index, and

• Stop here because all the columns required are there in the index, or

• Get the location of the records and go to the table to get the data The first method, fetching data directly from table, is called Sequential Scan (Seq Scan). This works best for small tables. In the second set of options, if there is no need to access the table, we have an Index Only Scan.

2.1 SEQUENTIAL SCAN ACCESS METHOD

The Seq Scan operation scans the entire relation (table) as stored on disk.

You can expect a Seq Scan when there isn't a useful index, or when such a large portion of the table is expected to be returned so that using an index would just add needless overhead. They'll also be used when there is only a very small amount of data to access; the index overhead is disproportionately large if the table takes up only a few pages on disk.

Note that a Seq Scan must read through all the dead rows in a table but will not include them in its output. It's therefore possible for their execution to take much longer than would be expected to produce all the required output if the table has been badly maintained and is quite bloated with dead rows.

So a sequential scan requires a single IO for each row - or even less because a page on the disk contains more than one row, so more than one row can be fetched with a single IO operation.

2.2 INDEX SCAN ACCESS METHODS

Unlike Sequential Scan, Index scan does not fetch all records sequentially. Rather it uses different data structure (depending on the type of index) corresponding to the index involved in the query and locate required data (as per predicate) clause with very minimal scans. Then the entry found using the index scan points directly to data in heap area, which is then fetched to check visibility as per the isolation level. So there are two steps for index scan:

1. Fetch data from index related data structure. It returns the TID of corresponding data in heap.
2. Then the corresponding heap page is directly accessed to get whole data. This additional step is required for the below reasons:
    a. Query might have requested to fetch columns more than whatever available in the corresponding index.
    b. Visibility information is not maintained along with index data. So in order to check the visibility of data as per isolation level, it needs to access heap data. (rows should be visible to the current transaction which means rows haven’t been deleted or replaced by a newer version)

An index scan requires several IO operations for each row (look up the row in the index, then retrieve the row from the heap).

An Index Scan is what you want if your query needs to return a value fast. If an index that is useful to satisfy a selective WHERE condition exists.

Regular index scans are the only type that will return rows that are already sorted. This makes them preferred by queries using a LIMIT, where the sort cost may otherwise be proportional to the full table size. There are also some upper-node types that need input in sorted order.

Index Only Scan is similar to Index Scan except for the second step i.e. as the name implies it only scans index data structure. There are two additional pre-condition in order to choose Index Only Scan compare to Index Scan:

1. Query should be fetching only key columns which are part of the index.
2. All records on the selected heap page should be visible. Index data structure does not maintain visibility information so in order to select data only from index we should avoid checking for visibility and this could happen if all data of that page are considered visible.

Note: Not every index type in Postgres supports index-only scans. Some, for example GIN indexes, don’t store all the underlying data that they index, so retrieving row data is impossible.

2.3 BITMAP SCAN ACCESS METHOD

Bitmap scan is a mix of Index Scan and Sequential Scan. It tries to solve the disadvantage of Index scan but still keeps its full advantage. For each data found in the index data structure, it needs to find corresponding data in heap page. So alternatively, it needs to fetch index page once and then followed by heap page, which causes a lot of I/O. So, bitmap scan method leverage the benefit of index scan without I/O.

Bitmap scans are a multi-step process that consist of a Bitmap Heap Scan, one or more Bitmap Index Scans.

Bitmap index scans are executed by reading the index first, create bitmap – one bit for one page of the table and populating the bitmap by 1 where needed row are placed.

The index bitmaps built by the scans can be combined with standard bit operations, including either AND (return rows that are on both lists) or OR (return rows that are on either list) when appropriate.

Bitmap Index Scan can be used instead of mutli-column index.

Bitmap Heap Scan uses a bitmap to get the heap tuple. When the number of index rows is small, it points directly to the heap tuple. If the number of rows to keep in the bitmap increases, it will point to the page that contains the row.

Bitmap Index Scans combines both cases: when you need many rows from the table, but not all of them, and when the rows that you will return are not in the same page (distributed).

3. ADDING DATA WITH INSERT AND COPY

Once you have created your table with the necessary specifications, the next logical step is to fill the table with data. There are generally three methods in PostgreSQL with which you can fill a table with data:

- Use the INSERT INTO command with a grouped set of data to insert new values.
- Use the INSERT INTO command in conjunction with a SELECT statement to insert existing values from another table.
- Use the COPY command to insert values from a system file.

3.1 INSERT AND COPY

INSERT creates new rows in a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query. In the case of INSERT, every statement has to check for locks, check for the existence of the table and the columns in the table, check for permissions, look up data types and so on.

Also, by default, Server will add new data into the first available free space in a table (vacated by vacuumed records).

A useful technique within PostgreSQL is to use the COPY command to insert values directly into tables from external files. Files used for input by COPY must either be in standard ASCII text format, whose fields are delimited by a uniform symbol, or in PostgreSQL's binary table format. Common delimiters for ASCII files are tabs and commas. When using an ASCII formatted input file with COPY, each line within the file will be treated as a row of data to be inserted and each delimited field will be treated as a column value.

The COPY FROM command operates much faster than a normal INSERT command because the data is read as a single transaction directly to the target table. On the other hand, it is a very strict format, and the entire COPY procedure will fail if just one line is malformed.

In the case of COPY, this is only done once, which is a lot faster. Whenever you want to write large amounts of data, data COPY is usually the way to go.

COPY is usually a lot better than plain inserts. The reason is that INSERT has a lot of overhead.

3.2 UPSERT

INSERT statement has the optional ON CONFLICT clause which specifies an alternative action to raising a unique violation or exclusion constraint violation error. For each individual row proposed for insertion, either the insertion proceeds, or, if an arbiter constraint or index specified by *conflict_target* is violated, the alternative *conflict_action* is taken.

ON CONFLICT DO NOTHING simply avoids inserting a row as its alternative action. ON CONFLICT DO UPDATE updates the existing row that conflicts with the row proposed for insertion as its alternative action.

conflict_target can perform unique index inference. When performing inference, it consists of one or more index_column_name columns and/or index_expression expressions, and an optional index_predicate. All table_name unique indexes that, without regard to order, contain exactly the conflict_target-specified columns/expressions are inferred (chosen) as arbiter indexes. If an index_predicate is specified, it must, as a further requirement for inference, satisfy arbiter indexes. Note that this means a non-partial unique index (a unique index without a predicate) will be inferred (and thus used by ON CONFLICT) if such an index satisfying every other criteria is available. If an attempt at inference is unsuccessful, an error is raised.

ON CONFLICT DO UPDATE guarantees an atomic INSERT or UPDATE outcome; provided there is no independent error, one of those two outcomes is guaranteed, even under high concurrency. This is also known as UPSERT — “UPDATE or INSERT”.

This feature is useful if you don’t know a record already exists in a table and rather than having the insert fail, you want it to either update the existing record or do nothing.

This feature requires a unique key, primary key, unique index, or exclusion constraint in place, that when violated, you’d want different behavior like updating the existing record or not doing anything.

4. SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Jayadevan Maymala, PostgreSQL for Data Architects, Packt Publishing, 2015
3. PostgreSQL 10 High PerformanceIbrar Ahmed, Gregory Smith, Enrico Pirozzi, 2018
4. Practical PostgreSQL by Joshua D. Drake, John C. Worsley