PostgreSQL and Query Processing

Srinivas Maddali

In all the RDBMS, the query processing is standardized. They are to follow the standards documented.

1. ANSI/ISO/IEC 9075:2003, "Database Language SQL", Parts 1 ("SQL/Framework"), 2 ("SQL/Foundation"), 3 ("SQL/CLI"), 4 ("SQL/PSM"), 9 ("SQL/MED"), 10 ("SQL/OLB"), 11("SQL/Schemata"), and 13 ("SQL/JRT")
2. ISO/IEC 9075:2003, "Database Language SQL", Parts 1 ("SQL/Framework"), 2 ("SQL/Foundation"), 3 ("SQL/CLI"), 4 ("SQL/PSM"), 9 ("SQL/MED"), 10 ("SQL/OLB"), 11("SQL/Schemata"), and 13 ("SQL/JRT")

XML standards are defined in

1. ANSI/ISO/IEC 9075-14:2006, "Database Language SQL", Part 14 ("SQL/XML")
2. ISO/IEC 9075-14:2006, "Database Language SQL", Part 14 ("SQL/XML")

Against every RDBMS the SQL statement issued belong to

1. Querying data - DQL
2. Inserting, updating, and deleting rows in a table – DML
3. Creating, replacing, altering, and dropping objects – DDL
4. Controlling access to the database and its objects – DCL
5. Guaranteeing database consistency and integrity - Version Management

PostgreSQL is NOT an exception to it as it also uses those standards. PostgreSQL conforms to SQL standard is ISO/IEC 9075 "Database Language SQL". A revised version of the standard is released from time to time; the most recent update appearing in 2016. The 2016 version is referred to as ISO/IEC 9075:2016, or simply as SQL:2016. The versions prior to that were SQL:2011, SQL:2008, SQL:2006, SQL:2003, SQL:1999, and SQL-92. Each version replaces the previous one, so claims of conformance to earlier versions have no official merit. PostgreSQL development aims for conformance with the latest official version of the standard where such conformance does not contradict traditional features or common sense. Many of the features required by the SQL standard are supported, though sometimes with slightly differing syntax or function. Further moves towards conformance can be expected over time. A detailed discussion is made in or rather documented in the below web page -
https://www.postgresql.org/docs/current/features.html

At the high level any query is processed in the below stages

1. Parser – submitted query (statement) is to be parsed. PostgreSQL does the same. Oracle saves the cursors – parsed and hence has soft parses where the stored cursor is reused and that saves parsing time.
2. Analyzing – this often is to get the data by analyzing the query which
   a. What are the listable items in case it is DQL (data querying language)
   b. Read the schema (bet practice is to prefix the table name with the owner so that the schema is scanned for that table name.
   c. If an alias is used for the table name, the same is used in the predicate and filters that reduces time instead of full table name (if if the time we save is a fraction of second, for a query intensive database, the fractions accumulate into minutes and hours).

d. If Order by is there using temporary memory the data is sorted as requested by the client.

3. Rewrite – this is based on:
   a. Rule system of PostgreSQL which is implemented:
      i. The first one worked using row level processing and was implemented deep in the executor. The rule system was called whenever an individual row had been accessed. This implementation was removed in 1995 when the last official release of the Berkeley Postgres project was transformed into Postgres95.
      ii. The second implementation of the rule system is a technique called query rewriting. The rewrite system is a module that exists between the parser stage and the planner/optimizer. This technique is still implemented.

      (read further - https://www.postgresql.org/docs/9.1/rule-system.html)

   b. The views are:
      i. Read only views.
      ii. Updatable views.
   c. The updatable views should fit into one of the below conditions:
      i. The view must have exactly one entry in its FROM list, which must be a table or another updatable view.
      ii. The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level.
      iii. The view definition must not contain set operations (UNION, INTERSECT or EXCEPT) at the top level.
      iv. The view's select list must not contain any aggregates, window functions or set-returning functions.

      (for more information please read https://www.postgresql.org/docs/current/sql-createview.html#SQL-CREATEVIEW-UPDATABLE-VIEWS)

   d. Planning and execution

      Every Plan of Execution is to have a tree which may be called as nodes. The nodes are fit into a flow diagram for the execution.
      i. FTS
      ii. Sequential Scan
      iii. Index Scan

      and various other methods of data access. When you have more than one table and they are joined (there is no cartesian join which is based on no predicate)

      a. Equijoin – in the predicate you seen a '=' join
      b. Right outer join (in tab 1 and tab 2) – that do not have any match in tab 2 also are to be listed along with the matched.
      c. Left outer join (in tab 1 and tab 2) – that do not have any match in tab 1 also are to be listed along with the matched.
      d. Intersection
      e. Union
      f. Minus
      g. Union All (which can have duplicates)
   e. Cost estimation of the plan for execution:
      a. For the estimation of cost, the stats are to be collected on the object/s else the default way would be implemented by any and every RDBMS. In PostgreSQL after every vacuum the stats are collected on the table because the version

management system in PostgreSQL is different from other RDBMS. Also the user has the capacity to collect the stats using "analyze" command.

b. The estimated cost is computed as (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost). By default, seq_page_cost is 1.0 and cpu_tuple_cost is 0.01 in PostgreSQL.
c. The cost estimation is based on multiple aspects in PostgreSQL.
    i. Startup cost – which is the cost of accessing the object required to fetch the data.
    ii. Run cost which includes:
        a. Sequential scan
        b. Index scan

            Types of Indexes and purposes

            1. Hash- Based on their structures and on metadata they can be categorized uder:
                a. Metadata of a page (like header data block called in Oracle) which contains info about that index and data stored.
                b. Bucket page (which contains data info of hash code – TID of the data block)
                c. Overflow Page – used to store overflow data of a bucket page.
                d. keep track of overflow pages that are currently clear and can be reused for other buckets.

    Read https://www.postgresql.org/docs/10/internals.html

    iii. SQL statement tree includes:
        a. For a SELECT statement, the following:
            1. DATA required in the select.
            2. TABLE/S in the "FROM" clause.
            3. PREDICATE which joins the tables if the data is from more than 1 table.
            4. RANGE and/or FILTER
        b. SELECT FOR UPDATE – used in the cursor to maintain the consistency of the data in the cursor. This
        c. INSERT statement:
            1. Identification of the SCHEMA
            2. TABLE identification
            3. COLUMN/S and their data types
            4. NULL ability of the columns
            5. PRECISION of the data where numbers are to be loaded with data precisions.
            6. LENGTH of the strings and/or NUMBERS wherever LENGTH is defined, and numbers are defined.
        d. UPDATE (costliest DML statement as a delete and an insert are there) statement:
            1. Identification of the SCHEMA
            2. COLUMN to be UPDATED.
            3. TABLE/S identification wherever there more than one table are referenced.
            4. PREDICATE that joins more than one table
            5. FILTER statement.
        e. DELETE statement:
            1. SCHEMA
            6. TABLE/s identification wherever there more than one tables are referenced.

2. PREDICATE that joins more than one table.
3. FILTER statement wherever that is required.

iv. DDL statement:
   a. CREATE table as select (CTAS):
      1. The source table FULL SCAN
      2. LOAD as SELECT.
   b. CREATE table:
      1. Create table statement.
   c. CREATE index:
      1. Index build non-unique.
      2. SORT of data
      3. FULL Table Access.
   d. ALTER command.
   e. DROP command.
   f. ANALYZE command.

v. DCL statements – data control language
   a. GRANT
   b. REVOKE

vi. TCL statements – transaction control language
   a. COMMIT – commit the transaction changes
   b. ROLLBACK – ROLLBACK the transaction changes
   c. SAVEPOINT – the SAVEPOINT statement to create a name for a system change number (SCN), to which roll back can tak3e place if needed during that transaction.

This is common in management by any RDBMS the locks, lathes, enqueues, something else. These days as a standard "performance schema" with all tables/views the Dynamic Performance Information and Data Dictionary Information is captured and shared with users.

Query processing is specialized in three technical features in PostgreSQL:

1. Foreign Data Wrappers (FDW) -
2. Parallel Query
3. JIT

FDW – Introduced in 9.1 Version

The PostgreSQL fdw module provides the foreign-data wrapper postgres_fdw, which can be used to access data stored in external PostgreSQL servers.

The functionality provided by this module overlaps substantially with the functionality of the older **dblink** module. But PostgreSQL fdw provides more transparent and standards-compliant syntax for accessing remote tables and can give better performance in many cases.

To prepare for remote access using PostgreSQL fdw:

Install the PostgreSQL-fdw extension using CREATE EXTENSION.

a. Create a foreign server object, using CREATE SERVER, to represent each remote database you want to connect to. Specify connection information, except user and password, as options of the server object.

b. Create a user mapping, using CREATE USER MAPPING, for each database user you want to allow to access each foreign server. Specify the remote user name and password to use as user and password options of the user mapping.

c. Create a foreign table, using CREATE FOREIGN TABLE or IMPORT FOREIGN SCHEMA, for each remote table you want to access. The columns of the foreign table must match the referenced remote table. You can, however, use table and/or column names different from the remote table's, if you specify the correct remote names as options of the foreign table object.

Now you need only SELECT from a foreign table to access the data stored in its underlying remote table. You can also modify the remote table using INSERT, UPDATE, or DELETE. (Of course, the remote user you have specified in your user mapping must have privileges to do these things.)

Note that PostgreSQL_fdw currently lacks support for INSERT statements with an ON CONFLICT DO UPDATE clause. However, the ON CONFLICT DO NOTHING clause is supported, provided a unique index inference specification is omitted.

It is generally recommended that the columns of a foreign table be declared with exactly the same data types, and collations if applicable, as the referenced columns of the remote table. Although PostgreSQL fdw is currently rather forgiving about performing data type conversions at need, surprising semantic anomalies may arise when types or collations do not match, due to the remote server interpreting WHERE clauses slightly differently from the local server.

Note that a foreign table can be declared with fewer columns, or with a different column order, than its underlying remote table has. Matching of columns to the remote table is by name, not position.

Parallel Query Processing in PostgreSQL (read for more info - https://www.postgresql.org/docs/11/when-can-parallel-query-be-used.html)

There is many a setting before the parallel processing is initialized.

a. max_parallel_workers_per_gather
b. dynamic_shared_memory_type

There are several settings which can cause the query planner not to generate a parallel query plan under any circumstances. In order for any parallel query plans whatsoever to be generated, the following settings must be configured as indicated.

max_parallel_workers_per_gather must be set to a value which is greater than zero. This is a special case of the more general principle that no more workers should be used than the number configured via max_parallel_workers_per_gather.

dynamic_shared_memory_type must be set to a value other than none. Parallel query requires dynamic shared memory in order to pass data between cooperating processes. (You may remember DISM of Oracle)

In addition, the system must not be running in single-user mode. Since the entire database system is running in single process in this situation, no background workers will be available.

Even when it is in general possible for parallel query plans to be generated, the planner will not generate them for a given query if any of the following are true:

The query writes any data or locks any database rows. If a query contains a data-modifying operation either at the top level or within a CTE, no parallel plans for that query will be generated. As an exception, the commands CREATE TABLE ... AS, SELECT INTO, and CREATE MATERIALIZED VIEW which create a new table and populate it can use a parallel plan.

The query might be suspended during execution. In any situation in which the system thinks that partial or incremental execution might occur, no parallel plan is generated. For example, a cursor created using DECLARE CURSOR will never use a parallel plan. Similarly, a PL/pgSQL loop

of the form FOR x IN query LOOP .. END LOOP will never use a parallel plan, because the parallel query system is unable to verify that the code in the loop is safe to execute while parallel query is active.

The query uses any function marked PARALLEL UNSAFE. Most system-defined functions are PARALLEL SAFE, but user-defined functions are marked PARALLEL UNSAFE by default.

The query is running inside of another query that is already parallel. For example, if a function called by a parallel query issues an SQL query itself, that query will never use a parallel plan. This is a limitation of the current implementation, but it may not be desirable to remove this limitation, since it could result in a single query using a very large number of processes.

The transaction isolation level is serializable. This is a limitation of the current implementation.

Even when parallel query plan is generated for a particular query, there are several circumstances under which it will be impossible to execute that plan in parallel at execution time. If this occurs, the leader will execute the portion of the plan below the Gather node entirely by itself, almost as if the Gather node were not present. This will happen if any of the following conditions are met:

No background workers can be obtained because of the limitation that the total number of background workers cannot exceed max_worker_processes.

No background workers can be obtained because of the limitation that the total number of background workers launched for purposes of parallel query cannot exceed max_parallel_workers.

The client sends an Execute message with a non-zero fetch count. See the discussion of the extended query protocol. Since libpq currently provides no way to send such a message, this can only occur when using a client that does not rely on libpq. If this is a frequent occurrence, it may be a good idea to set max_parallel_workers_per_gather to zero in sessions where it is likely, so as to avoid generating query plans that may be suboptimal when run serially.

The transaction isolation level is serializable. This situation does not normally arise, because parallel query plans are not generated when the transaction isolation level is serializable. However, it can happen if the transaction isolation level is changed to serializable after the plan is generated and before it is executed.

JIT (read for more info - https://www.investopedia.com/terms/j/jit.asp and https://llvm.org/devmtg/2016-09/slides/Melnik-PostgreSQLLLVM.pdf and https://www.postgresql.org/docs/12/jit-extensibility.html#JIT-PLUGGABLE

Just-in-Time (JIT) compilation is the process of turning some form of interpreted program evaluation into a native program, and doing so at run time. For example, instead of using general-purpose code that can evaluate arbitrary SQL expressions to evaluate a particular SQL predicate like WHERE a.col = 3, it is possible to generate a function that is specific to that expression and can be natively executed by the CPU, yielding a speedup. PostgreSQL has built in support to perform JIT compilation using LLVM when PostgreSQL is built with --with-llvm.

See src/backend/jit/README for further details.

JIT Accelerated Operations (source: PostgreSQL Docs)

Currently PostgreSQL's JIT implementation has support for accelerating expression evaluation and tuple deforming. Several other operations could be accelerated in the future.

Expression evaluation is used to evaluate WHERE clauses, target lists, aggregates and projections. It can be accelerated by generating code specific to each case.

Tuple deforming is the process of transforming an on-disk tuple (see Section 68.6.1) into its in-memory representation. It can be accelerated by creating a function specific to the table layout and the number of columns to be extracted.