EPAM Systems, RD Dep., RD Dep.

POSTGRESQL DB FOR DWH AND ETL BUILDING

Partitioning and Parallel Execution

CONTENTS

1. PARTITIONING

Given a default 8,000 blocks, PostgreSQL can store up to 32 TB of data inside a single table. If you compile PostgreSQL with 32,000 blocks, you can even put up to 128 TB into a single table. However, large tables such as this aren't necessarily convenient anymore, and it can make sense to partition tables to make processing easier and, in some cases, a bit faster. Starting with version 10.0, PostgreSQL offers improved partitioning, which will offer end users significantly easier handling of data partitioning.

Partition is a different sort of division—dividing the data. A partitioned table consists of several partitions, each of which is defined as a table. Each table row is stored in one of the partitions according to rules specified when the partitioned table is created.

Partition support is relatively new in PostgreSQL, and beginning with PG 10, improvements are made in every release, making partitioned tables easier to use. The most common case is range partitioning, meaning that each partition contains rows that have values of an attribute in the range assigned to the partition. Ranges assigned to different partitions cannot intersect, and a row that does not fit into any partition cannot be inserted.

Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions.
- When queries or updates access a large percentage of a single partition, performance can be improved by using a sequential scan of that partition instead of using an index, which would require random-access reads scattered across the whole table.
- Bulk loads and deletes can be accomplished by adding or removing partitions, if the usage pattern is accounted for in the partitioning design. Dropping an individual partition using `DROP TABLE`, or doing `ALTER TABLE DETACH PARTITION`, is far faster than a bulk operation. These commands also entirely avoid the `VACUUM` overhead caused by a bulk `DELETE`.
- Seldom-used data can be migrated to cheaper and slower storage media.

These benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.

How should a partitioning key for a table be selected? Based on the preceding observation, the partitioning key should be chosen so that it is used by the search conditions in either a large enough number of queries or in the most critical queries.

Partitions may have their own indexes that obviously are smaller than an index on the whole partitioned table. This option might be beneficial for short queries. However, this might significantly improve performance only if almost all queries extract data from the same partition. The cost of search in a B-tree is proportional to its depth. An index on a partition, most likely, will eliminate only one level of the B-tree, while the choice of needed partition also requires some amount of resources. These resources are likely comparable with the amount needed for an extra index level. Of course, a query may refer to a partition instead of the whole partitioned table, hiding the cost of choosing the partition to the application issuing the query.

1.1 DECLARATIVE PARTITIONING

PostgreSQL allows you to declare that a table is divided into partitions. The table that is divided is referred to as a partitioned table. The declaration includes the partitioning method, plus a list of columns or expressions to be used as the partition key.

The partitioned table itself is a “virtual” table having no storage of its own. Instead, the storage belongs to partitions, which are otherwise-ordinary tables associated with the partitioned table. Each partition stores a subset of the data as defined by its partition bounds. All rows inserted into a partitioned table will be routed to the appropriate one of the partitions based on the values of the partition key column(s). Updating the partition key of a row will cause it to be moved into a different partition if it no longer satisfies the partition bounds of its original partition.

Partitions may themselves be defined as partitioned tables, resulting in sub-partitioning. Although all partitions must have the same columns as their partitioned parent, partitions may have their own indexes, constraints and default values, distinct from those of other partitions.

It is not possible to turn a regular table into a partitioned table or vice versa. However, it is possible to add an existing regular or partitioned table as a partition of a partitioned table, or remove a partition from a partitioned table turning it into a standalone table; this can simplify and speed up many maintenance processes.

1.1.1 PARTITION METHODS

PostgreSQL offers built-in support for the following forms of partitioning:

- **List Partitioning**: The table is partitioned by explicitly listing which key value(s) appear in each partition.

  Example:

```
CREATE TABLE part_table (id INTEGER, val TEXT, num NUMERIC) PARTITION BY LIST(val);
CREATE TABLE part_table_a PARTITION OF part_table FOR VALUES IN ('A');
CREATE TABLE part_table_b PARTITION OF part_table FOR VALUES IN ('B');
CREATE TABLE part_table_def PARTITION OF part_table DEFAULT;
```

- **Range Partitioning**: The table is partitioned into “ranges” defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example, one might partition by date ranges, or by ranges of identifiers for particular business objects. Each range's bounds are understood as being inclusive at the lower end and exclusive at the upper end. For example, if one partition's range is from 1 to 100, and the next one's range is from 100 to 1000, then value 100 belongs to the second partition not the first.

  Example:

```
CREATE TABLE part_table (id INTEGER, val TEXT, num NUMERIC) PARTITION BY RANGE(num);
CREATE TABLE part_table_s PARTITION OF part_table FOR VALUES FROM (MINVALUE) TO (100);
CREATE TABLE part_table_l PARTITION OF part_table FOR VALUES FROM (100) TO (1000);
CREATE TABLE part_table_xl PARTITION OF part_table FOR VALUES FROM (1000) TO (MAXVALUE);
```

- **Hash Partitioning**: The table is partitioned by specifying a modulus and a remainder for each partition. Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.

  Example:

```
CREATE TABLE part_table (id INTEGER, val TEXT, num NUMERIC) PARTITION BY HASH(id);
CREATE TABLE part_table_p1 PARTITION OF part_table FOR VALUES WITH (MODULUS 3, REMAINDER 0);
CREATE TABLE part_table_p2 PARTITION OF part_table FOR VALUES WITH (MODULUS 3, REMAINDER 1);
CREATE TABLE part_table_p3 PARTITION OF part_table FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

If your application needs to use other forms of partitioning not listed above, alternative methods such as inheritance and UNION ALL views can be used instead. Such methods offer flexibility but do not have some of the performance benefits of built-in declarative partitioning.

1.1.2 PARTITION MAINTENANCE

Normally the set of partitions established when initially defining the table is not intended to remain static. It is common to want to remove partitions holding old data and periodically add new partitions for new data.

One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is to drop the partition that is no longer necessary by using `DROP TABLE` command.

This can very quickly delete millions of records because it doesn't have to individually delete every record. Note however that the above command requires taking an `ACCESS EXCLUSIVE` lock on the parent table.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right by using `ALTER TABLE ... DETACH PARTITION...`

This allows further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using `COPY`, pg_dump, or similar tools. It might also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports.

It is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later (`ATTACH PARTITION` command). This allows new data to be loaded, checked, and transformed prior to it appearing in the partitioned table.

The `ATTACH PARTITION` command requires taking a `SHARE UPDATE EXCLUSIVE` lock on the partitioned table.

Before running the `ATTACH PARTITION` command, it is recommended to create a `CHECK` constraint on the table to be attached that matches the expected partition constraint. That way, the system will be able to skip the scan which is otherwise needed to validate the implicit partition constraint. Without the `CHECK` constraint, the table will be scanned to validate the partition constraint while holding an `ACCESS EXCLUSIVE` lock on that partition. It is recommended to drop the now-redundant `CHECK` constraint after the `ATTACH PARTITION` is complete. If the table being attached is itself a partitioned table then each of its sub-partitions will be recursively locked and scanned until either a suitable `CHECK` constraint is encountered or the leaf partitions are reached.

Similarly, if the partitioned table has a `DEFAULT` partition, it is recommended to create a `CHECK` constraint which excludes the to-be-attached partition's constraint. If this is not done then the `DEFAULT` partition will be scanned to verify that it contains no records which should be located in the partition being attached. This operation will be performed whilst holding an `ACCESS EXCLUSIVE` lock on the `DEFAULT` partition. If the `DEFAULT` partition is itself a partitioned table then each of its partitions will be recursively checked in the same way as the table being attached.

It is possible to create indexes on partitioned tables so that they are applied automatically to the entire hierarchy. This is very convenient, as not only will the existing partitions become indexed, but also any partitions that are created in the future will.

One limitation is that it's not possible to use the `CONCURRENTLY` qualifier when creating such a partitioned index. To avoid long lock times, it is possible to use `CREATE INDEX ON ONLY` the partitioned table; such an index is marked invalid, and the partitions do not get the index applied automatically.

The indexes on partitions can be created individually using `CONCURRENTLY`, and then attached to the index on the parent using `ALTER INDEX .. ATTACH PARTITION`. Once indexes for all partitions are attached to the parent index, the parent index is marked valid automatically.

1.1.3 LIMITATIONS

The following limitations apply to partitioned tables:

- Unique constraints (and hence primary keys) on partitioned tables must include all the partition key columns. This limitation exists because the individual indexes making up the constraint can only directly enforce uniqueness within their own partitions; therefore, the partition structure itself must guarantee that there are not duplicates in different partitions.
- There is no way to create an exclusion constraint spanning the whole partitioned table. It is only possible to put such a constraint on each leaf partition individually. Again, this limitation stems from not being able to enforce cross-partition restrictions.
- `BEFORE ROW` triggers on `INSERT` cannot change which partition is the final destination for a new row.
- Mixing temporary and permanent relations in the same partition tree is not allowed. Hence, if the partitioned table is permanent, so must be its partitions and likewise if the partitioned table is temporary. When using temporary relations, all members of the partition tree have to be from the same session.

Individual partitions are linked to their partitioned table using inheritance behind-the-scenes. However, it is not possible to use all of the generic features of inheritance with declaratively partitioned tables or their partitions.

Notably, a partition cannot have any parents other than the partitioned table it is a partition of, nor can a table inherit from both a partitioned table and a regular table. That means partitioned tables and their partitions never share an inheritance hierarchy with regular tables.

Since a partition hierarchy consisting of the partitioned table and its partitions is still an inheritance hierarchy, tableoid and all the normal rules of inheritance apply, with a few exceptions:

- Partitions cannot have columns that are not present in the parent. It is not possible to specify columns when creating partitions with `CREATE TABLE`, nor is it possible to add columns to partitions after-the-fact using `ALTER TABLE`. Tables may be added as a partition with `ALTER TABLE ... ATTACH PARTITION` only if their columns exactly match the parent.
- Both `CHECK` and `NOT NULL` constraints of a partitioned table are always inherited by all its partitions. `CHECK` constraints that are marked `NO INHERIT` are not allowed to be created on partitioned tables. You cannot drop a `NOT NULL` constraint on a partition's column if the same constraint is present in the parent table.
- Using `ONLY` to add or drop a constraint on only the partitioned table is supported as long as there are no partitions. Once partitions exist, using `ONLY` will result in an error. Instead, constraints on the partitions themselves can be added and (if they are not present in the parent table) dropped.
- As a partitioned table does not have any data itself, attempts to use `TRUNCATE ONLY` on a partitioned table will always return an error.

1.1.4 BEST PRACTICES FOR DECLARATIVE PARTITIONING

The choice of how to partition a table should be made carefully, as the performance of query planning and execution can be negatively affected by poor design.

One of the most critical design decisions will be the column or columns by which you partition your data. Often the best choice will be to partition by the column or set of columns which most commonly appear in `WHERE` clauses of queries being executed on the partitioned table. `WHERE` clauses that are compatible with the partition bound constraints can be used to prune unneeded partitions. However, you may be forced into making other decisions by requirements for the `PRIMARY KEY` or a `UNIQUE` constraint.

Removal of unwanted data is also a factor to consider when planning your partitioning strategy. An entire partition can be detached fairly quickly, so it may be beneficial to design the partition strategy in such a way that all data to be removed at once is located in a single partition.

Choosing the target number of partitions that the table should be divided into is also a critical decision to make. Not having enough partitions may mean that indexes remain too large and that data locality remains poor which could result in low cache hit ratios.

However, dividing the table into too many partitions can also cause issues. Too many partitions can mean longer query planning times and higher memory consumption during both query planning and execution, as further described below.

When choosing how to partition your table, it's also important to consider what changes may occur in the future. For example, if you choose to have one partition per customer and you currently have a small number of large customers, consider the implications if in several years you instead find yourself with a large number of small customers. In this case, it may be better to choose to partition by `HASH` and choose a reasonable number of partitions rather than trying to partition by `LIST` and hoping that the number of customers does not increase beyond what it is practical to partition the data by.

Sub-partitioning can be useful to further divide partitions that are expected to become larger than other partitions. Another option is to use range partitioning with multiple columns in the partition key. Either of these can easily lead to excessive numbers of partitions, so restraint is advisable.

It is important to consider the overhead of partitioning during query planning and execution. The query planner is generally able to handle partition hierarchies with up to a few thousand partitions fairly well, provided that typical queries allow the query planner to prune all but a small number of partitions. Planning times become longer and memory consumption becomes higher when more partitions remain after the planner performs partition pruning. This is particularly true for the `UPDATE` and `DELETE` commands.

Another reason to be concerned about having a large number of partitions is that the server's memory consumption may grow significantly over time, especially if many sessions touch large numbers of partitions. That's because each partition requires its metadata to be loaded into the local memory of each session that touches it.

With data warehouse type workloads, it can make sense to use a larger number of partitions than with an OLTP type workload. Generally, in data warehouses, query planning time is less of a concern as the majority of processing time is spent during query execution.

With either of these two types of workload, it is important to make the right decisions early, as re-partitioning large quantities of data can be painfully slow. Simulations of the intended workload are often beneficial for optimizing the partitioning strategy. Never just assume that more partitions are better than fewer partitions, nor vice-versa.

1.2 PARTITIONING USING INHERITANCE

While the built-in declarative partitioning is suitable for most common use cases, there are some circumstances where a more flexible approach may be useful. Partitioning can be implemented using table inheritance, which allows for several features not supported by declarative partitioning, such as:

- For declarative partitioning, partitions must have exactly the same set of columns as the partitioned table, whereas with table inheritance, child tables may have extra columns not present in the parent.
- Table inheritance allows for multiple inheritance.
- Declarative partitioning only supports range, list and hash partitioning, whereas table inheritance allows data to be divided in a manner of the user's choosing. (Note, however, that if constraint exclusion is unable to prune child tables effectively, query performance might be poor.)
- Some operations require a stronger lock when using declarative partitioning than when using table inheritance. For example, removing a partition from a partitioned table requires taking an `ACCESS EXCLUSIVE` lock on the parent table, whereas a `SHARE UPDATE EXCLUSIVE` lock is enough in the case of regular inheritance.

1.2.1 PARTITION ALGORITHM

To create partitioning with table inheritance use the following steps:

1. Create the "master" table, from which all of the “child” tables will inherit. This table will contain no data. Do not define any check constraints on this table, unless you intend them to be applied equally to all child tables. There is no point in defining any indexes or unique constraints on it, either.

   Example:

   ```
   CREATE TABLE part_table (id INTEGER, val TEXT, num NUMERIC);
   ```

2. Create several "child" tables that each inherit from the master table. Normally, these tables will not add any columns to the set inherited from the master. Just as with declarative partitioning, these tables are in every way normal PostgreSQL tables (or foreign tables).
3. Add non-overlapping table constraints to the child tables to define the allowed key values in each. Ensure that the constraints guarantee that there is no overlap between the key values permitted in different child tables.

   Example:

   ```
   CREATE TABLE part_table_s(
            CHECK (num >= MINVALUE and num < 100)
   ) INHERITS (part_table);

   CREATE TABLE part_table_l(
            CHECK (num >= 100 and num < 1000)
   ) INHERITS (part_table);

   CREATE TABLE part_table_xl(
            CHECK (num >= 1000 and num < MAXVALUE)
   ) INHERITS (part_table);
   ```

4. For each child table, create an index on the key column(s), as well as any other indexes you might want.

   Example:

   ```
   CREATE INDEX part_table_s_num ON part_table_s (num);
   CREATE INDEX part_table_l_num ON part_table_l (num);
   CREATE INDEX part_table_xl_num ON part_table_xl (num);
   ```

5. We want our application to be able to say INSERT INTO part_table ... and have the data be redirected into the appropriate child table. We can arrange that by attaching a suitable trigger function to the master table. After creating the function, we create a trigger which calls the trigger function.

Example:

```
CREATE OR REPLACE FUNCTION part_table_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.num >= MINVALUE AND
         NEW.logdate < 100 ) THEN
        INSERT INTO part_table_s VALUES (NEW.*);
    ELSIF ( NEW.logdate >= 100 AND
            NEW.logdate < 1000 ) THEN
        INSERT INTO part_table_l VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= 1000 AND
            NEW.logdate < MAXVALUE ) THEN
        INSERT INTO part_table_xl VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the part_table_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER insert_part_table_trigger
    BEFORE INSERT ON part_table
    FOR EACH ROW EXECUTE FUNCTION part_table_insert_trigger();
```

A different approach to redirecting inserts into the appropriate child table is to set up rules, instead of a trigger, on the master table. A rule has significantly more overhead than a trigger, but the overhead is paid once per query rather than once per row, so this method might be advantageous for bulk-insert situations. In most cases, however, the trigger method will offer better performance.

6. Ensure that the constraint_exclusion configuration parameter is not disabled in postgresql.conf; otherwise child tables may be accessed unnecessarily.

1.2.2 PARTITION MAINTENANCE

To remove old data quickly, simply drop the child table that is no longer necessary by using `DROP TABLE` command.

Use `ALTER TABLE ... NO INHERIT...` command to remove the child table from the inheritance hierarchy table but retain access to it as a table in its own right.

To add a new child table to handle new data, create an empty child table just as the original children were created above.

Alternatively, one may want to create and populate the new child table before adding it to the table hierarchy. This could allow data to be loaded, checked, and transformed before being made visible to queries on the parent table.

There is no automatic way to verify that all of the `CHECK` constraints are mutually exclusive. It is safer to create code that generates child tables and creates and/or modifies associated objects than to write each by hand.

Indexes and foreign key constraints apply to single tables and not to their inheritance children.

If you are using manual `VACUUM` or `ANALYZE` commands, don't forget that you need to run them on each child table individually.

`INSERT` statements with `ON CONFLICT` clauses are unlikely to work as expected, as the `ON CONFLICT` action is only taken in case of unique violations on the specified target relation, not its child relations.

Triggers or rules will be needed to route rows to the desired child table, unless the application is explicitly aware of the partitioning scheme. Triggers may be complicated to write, and will be much slower than the tuple routing performed internally by declarative partitioning.

1.3 PARTITION PRUNING

Partition pruning is a query optimization technique that improves performance for declaratively partitioned tables.

With partition pruning enabled, the planner will examine the definition of each partition and prove that the partition need not be scanned because it could not contain any rows meeting the query's `WHERE` clause. When the planner can prove this, it excludes (*prunes*) the partition from the query plan.

By using the `EXPLAIN` command and the enable_partition_pruning configuration parameter, it's possible to show the difference between a plan for which partitions have been pruned and one for which they have not.

Note that partition pruning is driven only by the constraints defined implicitly by the partition keys, not by the presence of indexes. Therefore it isn't necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

Partition pruning during execution can be performed at any of the following times:

- During initialization of the query plan. Partition pruning can be performed here for parameter values which are known during the initialization phase of execution. Partitions which are pruned during this stage will not show up in the query's `EXPLAIN` or `EXPLAIN ANALYZE`. It is possible to determine the number of partitions which were removed during this phase by observing the “Subplans Removed” property in the `EXPLAIN` output.
- During actual execution of the query plan. Partition pruning may also be performed here to remove partitions using values which are only known during actual query execution. This includes values from subqueries and values from execution-time parameters such as those from parameterized nested loop joins. Since the value of these parameters may change many times during the execution of the query, partition pruning is performed whenever one of the execution parameters being used by partition pruning changes. Determining if partitions were pruned during this phase requires careful inspection of the loops property in the `EXPLAIN ANALYZE` output. Subplans corresponding to different partitions may have different values for it depending on how many times each of them was pruned during execution. Some may be shown as (never executed) if they were pruned every time.

Partition pruning can be disabled using the enable_partition_pruning setting.

1.4 CONSTRAINT EXCLUSION

Constraint exclusion is a query optimization technique similar to partition pruning. While it is primarily used for partitioning implemented using the legacy inheritance method, it can be used for other purposes, including with declarative partitioning.

Constraint exclusion works in a very similar way to partition pruning, except that it uses each table's `CHECK` constraints — which gives it its name — whereas partition pruning uses the table's partition bounds, which exist only in the case of declarative partitioning. Another difference is that constraint exclusion is only applied at plan time; there is no attempt to remove partitions at execution time.

The fact that constraint exclusion uses `CHECK` constraints, which makes it slow compared to partition pruning, can sometimes be used as an advantage: because constraints can be defined even on declaratively-partitioned tables, in addition to their internal partition bounds, constraint exclusion may be able to elide additional partitions from the query plan.

The default (and recommended) setting of constraint_exclusion is neither on nor off, but an intermediate setting called partition, which causes the technique to be applied only to queries that are likely to be working on inheritance partitioned tables. The on setting causes the planner to examine `CHECK` constraints in all queries, even simple ones that are unlikely to benefit.

Constraint exclusion only works when the query's `WHERE` clause contains constants (or externally supplied parameters).

Keep the partitioning constraints simple, else the planner may not be able to prove that child tables might not need to be visited. Use simple equality conditions for list partitioning, or simple range tests for range partitioning, as illustrated in the preceding examples. A good rule of thumb is that partitioning constraints should contain only comparisons of the partitioning column(s) to constants using B-tree-indexable operators, because only B-tree-indexable column(s) are allowed in the partition key.

All constraints on all children of the parent table are examined during constraint exclusion, so large numbers of children are likely to increase query planning time considerably. So the legacy inheritance based partitioning will work well with up to perhaps a hundred child tables; don't try to use many thousands of children.

2. PARALLEL EXECUTION CONCEPTS

Parallel execution can be viewed as yet another way to split up the query: the amount of work needed to execute a query is divided between processing units (processors or cores).

Any parallel algorithm has a certain part that must be executed on a single unit. Also, additional overheads appear as a cost of synchronizations between parallel processes. For these reasons, parallel processing is mostly beneficial when bulk amounts of data are processed.

Specifically, parallel execution is beneficial for massive scans and hash joins. Both scans and hash joins are typical for long queries, for which the speed-up is usually most significant.

In contrast, the speed-up for short queries is usually negligible. However, parallel execution of different queries may improve throughput, but this is not related to parallel execution of a single query.

Sometimes an optimizer may replace index-based access (that would be used within sequential execution) with a parallel table scan. This may be caused by imprecise cost estimation. In such cases, parallel execution may be slower than sequential execution.

In addition, whatever scalability benefits are provided by parallel execution cannot fix poor design or compensate for inefficient code for a simple mathematical reason: scalability benefits from parallelism are at best linear, while the cost of nested loops is quadratic.

The following are the most important operations that can be done in parallel:

- Parallel sequential scans
- Parallel index scans (btrees only)
- Parallel bitmap heap scans
- Parallel joins (all types of joins)

- Parallel btree creation (CREATE INDEX)
- Parallel aggregation
- Parallel append
- VACUUM
- CREATE INDEX

2.1 PARALLEL QUERYING

When the optimizer determines that parallel query is the fastest execution strategy for a particular query, it will create a query plan which includes a *Gather* or *Gather Merge* node.

```
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

A gather node contains exactly one plan, which it divides amongst what are called workers. Each worker runs as separate backend processes, each process working on a portion of the overall query. The results of workers are collected by a worker acting as the leader. The leader does the same work as other workers but has the added responsibility of collecting all the answers from fellow workers. If the gather node is the root node of a plan, the whole query will be run in parallel. If it's lower down, only the subplan it encompasses will be parallelized.

The total number of background workers that can exist at any one time is limited by both **max_worker_processes** and **max_parallel_workers**. Therefore, it is possible for a parallel query to run with fewer workers than planned, or even with no workers at all.

The cost of organizing additional workers (even one) significantly increases the total time of the query.

Generally, parallelization is rarely worthwhile for queries that finish in a few milliseconds. But for queries over a ginormous dataset that normally take seconds or minutes to complete, parallelization is worth the initial setup cost.

2.2 PARALLEL PLANS

Because each worker executes the parallel portion of the plan to completion, it is not possible to simply take an ordinary query plan and run it using multiple workers. Each worker would produce a full copy of the output result set, so the query would not run any faster than normal but would produce incorrect results.

Instead, the parallel portion of the plan must be what is known internally to the query optimizer as a *partial plan*; that is, it must be constructed so that each process which executes the plan will generate only a subset of the output rows in such a way that each required output row is guaranteed to be generated by exactly one of the cooperating processes. Generally, this means that the scan on the driving table of the query must be a parallel-aware scan.

2.2.1 PARALLEL SCANS

The following types of parallel-aware table scans are currently supported.

- In a *parallel sequential scan*, the table's blocks will be divided among the cooperating processes. Blocks are handed out one at a time, so that access to the table remains sequential.
- In a *parallel bitmap heap scan*, one process is chosen as the leader. That process performs a scan of one or more indexes and builds a bitmap indicating which table blocks need to be visited. These blocks are then divided among the cooperating processes as in a parallel sequential scan. In other words, the heap scan is performed in parallel, but the underlying index scan is not.
- In a *parallel index scan* or *parallel index-only scan*, the cooperating processes take turns reading data from the index. Currently, parallel index scans are supported only for b-tree indexes. Each process will claim a single index block and will scan and return all tuples referenced by that block; other processes can at the same time be returning tuples from a different index block. The results of a parallel b-tree scan are returned in sorted order within each worker process.

Other scan types, such as scans of non-b-tree indexes, may support parallel scans in the future.

2.2.2 PARALLEL JOINS

Just as in a non-parallel plan, the driving table may be joined to one or more other tables using a nested loop, hash join, or merge join. The inner side of the join may be any kind of non-parallel plan that is otherwise supported by the planner provided that it is safe to run within a parallel worker. Depending on the join type, the inner side may also be a parallel plan.

- In a *nested loop join*, the inner side is always non-parallel. Although it is executed in full, this is efficient if the inner side is an index scan, because the outer tuples and thus the loops that look up values in the index are divided over the cooperating processes.
- In a *merge join*, the inner side is always a non-parallel plan and therefore executed in full. This may be inefficient, especially if a sort must be performed, because the work and resulting data are duplicated in every cooperating process.
- In a *hash join* (without the "parallel" prefix), the inner side is executed in full by every cooperating process to build identical copies of the hash table. This may be inefficient if the hash table is large or the plan is expensive. In a *parallel hash join*, the inner side is a *parallel hash* that divides the work of building a shared hash table over the cooperating processes.

2.2.3 PARALLEL SAFETY

The planner classifies operations involved in a query as either *parallel safe*, *parallel restricted*, or *parallel unsafe*. A parallel safe operation is one which does not conflict with the use of parallel query. A parallel restricted operation is one which cannot be performed in a parallel worker, but which can be performed in the leader while parallel query is in use. Therefore, parallel restricted operations can never occur below a Gather or Gather Merge node but can occur elsewhere in a plan which contains such a node. A parallel unsafe operation is one which cannot be performed while parallel query is in use, not even in the leader. When a query contains anything which is parallel unsafe, parallel query is completely disabled for that query.

The following operations are always parallel restricted:

- Scans of common table expressions (CTEs).
- Scans of temporary tables.
- Scans of foreign tables, unless the foreign data wrapper has an IsForeignScanParallelSafe API which indicates otherwise.
- Plan nodes to which an InitPlan is attached.
- Plan nodes which reference a correlated SubPlan.

3. SOURCE BOOKS AND ARTICLES

1. PostgreSQL 13.4 Documentation, The PostgreSQL Global Development Group, 1996–2021
2. Regina O. Obe and Leo S. Hsu, PostgreSQL: Up and Running THIRD EDITION, O'Reilly Media, 2018
3. Henrietta Dombrovskaya, Boris Novikov, Anna Bailliekova PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries, Apress, 2021