A Performance Cheat Sheet for PostgreSQL

Sebastian Insausti
June 18, 2018

Related resources:

Blog
Why is PostgreSQL Running Slow? Tips & Tricks to Get to the Source

Blog
An Overview of Database Indexing in PostgreSQL

Blog
How to Benchmark PostgreSQL Performance

Blog
A Guide to Using pgBouncer for PostgreSQL

Technology
ClusterControl for PostgreSQL

Posted in: Performance Management PostgreSQL

Performance is one of the most important and most complex tasks when managing a database. It can be affected by the configuration, the hardware or even the design of the system. By default, PostgreSQL is configured with compatibility and stability in mind, since the performance depends a lot on the hardware and on our system itself. We can have a system with a lot of data being read but the information does not change frequently. Or we can have a system that writes continuously. For this reason, it is impossible to define a default configuration that works for all types of workloads.

In this blog, we will see how one goes about analyzing the workload, or queries, that are running. We shall then review some basic configuration parameters to improve the performance of our PostgreSQL database. As we mentioned, we will see only some of the parameters. The list of PostgreSQL parameters is extensive, we would only touch on some of the key ones. However, one can always consult the official documentation to delve into the parameters and configurations that seem most important or useful in our environment.

EXPLAIN

One of the first steps we can take to understand how to improve the performance of our database is to analyze the queries that are made.

PostgreSQL devises a query plan for each query it receives. To see this plan, we will use EXPLAIN.

The structure of a query plan is a tree of plan nodes. The nodes in the lower level of the tree are scan nodes. They return raw rows from a table. There are different

```
-----------------------------------------------------------------------
Nested Loop  (cost=0.00..734.81 rows=50662 width=144)
  ->  Seq Scan on city t1  (cost=0.00..93.19 rows=347 width=31)
        Filter: ((id > 100) AND (population > 700000))
  ->  Materialize  (cost=0.00..8.72 rows=146 width=113)
        ->  Seq Scan on country t2  (cost=0.00..7.99 rows=146 width=113)
              Filter: (population < 7000000)
(6 rows)
```

This command shows how the tables in our query will be scanned. Let's see what these values correspond to that we can observe in our EXPLAIN.

- The first parameter shows the operation that the engine is performing on the data in this step.
- Estimated start-up cost. This is the time spent before the output phase can begin.
- Estimated total cost. This is stated on the assumption that the plan node is run to completion. In practice, a node's parent node might stop short of reading all available rows.
- Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.
- Estimated average width of rows output by this plan node.

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. When comparing how effective one query is against the other, we will in practice be comparing the cost values of them.

It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan.

The costs are measured in arbitrary units determined by the planner's cost parameters. Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to 1.0 and the other cost parameters are set relative to that.

EXPLAIN ANALYZE

Let's see an example of the use of this tool.

```
world=# EXPLAIN ANALYZE SELECT * FROM city t1,country t2 WHERE id>100 AND t
                                                     QUERY PLAN
---------------------------------------------------------------------------
Nested Loop  (cost=0.00..734.81 rows=50662 width=144) (actual time=0.081..2
  ->  Seq Scan on city t1  (cost=0.00..93.19 rows=347 width=31) (actual tim
        Filter: ((id > 100) AND (population > 700000))
        Rows Removed by Filter: 3729
  ->  Materialize  (cost=0.00..8.72 rows=146 width=113) (actual time=0.000.
        ->  Seq Scan on country t2  (cost=0.00..7.99 rows=146 width=113) (a
              Filter: (population < 7000000)
              Rows Removed by Filter: 93
Planning time: 0.136 ms
Execution time: 24.627 ms
(10 rows)
```

If we do not find the reason why our queries take longer than they should, we can check this blog for more information.

VACUUM

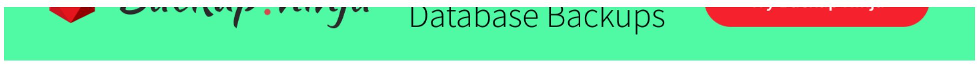
The VACUUM process is responsible for several maintenance tasks within the database, one of them recovering storage occupied by dead tuples. In the normal operation of PostgreSQL, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is performed. Therefore, it is necessary to do the VACUUM periodically, especially in frequently updated tables.

If the VACUUM is taking too much time or resources, it means that we must do it more frequently, so that each operation has less to clean.

In any case you may need to disable the VACUUM, for example when loading data in large quantities.

The VACUUM simply recovers space and makes it available for reuse. This form of the command can operate in parallel with the normal reading and writing of the table, since an exclusive lock is not obtained. However, the additional space is not returned to the operating system (in most cases); it is only available for reuse within the same table.

VACUUM FULL rewrites all the contents of the table in a new disk file without additional space, which allows the unused space to return to the operating system. This form is much slower and requires an exclusive lock on each table while processing.



ANALYZE collects statistics on the contents of the tables in the database and stores the results in pg_statistic. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

Configuration parameters

To modify these parameters we must edit the file $ PGDATA / postgresql.conf. We must bear in mind that some of them require a restart of our database.

max_connections

Determines the maximum number of simultaneous connections to our database. There are memory resources that can be configured per client, therefore, the maximum number of clients can suggest the maximum amount of memory used.

superuser_reserved_connections

In case of reaching the limit of max_connection, these connections are reserved for superuser.

shared_buffers

Sets the amount of memory that the database server uses for shared memory buffers. If you have a dedicated database server with 1 GB or more of RAM, a reasonable initial value for shared_buffers is 25% of your system's memory. Larger configurations for shared_buffers generally require a corresponding increase in max_wal_size, to extend the process of writing large amounts of new or modified data over a longer period of time.

temp_buffers

Sets the maximum number of temporary buffers used for each session. These are local session buffers used only to access temporary tables. A session will assign the temporary buffers as needed up to the limit given by temp_buffers.

be allowed to use as much memory as specified by this value before it starts to write data in temporary files.

This option was called sort_mem in older versions of PostgreSQL.

maintenance_work_mem

Specifies the maximum amount of memory that maintenance operations will use, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. Since only one of these operations can be executed at the same time by a session, and an installation usually does not have many of them running simultaneously, it can be larger than the work_mem. Larger configurations can improve performance for VACUUM and database restores.

When the autovacuum is executed, this memory can be assigned the number of times in which the autovacuum_max_workers parameter is configured, so we must take this into account, or otherwise, configure the autovacuum_work_mem parameter to manage this separately.

fsync

If fsync is enabled, PostgreSQL will try to make sure that the updates are physically written to the disk. This ensures that the database cluster can be recovered to a consistent state after an operating system or hardware crash.

While disabling fsync generally improves performance, it can cause data loss in the event of a power failure or a system crash. Therefore, it is only advisable to deactivate fsync if you can easily recreate your entire database from external data.

checkpoint_segments (PostgreSQL < 9.5)

Maximum number of record file segments between automatic WAL control points (each segment is normally 16 megabytes). Increasing this parameter can increase the amount of time needed to recover faults. In a system with a lot of traffic, it can affect the performance if it is set to a very low value. It is recommended to increase the value of checkpoint_segments on systems with many data modifications.

Also, a good practice is to save the WAL files on a disk other than PGDATA. This is

Maximum size the WAL is allowed to grow between the control points. The size of WAL can exceed max_wal_size in special circumstances. Increasing this parameter can increase the amount of time needed to recover faults.

min_wal_size (PostgreSQL >= 9.5)

When the WAL file is kept below this value, it is recycled for future use at a checkpoint, instead of being deleted. This can be used to ensure that enough WAL space is reserved to handle spikes in the use of WAL, for example when executing large batch jobs.

wal_sync_method

Method used to force WAL updates to the disk. If fsync is disabled, this setting has no effect.

wal_buffers

The amount of shared memory used for WAL data that has not yet been written to disk. The default setting is about 3% of shared_buffers, not less than 64KB or more than the size of a WAL segment (usually 16MB). Setting this value to at least a few MB can improve write performance on a server with many concurrent transactions.

effective_cache_size

This value is used by the query planner to take into account plans that may or may not fit in memory. This is taken into account in the cost estimates of using an index; a high value makes it more likely that index scans are used and a low value makes it more likely that sequential scans will be used. A reasonable value would be 50% of the RAM.

default_statistics_target

PostgreSQL collects statistics from each of the tables in its database to decide how queries will be executed on them. By default, it does not collect too much information, and if you are not getting good execution plans, you should increase

synchronous_commit

Specifies whether the transaction commit will wait for the WAL records to be written to disk before the command returns a "success" indication to the client. The possible values are: "on", "remote_apply", "remote_write", "local" and "off". The default setting is "on". When it is disabled, there may be a delay between the time the client returns, and when the transaction is guaranteed to be secure against a server lock. Unlike fsync, disabling this parameter does not create any risk of database inconsistency: a crash of the operating system or database may result in the loss of some recent transactions allegedly committed, but the state of the database will be exactly the same as if those transactions had been cancelled cleanly. Therefore, deactivating synchronous_commit can be a useful alternative when performance is more important than the exact certainty about the durability of a transaction.

Logging

There are several types of data to log that may be useful or not. Let's see some of them:

- log_min_error_statement: Sets the minimum logging level.
- log_min_duration_statement: Used to record slow queries in the system.
- log_line_prefix: Adheres information at the beginning of each log line.
- log_statement: You can choose between NONE, DDL, MOD, ALL. Using "all" can cause performance problems.

Design

In many cases, the design of our database can affect performance. We must be careful in our design, normalizing our schema and avoiding redundant data. In many cases it is convenient to have several small tables instead of one huge table. But as we said before, everything depends on our system and there is not a single possible solution.

Another very useful tool is the management of connection pool. If we have a system with a lot of load, we can use this to avoid saturating the connections in the database and to be able to reuse them.

Hardware

As we mentioned at the beginning of this blog, hardware is one of the important factors that directly affect the performance of our database. Let's see some points to keep in mind.

- Memory: The more RAM we have, the more memory data we can handle, and that means better performance. The speed of writing and reading on disk is much slower than in memory, therefore, the more information we can have in memory, the better performance we will have.
- CPU: Maybe it does not make much sense to say this, but the more CPU we have, the better. In any case it is not the most important in terms of hardware, but if we can have a good CPU, our processing capacity will improve and that directly impacts our database.
- Hard disk: We have several types of discs that we can use, SCSI, SATA, SAS, IDE. We also have solid state disks. We must compare quality / price, which we should use to compare its speed. But the type of disk is not the only thing to consider, we must also see how to configure them. If we want good performance, we can use RAID10, keeping the WALs on another disk outside the RAID. It is not recommended to use RAID5 since the performance of this type of RAID for databases is not good.

Conclusion

After taking into account the points mentioned in this blog, we can perform a benchmark to verify the behavior of the database.

It is also important to have our database monitored to determine if we are facing a performance problem and to be able to solve it as soon as possible. For this task there are several tools such as Nagios, ClusterControl or Zabbix, among others, that allow us not only to monitor, but with some of them, allows us to take proactive action before the problem occurs. With ClusterControl, in addition to performance. Hopefully, it gives a clearer picture of what things can become important and some of the basic parameters that can be configured. Do not hesitate to let us know if we've missed important ones.

‹ Previous blog Next blog ›

read more by:

Sebastian Insausti

Sebastian Insausti has loved technology since his childhood, when he did his first computer course using Windows 3.11. And from that moment he was decided on what his profession would be. He has since built up experience with MySQL, PostgreSQL, HAProxy, WAF (ModSecurity), Linux (RedHat, CentOS, OL, Ubuntu server), Monitoring (Nagios), Networking and Virtualization (VMWare, Proxmox, Hyper-V, RHEV). He's also a speaker and has given a few talks locally on InnoDB Cluster and MySQL Enterprise together with an Oracle team.

More from This Author

What's New in PostgreSQL 12

Oct 21, 2019 • by Sebastian Insausti

Which Time-Series Database is Better: TimescaleDB vs InfluxDB

May 16, 2019 • by Sebastian Insausti

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

The only management system you'll ever need to take control of your open source database infrastructure.

Learn more