Key PostgreSQL Configuration Parameters for Enhanced Performance

PostgreSQL is a widely used database known for its robust performance and reliability. To get the most out of PostgreSQL, tuning its parameters is crucial.

In this blog, we will explore the various PostgreSQL performance-related parameters and how to tune them effectively. By measuring Transactions Per Second (TPS) before and after tuning, and analyzing the results, we will demonstrate the significant impact of tuning on PostgreSQL performance.

I am going to use an AWS EC2 instance here are the specifications:

- Type = t2.large
- CPU = 2 Core
- RAM = 8GB
- Hard Drive = 100GB SSD
- Operating system = Ubuntu 22.04
- PostgreSQL version = 16.3

Install PostgreSQL by following community guidelines here.

We will be using the pgbench utility, which comes with the default PostgreSQL installation, to run the TPC-B benchmark. TPC-B is one of the predefined benchmark tests in pgbench, simulating a banking environment where multiple clients perform transactions on accounts. This workload involves various transactions, including updates, inserts, and selects.

We will run the benchmarks three times before tuning the parameters and three times after tuning them. By averaging the results of these tests, we will determine how much speed improvement we achieved through parameter tuning.

We first need to Initialize the database to run pgbench:

- The -i flag indicates the initialization of the database named Postgres.
- -s 50: The -s option specifies the scaling factor. A scaling factor of 50 means that the size of the dataset will be 50 times the default size. This affects the number of rows in the tables created by pgbench.

```
pgbench -i -s 50 postgres
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
5000000 of 5000000 tuples (100%) done (elapsed 10.25 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 13.99 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 10.33 s,
vacuum 0.23 s, primary keys 3.42 s).Copy to Clipboard
```

We are going to perform a benchmark based on following parameters:

- Runs a benchmarking test on the PostgreSQL database named Postgres
- Uses 10 client connections (-c 10)
- Utilizes 2 threads (-j 2)
- Executes a total of 10,000 transactions (-t 10000)

So that means each client will run total number of 1000 transactions

Getting the TPS without tuning

Try 1

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 4.769 ms
initial connection time = 15.628 ms
tps = 2097.002029 (without initial connection time)Copy to Clipboard
```

Try 2

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 4.627 ms
initial connection time = 15.163 ms
tps = 2161.168772 (without initial connection time)Copy to Clipboard
```

Try 3

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 4.540 ms
initial connection time = 16.365 ms
tps = 2202.622456 (without initial connection time)Copy to Clipboard
```

The average TPS we are getting without PostgreSQL tunning is about 2153.598

Now its time to tune the PostgreSQL.

Tune the Postgres

Change the following parameters inside postgresql.conf file available inside data directory and restart the PostgreSQL.

max_connections = 20
Description: Maximum number of concurrent connections to the database.
Purpose: Limits the number of clients that can connect to the database simultaneously.

shared_buffers = 2GB
Description: Amount of memory the database server uses for shared memory buffers.
Purpose: Determines how much memory is allocated for caching data in memory to improve performance.

effective_cache_size = 6GB
Description: An estimate of the memory available for disk caching by the operating system and within the database itself.
Purpose: Helps the query planner make better decisions about using indexes and other optimizations.

maintenance_work_mem = 1GB
Description: Maximum amount of memory to be used for maintenance operations like VACUUM, CREATE INDEX, and ALTER TABLE
Purpose: Allocates memory for maintenance tasks to improve their performance.

checkpoint_completion_target = 0.9
Description: Target duration for completing a checkpoint, as a fraction of the checkpoint interval.
Purpose: Spreads out the I/O load generated by checkpoints to avoid performance spikes.

wal_buffers = 16MB
Description: Amount of memory used for write-ahead log (WAL) data that has not yet been written to disk.
Purpose: Helps to improve write performance and ensure data durability.

default_statistics_target = 500
Description: Default number of samples used by the query planner to gather statistics for each column.
Purpose: Improves the accuracy of query planning by collecting more detailed statistics.

random_page_cost = 1.1
Description: Cost estimate for a non-sequentially fetched disk page.
Purpose: Helps the query planner estimate the cost of index scans versus sequential scans.

effective_io_concurrency = 200
Description: Number of concurrent disk I/O operations that can be executed simultaneously.
Purpose: Improves performance for systems with high I/O concurrency capabilities.

work_mem = 26214kB
Description: Amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files.
Purpose: Allocates memory for query operations to improve their performance.

huge_pages = off
Description: Controls the use of huge pages in memory allocation.
Purpose: Disables the use of huge pages, which can be beneficial for certain workloads but may not be necessary for all environments.

min_wal_size = 4GB
Description:
Minimum size of the write-ahead log (WAL) files.
Purpose: Ensures a minimum amount of WAL is always available to avoid frequent checkpoints.

max_wal_size = 16GB
Description: Maximum size of the write-ahead log (WAL) files.
Purpose: Limits the amount of disk space used by WAL files to avoid excessive disk usage.

Try 1

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 3.643 ms
initial connection time = 16.480 ms
tps = 2745.170716 (without initial connection time)Copy to Clipboard
```

Try 2

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 3.551 ms
initial connection time = 16.267 ms
tps = 2816.442368 (without initial connection time)Copy to Clipboard
```

Try 3

```
pgbench -c 10 -j 2 -t 10000 postgres
pgbench (16.3 (Ubuntu 16.3-1.pgdg22.04+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 2
maximum number of tries: 1
number of transactions per client: 10000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 3.622 ms
initial connection time = 16.698 ms
tps = 2760.782096 (without initial connection time)Copy to Clipboard
```

The average TPS we are getting after PostgreSQL tunning is about **2774.13** means.

That means we have observed a total of **28.8% improvement** in PostgreSQL performance after the tuning.

In conclusion, optimizing PostgreSQL for high TPS on modest hardware demonstrates the power of efficient database tuning. By fine-tuning parameters such as memory allocation, query optimization, and indexing strategies, we've shown that significant performance gains are

achievable even on smaller instances. This not only enhances application responsiveness and user experience but also minimizes the need for costly infrastructure upgrades.

Whether you're managing a startup environment or scaling a growing application, investing in PostgreSQL tuning can maximize performance while optimizing costs, proving that thoughtful configuration can truly amplify database efficiency without breaking the bank.

Further reading

Whitepaper

- Data Ingestion Benchmark Comparison between PostgreSQL and MongoDB

Professional Services

- PostgreSQL Optimization and Scaling
- Performance Tuning for PostgreSQL

Blogs

- PostgreSQL Tuning and DBtune
- Performance tuning in PostgreSQL using shared_buffers
- PostgreSQL performance tuning using work_mem