CYBERTEC
POSTGRESQL SERVICES & SUPPORT

CYBERTEC PGEE

BULK LOADING DATA

Date: 2024-12-18
Publisher: CYBERTEC PGEE team

TABLE OF CONTENTS

SCOPE OF THIS DOCUMENT

This document outlines some performance data we have seen with PGEE during bulk loading. The idea of this publication is to give users a feeling as well as some hard evidence to judge the amount of data one can load into PGEE easily.

The following topics will be covered:

- Generating sample data
- Bulk loading results
- Optimizations and speed ups
- Indexing large amounts of data

All performance data has been generated on local hardware (AMD Ryzen 7 5700G, 8 cores, local NVME storage). RedHat Enterprise Linux.

PGEE: LOADING VAST AMOUNTS OF DATA

In this section, we will discuss how large amounts of data can be managed.

PREPARING SIMPLE SAMPLE DATA

For the purpose of this example, it is necessary to first create some sample data which is used to outline various steps leading to better performance.

In this first example, we will use a simple table containing a handful of integer columns. While straightforward, this example provides an initial ballpark figure that we can build upon as the examples grow more complex:

```
benchmark=# CREATE TABLE t_sample (
       x1           int,
       x2           int,
       x3           int,
       x4           int,
       x5           int
);
CREATE TABLE
```

For bulk loading, flat files are often used to load data into PostgreSQL. To separate the time required for generating sample data from the actual loading process, we created a simple script that generates 10 million random rows, which can be repeatedly loaded.

```
#!/usr/bin/perl
use strict;
use warnings;

my $num_values = 10000000;
my $columns = 5;

# Generate random values for each column
for (my $i = 0; $i < $num_values; $i++) {
	my @random_values;
	for (my $j = 0; $j < $columns; $j++) {
		push @random_values, int(rand(1000000));	}
	print join(',', @random_values)."\n";
}
```

The output of the script looks as follows:

```
[hs@pgee benchmark]$ head data.txt
774330,561461,750376,181778,228203
646872,135167,577678,165632,965694
800245,769874,106714,142260,247270
124488,748993,552285,738894,485066
335652,599812,236180,362727,131632
199173,940807,98900,393187,564139
317877,751284,349036,206875,983297
732537,977025,328413,106460,916940
310871,181555,438948,890915,776944
163794,870753,767715,599221,274701
```

The file is comma separated and is therefore suitable for the **COPY** command provided by PostgreSQL

CONFIGURATION OF PGEE

The default configuration of PostgreSQL is usually not ideal for most workloads. Therefore, the following changes have been made to the configuration to improve efficiency:

```
shared_buffers = 64GB
work_mem = 64MB
maintenance_work_mem = 640MB
max_worker_processes = 16
max_parallel_workers_per_gather = 4
max_parallel_maintenance_workers = 4
max_parallel_workers = 16
checkpoint_timeout = 20min
max_wal_size = 50GB
min_wal_size = 50GB
effective_cache_size = 96GB
track_io_timing = on
track_wal_io_timing = on
shared_preload_libraries = 'pg_stat_statements'
```

All other settings are unchanged. The most noteworthy changes are related to checkpoints. Longer checkpoint distances will be very beneficial for all kinds of write operations in general.

Note: In case of a crash this also means that we are dealing with longer recovery times because PostgreSQL has to redo all the changes made since the previous checkpoint.

In general, the ideal configuration depends on the workload we expect. Bulk loading is typically just one of many operations on a server. As a result, configuration is often a compromise to achieve the best balance for your scenario.

LOADING DATA INTO PGEE

There are various ways to run the benchmark. The most common one is usually pgbench, which is shipped with PostgreSQL. For this paper, we used a simple script to repeatedly load data:

```
#!/usr/bin/perl

use strict;
use warnings;

use Time::HiRes qw(gettimeofday);

my $num_runs = 1000;
my $command_to_run =
      'psql benchmark < command.sql > /dev/null';

my $start_time = gettimeofday();
for my $i (1..$num_runs) {
       my $start_time = gettimeofday();
       system($command_to_run);
       my $end_time = gettimeofday();
       print "$i,", ($end_time - $start_time), "\n";
}
```

The script will return two columns- one for the number and another for the time in seconds it took to load 10 million rows:

```
1,3.19022703170776
2,3.19360613822937
3,3.20990705490112
4,3.20702505111694
5,3.20211005210876
```

For a first analysis, the dataset has been loaded 1000 times. Which leaves us with

> “10 million rows x 1000 = 10 billion rows”

Let’s inspect the results and dig into more detail in the next section.

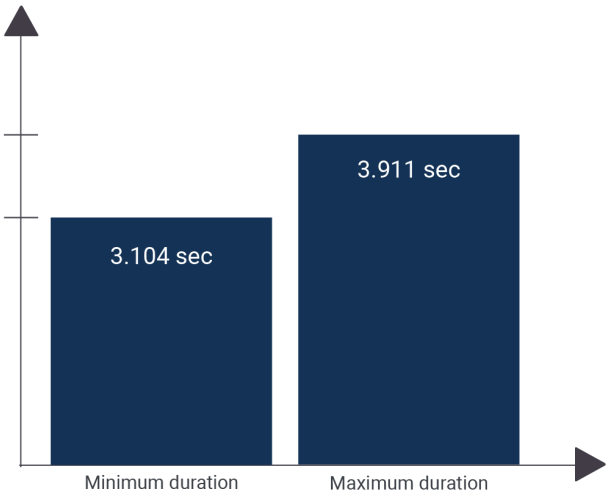
CHECKING THE RESULTS

The size of the table is **486 GB** which is around 4 times the size of RAM.
In this text, the average import of 10 million rows took 3.207 seconds.

RAW IMPORT SPEED AND RUNTIME VARIATIONS

This translates to around **3.18 million rows per second** (single core).
One often underestimated factor is runtime stability. What does this mean?
Specifically, what are the minimum and maximum import times?

Minimum duration: 3.104 seconds
Maximum duration: 3.911 seconds
Standard deviation: 67.13 milliseconds

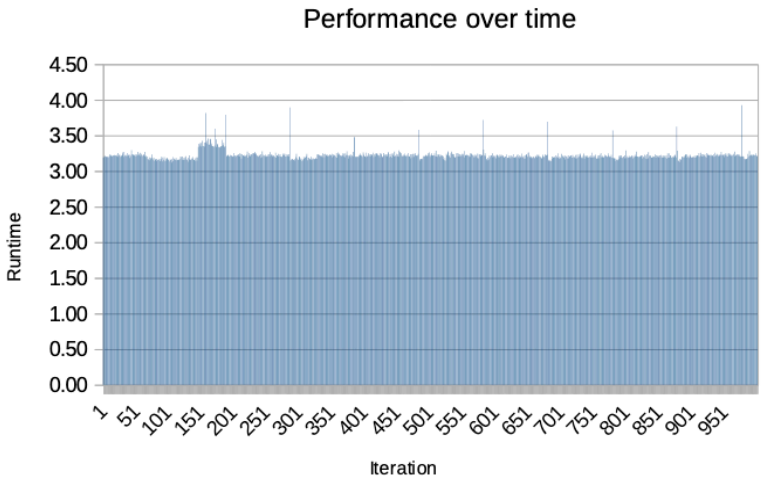


These numbers are promising, demonstrating stable behavior with very little variation, which indicates consistent performance.

The image shows how runtime evolved over 1000 iterations of 10 million rows each.

CHECKPOINTING BEHAVIOR

In this example, we increased the checkpoint distance to a much higher value (max_wal_size = 50 GB). Because of the fast import speed, we can observe that all checkpoints are "**requested**" and not "**timed**", which is a good sign for performance in general.



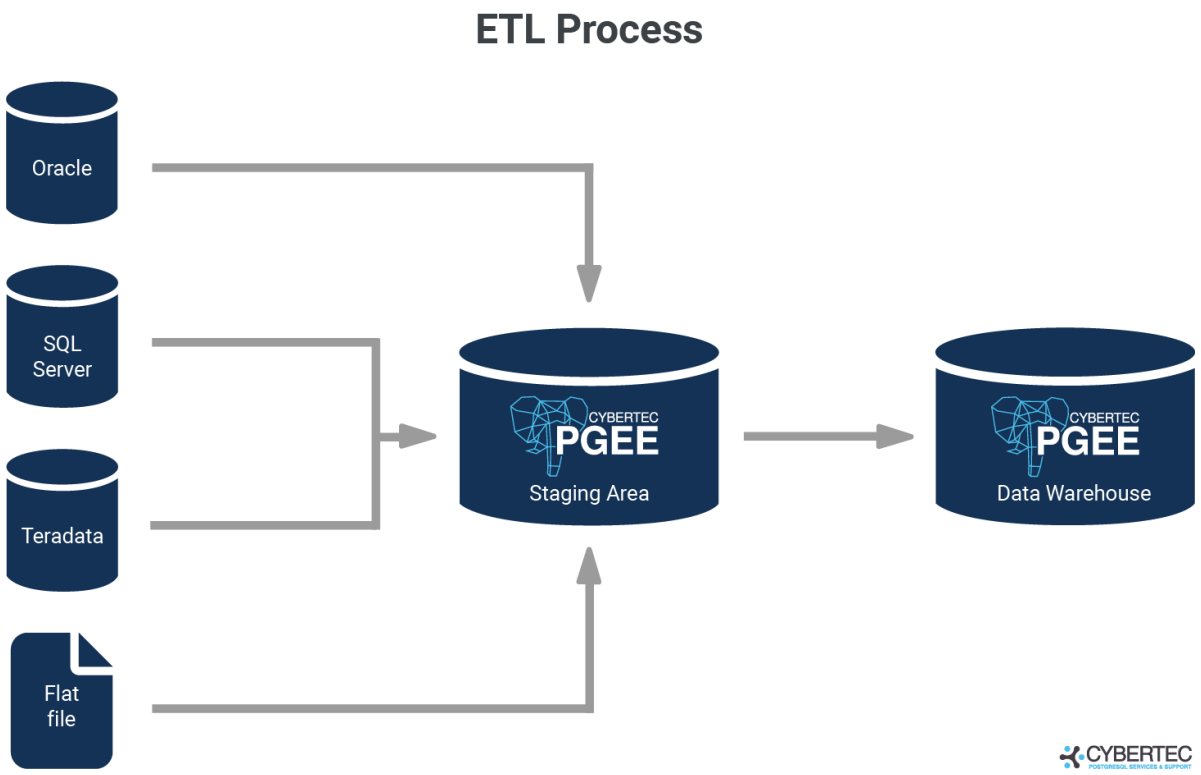
DATA WAREHOUSING: OPTIMIZED LOADING

In data warehousing, data is often loaded and processed in stages. Especially the first stage allows us to apply optimizations, which are otherwise not available.

Usually, a data warehousing project involves

- Importing large amounts of data into the staging area
- Clean data and load into data warehouse

The following image outlines how this works in practice:



The ETL process, particularly when pumping data into the “Staging area,” can be optimized in both PostgreSQL and PGEE.

USING UNLOGGED TABLES FOR FASTER LOADING

In a data warehouse, loading data into the staging area is critical. We can benefit from some characteristics which are intrinsically available:

- A failing import job can simply be restarted
- No need to replicate staging data
- No need to backup staging data

This allows us to apply optimizations such as:

- Bypassing the transaction log

The benefit is that we can avoid duplicate writes and reduce I/O load during loading, which can lead to higher throughput - especially on I/O bound systems.

CREATING UNLOGGED TABLES IN REAL LIFE

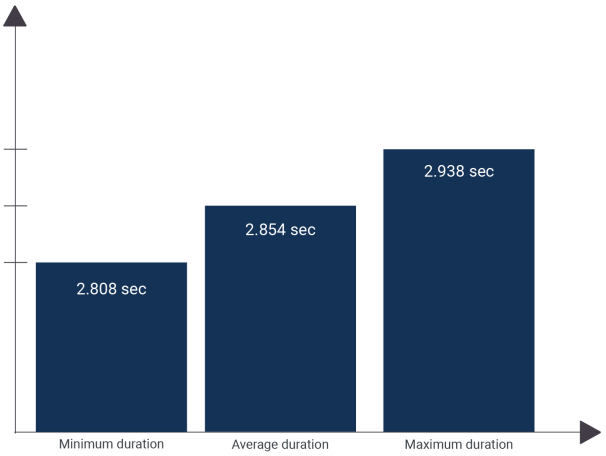
Let's rename the old table and create an unlogged table with the same structure. We can use the following two commands:

```
benchmark=# ALTER TABLE t_sample RENAME TO t_sample_full;
ALTER TABLE
benchmark=# CREATE UNLOGGED TABLE t_sample
       (LIKE t_sample_full);
CREATE TABLE
```

Importing data demonstrates approximately 12% better performance. Note that our import process is single-threaded and CPU bound. Still, we can measure a noteworthy difference which can be a lot bigger on comparatively slower I/O systems such as cloud storage.

Minimum duration: 2.808 seconds
Maximum duration: 2.938 seconds
Average duration: 2.854 seconds
Standard deviation: 16.86 ms

Once the data has been loaded, we can take a load at read performance to get a handle on how fast we can actually process data.



INSPECTING READ PERFORMANCE

The simplest test for raw throughput is a simple count. Note that the data set is 4x larger than memory, meaning caching of the data set is not possible:

```
benchmark=# \timing
Timing is on.
benchmark=# SELECT count(*) FROM t_sample;
    count
-------------
 10000000000
(1 row)

Time: 132325.580 ms (02:12.326)
```

Data can be read in **132 seconds,** which translates to a processing speed of **3.7 GB / second**. The test is 100% disk bound, as the storage subsystem tops out at around 3.8 GB / second.

On the CPU side, we are far from peak capacity. PostgreSQL requires only 5 processes, each running at no more than 65% peak load.

The important takeaway is that a majority of analytical workloads are I/O bound, which can be addressed by using **columnar storage** to shrink the overall size of data.

CREATING INDEXES ON LARGE TABLES

Indexing in PostgreSQL is a broad field that can be investigated in great detail.
For large amounts of data, we recommend the following index types:

- **btree:** Standard indexes
- **brin:** For pre-sorted data (timeseries, etc)
- **bloom:** To handle broad tables

In a default configuration, PGEE will use the following settings which control the way indexes are created:

```
maintenance_work_mem = 64 MB
max_parallel_maintenance_workers = 2
```

The default configuration leads to slow index creation:

```
benchmark=# CREATE INDEX ON t_sample_full (x1);
CREATE INDEX
Time: 3093540.167 ms (51:33.540)
```

Improving index creation can be achieved by adjusting those two parameters to higher values. The following table outlines the benefits we can achieve by setting those two values:

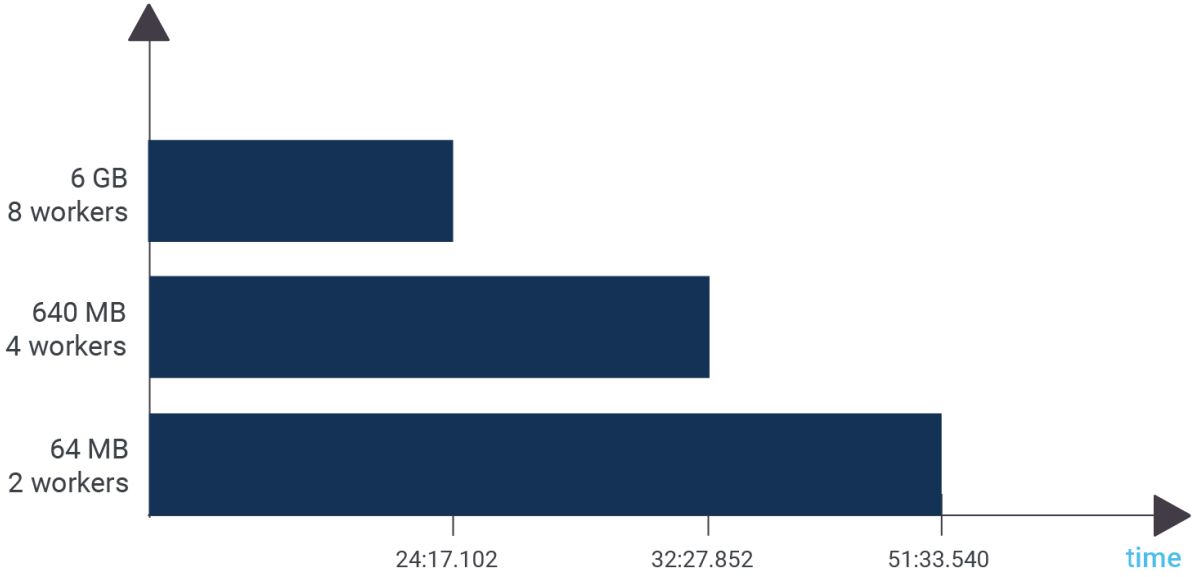
We can retry with higher values:

```
maintenance_work_mem = 640 MB
max_parallel_maintenance_workers = 4

benchmark=# CREATE INDEX ON t_sample_full (x1);
CREATE INDEX
Time: 1947852.321 ms (32:27.852)
```

Higher values can be even more beneficial:

```
maintenance_work_mem = 6 GB
max_parallel_maintenance_workers = 8

benchmark=# CREATE INDEX ON t_sample_full (x1);
CREATE INDEX
Time: 1457102.165 ms (24:17.102)
```

By adjusting the database configuration in a clever way, we can achieve significant gains:



SUPPORT AND GETTING HELP

REQUESTING HELP

Thank you for using CYBERTEC PGEE and **for being our valued customer.**
Your feedback is important to us and we look forward to hearing from you.
If you encounter any issues, or have technical questions, please reach out to our technical team. Our 24/7 support and ticketing system are here to assist you.

CYBERTEC Support Portal

Our consultants are eager to help you with any technical and business related issues.

If you need further information

For more information, or if you have any questions about our range of products, tools and services, contact us. There's no obligation—send us an inquiry via email or give us a call.

ISO 27001 CERTIFIED

Contact

CYBERTEC PostgreSQL International GmbH
Römerstraße 19
2752 Wöllersdorf
AUSTRIA

+ 43 (0) 2622 93022-0
sales@cybertec-postgresql.com

VERSION HISTORY

Version	Effective Date	Description	Author	Reviewed By	Approved By
1.0	2024-12-18	Content written	Hans-Jürgen Schönig	Patricia Horvath	Cornelia Biacsics