CYBERTEC
POSTGRESQL SERVICES & SUPPORT

CYBERTEC PGEE

SPEEDING UP LARGE SCALE AGGREGATIONS

CYBERTEC
PGEE

Date: 2024-12-13
Publisher: CYBERTEC PGEE Team

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

TABLE OF CONTENTS

PGEE: PERFORMANCE TUNING GUIDE

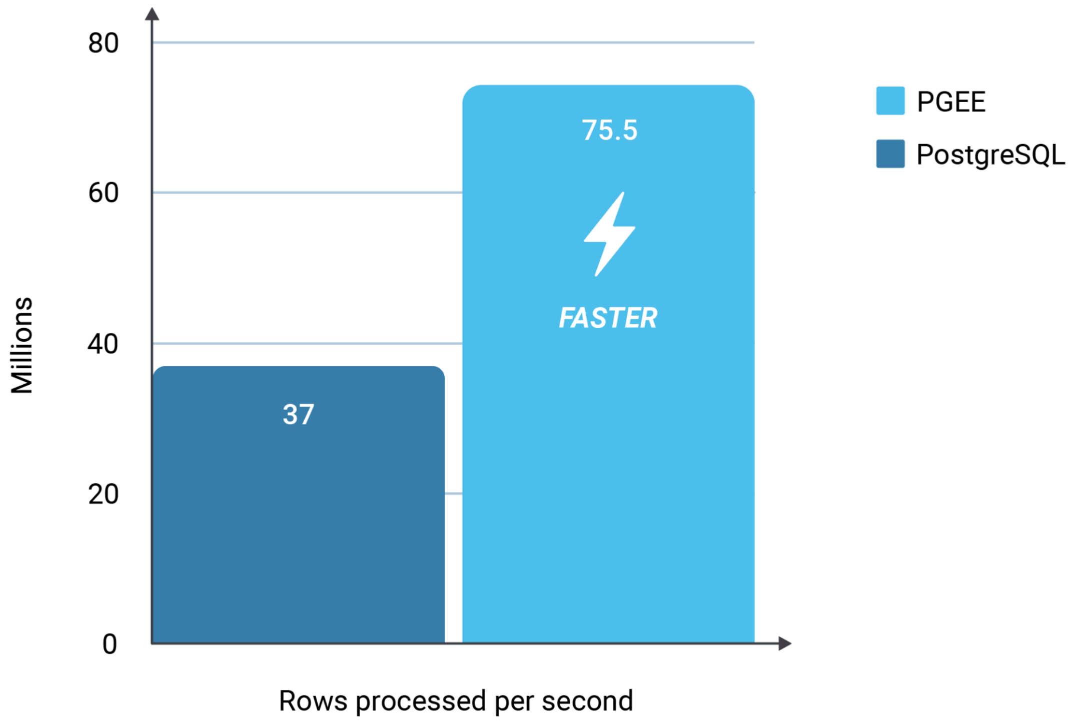
This document describes **performance tricks** that can be applied to PGEE to speed up **large-scale aggregations** and analytical workloads. We focus on some of the **optimizations** which are part of the PGEE query optimizer that have been specifically designed for enterprise-grade workloads in a **highly scalable** and often regulated environment.

The following topics are covered:

- Execution order of aggregation functions
- Adjusting PostgreSQL parameters for fast analytics
- Performance and plan inspection

"Better performance with PGEE"

PGEE aggregation speed compared to Open Source PostgreSQL



PREPARING 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.

After starting PGEE, we can create a sample database (from the Linux command line).

```
[hs@pgee ~]$ createdb benchmark
[hs@pgee ~]$ psql benchmark
psql (17.2 EE 1.4.0, server 17.2 EE 1.4.0)
 ____   ____ _____ _____
|  _ \ / ___| ____| ____|
| |_) | |  _|  _| |  _|
|  __/| |_| | |___| |___
|_|    \____|_____|_____|
PostgreSQL EE by CYBERTEC
Type "help" for help.

benchmark=#
```

Alternatively,, it is also possible to use other graphical tools such as pgadmin4 among many others.

In the next step, we will create two tables: one will store a handful of countries `(t_country)` and the second will handle sales `(t_sales)`. We can create and populate the data as follows:

```
benchmark=# CREATE TABLE t_country (
      id          int   PRIMARY KEY,
      name        text
);
CREATE TABLE

benchmark=# INSERT INTO t_country
      VALUES      (1, 'USA'),
                  (2, 'South Africa'),
                  (3, 'Germany'),
                  (4, 'India');
INSERT 0 4

benchmark=# CREATE TABLE t_sales (
      country_id      int,
      t               timestamptz DEFAULT clock_timestamp(),
      units           int
);
CREATE TABLE

benchmark=# INSERT INTO t_sales (country_id, units)
      SELECT     (id % 4) + 1, random()*10000
      FROM       generate_series(1, 200000000) AS id;
INSERT 0 200000000
```

As we can see here, the country table is really small - it includes only 4 countries. This is a very common case. In most large databases, there are typically various small lookup tables (to store data such as countries, currencies, etc) and a handful of really large tables. In our example, the large table serves as the sales table, populated with 200 million rows.

On disk this means:

```
benchmark=# \d+
           List of relations
 Schema |    Name     | … |  Size
--------+-----------+---+---------
 public | t_country | … | 16 kB
 public | t_sales   | … | 9955 MB
(2 rows)
```

While the lookup table is only 16 kB, the second table is close to 10 GB.

ADVANCED AGGREGATION AND GROUPING

If you are doing simple analysis and aggregation, a typical question to ask PostgreSQL would be:

"How many entries are there for each country?"

On the SQL level this translates to:

```
benchmark=# \timing
Timing is on.
benchmark=#
      SELECT     name, count(*)
      FROM       t_country AS c, t_sales AS s
      WHERE      s.country_id = c.id
      GROUP BY 1
      ORDER BY 2 DESC;
     name      |  count
--------------+----------
 Germany       | 50000000
 India         | 50000000
 South Africa | 50000000
 USA           | 50000000
(4 rows)

Time: 5387.018 ms (00:05.387)
```

Here, we join both tables using the country_id and apply the count function on top. On the test machine, the query takes 5.4 seconds. While this still translates to 37 million rows per second, we can do a lot better.

UNDERSTANDING POSTGRESQL AGGREGATION

Before we dig into the details, we have to inspect the execution plan standard PostgreSQL would give us when running the query:

```
benchmark=#
      explain SELECT   name, count(*)
      FROM       t_country AS c, t_sales AS s
      WHERE      s.country_id = c.id
      GROUP BY 1
      ORDER BY 2 DESC;
                                QUERY PLAN
--------------------------------------------------------------------
 Sort  (cost=2295202.29..2295202.30 rows=4 width=15)
   Sort Key: (count(*)) DESC
   ->  Finalize GroupAggregate  (cost=2295200.22..2295202.25 …)
       Group Key: c.name
       ->  Gather Merge  (cost=2295200.22..2295202.13 …)
           Workers Planned: 4
           ->  Sort  (cost=2294200.16..2294200.17 rows=4 width=15)
               Sort Key: c.name
               ->  Partial HashAggregate
                  (cost=2294200.08..2294200.12 rows=4 width=15)
                   Group Key: c.name
                  ->  Hash Join
                      (cost=1.09..2044199.96 rows=50000024 width=7)
                      Hash Cond: (s.country_id = c.id)
                      ->  Parallel Seq Scan on t_sales s
                       (cost=0.00..1773886.24 rows=50000024 …)
                      ->  Hash  (cost=1.04..1.04 rows=4 width=11)
                          ->  Seq Scan on t_country c
                           (cost=0.00..1.04 rows=4 width=11)
 JIT:
   Functions: 15
   Options: Inlining true, Optimization true,
            Expressions true, Deforming true
(18 rows)

Time: 1.658 ms
```

There are a few key points to note here. First of all, we can see:

- PostgreSQL uses a parallel query for higher efficiency

    - In this case, 4 parallel workers
    - This can be turned depending on various factors

What is more important here is the execution order the query is using:

- Read countries sequentially
- Read sales data sequentially
- Join the data
- Count the data

In reality, this means:

> "Check the name of 200 million country IDs"

In real life, this means a lot of overhead; however, this is the way the standard PostgreSQL optimizer operates.

LIGHTNING FAST QUERIES WITH PGEE

PGEE offers significantly better aggregation speed over Open Source PostgreSQL, as we will learn in this section.

CYBERTEC has greatly improved the PGEE optimizer to allow PostgreSQL to make some key changes to the way aggregations are handled:

"Aggregate first, join later"

By cleverly aggregating on numbers first, we can greatly reduce the number of lookups in a highly normalized data model. What does this mean in real life?

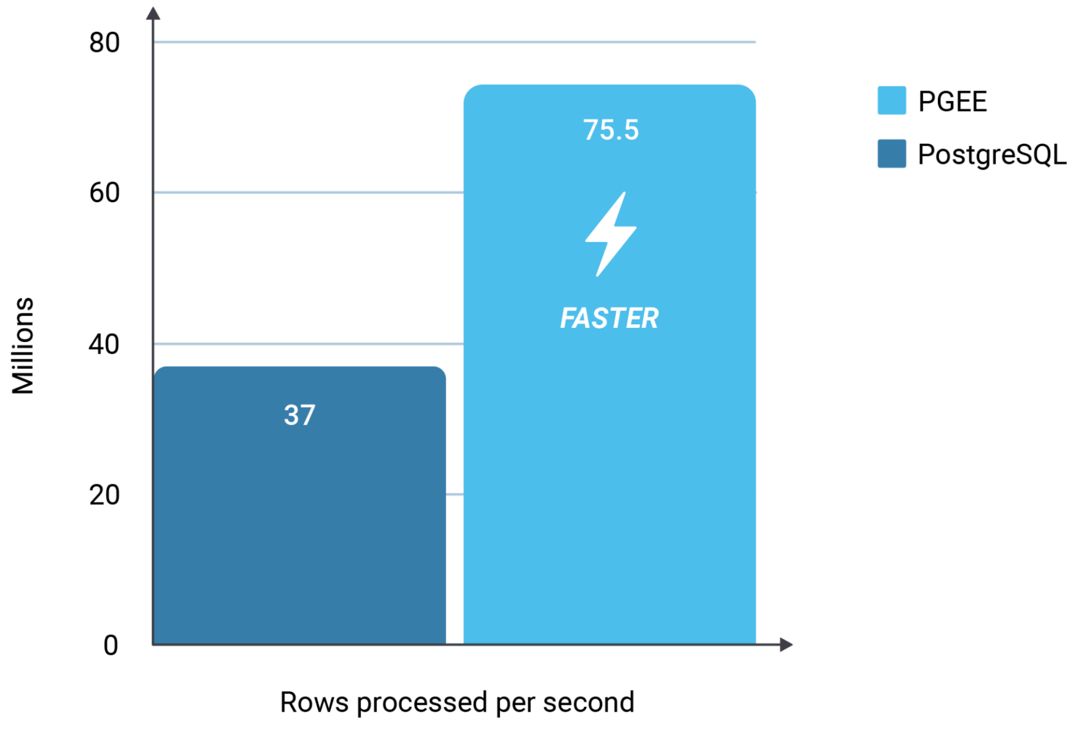
To benefit from PGEE optimizations, you can set `enable_agg_pushdown` to true in `postgresql.conf`, or set the variable at runtime:

```
benchmark=# SET enable_agg_pushdown = on;
SET
Time: 0.575 ms
benchmark=#
benchmark=# SELECT    name, count(*)
            FROM      t_country AS c, t_sales AS s
            WHERE     s.country_id = c.id
            GROUP BY 1
            ORDER BY 2 DESC;
     name     |  count
--------------+----------
 Germany      | 50000000
 India        | 50000000
 South Africa | 50000000
 USA          | 50000000
(4 rows)

Time: 2684.197 ms (00:02.684)
```

PGEE has processed 74.5 million rows / second vs 37 million rows.

PGEE aggregation speed compared to Open Source PostgreSQL



This query is using 4 CPU cores. This means that PGEE is processing close to 20 million rows per second per core (AMD Ryzen 7 5700G).

ADVANCED EXECUTION IN PGEE

The performance gains provided by PGEE are relevant. However, it is important to understand how we can observe those benefits when looking at the execution plan of the query in more detail:

```
benchmark=#
      explain    SELECT  name, count(*)
                 FROM     t_country AS c, t_sales AS s
                 WHERE   s.country_id = c.id
                 GROUP BY 1
                 ORDER BY 2 DESC;
                          QUERY PLAN
---------------------------------------------------------------------
 Sort (cost=2024887.63..2024887.64 rows=4 width=15)
   Sort Key: (count(*)) DESC
   -> Finalize GroupAggregate  (cost=2024887.05..2024887.59 …)
      Group Key: c.name
      ->  Gather Merge  (cost=2024887.05..2024887.53 …)
           Workers Planned: 4
           ->  Sort  (cost=2023886.99..2023887.00 rows=1 width=15)
               Sort Key: c.name
               ->  Nested Loop (cost=2023886.49..2023886.98 …)
                   ->  Partial HashAggregate
                       (cost=2023886.36..2023886.40 rows=4 width=12)
                       Group Key: s.country_id
                       ->  Parallel Seq Scan on t_sales s
                           (cost=0.00..1773886.24 rows=50000024 …)
                   ->  Index Scan using t_country_pkey on t_country c
                       (cost=0.13..0.15 rows=1 width=11)
                       Index Cond: (id = s.country_id)
 JIT:
   Functions: 10
   Options: Inlining true, Optimization true, Expressions true,
Deforming true
(17 rows)

Time: 1.528 ms
```

In this case, the Partial HashAggregate is directly operating on the data, returned by the sequential scan on the country table. As an aggregation key, we are able to use the country_id rather than the name, which is way more efficient.

Internally, this requires data type substitution, as well as some other advanced trickery which is beyond the scope of this document.

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

CYBERTEC POSTGRESQL SERVICES & SUPPORT

VERSION HISTORY

Version	Effective Date	Description	Author	Reviewed By	Approved By
1.0	2024-12-13	Content written	Hans-Jürgen Schönig	Christoph Berg	Cornelia Biacsics