PostgreSQL Performance: Monitoring and Enhancement

Table Of Contents

A. Intro

PostgreSQL is a widely used open-source relational database that offers robust functionality suitable for a variety of applications; However, performance-wise it can be significantly affected by numerous factors, and this is exactly what we

will delve into in this article, from the tools to investigate performance issues to their resolution through advanced optimization techniques.

> ***Remember, Remember the 28th of December! `PostgreSQL` performance improvements cannot be covered in just one article.***

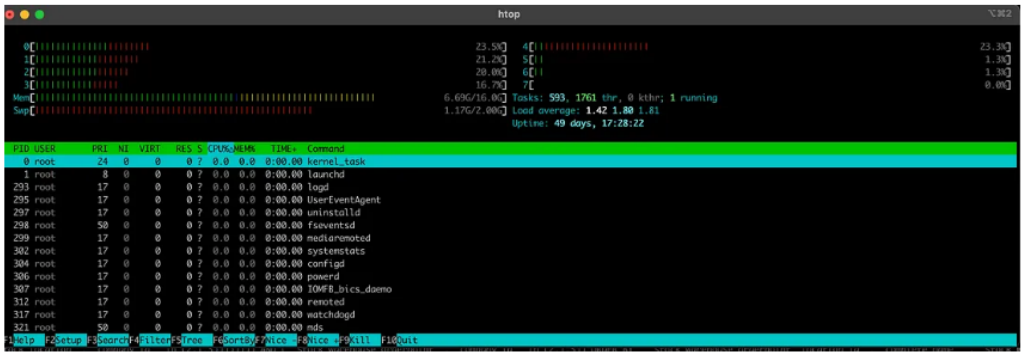
B. Investigating Performance Issues

B.1. Systematic Monitoring

This part covers some tools that can be used to investigate `PostgreSQL` performance issues.

B.1.1. Key Performance Metrics

Regular monitoring of CPU usage, memory, I/O, and disk space is critical, so using tools like `htop`, `iostat`, and `vmstat` offers real-time insights, for example: as you can see in the screenshot below, `htop` shows a comprehensive list of all running processes along with details such as PID, user, priority, memory consumption, and CPU usage.



htop

B.1.2. PostgreSQL-Specific Monitoring

You can rely on `PostgreSQL's` internal statistics collector, on this matter lights are shed on two important views as`pg_stat_activity` (or use `pg_activity`) and `pg_stat_statements`.

You can either use `pg_activity` or the query below which will help you identify active queries and who is running them, which is useful for pinpointing long-running or stuck queries.

```
SELECT pid, datname, usename, query, state
FROM pg_stat_activity
WHERE state = 'active';
```

While you can use `pg_stat_activity`to show all running queries, you can also manipulate `pg_stat_statements`to display the top 5 queries with the highest total execution time, which can reveal the potentially inefficient ones that may need optimization.

```
SELECT query, calls, total time, rows
FROM pg_stat_statements
ORDER BY total time DESC
LIMIT 5;
```

B.2. Analyzing Query Performance

This part covers some techniques that can be used to analyze `PostgreSQL` performance issues.

B.2.1. Log Analysis

Configuring `PostgreSQL` to log slow queries is crucial, it is pretty simple that you only need to locate the `postgresql.conf` file and edit the `log_min_duration_statement` line like below:

```
log_min_duration_statement = 1000  # MS
```

Important Note: don't forget to save and restart the `Postgres` server.

B.2.2. EXPLAIN ANALYZE

The `EXPLAIN ANALYZE` command is a life savior for understanding query performance:

```
EXPLAIN ANALYZE SELECT * FROM readers WHERE last_login >
CURRENT_DATE - INTERVAL '1 year';
```

As in the previous example, you add `EXPLAIN`before the query to generate the execution plan, and with `ANALYZE` it also executes the query and collects runtime statistics.

Using `EXPLAIN` only, can be enough to have an explanation for the query, but it is recommended to use `ANALYZE` to get the actual execution statistics.

> ***Please bear in mind, that `ANALYZE` executes the query so doing so on the production database with a `DELETE` query is like playing with explosives (smells like somebody is getting fiiiiiired).***

The most basic execution plan *(for the sake of the reader's mental health, complex plans are avoided in this article)* that you can get is the same as the one for the previous query, something that looks like the following:

```
Seq Scan on readers  (cost=0.00..1234.56 rows=4321 width=104)
  Filter: (last_login > (CURRENT_DATE - '1 year'::interval))
```

As you can see, the plan indicates inefficiency for large datasets and highlights the importance of considering *indexing*, since the filter's effectiveness depends on how many rows fall within the last year.

B.3. Profiling Database Load

B.3.1. pgBadger

For comprehensive log analysis, `pgBadger` is highly recommended because it generates detailed reports on query performance.

You can get it from the official website or through a package manager such as `brew`.

```
brew install pgbadger
```

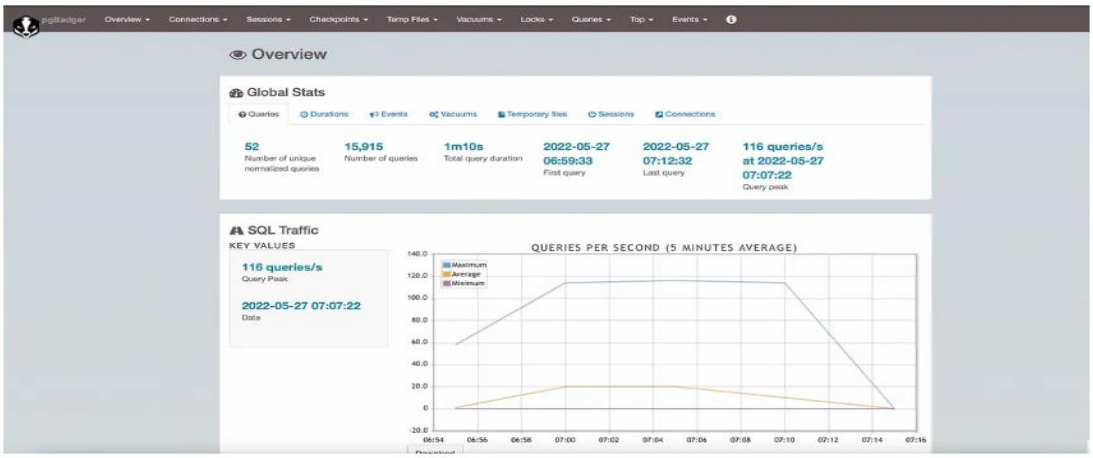
The next step is to change the `PostgreSQL` configuration in`postgresql.conf` file:

```
log_destination = 'stderr'
logging_collector = on
log_statement = 'all'
log_duration = on
log_line_prefix = '%t [%p-%l] %q%u@%d '
log_min_duration_statement = 0
```

Important Note: don't forget to save and reload the configuration with `pg_reload_conf()`

The last step is to generate the report of `PostgreSQL` performance that includes slow queries, lock waits...etc using `pgBadger` command

```
pgbadger /path/postgresql.log -o /path/output.html
```



pgbadger

Usually, the most common area to analyze in `output.html` is the section on slow or time-consuming queries, as optimizing these can significantly enhance overall database efficiency.

C. Performance Optimization Techniques

C.1. Indexing

For most people who work on improving database performance such as *Site Reliability Engineers, Database Administrators* ...etc in most cases, this step comes right after using the `EXPLAIN ANALYZE` , to the point that it became a meme.

indexing in production

We all know that a book's index helps readers find specific information quickly, so the database here is the book and the index serves as a guide for efficient access to particular data entries without the need to scan the entire dataset.

> ***With indexes, the database significantly reduces the time required for searching, sorting, and filtering operations.***

C.1.1. Right Index Type

- **B-Tree**: generally used when ordering data based on a specific column.

```
CREATE INDEX readers_name_index ON readers(name);
```

> ***Adding B-Tree index on the `name` column of the `readers` table would optimize queries that sort or search based on readers names.***

- **Hash**: To speed up data based on equality checks.

```
CREATE INDEX books_id_index ON books USING HASH (books_id);
```

> ***Adding Hash index on the `books_id` column of the `books` table would optimize queries that involve an equality condition, such as `SELECT * FROM books WHERE books_id = 28;`***

- **GIN**: Generally used for full-text search or operations on complex data like arrays, JSON ...etc

```
CREATE INDEX attachment_content_index ON attachments USING GIN (to_tsvector('english',
content));
```

> ***Adding a GIN index on the `content` column of the `attachments` table would optimize full-text search using the `to_tsvector` function for English-language text.***

C.1.2. Index Maintenance

Due to regular updates, deletions, and insertions, some indexes tend to consume more space than necessary, so it is important to frequently perform reindexing to manage index bloat.

To define index bloat, we can simply rely on `pg_stat_user_indexes` for example: we can retrieve *indexes > 1MB* sorted by size.

```
SELECT schemaname, tablename, indexname, pg_size_pretty(pg_relation_size(i.indexrelid)) AS
size,
       idx_scan as number_of_scans
FROM pg_stat_user_indexes JOIN pg_index i ON i.indexrelid = indexrelid
WHERE pg_relation_size(i.indexrelid) > '1MB'::bigint
ORDER BY pg_relation_size(i.indexrelid) DESC;
```

Afterwards, comes the part of rebuilding the indexes using the `REINDEX` command:

```
REINDEX INDEX CONCURRENTLY index_name;
```

> **Important Note:** ***Since reindexing can lock tables, it is crucial to schedule a downtime for maintenance. On a side note, using `CONCURRENTLY` reduces the downtime.***

C.2. Query Optimization

C.2.1. Writing Efficient SQL

- Select specific columns instead of `SELECT *`, specifying only the columns that are needed.

```
SELECT name, address, phone FROM readers;
```

- Practice proper use of `JOINs` focusing on the ones that align with specific data relationships. For example, the query below limits the range of the selected customers to those with reservations.

```
SELECT client.name, reservations.create_date FROM client INNER JOIN reservations ON client.id = reservations.client_id;
```

C.2.2. Advanced SQL Features

- **Partitioning**: To improve performance, you can split large tables into smaller and manageable pieces.

  For example: partitioning of the `reservations` table boosts query processing on specific date intervals.

```
CREATE TABLE reservations (reservation_id int NOT NULL,
                           reservation date date NOT NULL,
                           amount decimal)
PARTITION BY RANGE (reservation_date);
```

- **Parallel Query Execution**:
  Increasing `max_parallel_workers_per_gather` grants the

query `big_big_table` the privilege to use up to 7 CPU cores, which results in boosting execution time for large-scale data.

```
SET max parallel workers per gather TO 7;
SELECT blah, blah, blah FROM big_big_table WHERE whatever_conditions;
```

D. Database Configuration Tuning

> ***"PGTune is a useful tool that calculates PostgreSQL configurations for optimal performance based on hardware, but achieving the best setup requires considering database size, client number, and query complexity" ~ B.H***

D.1. Memory Settings

It's highly recommended to fine-tune `PostgreSQL` settings for efficient data storage. Specifically, adjusting `work_mem` is of value not only for effective memory management during sorting operations but also for optimizing query speed and reducing disk usage, leading to an overall improvement in database performance.

Below are the recommended values for each:

- `shared_buffers`: About 25% of the available RAM.
- `work_mem`: 4MB ~ 16MB per active query.

D.2. WAL Configuration

To maintain both speed and data safety in `PostgreSQL`, it is recommended to optimize `WAL` settings properly to boost the overall performance of the database while ensuring that changes are securely stored and can be recovered in case of failure.

E. Vacuuming and Autovacuum

Let's consider the `reservations` database where clients are processing and canceling reservations daily. Without vacuuming, the database would accumulate canceled reservations over time, leading to inefficiency and slower queries.

> ***Vacuuming ensures that canceled reservations are removed, keeping the database lean and responsive.***

- **Manual Vacuuming**: To reclaim the space and optimize a specific table, you can use `VACUUM` command:

```
VACUUM FULL a_specific_table;
```

- **Manual Analyze**: To update statistics you can add `ANALYZE` to the previous command or use it separately like below:

```
ANALYZE your_table;
```

- **Autovacuum Configuration**: Adjusting autovacuum settings in (duh not again!) the `postgresql.conf` file, tends to enable autovacuum, configure the thresholds for vacuuming and analyzing, and limit vacuum costs.

```
autovacuum = on
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
autovacuum_vacuum_cost_limit = 800
```

F. Hardware Optimization

F.1. Disk I/O

- SSDs for faster access.
- Separate `WAL` from data disks.

F.2. Network Performance

Ensure adequate network infrastructure.