PostgreSQL Performance: Advanced Techniques

Table Of Contents

1. Database Configuration Tuning

Managing connections efficiently can significantly improve PostgreSQL performance. So here, we are talking about how adjusting settings like ***max_connections*** and ***connection pooling*** can help maintain stability and performance.

1.1 Connection Pooling

Using a connection pooler like ***PgBouncer*** can help manage a large number of connections by reusing existing ones, which reduces the overhead of opening and closing connections.

Below is an example of the basic configuration for **PgBouncer**:

```
[databases]
postgres = host=localhost dbname=postgres

[pgbouncer]
listen_addr = 127.0.0.1
listen_port = 2828
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 100
default_pool_size = 20
```

1.2 Adjusting max_connections

Setting the ***max_connections*** setting to a value ensures it balances the number of concurrent users with the available system resources because if you allow too many connections, it can exhaust resources and degrade system performance.

```
max_connections = 200
```

2. Optimizing Network Settings

Efficient network configuration guarantees low latency and high throughput for database interactions. Therefore, adjusting the settings below can result in performance improvement.

2.1 TCP Settings

Adjusting TCP settings maintains stable and efficient connections, these include the following parameters:

- ***tcp_keepalives_idle:*** specifies the number of seconds a TCP connection must be idle before the first keepalive packet is sent.
- ***tcp_keepalives_interval:*** defines the interval again in seconds, between consecutive keepalive packets if no acknowledgment is received.
- ***tcp_keepalives_count:*** sets the maximum number of keepalive packets that will be sent before the connection is considered dead and terminated.

```
tcp_keepalives_idle = 60
tcp_keepalives_interval = 10
tcp_keepalives_count = 5
```

2.2 Compression and Encryption

Enabling SSL for secure connections and configuring compression for efficient data transfer can be very beneficial. The only downside here, is that encryption can add overhead, but it is essential for securing sensitive data.

```
ssl = on
ssl_cert_file = ‘server.crt’
ssl_key_file = ‘server.key’
ssl_prefer_server_ciphers = on
```

3. Advanced Vacuuming and Autovacuum Techniques

3.1 Vacuum Strategy

Effective vacuuming is important for maintaining database health and performance. Therefore, it’s advised to adopt a proactive vacuum strategy that considers the specific workload and usage patterns.

3.2 Aggressive Autovacuum Settings

For heavily updated tables, the need to configure more aggressive autovacuum settings arises to prevent bloat and ensure efficient space usage.

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_scale_factor = 0.02
autovacuum_analyze_threshold = 50
autovacuum_analyze_scale_factor = 0.01
```

3.3 Manual Vacuum Scheduling

In addition to autovacuum, as mentioned in the previous article; scheduling manual vacuum operations during low usage periods to ensure thorough cleaning of large tables is of importance.

```
VACUUM FULL big_table;
```

3.4 Monitoring Vacuum Efficiency

Here comes the step of using ***pg_stat_user_tables*** to monitor vacuum efficiency and adjust settings as needed, which ensures that the vacuum process effectively maintains table health.

```
SELECT relname, n_dead_tup, n_live_tup, last_vacuum, last_autovacuum
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
```

4. Hardware Optimization Techniques

4.1 CPU Optimization

Increasing CPU capabilities can significantly boost database performance, especially for compute-intensive operations.

4.2 CPU Affinity

Configuring CPU affinity to assign specific CPU cores to PostgreSQL processes helps reduce context switching, which occurs when the CPU switches between different tasks, and improves cache efficiency by keeping data closer to where it is processed.

```
taskset -c 0-3 postgres -D /var/lib/postgresql/data
```

4.3 Parallel Query Execution

To be able to take advantage of multi-core processors for large complex queries, we need to enable and tune parallel query execution.

```
max_parallel_workers_per_gather = 4
max_parallel_workers = 8
parallel_tuple_cost = 0.1
parallel_setup_cost = 1000
```

4.4 Advanced Storage Solutions

Optimizing storage solutions is critical for high performance databases, especially with large datasets. Therefore, here are a few suggestions:

- Use high performance SSDs or NVMe drives for data storage to reduce I/O latency and increase throughput.

- Implement RAID 10 for a balance of performance and redundancy ensuring data availability and faster read/write operations.

```
mdadm — create /dev/md0 — level=10 — raid-devices=4 /dev/sda /dev/sdb /dev/sdc /dev/sdd
```

5. Leveraging PostgreSQL Extensions

5.1 pg_stat_statements

This extension provides detailed statistics on SQL queries helping identify performance bottlenecks. Below is an example that shows how to enable and configure it to track query statistics.

```
shared_preload_libraries = ‘pg_stat_statements’
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

Using the following query we can get some useful data:

```
SELECT query, calls, total_time, mean_time, rows
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 28;
```

5.2 HypoPG

HypoPG is commonly used to create hypothetical indexes and test their impact on query performance, this is one of my favorite extensions because it helps evaluate potential performance improvements without affecting the actual database schema.

```
CREATE EXTENSION hypopg;
SELECT * FROM hypopg_create_index('CREATE INDEX ON freebies (category)');
EXPLAIN SELECT * FROM freebies WHERE category = 'tech';
```

6. Advanced Data Management

6.1 Partitioning Strategies

Partitioning large tables can significantly enhance query performance by reducing the amount of data scanned. Thus, some of the advisable techniques are to use:

- **Range Partitioning:** divides data based on a range of values such as dates. For instance, in an e-commerce website orders can be partitioned by year. Thus, the queries filtering by date can directly access the relevant partition which reduces scan times.

```
CREATE TABLE orders (
  order_id int,
  order_date date,
  amount numeric
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2023 PARTITION OF orders FOR VALUES FROM ('2022-12-28') TO ('2023-12-
28');
CREATE TABLE orders_2024 PARTITION OF orders FOR VALUES FROM ('2023-12-28') TO ('2024-12-
28');
```

- **List Partitioning:** segments data by specific values which is ideal for categorical data like status codes. For instance, in an e-commerce website orders can be partitioned by order status. Thus, the queries targeting a specific status access only the corresponding partition.

```
CREATE TABLE orders (
  order_id int,
  order_status text,
  amount numeric
) PARTITION BY LIST (order_status);

CREATE TABLE orders_new PARTITION OF orders FOR VALUES IN ('new');
CREATE TABLE orders_shipped PARTITION OF orders FOR VALUES IN ('shipped');
CREATE TABLE orders_cancelled PARTITION OF orders FOR VALUES IN ('cancelled');
```

- **Hash Partitioning:** distributes data evenly using a hash function on a key column, such as customer ID..etc which ensures balanced data distribution across partitions that is beneficial for evenly distributed workloads.

```
CREATE TABLE customers (
  customer_id int,
  customer_name text,
  email text
) PARTITION BY HASH (customer_id);

CREATE TABLE customers_part_1 PARTITION OF customers FOR VALUES WITH (MODULUS 4, REMAINDER
0);
CREATE TABLE customers_part_2 PARTITION OF customers FOR VALUES WITH (MODULUS 4, REMAINDER
1);
CREATE TABLE customers_part_3 PARTITION OF customers FOR VALUES WITH (MODULUS 4, REMAINDER
2);
CREATE TABLE customers_part_4 PARTITION OF customers FOR VALUES WITH (MODULUS 4, REMAINDER
3);
```

6.2 Archive and Cleanup

Regularly archive old data and clean up unnecessary records to maintain optimal performance and manage disk space efficiently. Thus, consider setting up policies for data archiving and automating the process using triggers or cron jobs. For instance, take a look at this example:

```
CREATE FUNCTION archive_old_data() RETURNS void AS $$
BEGIN
INSERT INTO archive_table
SELECT * FROM stock_move_line WHERE create_date < CURRENT_DATE - INTERVAL '1 year';
DELETE FROM stock_move_line WHERE create_date < CURRENT_DATE - INTERVAL '1 year';
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION trigger_archive_old_data() RETURNS trigger AS $$
BEGIN
PERFORM archive_old_data();
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- This can also be done using a cron job
CREATE TRIGGER archive_trigger
AFTER INSERT ON stock_move_line
FOR EACH ROW EXECUTE FUNCTION trigger_archive_old_data();
```