PostgreSQL Performance: Real-Life Scenarios & Solutions

Table Of Contents

Intro

We all know theory is different than practice, this article provides real-life situations that you can face with PostgreSQL performance and how to solve them.

1. Handling High Read & Write Throughput

Scenario

An e-commerce website experiences high read and write throughput, especially during peak shopping seasons like Christmas. The database should be able to handle thousands of transactions per second without degrading performance.

Solution

- **Connection Pooling**: use *PgBouncer* to manage and reuse database connections efficiently.

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

[pgbouncer]
listen_addr = 127.0.0.1
listen port = 6432
auth_type = md5
auth file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 500
default_pool_size = 50
```

- **Partitioning**: implement range partitioning on transaction date to optimize read and write operations.

```
CREATE TABLE transactions (
id serial PRIMARY KEY,
user_id int,
product_id int,
transaction_date date,
amount numeric
) PARTITION BY RANGE (transaction_date);

CREATE TABLE transactions_2024 PARTITION OF transactions FOR VALUES FROM ('2024–08–28') TO ('2025–08–28');
```

- **Indexing**: create appropriate indexes to speed up frequent queries.

```
CREATE INDEX idx_user_product ON transactions (user_id, product_id);
```

2. Slow Query Performance

Scenario

A SaaS application reports slow performance for several key analytical queries affecting the user experience.

Solution

- **Query Optimization**: analyze and rewrite inefficient queries using *EXPLAIN ANALYZE* to understand execution plans.

```
EXPLAIN ANALYZE
SELECT user_id, COUNT(*)
FROM user activity
WHERE activity_date > CURRENT_DATE — INTERVAL '1 month'
GROUP BY user_id;
```

- **Hypothetical Indexing**: use the HypoPG extension to test the impact of new indexes without actually creating them.

```
CREATE EXTENSION hypopg;
SELECT * FROM hypopg create index('CREATE INDEX ON user activity (activity date)');
EXPLAIN SELECT user_id, COUNT(*) FROM user_activity WHERE activity_date > CURRENT_DATE —
INTERVAL '1 month' GROUP BY user_id;
```

- **Materialized Views**: create materialized views to pre-compute expensive joins and aggregations.

```
CREATE MATERIALIZED VIEW monthly user activity AS
SELECT user_id, COUNT(*) AS activity_count
FROM user activity
WHERE activity_date > CURRENT_DATE — INTERVAL '1 month'
GROUP BY user_id;
```

3. Managing Large Data Volumes

Scenario

A financial services company needs to manage and analyze terabytes of historical transaction data to close the fiscal year while maintaining performance.

Solution

- **Data Archiving**: regularly archive old data to separate storage systems to keep the primary database lean.

```
CREATE TABLE transactions_archive (LIKE transactions INCLUDING ALL);

INSERT INTO transactions_archive
SELECT * FROM transactions
WHERE transaction date < CURRENT DATE - INTERVAL '1 year';
DELETE FROM transactions
WHERE transaction_date < CURRENT_DATE - INTERVAL '1 year';
```

- **Advanced Partitioning**: use list partitioning to segment data by year.

```
CREATE TABLE transactions (
 id serial PRIMARY KEY,
 user id int,
 product_id int,
 transaction date date,
 amount numeric
) PARTITION BY LIST (EXTRACT(YEAR FROM transaction date));

CREATE TABLE transactions_2024 PARTITION OF transactions FOR VALUES IN (2024);
```

- **Parallel Query Execution**: enable and configure parallel query execution for large-scale data processing.

```
SET max_parallel_workers_per_gather = 4;
SET max_parallel_workers = 8;
```

4. Addressing Lock Contention

Scenario

A content management system or simply CMS, experiences frequent lock contention causing delays and timeouts during peak usage.

Solution

- **Monitoring Locks**: use *pg_stat_activity* and *pg_locks* to identify and analyze lock contention.

```
SELECT pid, locktype, relation::regclass, mode, granted
FROM pg_locks
JOIN pg stat activity ON pg locks.pid = pg stat activity.pid
WHERE NOT granted;
```

- **Optimizing Transactions**: break long transactions into smaller units to reduce lock durations.

```
BEGIN;
UPDATE articles SET content = '…' WHERE id IN (1, 2, 3);
COMMIT;

BEGIN;
UPDATE articles SET content = '…' WHERE id IN (4, 5, 6);
COMMIT;
```

- **Index Maintenance**: regularly reindex to ensure indexes are efficient and reduce the likelihood of lock contention.

```
REINDEX TABLE articles;
```

5. High Latency in Replication

Scenario

A universal application with multiple data centers requires low-latency replication between primary and standby servers.

Solution

- **Synchronous Replication**: configure synchronous replication for critical data to ensure data consistency across regions.

```
ALTER SYSTEM SET synchronous_standby_names = 'node1, node2';
SELECT pg_reload_conf();
```

- **Replication Slots**: use replication slots to ensure data is not lost due to replication lag.

```
SELECT * FROM pg_create_physical_replication_slot('replication_slot1');
```

- **Network Optimization**: optimize TCP settings to reduce replication latency.

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