Improving PostgreSQL Performance with Partitioning

My recommended methodology for performance improvement of PostgreSQL starts with query optimization. The second step is architectural improvements, part of which is the partitioning of large tables.

Partitioning in PostgreSQL is one of those advanced features that can be a powerful performance booster. If your PostgreSQL tables are becoming very large and sluggish, partitioning might be the cure.

Table of Contents

The Big Table Problem

Large tables tend to grow uncontrollably, especially in OLTP or time-series workloads. As millions or billions of rows accumulate, you begin to notice:

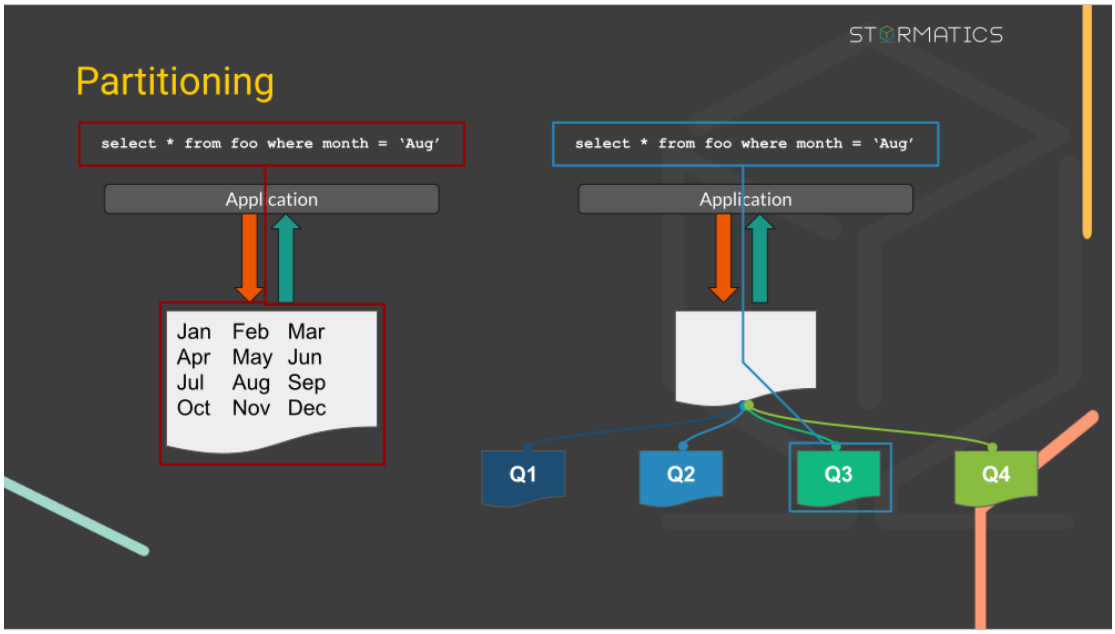
- **Slow queries** due to full table scans or massive indexes.
- **Heavy I/O usage**, especially when indexes cannot fit in memory.
- **Bloated memory** during operations like sorting or joining.
- **Increased maintenance cost**, with longer VACUUM, ANALYZE, and REINDEX times.
- **Hard-to-manage retention policies**, as purging old rows becomes expensive.
  These problems are amplified in cloud-hosted databases, where every IOPS, GB, or CPU upgrade increases cost.

What is Partitioning in PostgreSQL?

Partitioning is the process of dividing a large table into smaller, more manageable sub-tables called *partitions*, based on a defined key – called the partitioning key (e.g., date, category, or ID).

In PostgreSQL:

- Each partition is a real table.
- A *partitioned table* is a logical wrapper that unifies all partitions.
- Queries see a single table view, but under the hood, PostgreSQL targets individual partitions using a concept called *partition pruning*.



Benefits at a Glance

Benefit	How It Helps
Query performance	Scans only relevant partitions
Index efficiency	Smaller, faster per-partition indexes
Easier bulk operations	Load/delete on a single partition
Lower maintenance effort	Vacuum/analyze partitions separately
Archival simplicity	Drop old partitions instantly
Cost savings	Keep hot data on fast storage; move cold data

Partitioning Strategies in PostgreSQL

1. Range Partitioning

Range partitioning splits a table into partitions based on a continuous interval of values from a partition key, most commonly a date or numeric column. Each partition handles a specific range, with no overlap between them. This strategy is ideal for time-series or sequential data, where new data arrives chronologically and older data becomes less relevant over time.

It is one of PostgreSQL's most commonly used partitioning strategies due to its simplicity and strong alignment with real-world use cases like event logging, order tracking, and financial transactions.

This strategy is also my personal favorite!

How It Works

When a query filters on a range that overlaps with a specific partition's boundary, PostgreSQL will prune all other partitions, scanning only the relevant ones. This drastically reduces the number of rows scanned and improves query performance.

Use Case

Time-series data (logs, events), auto-increment IDs.

```
CREATE TABLE events (

id          BIGSERIAL PRIMARY KEY,

event_time  TIMESTAMPTZ NOT NULL,

payload     JSONB

) PARTITION BY RANGE (event_time);

-- Partition by year

CREATE TABLE events_2023 PARTITION OF events

FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE events_2024 PARTITION OF events

FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE events_2025 PARTITION OF events

FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
```

2. List Partitioning

List partitioning divides a table into partitions based on a discrete list of values from the partition key. Each partition holds rows that match one or more specific values, making it ideal for categorical data that does not follow a natural sequence or range.

Unlike range partitioning, where values are split by intervals, list partitioning is about grouping specific values, such as geographic regions, product types, or tenant identifiers.

How It Works

When a query filters using the exact values defined in the partition list, PostgreSQL applies partition pruning and accesses only the relevant partitions, bypassing the rest.

Use Case

Customer regions, product categories, tenant IDs.

```
CREATE TABLE orders (

id        SERIAL,

region    TEXT,

amount    NUMERIC

) PARTITION BY LIST (region);

CREATE TABLE orders_apac PARTITION OF orders

FOR VALUES IN ('Asia', 'Australia', 'New Zealand');

CREATE TABLE orders_emea PARTITION OF orders
```

```
FOR VALUES IN ('Europe', 'Middle East', 'Africa');
```

3. Hash Partitioning

Hash partitioning distributes rows across a fixed number of partitions using a hash function on the partition key. This strategy is best suited for datasets where there is no obvious natural range or finite list, but where you still want to distribute data evenly to avoid hotspots.

It is particularly useful in scenarios with uniform access patterns and large volumes of inserts, such as event logging, user activity tracking, or telemetry systems with millions of users.

How It Works

PostgreSQL computes a hash of the partition key and places the row into a partition based on the result's remainder modulo the number of partitions. Each partition gets approximately the same number of rows, ensuring even data and workload distribution.

Use Case

Uniformly spread customer IDs, UUIDs.

```
CREATE TABLE logs (

id      UUID,

message TEXT

) PARTITION BY HASH (id);

CREATE TABLE logs_0 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE logs_1 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE logs_2 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE logs_3 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

4. Composite Partitioning

Composite partitioning (also known as *sub-partitioning*) combines two partitioning strategies to create a two-level hierarchy of partitions. This is useful when a single partitioning method is not sufficient to manage complex data organization or high data volume.

It is commonly used in large-scale applications where datasets are naturally segmented along multiple dimensions, such as time and customer, or region and event type.

How It Works

The parent table is partitioned by one key, and each child partition is further partitioned by another key. This creates a tree of partitions where PostgreSQL can prune at both levels for more efficient queries.

Example Use Case

A SaaS platform that stores time-series data for many tenants.

```
-- Step 1: Create the top-level partitioned table (by date)

CREATE TABLE metrics (

recorded_at DATE NOT NULL,
```

```
tenant_id   INT NOT NULL,

value       NUMERIC

) PARTITION BY RANGE (recorded_at);

-- Step 2: Create a sub-table partitioned by tenant ID

CREATE TABLE metrics_2024 PARTITION OF metrics

FOR VALUES FROM ('2024-01-01') TO ('2025-01-01')

PARTITION BY LIST (tenant_id);

-- Step 3: Define sub-partitions per tenant

CREATE TABLE metrics_2024_tenant_1 PARTITION OF metrics_2024

FOR VALUES IN (1);

CREATE TABLE metrics_2024_tenant_2 PARTITION OF metrics_2024

FOR VALUES IN (2);
```

Composite partitioning is powerful but should be used with care. It introduces more complexity and administrative overhead, so it is best suited for high-scale systems where the data naturally aligns with multiple dimensions.

Real-World Performance Gains

PostgreSQL supports partition pruning, which skips scanning irrelevant partitions when a WHERE clause filters by the partition key.

Consider this query:

```
SELECT * FROM events WHERE event_time BETWEEN '2025-01-01' AND '2025-01-31';
```

If the events table is partitioned by month, only the partition for January 2025 is scanned. The rest are ignored.

Benefits in Practice

- **Faster Queries**
  When querying across multiple partitions, PostgreSQL may only access 1 relevant partition instead of scanning an entire monolithic table.
- **Parallelism**
  PostgreSQL can scan multiple partitions in parallel using separate workers.
- **Partition-wise JOINs and Aggregates**
  Smaller chunks of data mean faster processing with lower memory usage.

Maintenance, Backups, and Cost Optimization

One of the biggest advantages of partitioning in PostgreSQL, beyond query performance, is how it simplifies ongoing maintenance. Large tables are hard to manage. Operations like vacuum, analyze, reindexing, and backups get slower and riskier as the table grows. Partitioning breaks that problem down into smaller, manageable pieces.

Maintenance

Each partition is treated like a standalone table. That means you can vacuum, analyze, reindex, or cluster one partition at a time, without touching the rest. It is faster, safer, and creates less pressure on the database.

If your table has billions of rows, you do not want to reindex all of it just because one part is bloated. With partitioning, you can isolate the maintenance to just that chunk.

```
-- Reindex just the active partition

REINDEX TABLE transactions_2025;
```

This also helps with locking. A vacuum or reindex on a large monolithic table can block concurrent access or slow things down. When you target just one partition, the impact on live queries is much lower.

On top of that, PostgreSQL keeps planner statistics per partition. That leads to better query plans, especially if your partitions differ significantly in size or data distribution.

You can also parallelize your maintenance jobs. Run analyze on all recent partitions in parallel, or automate them using pg_cron or your scheduler of choice. It is routine work, and partitioning makes it less painful.

Backups and Restores

Backup strategies are easier to manage when your data is partitioned by time or category. You can:

- Back up only the recent partitions that are actively changing.
- Skip old partitions entirely or move them to archival storage.
- Restore just the hot data if needed, instead of the entire dataset.
  For example:

```
-- Detach old data and move it to cheaper storage

ALTER TABLE transactions DETACH PARTITION transactions_2024;

COPY transactions_2024 TO '/mnt/cold_storage/transactions_2024.csv';
```

That keeps your production database light and responsive, and gives you more control over what goes into your backups.

Cost Savings in the Cloud

In managed services like AWS RDS, Azure Flexible Server, or Google Cloud SQL, storage and IOPS are not free. Partitioning helps reduce the cost footprint by:

- Scanning less data = fewer IOPS.
- Keeping hot data small = smaller instance types.
- Offloading cold data = cheaper long-term storage.
  You are not forced to vertically scale every time your data grows. If most of your queries target recent data, and your partitioning strategy matches that, you can stay lean and efficient for a lot longer.

You also avoid long-running DELETE operations when purging old data. Just drop the partition. It is an instant metadata operation, not a row-by-row delete.

```
-- Drop last year’s data in seconds

DROP TABLE transactions_2024;
```

Partitioning gives you a clear boundary between hot and cold data. That makes it easier to tune, easier to scale, and easier to manage regardless of whether you are running Postgres on-prem or in the cloud.

Case Study: Capital One

I came across a very interesting post by Capital One. They use PostgreSQL to manage massive volumes of customer data. To serve recent transactions in milliseconds, they:

- Partitioned tables by event date.
- Placed recent data in dedicated partitions with customer ID indexes.
- Rarely accessed old partitions except for audits.
  As a result …

- 99.9% of recent data queries responded in milliseconds.
- Partitioning + indexing outperformed NoSQL alternatives.
- Maintenance and data retention were simple—drop a partition instead of running DELETE queries.
  Read more: Capital One's PostgreSQL Partitioning Tips

Best Practices for Partitioning in PostgreSQL

For teams that want hands-off automation of partition creation, pg_partman is my go-to tool. It simplifies recurring partition management, especially in time-series systems. Some other best practices are listed below:

1. Use the Partition Key in Queries

Partition pruning only works if your WHERE clause includes the partition key.

```
-- Good

SELECT * FROM events WHERE event_time >= '2024-01-01';

-- Bad (no pruning)

SELECT * FROM events WHERE id = 1234567890;
```

2. Pre-Create Partitions

Use tools like pg_partman or scheduled scripts to create partitions ahead of time.

```
# Cron job to create next year's partition

psql -c "CREATE TABLE events_2026 PARTITION OF events FOR VALUES FROM ('2026-01-01') TO
('2027-01-01');"
```

3. Monitor with EXPLAIN

Check if partition pruning is working.

```
EXPLAIN SELECT * FROM events WHERE event_time BETWEEN '2024-01-01' AND '2024-01-31';
```

Look for lines like: Seq Scan on events_2024

When Partitioning Backfires: The Cost of Over-Partitioning

Partitioning is not free. PostgreSQL must:

- Track partition metadata during query planning.
- Route inserts to the correct partition.
- Handle more objects (e.g., more tables = more stats).

Do Not Over-Partition

Thousands of tiny partitions (e.g., one per hour or customer) cause:

- **Query planning overhead**: Longer EXPLAIN times.
- **Higher memory usage**: Stats and locks on many tables.
- **Slow inserts**: Routing checks many partitions.
- **Unwieldy maintenance**: Difficult to monitor or index all.

The Sweet Spot

- Few dozen to a few hundred partitions: Reasonable size.
- Avoid partitions with <10,000 rows unless absolutely necessary.
- Monitor growth, and *merge* or *coalesce* partitions if needed.

Summary

Partitioning is one of PostgreSQL's most powerful tools when it comes to performance, scalability, and cost savings. But it must be implemented with care.

When It Works

- Tables with **billions of rows**
- Queries that filter on a **partition key**
- Maintenance and backups by **time or category**
- Cost-conscious deployments on **cloud platforms**

When It Hurts

- Too many **tiny partitions**
- Queries that do **not** filter by the partition key
- Need for **cross-partition constraints** or **global indexes**

Start small, plan for growth, monitor performance, and remember that partitioning is just one part of a larger performance strategy.