Table Partitioning in PostgreSQL

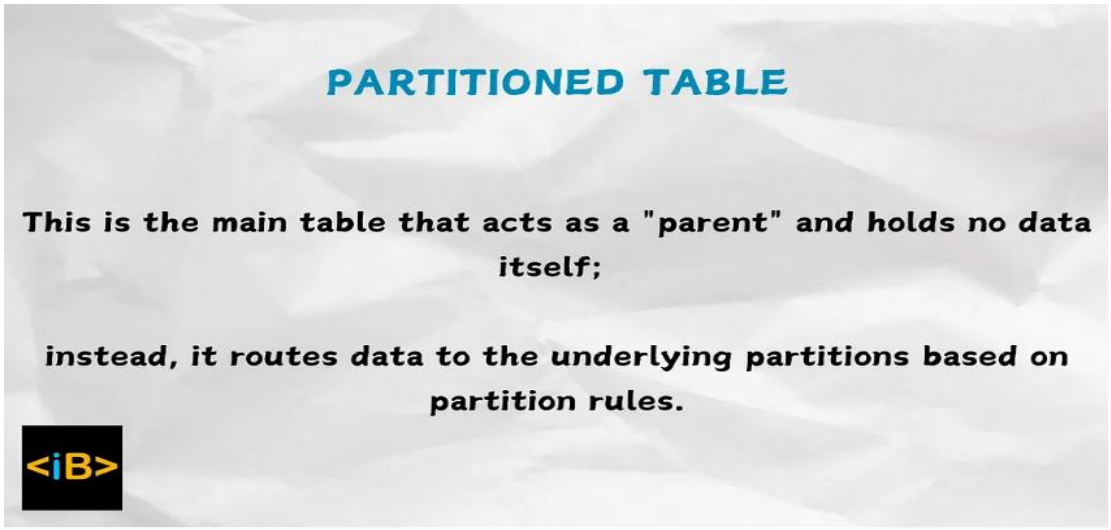
What is Table Partitioning?

Table partitioning is a way of dividing a large table into smaller, more manageable pieces called **partitions**. The main table acts as a “parent,” and the partitions, which are essentially child tables, store the actual data. Queries on the parent table are routed to the relevant partitions, reducing query execution time and optimizing resource usage.

Partitioning is particularly useful for:

- Large datasets that grow over time (e.g., logs, sales records).
- Queries that focus on a subset of data (e.g., a specific date range).
- Managing and archiving historical data.

What is a Partitioned Table?



Partitioned Table in PostgreSQL

What is a Child Table?

Child Table in PostgreSQL

Types of Table Partitioning

PostgreSQL supports three primary types of partitioning:

Range Partitioning: Divides data into ranges based on a partition key (e.g., date or numeric ranges).

- Example: Sales data partitioned by months.

List Partitioning: Divides data based on specific values in a partition key (e.g., categories or regions).

- Example: Data partitioned by geographic regions.

Hash Partitioning: Divides data based on the hash value of a partition key, ensuring even distribution.

- Example: Customers distributed across multiple partitions based on hashed customer IDs.

1. Range Partitioning

Range partitioning is ideal when your data is continuously growing over time. For example, consider an orders table where each order has an `order_date`. By partitioning data based on date ranges, queries can quickly target specific segments.

Step-by-Step Implementation

First, create and populate a simple `orders` table:

```
CREATE TABLE orders (
    order_id SERIAL,
    customer name TEXT NOT NULL,
    order_date DATE NOT NULL
);


INSERT INTO orders (customer name, order date) VALUES
('Ali', '2023-03-15'),
('Ayşe', '2023-07-20'),
('Mehmet', '2024-01-05'),
('Zeynep', '2024-06-12'),
('Hasan', '2025-02-28');
```

Next, rename the original table to prepare for partitioning:

```
ALTER TABLE orders RENAME TO orders_old;
```

Now, create a new partitioned table for orders using range partitioning:

```
CREATE TABLE orders (
    order_id SERIAL,
    customer_name TEXT NOT NULL,
    order_date DATE NOT NULL
) PARTITION BY RANGE (order_date);
```

Create partitions for specific year ranges:

```
CREATE TABLE orders_2023 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_2024 PARTITION OF orders
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE orders_2025 PARTITION OF orders
    FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
```

Migrate data from the old table into the new partitioned structure:

```
INSERT INTO orders (order_id, customer_name, order_date)
SELECT order_id, customer_name, order_date FROM orders_old;
```

You can then verify each partition:

```
test=# SELECT * FROM orders_2023;
 order_id | customer_name | order_date
----------+---------------+------------
        1 | Ali           | 2023-03-15
        2 | Ayşe          | 2023-07-20
(2 rows)

test=# SELECT * FROM orders_2024;
 order_id | customer_name | order_date
----------+---------------+------------
        3 | Mehmet        | 2024-01-05
        4 | Zeynep        | 2024-06-12
(2 rows)

test=# SELECT * FROM orders_2025;
 order_id | customer_name | order_date
----------+---------------+------------
        5 | Hasan         | 2025-02-28
(1 row)

test=# select * from orders;
 order_id | customer_name | order_date
----------+---------------+------------
        1 | Ali           | 2023-03-15
        2 | Ayşe          | 2023-07-20
        3 | Mehmet        | 2024-01-05
        4 | Zeynep        | 2024-06-12
        5 | Hasan         | 2025-02-28
(5 rows)
```

This approach allows PostgreSQL to efficiently route queries based on the order date.

2. List Partitioning

List partitioning works well when you need to separate data by specific, discrete values. In our example, we'll create an `employees` table partitioned by department.

Example Setup

First, define the table with a composite primary key:

```
CREATE TABLE employees (
    emp id SERIAL,
    name TEXT NOT NULL,
    department TEXT NOT NULL,
    PRIMARY KEY (emp_id, department)
) PARTITION BY LIST (department);
```

Next, create partitions for specific departments:

```
CREATE TABLE employees hr PARTITION OF employees
    FOR VALUES IN ('HR');

CREATE TABLE employees_it PARTITION OF employees
    FOR VALUES IN ('IT');

CREATE TABLE employees_sales PARTITION OF employees
    FOR VALUES IN ('Sales');

CREATE TABLE employees_other PARTITION OF employees
    DEFAULT;
```

After inserting sample data into each partition, you can verify the contents by running:

```
SELECT * FROM employees hr;
SELECT * FROM employees it;
SELECT * FROM employees_sales;
SELECT * FROM employees_other;
SELECT * FROM employees;
```

This method keeps departmental data neatly organized, boosting query performance for department-specific requests.

3. Hash Partitioning

Hash partitioning distributes data evenly across a number of partitions based on a hash function, which is particularly useful when you require balanced data distribution.

Implementation Example

Start by creating a basic `students` table partitioned by hash:

```
CREATE TABLE students (
    id SERIAL,
    y TEXT
) PARTITION BY HASH (id);
```

Then, define partitions with a modulus to ensure even distribution:

```
CREATE TABLE students 0 PARTITION OF students FOR VALUES WITH (MODULUS 5, REMAINDER 0);
CREATE TABLE students_1 PARTITION OF students FOR VALUES WITH (MODULUS 5, REMAINDER 1);
CREATE TABLE students 2 PARTITION OF students FOR VALUES WITH (MODULUS 5, REMAINDER 2);
CREATE TABLE students_3 PARTITION OF students FOR VALUES WITH (MODULUS 5, REMAINDER 3);
CREATE TABLE students_4 PARTITION OF students FOR VALUES WITH (MODULUS 5, REMAINDER 4);
```

Populate the table:

```
INSERT INTO students (y)
SELECT 'Row ' || generate_series(1, 100);
```

This ensures your data is balanced across the five partitions.

4. Automating Partitioning with pg_partman

For those looking to automate partition management in PostgreSQL, **pg_partman** is a must-have extension. It simplifies the process of creating, maintaining, and managing partitions — especially in environments with continuously growing data.

Setting Up pg_partman

Before starting, ensure you have the necessary contrib packages installed:

```
dnf install pg_partman_13
```

Then, create the schema and extension:

```
CREATE SCHEMA IF NOT EXISTS partman;

CREATE EXTENSION pg_partman SCHEMA partman;
```

Native Partitioning Example with pg_partman

Let's partition a `store.transactions` table by range, based on `sale_date`:

```
CREATE TABLE store.transactions (
    id SERIAL NOT NULL,
    sale_date TIMESTAMP NOT NULL DEFAULT now(),
```

```
    amount NUMERIC NOT NULL,
    PRIMARY KEY (id, sale_date)
) PARTITION BY RANGE (sale_date);
```

Now, use pg_partman to automatically create monthly partitions:

```
SELECT partman.create_parent(
    p_parent_table := 'store.transactions',
    p_control := 'sale_date',
    p_type := 'native',
    p_interval := '1 month'
);
```

Insert some sample transactions:

```
INSERT INTO store.transactions (sale_date, amount)
VALUES
    ('2024-11-01', 100),
    ('2024-12-01', 150),
    ('2024-03-01', 200),
    ('2024-04-01', 250),
    ('2024-05-01', 300),
    ('2024-06-01', 350);
```

Finally, verify partition creation with:

```
SELECT * FROM partman.show_partitions('store.transactions');
```

You can also inspect individual partitions:

```
SELECT * FROM store.transactions_p2024_11;
SELECT * FROM store.transactions_p2024_12;
```

Why pg_partman Stands Out

pg_partman dramatically reduces the overhead of manual partition management. It automatically creates new partitions as data grows, handles maintenance tasks, and ensures that your database remains performant over time. This is especially crucial for applications with high data ingestion rates and time-sensitive information.

Benefits of Table Partitioning

Performance Optimization:

- Queries on partitioned tables are faster because PostgreSQL scans only the relevant partitions.

Efficient Data Management:

- Partitions can be managed independently (e.g., archived, dropped, or vacuumed).

Scalability:

- You can add new partitions as your data grows, avoiding the need to restructure existing data.

Indexing:

- Each partition can have its own indexes, tailored to the data it holds.

Note :-

✅ If you are using pg_partman, the next month's partition will be created automatically.
❌ If you are using manual partitioning, you need to create partitions yourself.