In this PDF, you can learn the steps and use cases of database partitioning. I have provided the queries along with detailed, step-by-step instructions. Please read carefully and dive deep into each step for a comprehensive understanding.

This practice has been done in PostgreSQL; you can do it in any Structured Query Language

Definition of Database Partitioning in PostgreSQL (As a PostgreSQL Developer)

Database partitioning is the process of splitting large tables into smaller, more manageable pieces, called partitions, while still maintaining them as a single logical entity. In PostgreSQL, partitioning is a way to distribute the data across different physical storage locations based on specific criteria (e.g., range of values, list of values, or hash distribution). The goal of partitioning is to improve query performance, data management, and scalability for large datasets.

There are three primary types of partitioning strategies in PostgreSQL:

1. **Range Partitioning**: Data is divided into partitions based on a specified range of values. This is commonly used when working with date, time, or numeric columns. For example, partitioning sales data by year (2020, 2021, etc.) allows for more efficient querying of data within specific time ranges.
2. **List Partitioning**: Data is divided into partitions based on predefined lists of values. For instance, partitioning data by specific categories such as product types or regions, where each partition contains only a subset of the data that belongs to one of the predefined categories.
3. **Hash Partitioning**: Data is distributed across partitions based on a hash function applied to a specific column (e.g., customer ID or order ID). This is useful when there is no natural range or list for partitioning but you want to evenly distribute data across partitions.

Benefits of Partitioning in PostgreSQL:

- **Improved Query Performance**: Queries that filter on the partition key can skip irrelevant partitions (partition pruning), leading to faster query execution.
- **Efficient Data Management**: Partitioning makes it easier to manage and archive data, as entire partitions can be added, dropped, or archived without affecting other partitions.
- **Scalability**: As data grows, partitions allow for more efficient indexing, backups, and storage management, preventing a single large table from becoming a performance bottleneck.
- **Improved Maintenance**: Large tables can be easier to maintain when they are partitioned, as maintenance operations like vacuuming, indexing, and backups can be performed on individual partitions instead of the entire table.

IMPLEMENTATION:

1. **Range Partitioning**: Let see you have the **sales (table)** data and you want to do the partition based on Year for better query optimization.

   Query for partition based on year

```
CREATE TABLE sales_2023 PARTITION OF sales
FOR VALUES FROM ('2023-01-01') TO ('2023-12-31');

CREATE TABLE sales_2024 PARTITION OF sales
FOR VALUES FROM ('2024-01-01') TO ('2024-12-31');
```

   Explanation : sales_2023 partition name

   sales table name

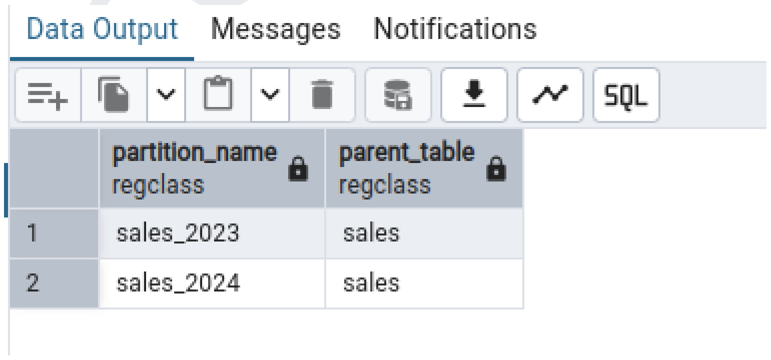
   FROM (DATE)TO (DATE) is range for the sales_2023 partition

Output: Your sales table partition has been completed

See the partition list name

```
SELECT inhrelid::regclass AS partition_name, inhparent::regclass AS parent_table
FROM pg_inherits
WHERE inhparent = 'sales'::regclass;
```

Output: Now you can see the Below partition_name



	partition_name regclass	parent_table regclass
1	sales_2023	sales
2	sales_2024	sales

Yeh.. now you can run the query based on partition name and see the result:

```
SELECT * FROM sales_2024;
```

Output: You will get the data for the year 2024 by writing a small query in less time.

If you want to get data for two partitions or two years, you just need to run the query below.

```
SELECT *
FROM sales_2023
UNION ALL
SELECT *
FROM sales_2024;
```

Output: You will get the both year of or both partitions data

Advance 👍 **If you want to partition data for N number of years, you can define a dynamic method to create partitions in the database based on the year range. Find the dynamic method below for creating dynamic partitions.**

```
CREATE OR REPLACE FUNCTION create_partitions(start_year INT, end_year INT)
RETURNS VOID AS
$$
DECLARE
    year INT;
    partition_name TEXT;
BEGIN
    FOR year IN start_year..end_year LOOP
        partition_name := 'sales_' || year;
        EXECUTE format('
            CREATE TABLE %I PARTITION OF sales
            FOR VALUES FROM (''%s-01-01'') TO (''%s-01-01'')',
            partition_name,
            year,
            year + 1
        );
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Output: After creating the method or function, you just need to call the function by passing the year range, from year to year, as shown below

```
SELECT create_partitions(2020, 2024);
```

Output: It will create the partition for each year from 2020 to 2024

2. **List Partitioning**: Data is divided into partitions based on predefined lists of values. For instance, partitioning data by specific categories such as product types or regions, where each partition contains only a subset of the data that belongs to one of the predefined categories.

   In the Above example we saw the partition based on Years now we are going to create the partition based on **specific categories like region:**

Let see in the same table sale_order we have the Region colum we need to do the partition for that you can hit the below query:

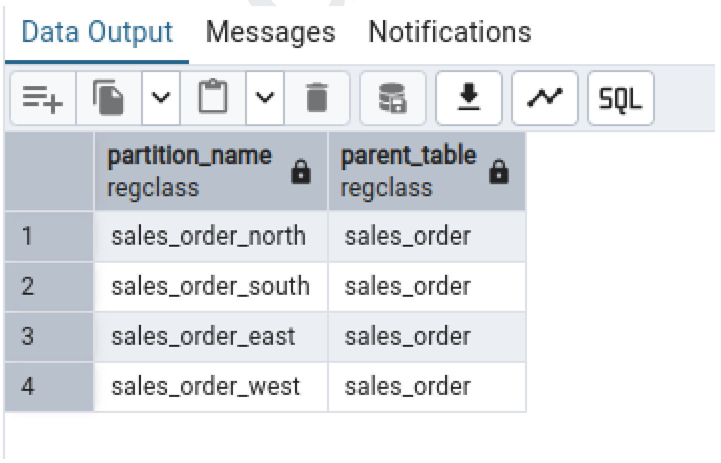
```
CREATE TABLE sales_order_north PARTITION OF sales_order
    FOR VALUES IN ('North');

-- Create partition for 'South' region
CREATE TABLE sales_order_south PARTITION OF sales_order
    FOR VALUES IN ('South');

-- Create partition for 'East' region
CREATE TABLE sales_order_east PARTITION OF sales_order
    FOR VALUES IN ('East');

-- Create partition for 'West' region
CREATE TABLE sales_order_west PARTITION OF sales_order
    FOR VALUES IN ('West');
```

Output: Successfully we created the partitions based on region like North, South, east and West

You can check the partition name with below query

```
SELECT inhrelid::regclass AS partition_name, inhparent::regclass AS parent_table
FROM pg_inherits
WHERE inhparent = 'sales_order'::regclass;
```

Output:



	partition_name regclass	parent_table regclass
1	sales_order_north	sales_order
2	sales_order_south	sales_order
3	sales_order_east	sales_order
4	sales_order_west	sales_order

Now try Hash Partitioning from your side and let me know if you are facing an issue.

Email : rahulsinghyadav64@gmail.com