How To Partition PostgreSQL Database

Scripts And Commands with Examples

Asfaw Gedamu

Standard Operating Procedure (SOP) for PostgreSQL Database Partitioning

Database partitioning refers to dividing a database into smaller, manageable units based on certain criteria. It offers several benefits, including improved performance, scalability, and manageability. Here's a breakdown of the different types of database partitioning:

Purpose: This SOP outlines the steps for partitioning PostgreSQL databases using different techniques, including horizontal, vertical, hybrid, functional, and composite partitioning.

Target Audience: PostgreSQL database administrators and developers.

Prerequisites:

- Familiarity with PostgreSQL commands and concepts.
- Access to a PostgreSQL database with administrator privileges.
- Script editor and command-line interpreter.

1. General Steps:

- **Planning:** Analyze your database structure, access patterns, and performance bottlenecks to determine the optimal partitioning strategy.
- **Pre-processing:** Create temporary tables or backup your data before performing any partitioning operations.
- **Partitioning:** Execute the relevant script for your chosen partitioning type.
- **Validation:** Verify the validity and functionality of the partitioned database.
- **Maintenance:** Schedule maintenance tasks like monitoring, reorganizing, and adding/dropping partitions as needed.

2. Partitioning Types:

2.1. Horizontal Partitioning:

- Divides data rows across multiple databases or tables based on a specific key, like customer ID or date range.
- Ideal for large datasets, distributing workload across servers, and enabling parallel queries.
- Popular variations include range partitioning (by value ranges) and list partitioning (by specific values).

SQL Script:

```
CREATE TABLE orders_horiz (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product_id INTEGER,
```

```
  order_date DATE
) PARTITION BY RANGE (order_date) (
  VALUE LESS THAN '2023-01-01' STORE IN PARTITION orders_2022,
  VALUE LESS THAN '2024-01-01' STORE IN PARTITION orders_2023
);

INSERT INTO orders_horiz (order_id, customer_id, product_id, order_date)
VALUES (1, 123, 456, '2022-12-31');

INSERT INTO orders_horiz (order_id, customer_id, product_id, order_date)
VALUES (2, 456, 789, '2023-01-02');

SELECT * FROM orders_horiz WHERE order_date >= '2023-01-01';
```

Output: The first INSERT will fail as the date falls before the specified range in orders_horiz, while the second one will succeed. The SELECT will only return orders from 2023.

2.2. Vertical Partitioning:

- Splits data columns into separate tables based on logical units, like customer information and order details.
- Reduces storage footprint for frequently accessed columns and simplifies queries focused on specific data subsets.
- Less common than horizontal partitioning but advantageous for specific scenarios.

SQL Script:

```
CREATE TABLE customer_data (
  customer_id INTEGER PRIMARY KEY
);

CREATE TABLE order_details (
  order_id INTEGER REFERENCES orders(order_id) PRIMARY KEY,
  product_id INTEGER,
  order_date DATE
);

INSERT INTO customer_data (customer_id) VALUES (123);

INSERT INTO order_details (order_id, product_id, order_date) VALUES (1, 456, '2023-01-02');

SELECT * FROM customer_data c JOIN order_details od ON c.customer_id = od.order_id;
```

Output: Separate tables store customer and order details, improving access efficiency for specific data types.

2.3. Hybrid Partitioning:

- Combines both horizontal and vertical partitioning, further dividing data horizontally within vertically partitioned tables.
- Offers the most granular control and optimization for highly complex datasets with diverse access patterns.
- Requires careful planning and management due to its intricate structure.

SQL Script:

```
CREATE TABLE orders_hybrid (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product_id INTEGER,
  order_date DATE
) PARTITION BY RANGE (order_date) (
  VALUE LESS THAN '2023-01-01' STORE IN PARTITION orders_2022_europe (
    VALUES LESS THAN '2022-12-15' STORE IN PARTITION orders_2022_europe_west,
    VALUES LESS THAN '2022-12-31' STORE IN PARTITION orders_2022_europe_east
  ),
  VALUE LESS THAN '2024-01-01' STORE IN PARTITION orders_2023_america
);

INSERT INTO orders_hybrid (order_id, customer_id, product_id, order_date)
VALUES (1, 123, 456, '2022-12-10');

INSERT INTO orders_hybrid (order_id, customer_id, product_id, order_date)
VALUES (2, 456, 789, '2023-01-02');

SELECT * FROM orders_hybrid WHERE order_date >= '2023-01-01' AND customer_id = 123;
```

Output: This combines horizontal and vertical partitioning, with separate tables for each year and further

4. Functional Partitioning:

- Separates data based on its usage or function, like transactional versus analytical data.
- Enables independent scaling and resource allocation for different workloads.
- More common in data warehousing and business intelligence applications.

SQL Script:

```
-- Create tables for transactional and analytical data
CREATE TABLE orders_trans (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product_id INTEGER,
  order_date DATE
);

CREATE TABLE orders_analyt (
  order_id INTEGER PRIMARY KEY REFERENCES orders_trans(order_id),
  product_id INTEGER,
  order_date DATE,
  total_amount DECIMAL,
  region VARCHAR(50)
);

-- Insert data into appropriate tables based on usage
INSERT INTO orders_trans (order_id, customer_id, product_id, order_date)
VALUES (1, 123, 456, '2023-01-02');

INSERT INTO orders_analyt (order_id, product_id, order_date, total_amount, region)
VALUES (1, 456, '2023-01-02', 100.00, 'US');

-- Analyze performance for different types of queries
EXPLAIN SELECT * FROM orders_trans WHERE order_date = '2023-01-02';
EXPLAIN SELECT SUM(total_amount) FROM orders_analyt WHERE region = 'US';
```

Output: This example separates transactional data (orders_trans) from analytical data (orders_analyt) for distinct access patterns, optimizing both transactional processing and complex analytical queries.

5. Composite Partitioning:

- Utilizes multiple partitioning criteria simultaneously, like date and region.
- Provides extreme granularity and targeted data access for very large and complex datasets.
- Highly technical and requires advanced database administration skills.

SQL Script:

```
CREATE TABLE orders_comp (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product_id INTEGER,
  order_date DATE,
  region VARCHAR(50)
) PARTITION BY RANGE (order_date) (
  VALUE LESS THAN '2023-01-01' STORE IN PARTITION orders_2022 (
    VALUES LESS THAN '2022-12-15' STORE IN PARTITION orders_2022_west
  ),
  VALUE LESS THAN '2024-01-01' STORE IN PARTITION orders_2023
);

INSERT INTO orders_comp (order_id, customer_id, product_id, order_date, region)
VALUES (1, 123, 456, '2022-12-10', 'EU');

INSERT INTO orders_comp (order_id, customer_id, product_id, order_date, region)
VALUES (2, 456, 789, '2023-01-02', 'US');

SELECT * FROM orders_comp WHERE order_date >= '2023-01-01' AND region = 'US';
```

Output: This combines horizontal partitioning based on date with vertical partitioning for region, offering fine-grained control and efficient access for specific date and region combinations.

Note: Remember to adjust these scripts and commands to fit your specific data schema and partitioning needs. Consider carefully planning your partitioning strategy before implementation to ensure optimal performance and manageability.

Additional Factors:

- Partitioning strategies should be chosen based on specific database size, access patterns, and performance requirements.
- Effective partitioning requires careful planning, implementation, and ongoing maintenance.
- Modern database systems offer native partitioning functionalities and tools to simplify its implementation.

By understanding these different types of database partitioning, you can choose the best approach to optimize your database for improved performance, scalability, and manageability.

6. What are the main differences between Oracle DB partitioning and PostgreSQL DB partitioning?

The table below provides the main differences.

Table 1Differences between Oracle DB Partitioning and PostgreSQL DB Partitioning

Feature	Oracle DB	PostgreSQL
Partitioning types offered	Range, List, Hash, Composite (List-List, List-Range, List-Hash, Range-Range, Range-List, Range-Hash, Hash-Hash, Hash-List, Hash-Range)	Range, List, Hash, Composite (multiple levels supported)
Declarative vs. procedural partitioning	Procedural (requires creating tables for each partition)	Declarative (partitions defined within the table definition)
Global vs. local indexes	Supports global indexes across partitions	Generally relies on local indexes, although global indexes possible with partitioned tables
Partition management tools	DBMS_REPAIR, DBMS_PARTITIONS	pg_partman, ALTER TABLE
Partition exchange and split	Supported	Not natively supported, requires manual data manipulation
Foreign key constraints	Supported on parent table in some cases	Not supported on parent table
Performance comparison	Generally considered faster for OLTP workloads	Can be faster for large datasets and specific queries
Complexity	More complex setup and management due to procedural nature	Simpler due to declarative nature

Additional Notes:

- PostgreSQL supports nested partitioning (multiple levels) while Oracle supports composite partitioning with predefined combinations.

- Oracle offers more partitioning types like Hash and List-Hash, but PostgreSQL's declarative approach simplifies management.
- Oracle has dedicated tools for partition management, while PostgreSQL relies on its native ALTER TABLE command.
- Oracle allows partition exchange and split, while PostgreSQL requires manual data manipulation.
- Foreign key constraints on parent tables in partitioned tables are limited in PostgreSQL.
- Performance can vary depending on workload, schema design, and database configuration.

Overall:

Both Oracle and PostgreSQL offer robust partitioning functionality. Oracle generally excels in speed and complexity, while PostgreSQL shines in simplicity and flexibility. The choice between them depends on specific needs, priorities, and existing database environment.