How to use table partitioning to scale PostgreSQL

With huge data being stored in databases, performance and scaling are two main factors that are affected. As table size increases with data load, more data scanning, swapping pages to memory, and other table operation costs also increase. Partitioning may be a good solution, as It can help divide a large table into smaller tables and thus reduce table scans and memory swap problems, which ultimately increases performance.

Partitioning helps to scale PostgreSQL by splitting large logical tables into smaller physical tables that can be stored on different storage media based on uses. Users can take better advantage of scaling by using declarative partitioning along with foreign tables using postgres_fdw

Benefits of partitioning

- PostgreSQL declarative partitioning is highly flexible and provides good control to users. Users can create any level of partitioning based on need and can modify, use constraints, triggers, and indexes on each partition separately as well as on all partitions together.
- Query performance can be increased significantly compared to selecting from a single large table.
- Partition-wise-join and partition-wise-aggregate features increase complex query computation performance as well.
- Bulk loads and data deletion can be much faster, as based on user requirements these operations can be performed on individual partitions.
- Each partition can contain data based on its frequency of use and so can be stored on media that may be cheaper or slower for low-use data.

When to use partitioning

Most benefits of partitioning can be enjoyed when a single table is not able to provide them. So we can say that if a lot of data is going to be written on a single table at some point, users need partitioning. Apart from data, there may be other factors users should consider, like update frequency of the data, use of data over a time period, how small a range data can be divided, etc. With good planning

and taking all factors into consideration, table partitioning can give a great performance boost and scale your PostgreSQL to larger datasets.

How to use partitioning

As of PostgreSQL12 release List, Range, Hash and combinations of these partition methods at different levels are supported. Let's explore what these are and how users can create different types of partitions with examples. For this article we will use the same table, which can be created by different partition methods.

LIST PARTITION

A list partition is created with predefined values to hold in a partitioned table. A default partition (optional) holds all those values that are not part of any specified partition.

```
CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY
LIST(status);
CREATE TABLE cust_active PARTITION OF customers FOR VALUES IN ('ACTIVE');
CREATE TABLE cust_archived PARTITION OF customers FOR VALUES IN ('EXPIRED');
CREATE TABLE cust_others PARTITION OF customers DEFAULT;
```

```
INSERT INTO customers VALUES (1,'ACTIVE',100), (2,'RECURRING',20), (3,'EXPIRED',38),
(4,'REACTIVATED',144);
select * from customers;

SELECT tableoid::regclass,* FROM customers;
```

RANGE PARTITION

A range partition is created to hold values within a range provided on the partition key. Both minimum and maximum values of the range need to be specified, where minimum value is inclusive and maximum value is exclusive

```
CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY
RANGE(arr);
 CREATE TABLE cust_arr_small PARTITION OF customers FOR VALUES FROM (MINVALUE) TO
(25);
 CREATE TABLE cust_arr_medium PARTITION OF customers FOR VALUES FROM (25) TO (75);
 CREATE TABLE cust_arr_large PARTITION OF customers FOR VALUES FROM (75) TO
(MAXVALUE);
```

```
 INSERT INTO customers VALUES (1,'ACTIVE',100), (2,'RECURRING',20), (3,'EXPIRED',38),
(4,'REACTIVATED',144);
  SELECT tableoid::regclass,* FROM customers;
```

HASH PARTITION

A hash partition is created by using modulus and remainder for each partition, where rows are inserted by generating a hash value using these modulus and remainders.

```
CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY HASH(id);
 CREATE TABLE cust_part1 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder
0);
 CREATE TABLE cust_part2 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder
1);
 CREATE TABLE cust_part3 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder
2);
  INSERT INTO customers VALUES (1,'ACTIVE',100), (2,'RECURRING',20),
(3,'EXPIRED',38), (4,'REACTIVATED',144);
   SELECT tableoid::regclass,* FROM customers;
```

Example:

```
CREATE TABLE customers (id INTEGER, status TEXT, name varchar) PARTITION BY
RANGE(id);

CREATE TABLE customers_name_small PARTITION OF customers FOR VALUES FROM (MINVALUE)
TO (25000);
 CREATE TABLE customers_name_medium PARTITION OF customers FOR VALUES FROM (25000) TO
(2500000);
CREATE TABLE customers_name_large PARTITION OF customers FOR VALUES FROM (2500000) TO
(MAXVALUE);


INSERT INTO customers
select x,'true','bibek'
```

```
from pg_catalog.generate_series(1,250000) as x;

explain select * from customers
where id between 25000 and 2500000;



INSERT INTO customers
select x,'true','bibek'
from pg_catalog.generate_series(2500000,25000000) as x;


select count(*) from customers_name_large;



explain select * from customers
where id=3500000;


explain select * from customers_name_large
where id=3500000;


ALTER TABLE customers DROP PARTITION customers_name_small;
ALTER TABLE customers DROP PARTITION customers_name_medium;
ALTER TABLE customers DROP PARTITION customers_name_large;


explain select * from customers
where id=3500000;
```