Partitioning

The process of dividing the tables into smaller manageable parts is called partitioning, and these smaller manageable parts are called partitions. PostgreSQL supports partitioning via table inheritance. Basically, you have to create each partition as a child table of the master table. So, the partitioning is made in such a way that every child table inherits single parent table. Parent table is empty and it exists just to describe the whole data set.

PostgreSQL partitioning can be implemented in range partitioning or list partitioning.

1. Generally, if you want to split data into specific ranges, then use range partitioning. Date and timestamp values are good examples for this. For example, you may want to split yearly data into quarterly partitions.

   Range partitioning can be done for example by ID ranges (like 0-100 000, 100 001-200 000, 200 001-300 000…) or Date ranges (like 2009-11-01 – 2009-11-30, 2009-12-01 – 2009-12-31…).

2. Similarly, if you want to split data based on a specific list of values, then use list partitioning. For this, you can consider creating partitions for each state of a country.

   List partitioning can be done for example by list of cities (like New York, Los Angeles, Chicago, Houston, Philadelphia…) or list of categories (like Programming, Home, Food…).

How to create partition?

Five easy steps how to make a partitioned table in PostgreSQL

1. Create master table
2. Create child tables without overlapping table constraints
3. Create indexes
4. Create trigger function to inserting data to child tables
5. Enable constraint exclusion

1. **Create very simple master table. This table can contain simple data**

```
create table logs (
    id serial primary key,
    created_at timestamp without time zone default now(),
    content text
);
```

2. **Create child tables, which inherits the master table and adds checks for dates, because we want ensure that we have only right data on each partition. We will partition table by date into four quarters of the year.**

```
create table logs_q1
   (check (created_at >= date '2014-01-01' and created_at <= date '2014-03-31')) inherits (logs);
create table logs_q2
   (check (created_at >= date '2014-04-01' and created_at <= date '2014-06-30')) inherits (logs);
create table logs_q3
   (check (created_at >= date '2014-07-01' and created_at <= date '2014-09-30')) inherits (logs);
create table logs_q4
   (check (created_at >= date '2014-10-01' and created_at <= date '2014-12-31')) inherits (logs);
```

3. **Create indexes to child tables to speed up day field usage, because almost all queries (INSERTs, SELECTs and UPDATEs) on the date field.**

```
create index logs_q1_created_at on logs_q1 using btree (created_at);
create index logs_q2_created_at on logs_q2 using btree (created_at);
create index logs_q3_created_at on logs_q3 using btree (created_at);
create index logs_q4_created_at on logs_q4 using btree (created_at);
```

4. **Then we need insert trigger and of course trigger function to master table. Conditions must be exactly the same as what the child tables checks**

```
create or replace function on_logs_insert() returns trigger as $$
begin
if ( new.created_at >= date '2014-01-01' and new.created_at <= date '2014-03-31') then
            insert into logs_q1 values (new.*);
 elsif ( new.created_at >= date '2014-04-01' and new.created_at <= date '2014-06-30') then
            insert into logs_q2 values (new.*);
 elsif ( new.created_at >= date '2014-07-01' and new.created_at <= date '2014-09-30') then
            insert into logs_q3 values (new.*);
 elsif ( new.created_at >= date '2014-10-01' and new.created_at <= date '2014-12-31') then
            insert into logs_q4 values (new.*);
 else
            raise exception 'created_at date out of range';
end if;
return null;
end;
```

```
$$ language plpgsql;
```

Let's attach the trigger function defined above to logs table.

```
create trigger logs_insert
 before insert on logs
for each row execute procedure on_logs_insert();
```

5. **Make sure that Constraint Exclusion is enabled. Constraint exclusion is driven by CHECK constraints. If constraint exclusion is disabled then query is not using check constraints and every query scans thru whole all child tables. So constraint exclusion is very important when using partitioned tables.**

Set constraint exclusion on with following row on postgresql.conf:

constraint_exclusion = on

Set constraint exclusion on following command on psql or psqlrc

SET constraint_exclusion = on;

Finally, let's insert some data into logs table to see the partitioning in work.

```
insert into logs (created_at, content)
values (date '2014-02-03', 'Content 1'), (date '2014-03-11', 'Content 2'),
        (date '2014-04-13', 'Content 3'), (date '2014-07-08', 'Content 4'),
        (date '2014-10-23', 'Content 5');
```

Now check the values,

```
select * from logs;
select * from logs_q1;
select * from logs_q2;
select * from logs_q3;
select * from logs_q4;
```

How to update the partition key value?

A simple way to handle queries that do update the partitioning key is to capture the UPDATE query, delete the related record from the partition, and then fire an INSERT on the base table. Then the partitioning mechanism will kick in and redirect the record to the appropriate partition.

So, for this you are required to create trigger on child tables. Basically, any change to the partitioning key value might result in the movement of that record to another partition, which is termed as the row movement. And to handle this you need to create trigger on every child table.

Create child_update trigger function:

```
create or replace function on_logs_q1_update() returns trigger as $$
begin
if (new.created_at != OLD.created_at) then
delete from logs_q1 where old.created_at=created_at;
insert into logs values (NEW.*);
return null;
else
return new;
end if;
return null;
end;
$$
language plpgsql;
```

Create trigger:

```
create trigger logs_update
before update on logs_q1
for each row execute procedure on_logs_q1_update();
```

Same for other partitions also…

How to create list partition?

The Steps of Creating List Partition is almost same:

1. Create master table
2. Create child tables as per List partition
3. Create indexes to child tables
4. Create trigger function to inserting data to child tables

1. **Create master table**

```
CREATE TABLE sales_record_listpart
(
id NUMERIC primary key,
sales_date date,
sales_amount NUMERIC,
city text
);
```

2. **Child Table for List Partition**

```
CREATE TABLE sales_record_list1
(
PRIMARY KEY (id, city),
CHECK (city IN ('new york', 'sydney'))
)
INHERITS (sales_record_listpart);
```

Note : You can create many child tables as per your need.

3. **Add Index to the child tables**

```
CREATE INDEX list1_index ON
sales_record_list1(city);
```

4. Create the Trigger for child tables

```
CREATE OR REPLACE FUNCTION
sales_record_list_insert()
RETURNS TRIGGER AS $$
BEGIN
IF (NEW.city IN ('new york', 'sydney')) THEN
INSERT INTO sales_record_list1 VALUES (NEW.*);
ELSEIF (NEW.city IN ('Islamabad', 'Boston', 'London')) THEN
INSERT INTO sales_record_list2 VALUES (NEW.*);
```

```
ELSE
RAISE EXCEPTION 'CITY not present in this lists';
END IF;
RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER sales_day_trigger
BEFORE INSERT ON sales_record_listpart
FOR EACH ROW
EXECUTE PROCEDURE sales_record_list_insert();
```

Insert some records and check:

```
INSERT INTO sales_record_listpart
(id, sales_date, sales_amount, city)
VALUES
(1,'15-APR-2008',1200,'sydney');

Select * from sales_record_list1
Select * from sales_record_list2
```

***********Postgresql 10 ********

Example of Range Partition :

```
CREATE TABLE measurement (
   logdate        date not null,
    peaktemp        int default 1,
    unitsales       int
) PARTITION BY RANGE (logdate);

ALTER TABLE measurement ADD CHECK (peaktemp > 0);
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');

keith@keith=# \d+ measurement
                              Table "public.measurement"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----------+---------+-----------+----------+---------+---------+--------------+-------------
 logdate   | date    |           | not null |         | plain   |              |
 peaktemp  | integer |           |          | 1       | plain   |              |
 unitsales | integer |           |          |         | plain   |              |
Partition key: RANGE (logdate)
Check constraints:
    "measurement_peaktemp_check" CHECK (peaktemp > 0)
Partitions: measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')

=# \d measurement_y2016m07
        Table "public.measurement_y2016m07"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 logdate   | date    |           | not null |
 peaktemp  | integer |           |          | 1
 unitsales | integer |           |          | 0
Partition of: measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
Check constraints:
    "measurement_peaktemp_check" CHECK (peaktemp > 0)
```

Example of List Partition :

```
CREATE TABLE cities (
    city_id         bigserial not null,
    name        text not null,
    population   int
) PARTITION BY LIST (initcap(name));

CREATE TABLE cities_west
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('Los Angeles', 'San Francisco');

keith@keith=# \d+ cities
                                          Table "public.cities"
  Column    |  Type    | Collation | Nullable |               Default                | Storage  | Stats target
| Description
------------+---------+-----------+----------+--------------------------------------+----------+--------------
+-------------
 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
|
 name       | text    |           | not null |                                      | extended |              |
 population | integer |           |          |                                      | plain    |              |
Partition key: LIST (initcap(name))
Partitions: cities_west FOR VALUES IN ('Los Angeles', 'San Francisco')

keith@keith=# \d+ cities_west
                                        Table "public.cities_west"
  Column    |  Type    | Collation | Nullable |               Default                | Storage  | Stats target
| Description
------------+---------+-----------+----------+--------------------------------------+----------+--------------
+-------------
```

```
 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |              |
 name       | text    |           | not null |                                         | extended |              |
 population | integer |           |          |                                         | plain    |              |
Partition of: cities FOR VALUES IN ('Los Angeles', 'San Francisco')
Check constraints:
    "city_id_nonzero" CHECK (city_id <> 0)
```

Sub-partitioning is also possible and you can easily mix and match both range and list partitioning types. Here we redefine the cities_west table to be partitioned as well

```
CREATE TABLE cities_west
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('Los Angeles', 'San Francisco') PARTITION BY RANGE (population);

CREATE TABLE cities_west_10000_to_100000
    PARTITION OF cities_west FOR VALUES FROM (10000) TO (100000);

keith@keith=# \d+ cities
                                            Table "public.cities"
   Column   |  Type   | Collation | Nullable |                 Default                 | Storage  | Stats target | Description
------------+---------+-----------+----------+-----------------------------------------+----------+--------------+-------------
 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |              |
 name       | text    |           | not null |                                         | extended |              |
 population | integer |           |          |                                         | plain    |              |
Partition key: LIST (initcap(name))
Partitions: cities_west FOR VALUES IN ('Los Angeles', 'San Francisco')
```

```
keith@keith=# \d+ cities_west

                              Table "public.cities_west"

  Column   |  Type   | Collation | Nullable |            Default             | Storage  | Stats target
| Description

------------+---------+-----------+----------+---------------------------------------+----------+--------------
+-------------

 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
|

 name       | text    |           | not null |                                | extended |              |

 population | integer |           | not null |                                | plain    |              |

Partition of: cities FOR VALUES IN ('Los Angeles', 'San Francisco')

Partition key: RANGE (population)

Check constraints:

   "city_id_nonzero" CHECK (city_id <> 0)

Partitions: cities_west_10000_to_100000 FOR VALUES FROM (10000) TO (100000)


keith@keith=# \d+ cities_west_10000_to_100000

                         Table "public.cities_west_10000_to_100000"

  Column   |  Type   | Collation | Nullable |            Default             | Storage  | Stats target
| Description

------------+---------+-----------+----------+---------------------------------------+----------+--------------
+-------------

 city_id    | bigint  |           | not null | nextval('cities_city_id_seq'::regclass) | plain    |
|

 name       | text    |           | not null |                                | extended |              |

 population | integer |           | not null |                                | plain    |              |

Partition of: cities_west FOR VALUES FROM (10000) TO (100000)

Check constraints:

   "city_id_nonzero" CHECK (city_id <> 0)
```

Partition Maintenance :

The simplest option for removing old data is to drop the partition that is no longer necessary:

```
DROP TABLE measurement_y2006m02;
```

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above:

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

As an alternative, it is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later. This allows the data to be loaded, checked, and transformed prior to it appearing in the partitioned table:

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```