PARTITIONING

@JuanitoFatas

Warning: I am not DBA, these are toy examples. Do not use on Production.

Background

Partitioning is technique to split large table into smaller ones. We can split large table into smaller chunks based on key. Key could be Range, List, or Hash.

For use cases like time-series or serial-based applications, we can partition incoming data for ranges such as daily or based on any custom format like timestamped ID. Each partition contains only the data according to your range (daily). The partition key controls the size of a partition.

When `INSERT` or `UPDATE` on a partitioned table, PostgreSQL will do it on the right partition based on criteria you defined (daily). Each partitioned table partitions will be stored as child table of the parent table.

When reading data from partitioned table, PostgreSQL optimizer examines the `WHERE` clause and only scan the relevant partitions if possible.

Use partition when table is bigger than memory of database.

Trigger-based Partitioning

Before Postgres adds **native partitioning** (PostgreSQL 10+), we can use table inheritance + constraints + triggers to implement Partitioning. That’s why I call it Trigger-based Partitioning.

Let’s take a look at inheritance and triggers.

Inheritance

First create a parent table `logs` and child table `sql_logs` :

```
CREATE TABLE logs (id serial, created_at timestamp without time zone);
CREATE TABLE sql_logs(sql text) INHERITS(logs);
```

Now let’s add a row to both tables:

```
INSERT INTO logs VALUES (1, '2020-05-10');
INSERT INTO sql_logs VALUES (2, '2020-05-11', 'SELECT 1');
```

Select them you see `logs` showing record from child tables:

```
SELECT * FROM logs ;
 id |     created_at
----+---------------------
  1 | 2020-05-10 00:00:00
  1 | 2020-05-11 00:00:00
(2 rows)

SELECT * FROM sql_logs ;
 id |     created_at      |   sql
----+---------------------+----------

  1 | 2020-05-11 00:00:00 | SELECT 1
(1 row)
```

If you only want to see records from parent table, add `ONLY` :

```
SELECT * FROM ONLY logs ;
 id |     created_at
----+---------------------
  1 | 2020-05-10 00:00:00
(1 row)
```

Same for `UPDATE` and `DELETE` . If you only want to update the child table, add the `ONLY` .

Example creating partitioned tables

Let’s say we want to partition our `logs` table based on `created_at` time. We can use Check constraints.

First drop all tables we created:

```
DROP TABLE logs CASCADE
;
```

Then let’s create 13 tables!

```
CREATE TABLE logs (id serial, created
_at timestamp without time zone)
;

CREATE TABLE logs_old (
  CHECK ( created_at < '2020-01-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202001 (
  CHECK ( created_at >= '2020-01-01'
and created_at < '2020-02-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202002 (
  CHECK ( created_at >= '2020-02-01'
and created_at < '2020-03-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202003 (
  CHECK ( created_at >= '2020-03-01'
and created_at < '2020-04-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202004 (
  CHECK ( created_at >= '2020-04-01'
and created_at < '2020-05-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202005 (
  CHECK ( created_at >= '2020-05-01'
and created_at < '2020-06-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202006 (
  CHECK ( created_at >= '2020-06-01'
and created_at < '2020-07-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202007 (
  CHECK ( created_at >= '2020-07-01'
and created_at < '2020-08-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202008 (
  CHECK ( created_at >= '2020-08-01'
and created_at < '2020-09-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202009 (
  CHECK ( created_at >= '2020-09-01'
and created_at < '2020-10-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202010 (
  CHECK ( created_at >= '2020-10-01'
and created_at < '2020-11-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202011 (
  CHECK ( created_at >= '2020-11-01'
and created_at < '2020-12-01' )
) INHERITS(logs)
;

CREATE TABLE logs_202012 (
  CHECK ( created_at >= '2020-12-01'
and created_at < '2021-01-01' )
) INHERITS(logs)
;
```

Now we have 13 tables, 12 tables for each month in 2020. `logs` table for everything before 2020:

```
\d
            List of relations
 Schema |    Name     |   Type   | Ow
ner
--------+-------------+----------+---
----
```

```
 public | logs        | table     | hh
h
 public | logs_202001 | table     | hh
h
 public | logs_202002 | table     | hh
h
 public | logs_202003 | table     | hh
h
 public | logs_202004 | table     | hh
h
 public | logs_202005 | table     | hh
h
 public | logs_202006 | table     | hh
h
 public | logs_202007 | table     | hh
h
 public | logs_202008 | table     | hh
h
 public | logs_202009 | table     | hh
h
 public | logs_202010 | table     | hh
h
 public | logs_202011 | table     | hh
h
 public | logs_202012 | table     | hh
h
 public | logs_old    | table     | hh
h
```

Add relevant indexes for these tables to be able to quickly query based on `created_at` later on.

```
CREATE INDEX index_created_at_on_logs
_old ON logs_old USING btree (created
_at);
CREATE INDEX index_created_at_on_logs
_202001 ON logs_202001 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202002 ON logs_202002 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202003 ON logs_202003 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202004 ON logs_202004 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
```

```
_202005 ON logs_202005 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202006 ON logs_202006 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202007 ON logs_202007 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202008 ON logs_202008 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202009 ON logs_202009 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202010 ON logs_202010 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202011 ON logs_202011 USING btree (c
reated_at);
CREATE INDEX index_created_at_on_logs
_202012 ON logs_202012 USING btree (c
reated_at);
```

Triggers

Now we can add the **trigger function** to decide where data go when we do DML operations[1]. Let's take `INSERT` as example:

```
CREATE OR REPLACE FUNCTION logs_inser
t_trigger()
  RETURNS trigger
AS $function$
BEGIN
  IF ( NEW.created_at < '2020-01-01'
) THEN
    INSERT INTO logs_old VALUES (NEW.
*);
  ELSIF ( NEW.created_at >= '2020-01-
01' and NEW.created_at < '2020-02-01'
) THEN
    INSERT INTO logs_202001 VALUES (N
EW.*);
  ELSIF ( NEW.created_at >= '2020-02-01' and NEW.created_at < '2020-03-01' ) THEN
    INSERT INTO logs_202002 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-03-01' and NEW.created_at < '2020-04-01' ) THEN
    INSERT INTO logs_202003 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-04-01' and NEW.created_at < '2020-05-01' ) THEN
    INSERT INTO logs_202004 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-05-01' and NEW.created_at < '2020-06-01' ) THEN
    INSERT INTO logs_202005 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-06-01' and NEW.created_at < '2020-07-01' ) THEN
    INSERT INTO logs_202006 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-07-01' and NEW.created_at < '2020-08-01' ) THEN
    INSERT INTO logs_202007 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-08-01' and NEW.created_at < '2020-09-01' ) THEN
    INSERT INTO logs_202008 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-09-01' and NEW.created_at < '2020-10-01' ) THEN
    INSERT INTO logs_202009 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-10-01' and NEW.created_at < '2020-11-01' ) THEN
    INSERT INTO logs_202010 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-11-01' and NEW.created_at < '2020-12-01' ) THEN
```

```
    INSERT INTO logs_202011 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-12-01' and NEW.created_at < '2021-01-01' ) THEN
    INSERT INTO logs_202012 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2021-01-01' and NEW.created_at < '2021-02-01' ) THEN
    INSERT INTO logs_202101 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$function$ LANGUAGE plpgsql;
```

It took me 20 minutes to type all these, look up the manual, to get this thing right, you can see how this quickly became annoyed :/

We have `logs_insert_trigger` trigger function defined, now creates a trigger on our `logs` table, so when we insert row to logs table, it goes to the right partition.

```
CREATE TRIGGER insert_logs_trigger
BEFORE INSERT ON logs
FOR EACH ROW EXECUTE PROCEDURE logs_insert_trigger();
```

Now when we `INSERT` a row into `logs` table. It will goes to the right partitioned tables. Let’s insert 100 random rows in randome times into logs

```
INSERT INTO logs (id, created_at)
SELECT round(1000000000*random()), generate_series('2019-01-01'::timestamp, '2020-12-31'::timestamp, '10 seconds');

SELECT COUNT(1) FROM logs;
  count
```

```
---------
 6307201
```

We have about 6M rows.

You can see the parent table `logs` has no data while its partitions have:

```
\d+
                          List of rela
tions
 Schema |     Name     |   Type    | Ow
ner |     Size     | Description
--------+--------------+-----------+---
----+------------+------------
 public | logs         | table     | hh
h    | 0 bytes     |
 public | logs_202001 | table     | hh
h    | 11 MB       |
 public | logs_202002 | table     | hh
h    | 11 MB       |
 public | logs_202003 | table     | hh
h    | 11 MB       |
 public | logs_202004 | table     | hh
h    | 11 MB       |
 public | logs_202005 | table     | hh
h    | 11 MB       |
 public | logs_202006 | table     | hh
h    | 11 MB       |
 public | logs_202007 | table     | hh
h    | 11 MB       |
 public | logs_202008 | table     | hh
h    | 11 MB       |
 public | logs_202009 | table     | hh
h    | 11 MB       |
 public | logs_202010 | table     | hh
h    | 11 MB       |
 public | logs_202011 | table     | hh
h    | 11 MB       |
 public | logs_202012 | table     | hh
h    | 11 MB       |
 public | logs_old     | table     | hh
h    | 133 MB      |
(15 rows)
```

The data are inserted to respective partitions. Our parent table `logs` stay unchanged.

parent table `logs` stay unchanged.

```
SELECT * FROM logs_old limit 1;

 id  |      created_at
-----+--------------------
277145595 | 2019-01-01 00:00:00
```

When we want to find log given specific date. We can either query the parent table or go directly to specific table. Let’s see what is the performance implications doing both:

```
EXPLAIN ANALYZE SELECT * FROM logs WH
ERE created_at > '2020-01-01' AND cre
ated_at < '2020-01-02';


QUERY PLAN
-------------------------------------
-------------------------------------
-------------------------------------
-------------------------------------
-----------
 Append  (cost=0.00..365.13 rows=8709
width=12) (actual time=0.016..2.355 r
ows=8639 loops=1)
   ->  Seq Scan on logs  (cost=0.00..
0.00 rows=1 width=12) (actual time=0.
003..0.003 rows=0 loops=1)
         Filter: ((created_at > '2020
-01-01 00:00:00'::timestamp without t
ime zone) AND (created_at < '2020-01-
02 00:00:00'::timestamp without time
zone))
   ->  Index Scan using index_created
_at_on_logs_202001 on logs_202001  (c
ost=0.42..321.58 rows=8708 width=12)
(actual time=0.013..1.496 rows=8639 l
oops=1)
         Index Cond: ((created_at >
'2020-01-01 00:00:00'::timestamp with
out time zone) AND (created_at < '202
0-01-02 00:00:00'::timestamp without
time zone))
 Planning Time: 1.162 ms
 Execution Time: 2.832 ms
(7 rows)
```

```
(7 rows)


EXPLAIN ANALYZE SELECT * FROM logs_20
2001 WHERE created_at > '2020-01-01'
AND created_at < '2020-01-02';

QUERY PLAN
-------------------------------------
-------------------------------------
-------------------------------------
-------------------------------------
-----
 Index Scan using index_created_at_on
_logs_202001 on logs_202001  (cost=0.
42..321.58 rows=8708 width=12) (actua
l time=0.012..1.450 rows=8639 loops=1
)
   Index Cond: ((created_at > '2020-0
1-01 00:00:00'::timestamp without tim
e zone) AND (created_at < '2020-01-02
00:00:00'::timestamp without time zon
e))
 Planning Time: 0.064 ms
 Execution Time: 2.014 ms
(4 rows)
```

As you can see and probably figured, we should directly query the specific table. Both planning time and execution time are saved. Your application code should decide which table to run the query.

Time flies fast. We are approaching 2020-12-31. We need to add new partition for 2021. We can follow above to create tables for 2021. Then update the `logs_insert_trigger()` code. But this requires some extra steps because we don’t want to inherit from `logs` before we updated the trigger function. Otherwise during the time you’re adding the table’s DML operations will go to the wrong table.

```
CREATE TABLE logs_202101 (LIKE logs I
NCLUDING ALL);

ALTER TABLE logs_202101 ADD CONSTRAINT
ALTER TABLE logs_202101 ADD CONSTRAINT logs_202101_created_at_check
  CHECK ( created_at >= '2021-01-01' and created_at < '2021-02-01' );

CREATE OR REPLACE FUNCTION logs_insert_trigger()
  RETURNS trigger
AS $function$
BEGIN
  IF ( NEW.created_at < '2020-01-01' ) THEN
    INSERT INTO logs_old VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-01-01' and NEW.created_at < '2020-02-01' ) THEN
    INSERT INTO logs_202001 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-02-01' and NEW.created_at < '2020-03-01' ) THEN
    INSERT INTO logs_202002 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-03-01' and NEW.created_at < '2020-04-01' ) THEN
    INSERT INTO logs_202003 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-04-01' and NEW.created_at < '2020-05-01' ) THEN
    INSERT INTO logs_202004 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-05-01' and NEW.created_at < '2020-06-01' ) THEN
    INSERT INTO logs_202005 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-06-01' and NEW.created_at < '2020-07-01' ) THEN
    INSERT INTO logs_202006 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-07-01' and NEW.created_at < '2020-08-01' ) THEN
    INSERT INTO logs_202007 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-08-
01' and NEW.created_at < '2020-09-01' ) THEN
    INSERT INTO logs_202008 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-09-01' and NEW.created_at < '2020-10-01' ) THEN
    INSERT INTO logs_202009 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-10-01' and NEW.created_at < '2020-11-01' ) THEN
    INSERT INTO logs_202010 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-11-01' and NEW.created_at < '2020-12-01' ) THEN
    INSERT INTO logs_202011 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2020-12-01' and NEW.created_at < '2021-01-01' ) THEN
    INSERT INTO logs_202012 VALUES (NEW.*);
  ELSIF ( NEW.created_at >= '2021-01-01' and NEW.created_at < '2021-02-01' ) THEN
    INSERT INTO logs_202101 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$function$ LANGUAGE plpgsql;

ALTER TABLE logs_202101 INHERIT logs;
```

Delete a partition table is awesome. You can directly drop the table:

```
DROP TABLE logs_202001
```

If you want to see all parent/children tables relationships:

```
SELECT
  parent.relname AS parent,
  child.relname AS child
FROM
  pg_inherits JOIN pg_class parent
    ON pg_inherits.inhparent = parent
.oid JOIN pg_class child
    ON pg_inherits.inhrelid = child.o
id JOIN pg_namespace parent_namespace
    ON parent_namespace.oid = parent.
relnamespace JOIN pg_namespace child_
namespace
    ON child_namespace.oid = child.re
lnamespace;

 parent |    child
--------+-------------
 logs   | logs_old
 logs   | logs_202001
 logs   | logs_202002
 logs   | logs_202003
 logs   | logs_202004
 logs   | logs_202005
 logs   | logs_202006
 logs   | logs_202007
 logs   | logs_202008
 logs   | logs_202009
 logs   | logs_202010
 logs   | logs_202011
 logs   | logs_202012
 logs   | logs_202101
```

So this was Partitioning before **Postgres's native partitioning**. With native partitioning, you don't need to manage the triggers. Postgres does it for you!

[1] INSERT, DELETE, UPDATE ↩

*This is one of the posts from the **postgres Series**.*