Dealing with Gigantic Tables

Linas Valiukas

PostgresConf Silicon Valley 2018

Who?

- Contractor in academia
- 12 TB worth of stuff on PostgreSQL
- DBA out of necessity

Andrew is amazing ❤

```
[10:33] == RhodiumToad [~andrew@82-68-208-21.dsl.in-addr.zen.co.uk]
[10:33] ==  realname : Andrew { http://www.rhodiumtoad.org.uk/ }
[10:33] ==  channels : #postgresql
[10:33] ==  server   : barjavel.freenode.net [Paris, FR, EU]
[10:33] ==  account  : RhodiumToad
[10:33] == End of WHOIS
```

Sample gigantic table

```
CREATE TABLE friends (
    friends_id  BIGSERIAL  PRIMARY KEY,
    first_name  VARCHAR    NOT NULL,
    last_name   VARCHAR    NOT NULL,
    dob         DATE       NOT NULL,
    motto       TEXT       NULL
);
```

Sample gigantic table

```
INSERT INTO friends (first_name, last_name, dob, motto)
    SELECT
        random()::text AS first_name,
        random()::text AS last_name,
        TIMESTAMP '1990-01-01' + random() * (TIMESTAMP '1990-01-01' - TIMESTAMP
'2014-12-31') AS dob,

        -- Approx. every 1000th motto is the same
        'The quick brown fox jumps over the ' || floor(random() * 1000 +
1)::int::text || ' dog.' AS motto

    -- 10b rows, ~1 TB of data
    FROM generate_series(1, 10 * 1000 * 1000 * 1000);
```

½ hash index

½ hash index: simple query

```
SELECT *
FROM friends
WHERE motto = 'The quick brown fox jumps over the 42 dog.';
```

½ hash index: B-tree index

```
CREATE INDEX friends_motto_btree ON friends USING btree (motto);
```

```
 table_name |          index_name     | num_rows | table_size | index_size
------------+-------------------------+----------+------------+------------
 friends    | friends_motto_btree     |  1000000 | 113 MB     | 65 MB
```

½ hash index: hash index

```
CREATE INDEX friends_motto_btree ON friends USING btree (motto);
CREATE INDEX friends_motto_hash ON friends USING hash (motto);
```

```
 table_name |          index_name     | num_rows | table_size | index_size
------------+-------------------------+----------+------------+------------
 friends    | friends_motto_btree     |  1000000 | 113 MB     | 65 MB
 friends    | friends_motto_hash      |  1000000 | 113 MB     | 44 MB
```

½ hash index: `half_md5()`

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE OR REPLACE FUNCTION half_md5(string TEXT) RETURNS bytea AS $$

    SELECT SUBSTRING(public.digest(string, 'md5'::text), 0, 9);

$$ LANGUAGE SQL IMMUTABLE STRICT;
```

½ hash index: `half_md5()` index

```
CREATE INDEX friends_motto_btree ON friends USING btree (motto);
CREATE INDEX friends_motto_hash ON friends USING hash (motto);
CREATE INDEX friends_motto_half_md5_btree ON friends USING btree (half_md5(motto));
```

```
 table_name |         index_name           | num_rows | table_size | index_size
------------+------------------------------+----------+------------+------------
 friends    | friends_motto_btree          |  1000000 | 113 MB     | 65 MB
 friends    | friends_motto_hash           |  1000000 | 113 MB     | 44 MB
 friends    | friends_motto_half_md5_btree |  1000000 | 113 MB     | 30 MB
```

½ hash index: querying with `half_md5()` index

```
SELECT *
FROM friends

-- Index scan:
WHERE half_md5(motto) = half_md5('The quick brown fox jumps over the 42 dog.')

-- Filter for false positives:
  AND motto = 'The quick brown fox jumps over the 42 dog.';
```

Partitioning

Partitioning: pre: tables, chunk size

```
ALTER TABLE friends RENAME TO friends_nonpartitioned;

CREATE TABLE friends_partitioned AS TABLE friends_nonpartitioned WITH NO DATA;

SELECT setval(
    pg_get_serial_sequence('enemies_partitioned', 'enemies_id'),
    (SELECT MAX(enemies_id) FROM enemies_nonpartitioned)
);

CREATE OR REPLACE FUNCTION friends_id_chunk_size() RETURNS BIGINT AS $$
BEGIN
     RETURN 100 * 1000 * 1000;   -- 100m friends in each partition
END; $$ LANGUAGE plpgsql IMMUTABLE;
```

Partitioning: pre: partition name helper

```
CREATE OR REPLACE FUNCTION friends_partition_name(friends_id BIGINT) RETURNS TEXT AS $$
DECLARE
    to_char_format CONSTANT TEXT := '00'; -- ...
BEGIN
    SELECT friends_id / friends_id_chunk_size() INTO friends_id_chunk_number;

    SELECT 'friends_partitioned_' || TRIM(leading ' ' FROM
TO_CHAR(friends_id_chunk_number, to_char_format)) INTO table_name;

    RETURN table_name;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

Partitioning: pre: partition name helper

```
pypt=# select friends_partition_name(1);
 friends_partition_name
------------------------
 friends_partitioned_00
(1 row)


pypt=# select friends_partition_name(300000000);
 friends_partition_name
------------------------
 friends_partitioned_03
(1 row)
```

Partitioning: pre: insert trigger

```
CREATE FUNCTION friends_partitioned_insert_trigger() RETURNS TRIGGER AS $$
BEGIN
     EXECUTE '
         INSERT INTO ' || friends_partition_name(NEW.friends_id) || '
              SELECT $1.*
     ' USING NEW;
     RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER friends_partitioned_insert_trigger
     BEFORE INSERT ON friends_partitioned
     FOR EACH ROW EXECUTE PROCEDURE friends_partitioned_insert_trigger();
```

Partitioning: pre: schema helper

```
CREATE FUNCTION table_exists(test_table_name VARCHAR) RETURNS BOOLEAN AS $$
DECLARE
    schema_position INT;
BEGIN

    RETURN EXISTS (
        SELECT 1
        FROM information_schema.tables
        WHERE table_schema = CURRENT_SCHEMA()
         AND table_name = test_table_name
    );

END;
$$ LANGUAGE plpgsql;
```

Partitioning: pre: partition creation helper

```
CREATE OR REPLACE FUNCTION friends_create_partitions() RETURNS VOID AS $$

-- DECLARE ...

BEGIN

    SELECT friends_id_chunk_size() INTO chunk_size;

    -- Create +1 partition for future insertions
    SELECT COALESCE(MAX(friends_id), 0) + chunk_size + 1
    FROM friends_partitioned
    INTO max_friends_id;

    -- ...
```

Partitioning: pre: partition creation helper

```
FOR partition_friends_id IN 1..max_friends_id BY chunk_size LOOP

    SELECT friends_partition_name(partition_friends_id) INTO target_table_name;
    IF NOT table_exists(target_table_name) THEN

        SELECT (partition_friends_id / chunk_size) * chunk_size INTO friends_id_start;
        SELECT ((partition_friends_id / chunk_size) + 1) * chunk_size INTO
friends_id_end;

         EXECUTE 'CREATE TABLE ' || target_table_name || ' (PRIMARY KEY (friends_id),
CONSTRAINT ' || target_table_name || '_friends_id CHECK (friends_id >= ' ||
friends_id_start || ' AND friends_id <  ' || friends_id_end   || ')) INHERITS
(friends_partitioned);';

    END IF;
END LOOP;
```

Partitioning: pre: partition creation helper

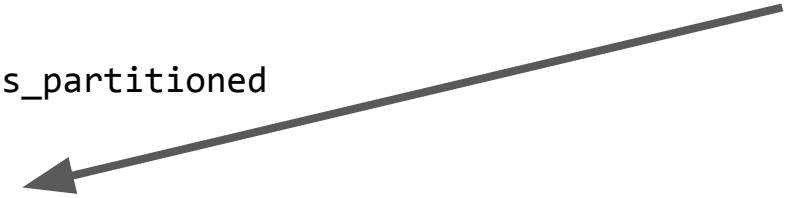
```
SELECT friends_create_partitions();
```

```
pypt=# \d+
                              List of relations
  Schema   |          Name              |   Type   | Owner |Size        | Description
-----------+----------------------------+----------+-------+------------+-------------
 public    | friends_nonpartitioned     | table    | pypt  | 1 TB       |
 public    | friends_partitioned        | table    | pypt  | 8192 bytes |
 public    | friends_partitioned_00     | table    | pypt  | 8192 bytes |
 public    | friends_partitioned_01     | table    | pypt  | 8192 bytes |
(5 rows)
```

Partitioning: pre: friends updatable view

```
CREATE VIEW friends AS

    SELECT *
    FROM (
        SELECT *
        FROM friends_partitioned

        UNION ALL

        SELECT *
        FROM friends_nonpartitioned

    ) AS friends_union;
```

(the whole point of this presentation)



Partitioning: pre: friends updatable view

```
-- https://wiki.postgresql.org/wiki/INSERT_RETURNING_vs_Partitioning
ALTER VIEW friends
    ALTER COLUMN friends_id
    SET DEFAULT nextval(
        pg_get_serial_sequence('friends_partitioned', 'friends_partitioned_id')
    );
```

Partitioning: pre: updatable view trigger

```
CREATE FUNCTION friends_insert_update_delete() RETURNS trigger AS $$
BEGIN

    IF (TG_OP = 'INSERT') THEN

        -- All new INSERTs go to partitioned table only.
        --
        -- By INSERTing into the master table, we're letting triggers choose
        -- the correct partition and partition's own triggers to get fired (if any).
        INSERT INTO friends_partitioned SELECT NEW.*;

        -- Return inserted row
        RETURN NEW;
```

Partitioning: pre: updatable view trigger

```
ELSIF (TG_OP = 'UPDATE') THEN

    -- UPDATE both tables

    UPDATE friends_partitioned
    SET first_name = NEW.first_name, last_name = NEW.last_name, dob = NEW.dob, motto
= NEW.motto
    WHERE friends_id = OLD.friends_id;

    UPDATE friends_nonpartitioned
    SET first_name = NEW.first_name, last_name = NEW.last_name, dob = NEW.dob, motto
= NEW.motto
     WHERE friends_id = OLD.friends_id;

     -- Return updated row
     RETURN NEW;
```

Partitioning: pre: updatable view trigger

```
ELSIF (TG_OP = 'DELETE') THEN

    -- DELETE from both tables

    DELETE FROM friends_partitioned
    WHERE friends_id = OLD.friends_id;

    DELETE FROM friends_nonpartitioned
    WHERE friends_id = OLD.friends_id;

    -- Return deleted row
    RETURN OLD;
```

Partitioning: pre: updatable view trigger

```
CREATE TRIGGER friends_insert_update_delete
     INSTEAD OF INSERT OR UPDATE OR DELETE ON friends -- the view
     FOR EACH ROW EXECUTE PROCEDURE friends_insert_update_delete();
```

Partitioning: copying data

```
CREATE FUNCTION copy_chunk_of_nonpartitioned_friends_to_partitions(start_friends_id
BIGINT, end_friends_id BIGINT) RETURNS VOID AS $$

DECLARE
    start_friends_table_name TEXT;

BEGIN

    PERFORM pid
    FROM pg_stat_activity, LATERAL pg_cancel_backend(pid) f
    WHERE backend_type = 'autovacuum worker'
      AND query ~ 'friends';
```

Partitioning: copying data

```
SELECT friends_partition_name(start_friends_id) INTO start_friends_table_name;

-- Test whether start_friends_id and end_friends_id is within the same partition here

EXECUTE '
    WITH deleted_rows AS (
        DELETE FROM friends_nonpartitioned
        WHERE friends_id BETWEEN ' || start_friends_id || ' AND ' || end_friends_id
|| '
        RETURNING *
    )
    INSERT INTO ' || start_friends_table_name || '
        SELECT *
        FROM deleted_rows;
     ';
```

Incremental backup using ZFS snapshots

Incremental backup using ZFS snapshots

```
# Create initial snapshot
zfs snapshot pool/postgresql@backup_1

# Send initial snapshot to "backup_host"
zfs send pool/postgresql@backup_1 | \
     pv | \
     ssh backup_host "mbuffer -s 128k -m 1G | zfs receive -F pool/postgresql"
```

Incremental backup using ZFS snapshots

```
# Create a new snapshot
zfs snapshot pool/postgresql@backup_2

# Generate and send an incremental stream from "backup_1" to "backup_2"
zfs send -i pool/postgresql@backup_1 pool/postgresql@backup_2 | \
     pv | \
     ssh backup_host "mbuffer -s 128k -m 1G | zfs receive -F pool/postgresql"

# Destroy old snapshot (do it on "backup_host" too!)
zfs destroy pool/postgresql@backup_1
```

Thanks!

linas.valiukas@gmail.com

www.pypt.lt