Real-time Analytics with PostgreSQL

Andres Freund
andres@citusdata.com

Marco Slot
marco@citusdata.com

Tutorial Logistics

Slides:

http://goo.gl/citus-slides

Github repo:

https://github.com/citusdata/pgconfsv-tutorial

SSH IPs at the door.

Username: **admin**

Password: **LpsEiksef2**

Update tutorial files:

```
cd pgconfsv-tutorial
git pull
```

Exercise file names

loader.py

Script to load github archive data into the database

exercises/*

Functions that we'll define in the slides

createagg.sql

Scripts for automating view creation

Real-time Analytics

I have a lot of real-time events data from servers, sensors, ...

In most cases time-series data, also high volume transactional data.

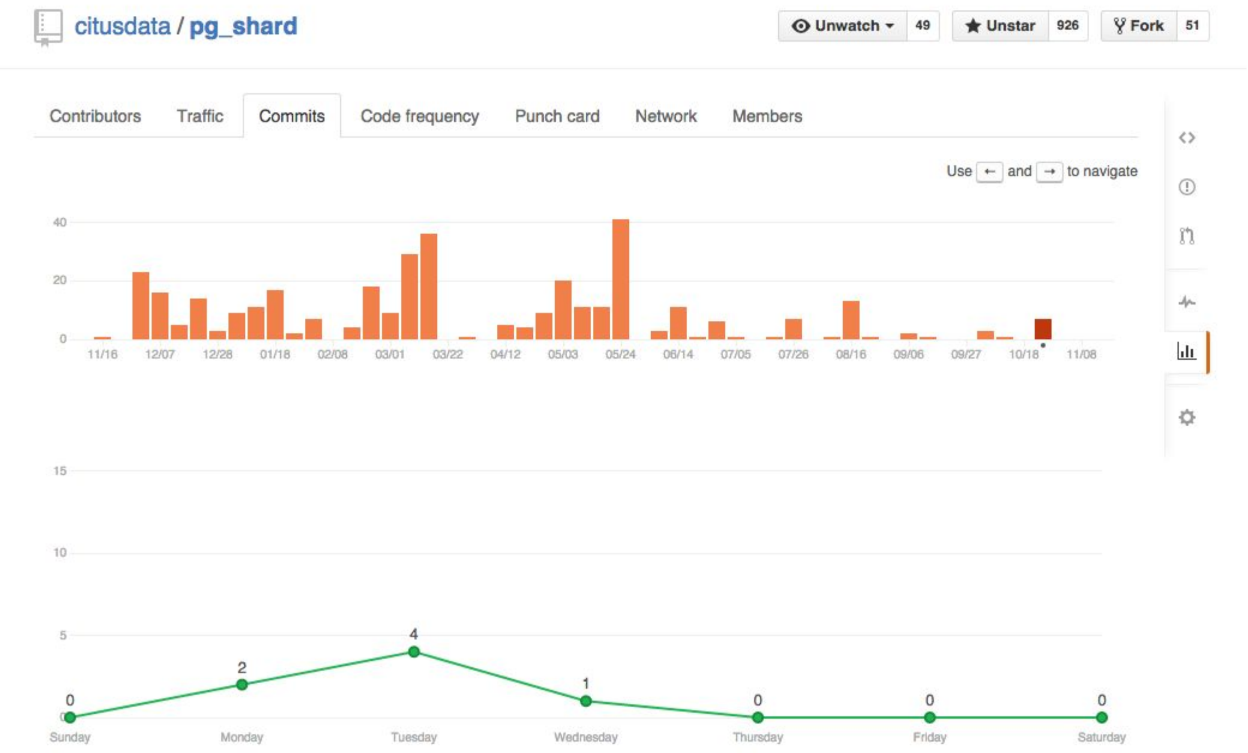
I want real-time analytics for dashboards, alarms, ...

but...

- Computing aggregates over large datasets is expensive
- Can cache aggregates, but frequently updated

Use-case: Analytical dashboards

Github events dashboard with real-time charts:



Data flow

Typical flow of data:

event $\rightarrow$ collect data $\rightarrow$ store fact $\rightarrow$ compute aggregates $\rightarrow$ retrieve aggregate

Common architecture:

event $\rightarrow$ kafka $\rightarrow$ HDFS $\rightarrow$ Spark $\rightarrow$ Cassandra $\rightarrow$ dashboard

The PostgreSQL architecture:

event $\rightarrow$ kafka $\rightarrow$ PostgreSQL $\rightarrow$ dashboard

Different aggregates

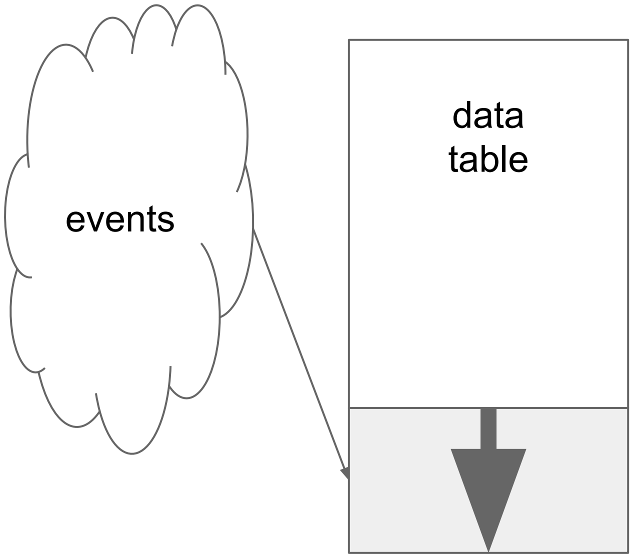
Typically many different views on the data:

- number of events per day
- number of events per day by repository
- number of events per day by repository and type
- number of events per week by type
- number of unique visitors per day by repository
- number of unique visitors per day by referral
- …

Note: some aggregates can be derived from finer-grained ones.

Real-time Analytics

Data grows: Queries slow down



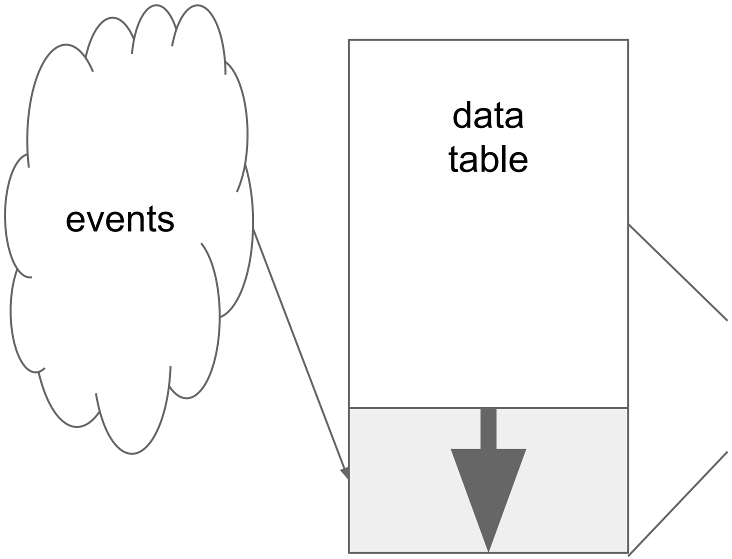

```
Day 1:
SELECT type, count(*) FROM data
GROUP BY type ORDER BY 2 DESC LIMIT 5;
Time: 176.043 ms

Day 10:
SELECT type, count(*) FROM data
GROUP BY type ORDER BY 2 DESC LIMIT 5;
Time: 1812.932 ms

Day 31:
SELECT type, count(*) FROM data
GROUP BY type ORDER BY 2 DESC LIMIT 5;
Time: 93926.968 ms
```

Real-time Analytics

Many queries only concern recent data, but there can be a lot of data:




```
SELECT type, count(*)
FROM data
WHERE created_at:date BETWEEN '2015-10-17' AND '2015-11-17'
GROUP BY type ORDER BY 2 DESC;
Time: 103045.259 ms
```

Exercise: Github Events Data

Github publishes every user-initiated event in JSON format on:

https://www.githubarchive.org

20+ different event types:

- PushEvent - User pushes commits into github
- WatchEvent - User starts following a repository
- IssueCommentEvent - User adds comment to an open issue
- …

Moderately high volume with wide rows: ~1.6GB/day ~600k events/day

Exercise: Github Events Data

```
[ec2-user@ip-10-146-212-13 ~]$ psql
psql (9.4.5)
Type "help" for help.

postgres=# \d
                        List of relations
 Schema |              Name           |       Type         |  Owner
--------+-----------------------------+--------------------+----------
 public | data                        | table              | ec2-user
```

Exercise: Github Events Data

Table schema for github events data:

```
CREATE TABLE data (
    id bigserial primary key,
    github_id bigint not null,
    type text not null,
    public bool not null,
    created_at timestamp NOT NULL,
    actor jsonb,
    repo jsonb,
    org jsonb,
    payload jsonb
);
```

Querying JSONB

The data table uses JSONB fields, you can query them using `->>'...'` for values and `->'...'` for values.

```
postgres=# SELECT repo->>'name' AS repo FROM data LIMIT 1;
             repo
-------------------------------
 rdpeng/ProgrammingAssignment2
(1 row)


postgres=# SELECT payload->'issue' AS issue FROM data WHERE type = 'IssuesEvent'
LIMIT 1;
                    issue
---------------------------------------------
 {"id": 53324175, "url": "...", "body" : "...
(1 row)
```

Exercise: Load Github Events data

Load data from the github archive into the table:

```
$ cd pgconfsv-tutorial
$ python loader.py dbname=postgres 2015-02-01 2015-02-02
```

```
[8240] loaded 19719 rows from data/2015-02-01-20.json.gz in 8.938035 seconds, 2206.189613 rows/sec
[8234] loaded 18595 rows from data/2015-02-01-21.json.gz in 8.085750 seconds, 2299.724863 rows/sec
[8231] loaded 17746 rows from data/2015-02-01-22.json.gz in 6.726284 seconds, 2638.306668 rows/sec
[8236] loaded 16107 rows from data/2015-02-01-23.json.gz in 5.928186 seconds, 2717.019931 rows/sec

...
```

(Data loaded in parallel)

Analyzing raw data

Let's try some queries:

```
postgres=# SELECT type, count(*) AS value FROM data GROUP BY type ORDER BY 2 DESC
LIMIT 5;
       type         |  value
--------------------+---------
 PushEvent          | 5943741
 CreateEvent        | 1400776
 WatchEvent         | 1126096
 IssueCommentEvent  | 1113041
 IssuesEvent        |  585477
(5 rows)

Time: 93926.968 ms
```

Materialized views

PostgreSQL offers materialized views, essentially cached query results.

Pros:

- Very fast queries
- Create indexes on results

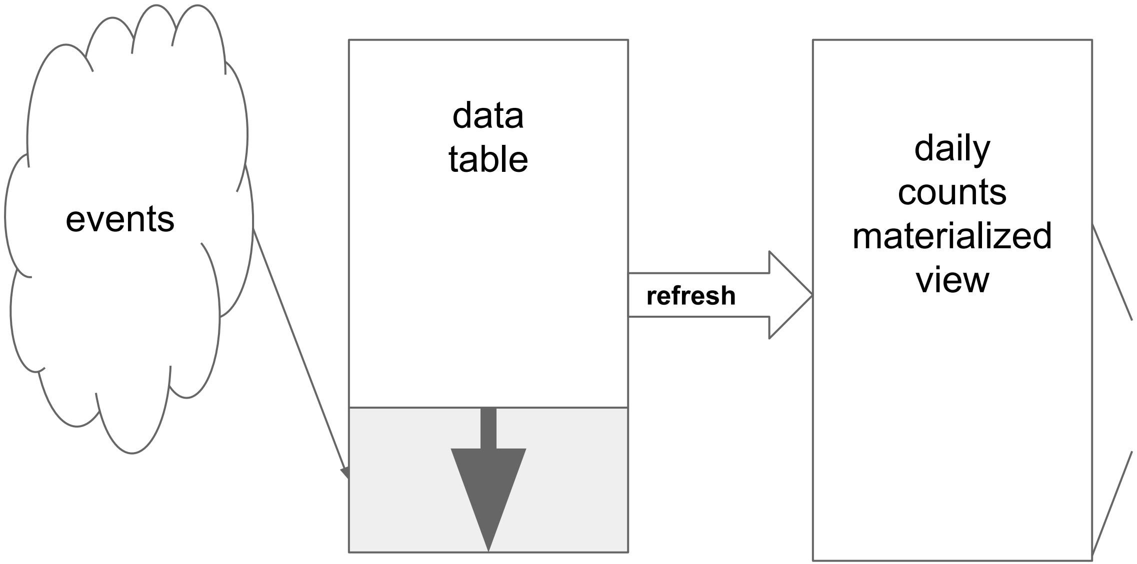
Cons:

- No incremental computation

```
CREATE MATERIALIZED VIEW view_name AS SELECT …;
REFRESH MATERIALIZED VIEW [CONCURRENTLY] view_name;
```

Materialized views

What if we cache aggregates in a materialized view?




```
SELECT * FROM data_count_daily_view
WHERE date BETWEEN ...
ORDER BY date ASC;
```

Time: 0.532 ms

Setting up materialized views

Creating the view:

```
postgres=# CREATE MATERIALIZED VIEW data_daily_counts_view AS
SELECT created_at::date AS created_date, count(*) FROM data GROUP BY created_date;
Time: 61379.565 ms
```

Fast queries:

```
postgres=# SELECT * FROM data_daily_counts_view
WHERE date BETWEEN '2015-01-01' AND '2015-01-31' ORDER BY created_date ASC;
Time: 0.532 ms
```

Setting up materialized views

Refreshing the view after new data was added:

```
postgres=# CREATE UNIQUE INDEX ON data_daily_counts_view (created_date);
```

```
postgres=# REFRESH MATERIALIZED VIEW CONCURRENTLY data_daily_counts_view;
REFRESH MATERIALIZED VIEW
Time: 49671.363 ms
```

By the time materialized view is updated, data is already old… Not real-time.

When to use materialized views

Materialized views can work well in certain situations.

Works well:

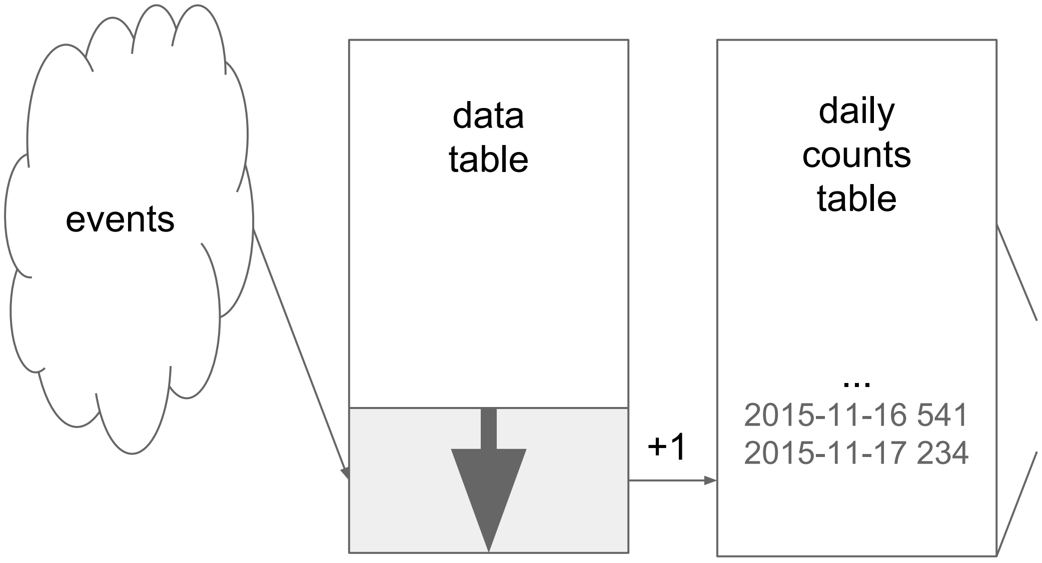
- Moderate amounts of data
- Periodic background refreshes

Works not so well:

- Aggregation over very large data set
- Very high rate of ingest
- Near real-time views

Caching aggregates

What if we cached aggregates in a table and update them in real-time?




```
SELECT * FROM data_count_daily WHERE
date BETWEEN '2015-10-17' AND '2015-11-17'
ORDER BY created_date ASC;
```
Time: 0.495 ms (cached)

use trigger to update counts table on every write

Exercise: Cached Aggregates

Create a cached daily count table:

```
postgres=# CREATE TABLE data_daily_counts_cached (
    created_date date PRIMARY KEY,
    value int
);
```

```
postgres=# INSERT INTO data_daily_counts_cached
SELECT created_at::date AS created_date, count(*)
FROM data
GROUP BY created_date;
```

We will update the count in this table whenever an event is inserted.

Exercise: Updated Cached Aggregates

Set up a trigger function that updates or inserts into the counts table:

```
postgres=# CREATE OR REPLACE FUNCTION update_daily_aggregates()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $body$
BEGIN
     LOOP
          UPDATE data_daily_counts_cached
          SET value = value+1 WHERE created_date = NEW.created_at::date;
          IF found THEN
               RETURN NULL;
          END IF;
          BEGIN
               INSERT INTO data_daily_counts_cached VALUES (NEW.created_at::date, 1);
               RETURN NULL;
          EXCEPTION WHEN unique_violation THEN
               -- retry
          END;
     END LOOP;
END;
$body$;
```

Easier with UPSERT in PostgreSQL 9.5!

Exercise: Set Up Trigger

Add the trigger to the data table:

```
postgres=# CREATE TRIGGER data_change
AFTER INSERT ON data
FOR EACH ROW
EXECUTE PROCEDURE update_daily_aggregates();
```

Trigger Exercise

Let’s load some more data!

```
$ python loader.py dbname=postgres 2015-02-03 2015-02-04
```

Trigger Exercise

Let’s load some more data!

```
$ python loader.py dbname=postgres 2015-02-03 2015-02-04

[1727] loaded 14150 rows from data/2015-02-01-0.json.gz in 36.770394 seconds, 384.820461 rows/sec
[1725] loaded 12490 rows from data/2015-02-01-2.json.gz in 62.330867 seconds, 200.382260 rows/sec
[1728] loaded 11830 rows from data/2015-02-01-9.json.gz in 73.339403 seconds, 161.304831 rows/sec
[1726] loaded 12906 rows from data/2015-02-01-1.json.gz in 127.926202 seconds, 100.886291 rows/sec
```

:-(

Exercise: Updated Cached Aggregates

Set up a trigger function that updates or inserts into the counts table:

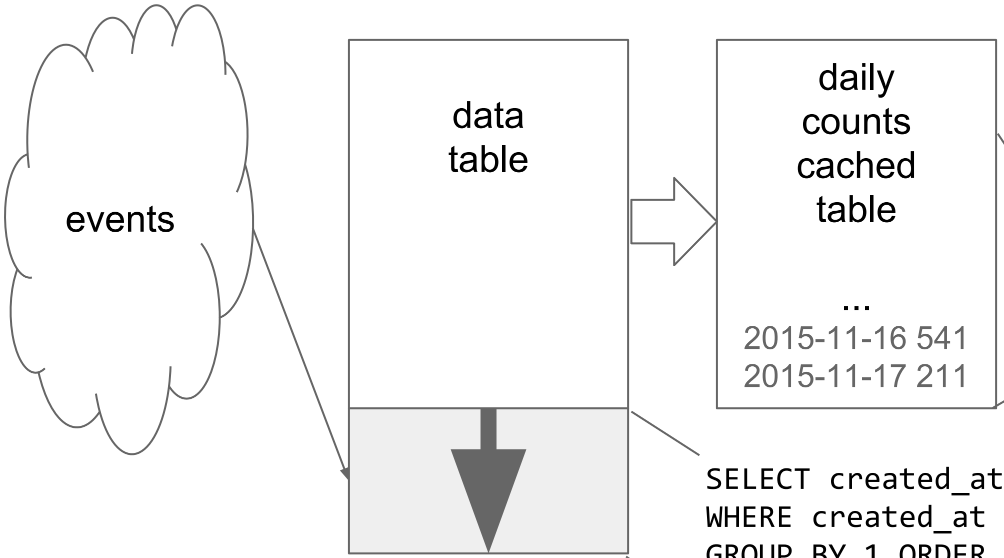
bloat

lock on created_date!

```
postgres=# CREATE OR REPLACE FUNCTION update_daily_aggregates()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $body$
BEGIN
      LOOP
            UPDATE data_daily_counts SET value = value+1 WHERE created_date = NEW.created_at::date;
            IF found THEN
                  RETURN;
            END IF;
            BEGIN
                  INSERT INTO data_daily_counts VALUES (NEW.created_at::date, 1);
                  RETURN;
            EXCEPTION WHEN unique_violation THEN
                  -- retry
            END;
      END LOOP;
END;
$body$;
```

Caching older aggregates

What if we partition the data and cache only older aggregates?



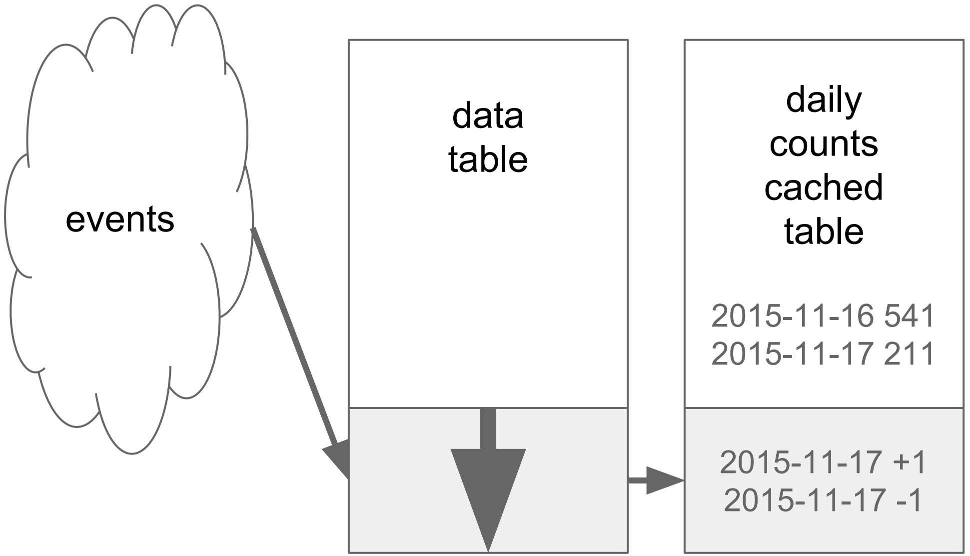

```
SELECT * FROM data_daily_counts_cached
WHERE created_date BETWEEN … AND '2015-11-16
ORDER BY created_date ASC;
Time: 0.468 ms (cached)
```

```
SELECT created_at::date, count(*) FROM data
WHERE created_at BETWEEN '2015-11-17' AND '2015-11-18'
GROUP BY 1 ORDER BY 1 ASC;
Time: 373.581 ms (not cached)
```

Updates/deletes for cached aggregations not supported, >500x slower than full cache.

Aggregate Change Queue

What if we kept a queue of changes to aggregates after insert/update/delete?

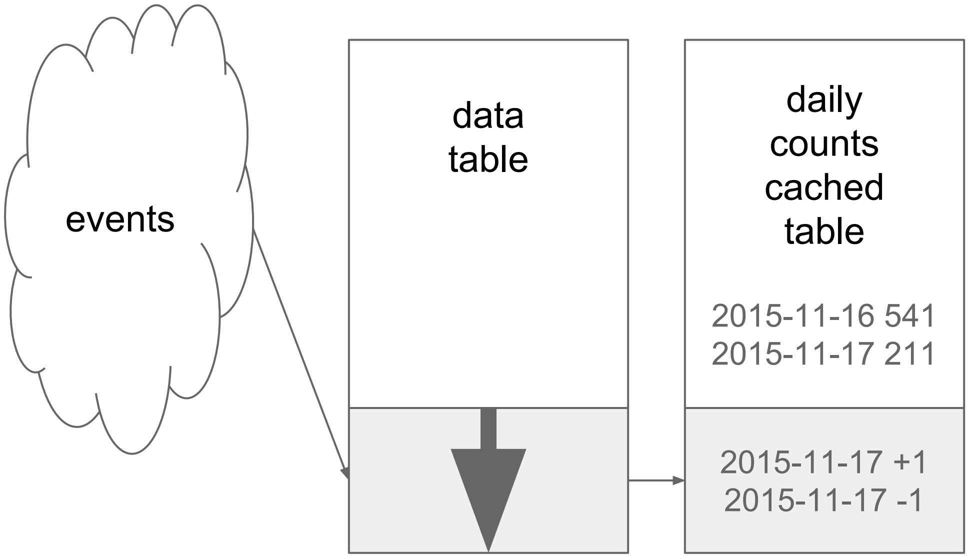


INSERT +1 on date of event
DELETE -1 on date of event
UPDATE -1 on date of old event, +1 on new

use trigger to add to queue on every write

Aggregate Change Queue

What if we kept a queue of changes to aggregates after insert/update/delete?




```
SELECT date, SUM(value)
FROM (
    SELECT created_date, value
    FROM data_daily_counts_cached
  UNION ALL
    SELECT created_date, diff AS value
    FROM data_daily_counts_queue
) combine
WHERE date BETWEEN ...
GROUP BY date;

Time: 27.515 ms
```

Can create a view to make this query transparent.

Exercise: Cached Aggregates

Queue table contains changes for every record:

```
postgres=# CREATE TABLE data_daily_counts_queue (
    created_date date,
    diff int
);
```

We will add a change to this table whenever an event is inserted.

Exercise: Change Queue

Set up a trigger function that appends to the queue table:

```
postgres=# CREATE OR REPLACE FUNCTION append_to_daily_counts_queue()
RETURNS TRIGGER LANGUAGE plpgsql
AS $body$
BEGIN
    CASE TG_OP
    WHEN 'INSERT' THEN
        INSERT INTO data_daily_counts_queue(created_date, diff)
        VALUES (date_trunc('day', NEW.created_at), +1);
    WHEN 'UPDATE' THEN
        …
    WHEN 'DELETE' THEN
        …
    END CASE;

    IF random() < 0.0001 THEN  /* 1/10,000 probability */
       PERFORM flush_daily_counts_queue();
    END IF;
    RETURN NULL;
END;
$body$;
```

Exercise: Flush Queue Table 1/5

Flush queue into cached aggregates table:

```
WITH aggregated_queue AS (
         /* Aggregate the diffs in the queue */
),   preexist AS (
         /* Find out which values already exist in the cached table */
),   perform_updates AS (
         /* UPDATE existing values in the cached aggregates table */
),   perform_inserts AS (
         /* INSERT new values into the cached aggregates table */
),   perform_prune AS (
         /* DELETE the aggregated values from the queue */
)    SELECT (SELECT count(*) FROM perform_updates) updates,
            (SELECT count(*) FROM perform_inserts) inserts,
            (SELECT count(*) FROM perform_prune) prunes
```

Exercise: Flush Queue Table 2/5

Aggregate the queue table by summing the differences:

```
WITH aggregated_queue AS (
  SELECT created_date, SUM(diff) AS value
  FROM data_daily_counts_queue
  GROUP BY created_date
)
```

We need to check which items already exist to do either INSERT or UPDATE:

```
preexist AS (
  SELECT *, EXISTS(
              SELECT *
              FROM data_daily_counts_cached materialized
              WHERE materialized.created_date = aggregated_queue.created_date
            ) does_exist
  FROM aggregated_queue
)
```

Exercise: Flush Queue Table 3/5

Update existing values in cached aggregates table:

```
perform_updates AS (
  UPDATE data_daily_counts_cached AS materialized
  SET value = materialized.value + preexist.value
  FROM preexist
  WHERE preexist.does_exist AND materialized.created_date = preexist.created_date
  RETURNING 1
),
```

Insert new values into cached aggregates table:

```
perform_inserts AS (
  INSERT INTO data_daily_counts_cached
  SELECT created_date, value
  FROM preexist
  WHERE NOT preexist.does_exist
  RETURNING 1
),
```

Exercise: Flush Queue Table 4/5

Delete values from the queue:

```
perform_prune AS (
  DELETE FROM data_daily_counts_queue
  RETURNING 1
),
```

Finally, finish the query:

```
SELECT (SELECT count(*) FROM perform_updates) updates,
       (SELECT count(*) FROM perform_inserts) inserts,
       (SELECT count(*) FROM perform_prune) prunes;
```

Exercise: Flush Queue Table 5/5

The flush function that is called from the trigger, make sure it doesn’t run concurrently:

```
CREATE OR REPLACE FUNCTION flush_daily_counts_queue()
RETURNS bool LANGUAGE plpgsql
AS $body$
BEGIN
    IF NOT pg_try_advisory_xact_lock(
                'data_daily_counts_queue'::regclass::oid::bigint) THEN
        RAISE NOTICE 'skipping queue flush';
        RETURN false;
    END IF;

    WITH aggregated_queue AS (
        …
    ) SELECT (SELECT count(*) FROM perform_updates) updates, …;

    RETURN true;
END;
$body$
```

Exercise: Set Up Trigger

Add the trigger to the data table:

```
postgres=# CREATE TRIGGER data_change after
INSERT OR UPDATE OR DELETE
ON data FOR each row
EXECUTE PROCEDURE   append_to_daily_counts_queue();
```

Exercise: Create the View

Create a view that sums the pending differences in the queue with the cached aggregates:

```
CREATE OR REPLACE VIEW data_daily_counts AS
SELECT created_date, SUM(value) AS value
FROM (
    SELECT created_date, value
    FROM data_daily_counts_cached

  UNION ALL

    SELECT created_date, diff AS value
    FROM data_daily_counts_queue

) combine

GROUP BY created_date;
```

Querying the views

Get the daily number of events:

```
postgres=# SELECT * FROM data_daily_counts ORDER BY created_date;
 created_date | value
--------------+--------
 2015-01-01   | 218934
 2015-01-02   | 335668
 2015-01-03   | 292802


Time: 0.248 ms
```

Loading Data

Data loading is still fast:

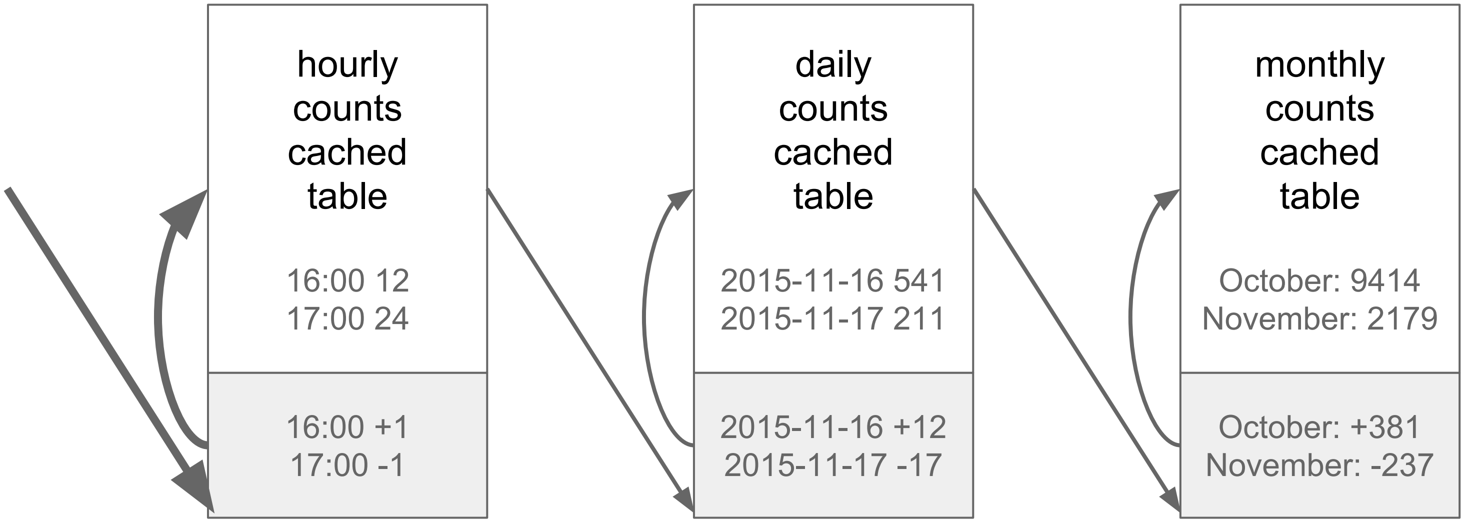
```
$ python loader.py dbname=postgres 2015-02-03 2015-02-03

[2692] loaded 15345 rows from data/2015-02-03-5.json.gz in 8.054836 seconds, 1905.066712 rows/sec
[2688] loaded 15038 rows from data/2015-02-03-4.json.gz in 8.099043 seconds, 1856.762553 rows/sec
[2694] loaded 16306 rows from data/2015-02-03-6.json.gz in 8.812405 seconds, 1850.346165 rows/sec
[2687] loaded 15759 rows from data/2015-02-03-2.json.gz in 9.026315 seconds, 1745.895201 rows/sec

...
```

Cascading aggregates

Some aggregates can be derived from finer-grained ones.



Automatically Generating Aggregates

Setting up a new view requires a lot of effort, but it can be automated:

```
SELECT pagg.create_cascaded_rollup(
    tablename        := 'data',
    rollupname       := 'data_by_type',
    group_by         := array['type'],
    cascade          := array[$$date_trunc('hour', created_at)$$,
                              $$date_trunc('day', created_at)$$,
                              $$date_trunc('month', created_at)$$],
    cascade_names    := array['hourly',
                              'daily',
                              'monthly'],
    cascade_name     := 'created_at',
    agg_count        := array['*'],
    agg_count_names  := array['countstar']
);
```

Automatically Generating Aggregates

Setting up a new view requires a lot of effort, but it can be automated:

```
SELECT pagg.create_cascaded_rollup(
    tablename      := 'data',
    rollupname     := 'data_by_type_repo',

    group_by       := array['type', $$repo->>'name'$$],
    group_by_names := array['type','reponame'],

    cascade        := array[$$date_trunc('hour', created_at)$$,
                            $$date_trunc('day', created_at)$$,
                            $$date_trunc('month', created_at)$$],
    cascade_names  := array['hourly',
                            'daily',
                            'monthly'],
    cascade_name   := 'created_at',

    agg_sum        := array[$$(payload->>'distinct_size')::int$$],
    agg_sum_names  := array['num_commits']
);
```

Querying the views

Get the daily number of events:

```
SELECT created_at, to_char(created_at, 'Day'), num_commits
FROM   data_by_type_repo_daily
WHERE  reponame = 'postgres/postgres'
       AND TYPE = 'PushEvent'
ORDER  BY created_at;
|      created_at       |  to_char   | num_commits |
+-----------------------+------------+-------------+
| 2015-01-03 00:00:00 | Saturday   |          16 |
| 2015-01-04 00:00:00 | Sunday     |          22 |
| 2015-01-05 00:00:00 | Monday     |           2 |

Time: 1.302 ms
```

Automatically Generating Aggregates

Queue tables:

```
pagg_queues.data_by_type_queue_hourly
pagg_queues.data_by_type_queue_daily
pagg_queues.data_by_type_queue_monthly
```

Materialized view tables (stale):

```
pagg_data.data_by_type_mat_hourly    + trigger into daily queue
pagg_data.data_by_type_mat_daily + trigger into monthly queue
pagg_data.data_by_type_mat_monthly
```

Views (up-to-date):

```
pagg.data_by_type_hourly        (mat. hourly  + queue hourly)
pagg.data_by_type_daily         (mat. daily   + queue daily & hourly)
pagg.data_by_type_monthly       (mat. monthly + queue monthly & daily & hourly)
```

Aggregate Size

Data size:	19GB (January)
Hourly by type:	1840 kB
Daily by type:	144 kB
Monthly by type:	24 kB
Hourly by type and repository:	440MB
Daily by type and repository:	684MB
Monthly by type and repository:	214MB

Incrementally computed Aggregates

`count(*)`: well, we did that

`sum(field)`: insert actual value into queue

`avg(field)`: compute both `count & sum`:

`avg(field) = sum(field)/ count(*)`

`count(distinct field)`: Efficient approximate solutions available (HyperLogLog)

`top-n(field)`: Approximate solutions available (count-min sketch)

HyperLogLog

Distinct count on large data sets can be very expensive.

- Need to store every item: Unbounded memory usage
- Need to do a lookup whenever a new item is counted

HyperLogLog (HLL) is an approximation algorithm for distinct counts.

- Faster counting
- Fixed memory size
- HLL datum can be incrementally updated
- Supports unions across multiple HLL data

HyperLogLog Algorithm

HyperLogLog starts by taking a hash of items counted:

```
hll_hash_text('citusdata/pg_shard')
```

The hash function will produce a uniformly distributed bit string. Unlikely patterns occurring indicates high cardinality.

Hash value with n 0-bits is observed $\rightarrow$ roughly 2^n distinct items

HyperLogLog divides values into m streams and averages the results.

HyperLogLog Algorithm

To count an item:

```
hll_add(uniques, hll_hash_text('citusdata/pg_shard'))
```

e.g. hash is 0110000011010110

First bits determine register i

4 leading 0s

$$E := \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]}\right)$$

max(4,M[i])

2	2	5	4	3	3	**4**	2	4	1	5	4	4	3	2	4

HyperLogLog for PostgreSQL

Extension to add HLL type and functions to PostgreSQL:

https://github.com/aggregateknowledge/postgresql-hll

Create a table that will track the number of distinct repositories used per day:

```
postgres=# CREATE EXTENSION hll;

postgres=# CREATE TABLE data_daily_unique_repos (
    date date,
    uniques hll
);
```

HyperLogLog for PostgreSQL

Initialize an hll:

```
INSERT INTO data_daily_unique_repos VALUES ('2015-11-17', hll_empty());
```

Count an item:

```
UPDATE data_daily_unique_repos
SET uniques = hll_add(uniques, hll_hash_text('citusdata/pg_shard'))
WHERE date = '2015-11-17';
```

Get the distinct count:

```
SELECT date, hll_cardinality(uniques) FROM data_daily_unique_repos;
    date    | hll_cardinality
------------+-----------------
 2015-11-17 |               1
```

Count distinct vs HyperLogLog

Regular distinct count:

```
SELECT created_at::DATE AS date,
       count(DISTINCT repo ->> 'id')
FROM   data
WHERE  date BETWEEN '2015-01-01' AND '2015-01-31'
GROUP BY date ORDER BY date;
```

```
    date    | count
------------+--------
 2015-01-01 |  66303
 2015-01-02 |  93976
 2015-01-03 |  85527
...
Time: 330221.164 ms
```

Count distinct vs HyperLogLog

Live HLL distinct count:

```
SELECT created_at::AS date,
       hll_cardinality(hll_add_agg(hll_hash_text(repo->>'id')))::int AS count
FROM   data
WHERE  created_at::date BETWEEN '2015-01-01' AND '2015-01-31'
GROUP  BY date
ORDER  BY date;
    date     | count
-------------+---------
 2015-01-01 |  64134
 2015-01-02 |  94512
 2015-01-03 |  86034
…
Time: 43217.501 ms
```

Count distinct vs HyperLogLog

Stored HLL distinct count:

```
SELECT date,
       hll_cardinality(uniques)
FROM   data_daily_unique_repos
WHERE  date BETWEEN '2015-01-01' AND '2015-01-31'
ORDER  BY date;
```

```
    date     | count
------------+--------
 2015-01-01 |  66265
 2015-01-02 |  94843
 2015-01-03 |  86497
```

Time: **0.830 ms**

Count distinct vs HyperLogLog

Average error:

```
SELECT avg(abs(hll.count::float / regular.count - 1))
FROM   (SELECT created_at::date AS date, count(DISTINCT repo ->> 'id') AS count
        FROM   data GROUP  BY date) regular
JOIN   (SELECT date, hll_cardinality(uniques) AS count
        FROM   data_daily_uniques) hll USING (date)
WHERE  date BETWEEN '2015-01-01' AND '2015-01-31';

        avg
-------------------
 0.017541495072427
(1 row)

Time: 147121.944 ms
```

Improving Accuracy

HLL type is configurable:

```
CREATE TABLE data_daily_unique_repos (
    date    DATE,
    uniques HLL (log2m=11, regwidth=5, expthresh=6)
);
```

log2m: Use 2^{log2m} registers (2048) in the hll data structure

Higher gives better accuracy, but uses more memory

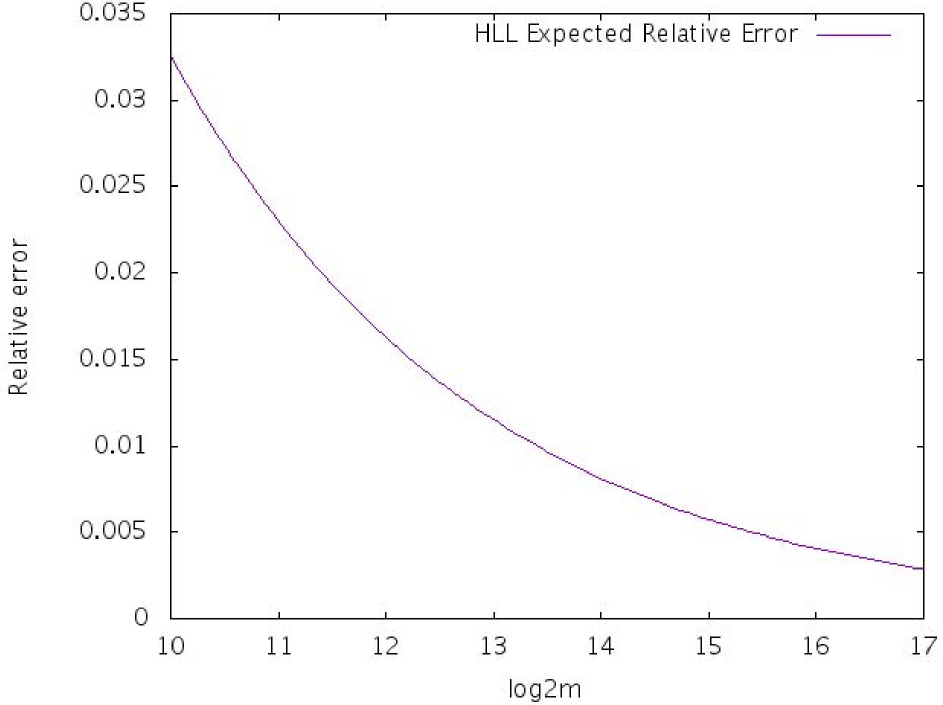
regwidth: Number of bits per register (can track up to 32 leading 0s with regwidth=5)

Higher allows counting up to higher cardinalities, but uses more memory

expthres: Keep an explicit list of hashes until reaching $2^{expthresh}$ - 1 items

HyperLogLog Accuracy

Expected relative error: $1.04/\sqrt{(2^{log2m})}$



HyperLogLog Aggregation

You can take the union of 2 hlls:

```
postgres=# SELECT ( hll_empty() || hll_hash_text('Andres') ) ||
                  ( hll_empty() || hll_hash_text('Marco') );
+--------------------------------------------+
|                   ?column?                 |
+--------------------------------------------+
| \x128b7fe3040b48478df6f901efbf7fb3771635 |
+--------------------------------------------+
(1 row)
postgres=# SELECT hll_union_agg(hll_empty() || hll_hash_text(name))
           FROM (VALUES ('Andres'),('Marco')) d(name);
```

HyperLogLog Aggregation

Since you can take the union of 2 HLLs, you can create incremental materialized views:

```
postgres=# CREATE OR REPLACE VIEW data_daily_uniques AS
SELECT created_date, hll_union_agg(uniques)
FROM (
    SELECT created_date, uniques
    FROM data_daily_uniques_cached
  UNION ALL
    SELECT created_date, increment AS value
    FROM data_daily_uniques_queue
) combine
GROUP BY created_date;
```

INSERT only!

(and set up triggers, flush function, etc...)

Distributed tables

Tables can be distributed using extensions such as pg_shard, postgres_fdw, CitusDB.

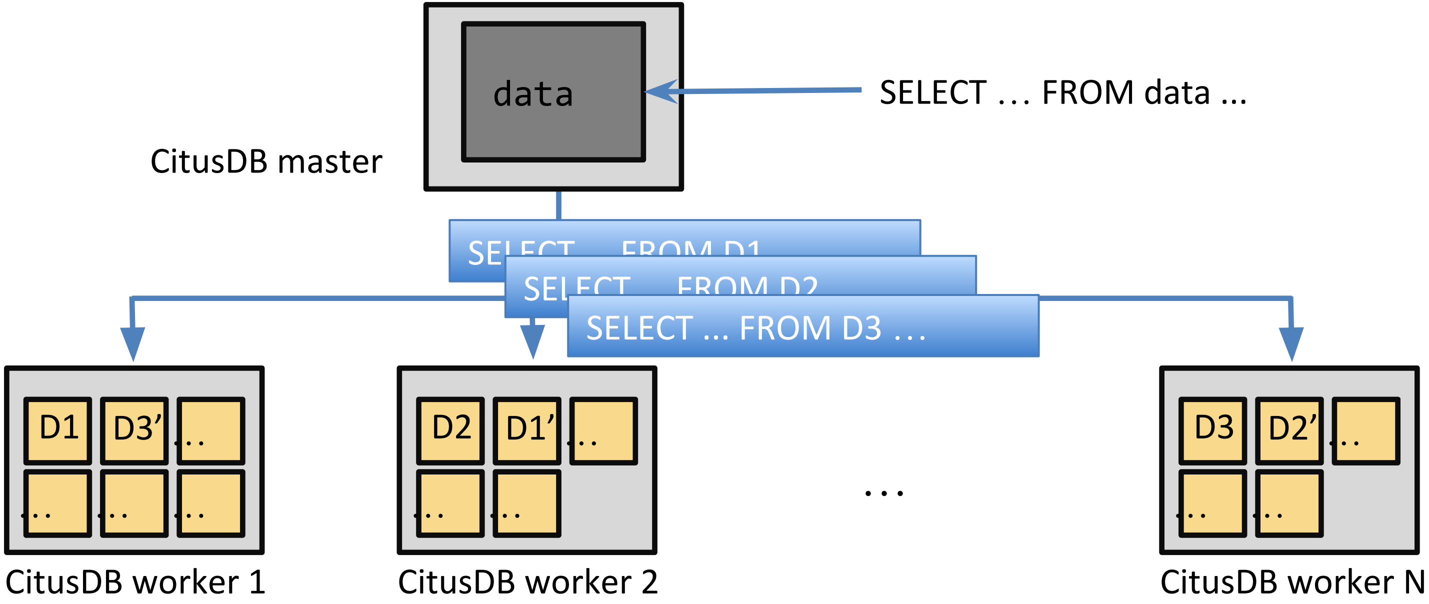
Distributed tables can be sharded using different schemes:

- **Hash-partitioning**
  Can use incrementally updated materialized view for each shard, as long as both data and aggregations are partitioned by the same key (a GROUP BY column)

- **Range-partitioning** (typically by time)
  Can use incrementally updated materialized view on the latest shard, but roll-ups over longer periods require data from multiple machines

Parallel Querying

When data is sharded, it can be queried in parallel using all the cores in the cluster.



In many cases, aggregation queries can be fast enough to be “human real-time”.

Distributed tables

Types of aggregation queries:

- **GROUP BY on partition column**:
  Queries can be pushed down directly.
  No further aggregation necessary.

- **GROUP BY only on other columns:**
  Need to aggregate across results from many shards.
  Can be expensive, especially with many rows or hlls.

- **Query on a pre-aggregated view (without GROUP BY):**
  Can be pushed down directly.

Final notes

Slides: http://goo.gl/citus-slides

Github repo: https://github.com/citusdata/pgconfsv-tutorial

andres@citusdata.com marco@citusdata.com