
Around the world with
Postgres Extensions
®

About me

- Run Citus Cloud

- Now part of Microsoft

- Previously ran Heroku Postgres
- Curate Postgres Weekly

What are extensions

- Low level hooks that allow you to change/extend behavior of Postgres
- Can be written in C or higher level language
- Can be for new functionality, new data types, or very dramatically change behavior

Extensions continued

- Postgres ships with some sort of native ones (known as contrib)

- Other extensions have to be built and installed

- Hosting providers have some set of extensions they support

Contrib extensions

- pg_stat_statements
- earthdistance
- hstore
- citext
- Postgres fdw
- UUID

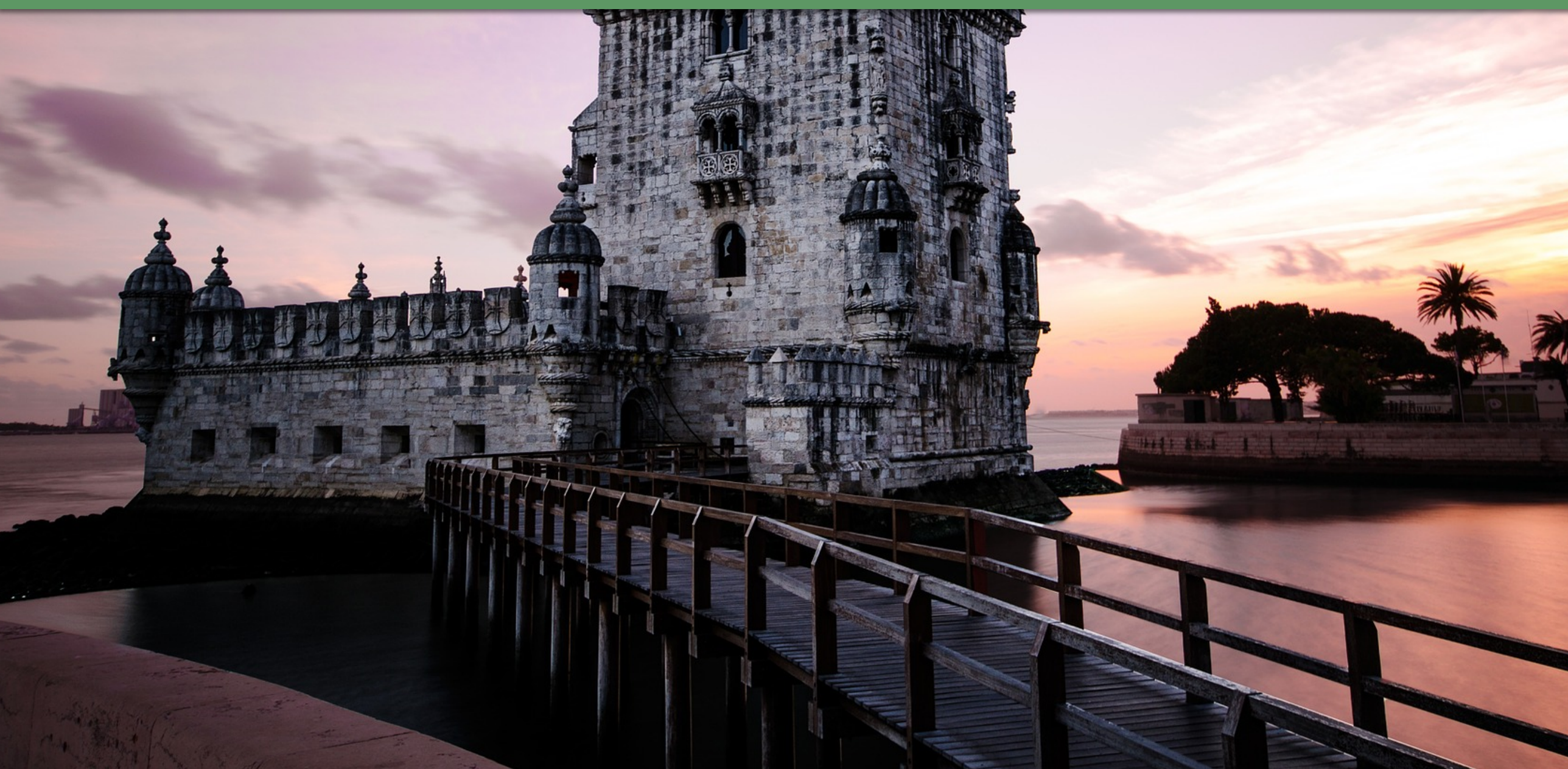
A few non-contrib

- PostGIS
- Citus
- Zombo
- HyperLogLog
- TopN
- Timescale

Today

- pg_stat_statements
- PostGIS
- HyperLogLog
- TopN
- Timescale
- pg_partman
- Citus
- FDWs

pg_stat_statements



What's it do

- Records queries that were run and all sorts of stats
  - I don't understand those stats
- Think of it as parameterizing a query,
  - How many times
  - How long it ran

What's all this?

```
SELECT *
FROM pg_stat_statements
WHERE query ~ 'from users where email';


userid              | 16384
dbid                | 16388
query               | select * from users where email = ?;
calls               | 2
total_time          | 0.000268
rows                | 2
shared_blks_hit     | 16
shared_blks_read    | 0
shared_blks_dirtied | 0
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
...
```

```sql
SELECT
  (total_time / 1000 / 60) as total,
  (total_time/calls) as avg,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;
```

Most expensive queries

```
 total  |   avg   |          query
--------+---------+--------------------------
 295.76 |  10.13 | SELECT id FROM users...
 219.13 |  80.24 | SELECT * FROM ...
(2 rows)
```

Small plug

citus_stat_statements

Citus stat statements

- Per tenant stats
- Persists tenant id
  - With all the benefits of pg_stat_statements
- Can answer
  - Which tenant is most noisy
  - Which tenant consumes most resources

PostGIS

PostGIS

The most advanced open source geospatial database

PostGIS

New datatypes

- Point
- Pointz
- Linestring
- Polygon
- Multipoint
- etc.

PostGIS

Indexing and built-in operators

GiST

```
SELECT ST_Distance(
    'POINT(37.773972, -122.431297)’::geography,
    ‘POINT(38.736946, -9.1426855)'::geography);
```

PostGIS

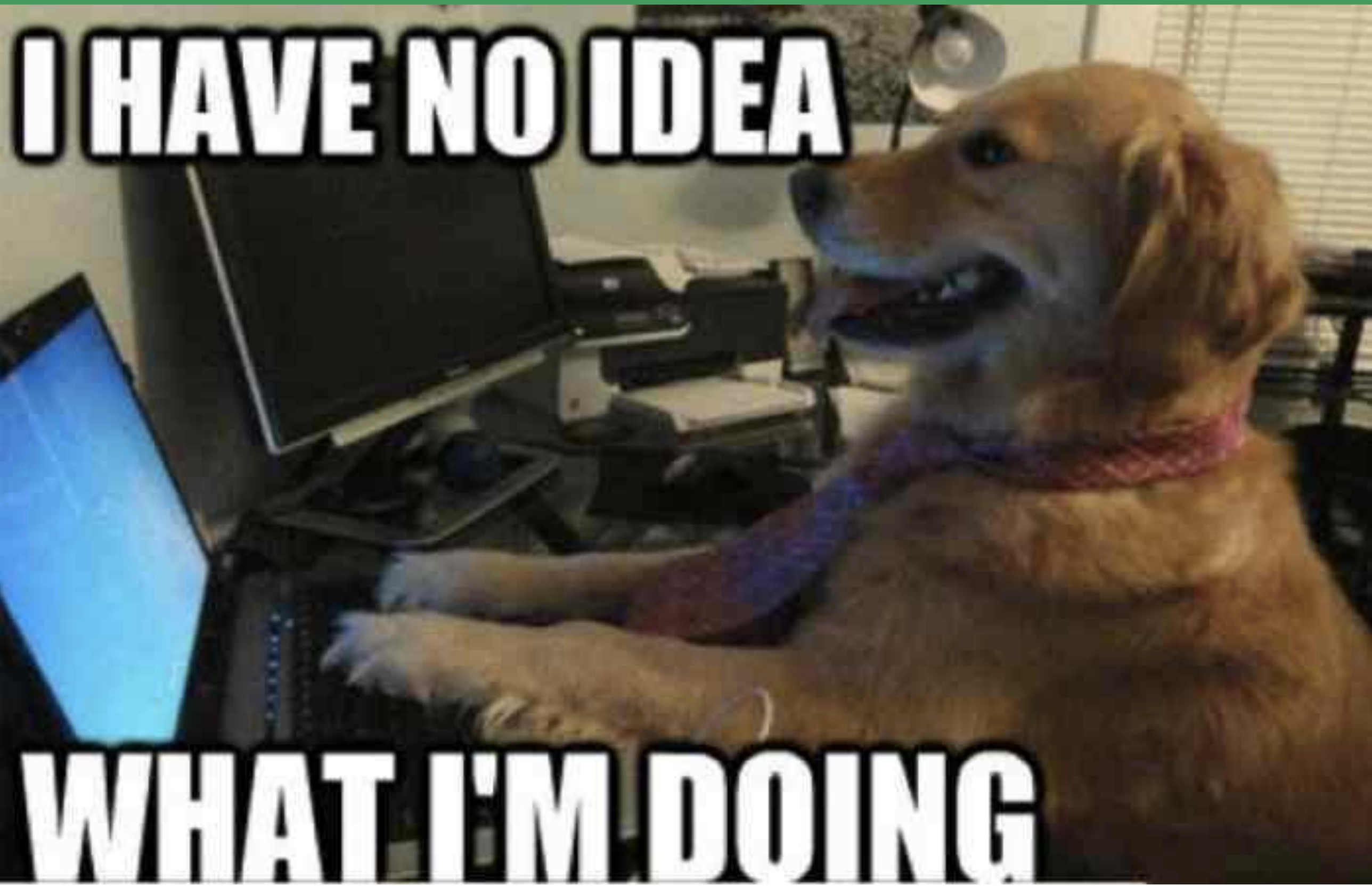
Other extensions

PgRouting - Building routing applications
ogrfdw - Query remote geospatial sources
pgpointcloud - Compress lidar data

HyperLogLog

HyperLogLog

- KMV - K minimum value
- Bit observable patterns
- Stochastic averaging
- Harmonic averaging


I HAVE NO IDEA
WHAT I'M DOING

Probabilistic uniques with a small footprint

~~Probabilistic uniques~~ with a small footprint

Close enough uniques with a small footprint

```
CREATE EXTENSION hll;
CREATE TABLE helloworld (
    id serial,
    set hll);
```

```
UPDATE helloworld
SET set = hll_add(set, hll_hash_integer(12345))
WHERE id = 1;

UPDATE helloworld
SET set = hll_add(set, hll_hash_text('hello
world’))
WHERE id = 1;
```

In the real world

```
CREATE TABLE daily_uniques (
    date    date unique,
    users    hll
);
```

Real world

```
INSERT INTO daily_uniques (date, users)
  SELECT date,
         hll_add_agg(hll_hash_integer(user_id))
    FROM users
    GROUP by 1;
```

Real world

```
SELECT users
FROM daily_uniques;

users |
\x128b7f80c8b96a7321f6cec13b12ffb2242de7db3caddf
2d017737de20d6599165344df333908c162802510d314fd3
0926f3b50f16075cba0b38e94becbcd8bc8625e56066c5e4
cf119639775c69d027242eec7da2b22671431358
```

```sql
SELECT
    EXTRACT (month from date) as month,
    hll_cardinality(hll_union_agg(users))
FROM daily_uniques
WHERE date >= '2018-08-01'
  AND date <  '2018-11-01'
GROUP BY 1;
```

Some best practices

- It uses update
  - Do as batch in most cases
- Tweak the config

Tuning parameters

- log2m - log base 2 of registers
  - Between 4 and 17
  - An increase of 1 doubles storage
- regwidth - bits per register
- expthresh - threshold for explicit vs. sparse
- spareson - toggle sparse on/off

Is it better?

1280 bytes
Estimate count of 10s of billions
Few percent error

Common use cases

Close enough uniques
  Ad networks
  Web analytics

Bonus: HLL is composable, unions, intersections, etc.

TopN

TopN

Top list of people that have done X/Y

Generally accurate on who is in the top

i.e.

- Top 10 websites browsed
- Top most frequent visitors
- Top 10 search terms

TopN

```
CREATE TABLE aggregated_topns (day date, topn jsonb);

INSERT INTO
            aggregated_topns select date_trunc('day', created_at),
            topn_add_agg((repo::json)->> 'name') as topn
FROM github_events
GROUP BY 1;
```

TopN leverages JSONB

```
SELECT top_users
FROM page_views;

top_users_1000          | {"490": 5, "1958": 4,
"5260": 4, "5678": 4, "5864": 3, "6042": 3,
"6498": 3, "7466": 3, "8343": 3, "8843": 3,
"8984": 3}
```

```
SELE\CT (topn(topn_union_agg(topn), 10)).*
FROM aggregated_topns
WHERE day IN ('2018-01-02', '2018-01-03');
                      item                       | frequency
-------------------------------------------------+-----------
 dipper-github-fra-sin-syd-nrt/test-ruby-sample |     12489
 wangshub/wechat_jump_game                       |      6402
 shenzhouzd/update                               |      6170
 SCons/scons                                     |      4593
 TheDimPause/thedimpause.github.io               |      3964
 nicopeters/sigrhtest                            |      3740
 curtclifton/curtclifton.github.io               |      3345
 CreatorB/hackerdroid                            |      3206
 dipper-github-icn-bom-cdg/test-ruby-sample      |      3126
 dotclear/dotclear                               |      2992
(10 rows)
```

```
select day, (topn(topn, 2)).* from aggregated_topns where day IN
('2018-01-01', '2018-01-02', '2018-01-03');
    day     |                          item                          | frequency
------------+--------------------------------------------------------+
-----------
 2018-01-01 | dipper-github-fra-sin-syd-nrt/test-ruby-sample |      9179
 2018-01-01 | shenzhouzd/update                                      |      4543
 2018-01-02 | dipper-github-fra-sin-syd-nrt/test-ruby-sample |      7151
 2018-01-02 | SCons/scons                                            |      4593
 2018-01-03 | dipper-github-fra-sin-syd-nrt/test-ruby-sample |      5338
 2018-01-03 | CreatorB/hackerdroid                                   |      3206
(6 rows)
```

Timescale

Timescale

- **Time-centric**: Data records always have a timestamp.
- **Append-only**: Data is almost solely append-only (INSERTs).
- **Recent**: New data is typically about recent time intervals, and we more rarely make updates or backfill missing data about old intervals.

Timescale

```
CREATE TABLE "rides"(
    vendor_id TEXT,
    pickup_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    dropoff_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    passenger_count NUMERIC,
    trip_distance NUMERIC,
    payment_type INTEGER,
    fare_amount NUMERIC,
    extra NUMERIC,
    mta_tax NUMERIC,
    tip_amount NUMERIC,
    tolls_amount NUMERIC,
    improvement_surcharge NUMERIC,
    total_amount NUMERIC
);
```

```
SELECT create_hypertable('rides',
'pickup_datetime');
```

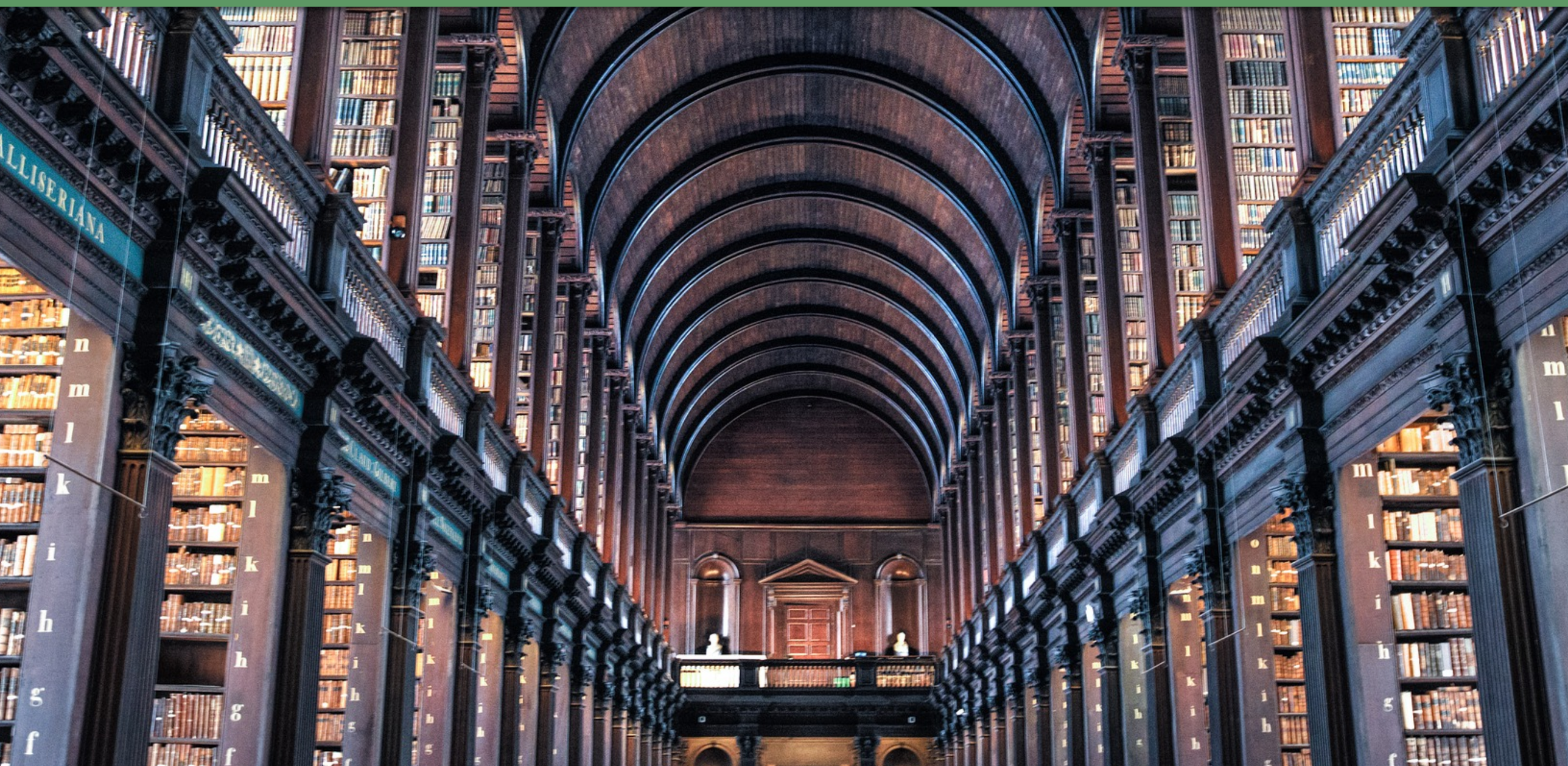
```
SELECT date_trunc('day', pickup_datetime) as day,
       avg(fare_amount)
FROM rides
WHERE passenger_count > 1
  AND pickup_datetime < '2016-01-08'
GROUP BY day
ORDER BY day;
```

```
         day          |        avg
----------------------+--------------------
2016-01-01 00:00:00 | 13.3990821679715529
2016-01-02 00:00:00 | 13.0224687415181399
2016-01-03 00:00:00 | 13.5382068607068607
2016-01-04 00:00:00 | 12.9618895561740149
2016-01-05 00:00:00 | 12.6614611935518309
2016-01-06 00:00:00 | 12.5775245695086098
2016-01-07 00:00:00 | 12.5868802584437019
(7 rows)
```

```
SELECT time_bucket('5 minute', pickup_datetime) AS five_min,
       count(*)
FROM rides
WHERE pickup_datetime < '2016-01-01 02:00'
GROUP BY five_min
ORDER BY five_min;
```

```
      five_min         | count
-----------------------+-------
 2016-01-01 00:00:00 |   703
 2016-01-01 00:05:00 |  1482
 2016-01-01 00:10:00 |  1959
 2016-01-01 00:15:00 |  2200
 2016-01-01 00:20:00 |  2285
```

pg_partman



pg_partman

Another time partitioning extension

Builds on native Postgres time partitioning

pg_partman

```
CREATE SCHEMA github;

CREATE TABLE github.events (
  event_id bigint,
  event_type text,
  event_public boolean,
  repo_id bigint,
  payload jsonb,
  repo jsonb, actor jsonb,
  org jsonb,
  created_at timestamp
) PARTITION BY RANGE (created_at);
```

```
SELECT partman.create_parent('github.events',
'created_at', 'native', 'hourly');

UPDATE partman.part_config SET
infinite_time_partitions = true;
```

```
                  List of relations
 Schema |             Name              |   Type   |  Owner
--------+-------------------------------+----------+----------
 public | events                        | table    | citus
 public | events_event_id_seq           | sequence | citus
 public | events_p2018_10_23_0900       | table    | citus
 public | events_p2018_10_23_0905       | table    | citus
 public | events_p2018_10_23_0910       | table    | citus
 public | events_p2018_10_23_0915       | table    | citus
 public | events_p2018_10_23_0920       | table    | citus
 public | events_p2018_10_23_0925       | table    | citus
 public | events_p2018_10_23_0930       | table    | citus
 public | events_p2018_10_23_0935       | table    | citus
```

```
SELECT * from partman.part_config;
-[ RECORD 1 ]--------------+-----------------------------
parent_table               | public.events
control                    | event_time
partition_type             | native
partition_interval         | 00:05:00
constraint_cols            | #
premake                    | 4
optimize_trigger           | 4
optimize_constraint        | 30
epoch                      | none
inherit_fk                 | t
retention                  | #
retention_schema           | #
retention_keep_table       | t
retention_keep_index       | t
infinite_time_partitions   | t
datetime_string            | YYYY_MM_DD_HH24MI
automatic_maintenance      | on
jobmon                     | t
sub_partition_set_full     | f
undo_in_progress           | f
trigger_exception_handling | f
upsert                     |
trigger_return_null        | t
template_table             | partman.template_public_events
publications               | #
```

Why not native partitioning

Partman is native, with extra bells and whistles

Automates creating/dropping partitions

Citus

Citus

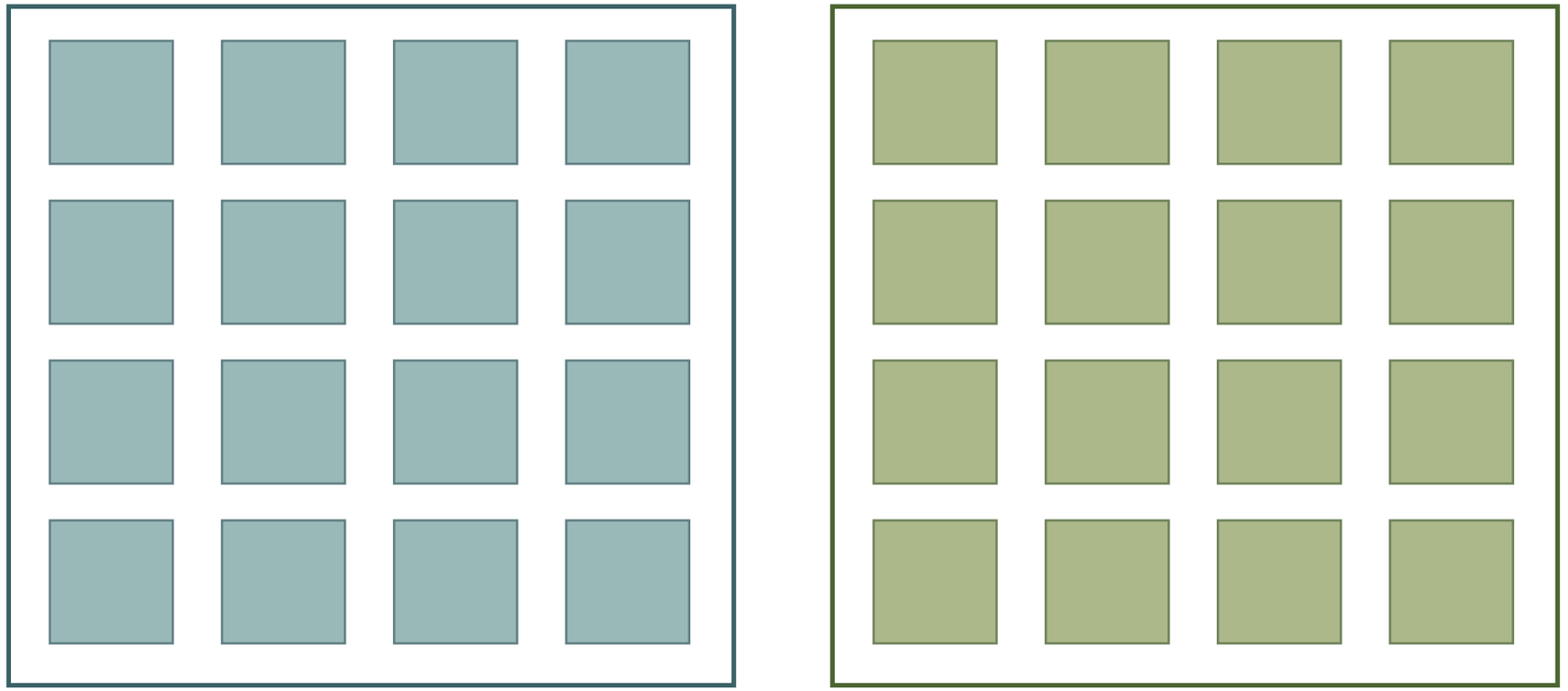
Turns postgres into a distributed, horizontally scalable database

You application still thinks it's a single node, under the covers, it is all sharded

What is sharding

Splitting database into smaller parts

Nodes contain shards



More details

- Hash based on some id
- Postgres internal hash can work fine, or so can your own
- Define your number of shards up front, make this larger than you expect to grow to in terms of nodes
  - (2 is bad)
  - (2 million is also bad)
  - Factors of 2 are nice, but not actually required

Don’t just route values

- 1-10 -> shard 1
- 2-20 -> shard 2

Create range of hash values

- hash 1 = 46154
- hash 2 = 27193

- Shard 13 = ranges 26624 to 28672

How does sharding work

- Events table
  - Events_001
  - Events_002
  - Events_003

Github event data

```
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    user_id bigint,
    org jsonb,
    created_at timestamp
);

CREATE TABLE github_users
(
    user_id bigint,
    url text,
    login text,
    avatar_url text,
    gravatar_id text,
    display_login text
);
```

Distributing data

```
SELECT create_distributed_table('github_events',
'user_id');
SELECT create_distributed_table('github_users',
'user_id');
```

```
SELECT count(*) from github_events;
 count
--------
 126245
(1 row)
```

Real-time executor

Parallelizes queries across all nodes

```
SELECT count(*)
FROM events
```

```
Aggregate  (cost=0.00..0.00 rows=0 width=0)
   ->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00
rows=0 width=0)
         Task Count: 32
         Tasks Shown: One of 32
         ->  Task
               Node: host=ec2-23-22-189-35...
               ->  Aggregate  (cost=488.05..488.06
rows=1 width=8)
```

Real-time executor

Parallelizes queries across all nodes

```
SELECT count(*)
FROM events
```

```
Aggregate  (cost=0.00..0.00 rows=0 width=0)
   ->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00
rows=0 width=0)
         Task Count: 32
         Tasks Shown: One of 32
         ->  Task
               Node: host=ec2-23-22-189-35...
               ->  Aggregate  (cost=488.05..488.06
rows=1 width=8)
```

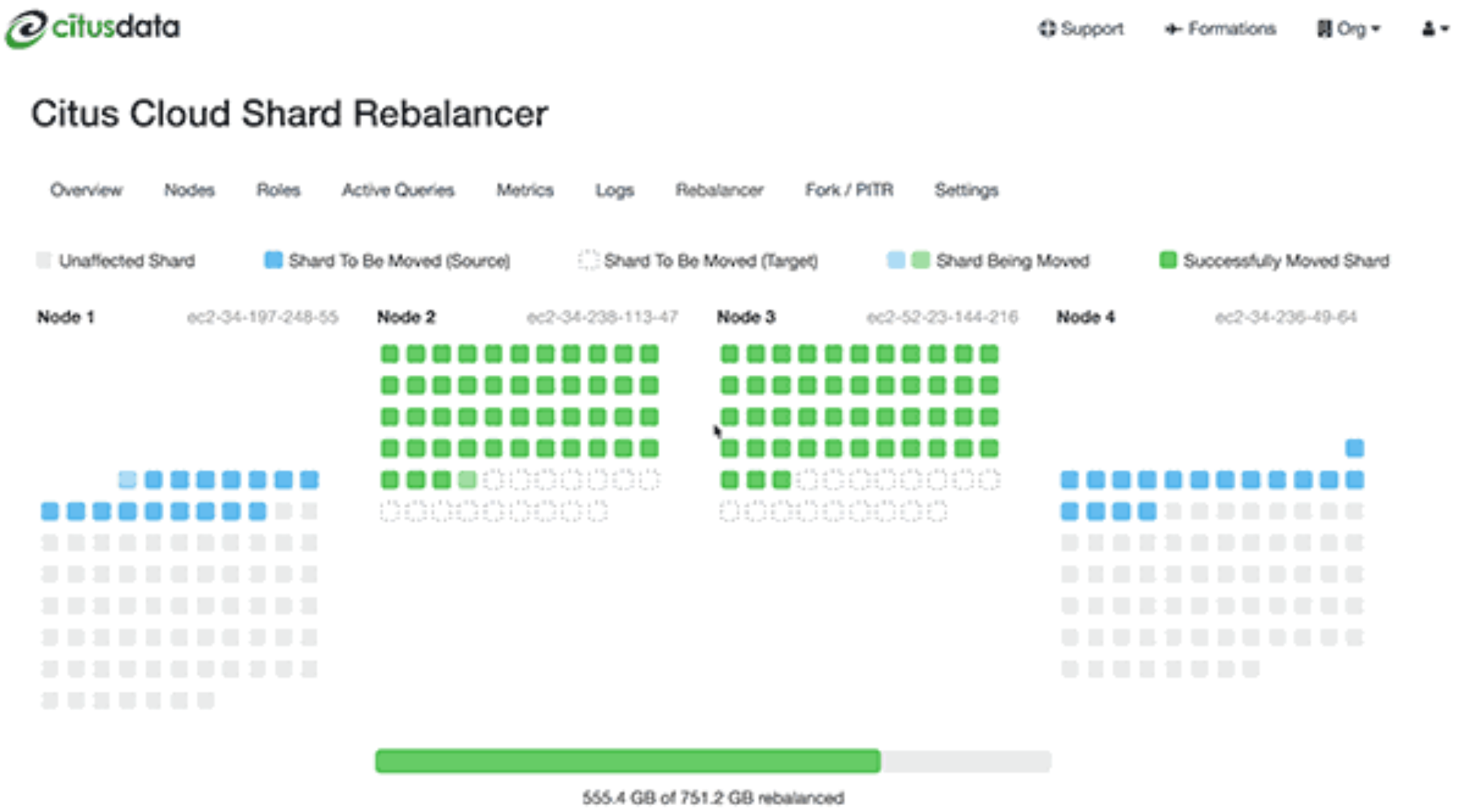
Router executor

Routes queries to single shard

```
SELECT count(*)
FROM events
WHERE customer_id = 1
```

```
 Custom Scan (Citus Router)  (cost=0.00..0.00
rows=0 width=0)
   Task Count: 1
   Tasks Shown: All
   ->  Task
         Node: host=ec2-35-173-16-44...
         ->  Aggregate  (cost=574.21..574.22
rows=1 width=8)
```

Rebalancing moves shards for scaling



FDWs

FDWs

- Foreign Data Wrapper
- Connect from within Postgres to something else
  - (or to Postgres)
- Examples
  - Redis
  - Mongo
  - cstore
  -

Create foreign server

```
CREATE SERVER redis_server
    FOREIGN DATA WRAPPER redis_fdw
    OPTIONS (address '127.0.0.1', port '6379');

CREATE FOREIGN TABLE redis_db0 (key text, value text)
    SERVER redis_server
    OPTIONS (database '0');

CREATE USER MAPPING FOR PUBLIC
    SERVER redis_server
    OPTIONS (password 'secret');
```

```
# \d
                List of relations
 Schema |       Name        |     Type      | Owner
--------+-------------------+---------------+-------
 public | products          | table         | craig
 public | purchases         | table         | craig
 public | redis_db0         | foreign table | craig
 public | users             | table         | craig
(4 rows)
```

```
SELECT *
FROM redis_db0
LIMIT 5;
   key    | value
----------+-------
 user_40 | 44
 user_41 | 32
 user_42 | 11
 user_43 | 3
 user_80 | 7
(5 rows)
```

```
SELECT
  id,
  email,
  value as visits
FROM
  users,
  redis_db0
WHERE
      ('user_' || cast(id as text)) = cast(redis_db0.key as text)
  AND cast(value as int) > 40;

 id |            email             | visits
----+------------------------------+--------
 40 | Cherryl.Crissman@gmail.com | 44
 44 | Brady.Paramo@gmail.com     | 44
 46 | Laronda.Razor@yahoo.com    | 44
 47 | Karole.Sosnowski@gmail.com | 44
 12 | Jami.Jeon@yahoo.com        | 49
 14 | Jenee.Morrissey@gmail.com  | 47
(6 rows)
```

In conclusion

Postgres is more than Postgres

The next time you want something you think Postgres doesn’t do, explore extensions, or consider writing one

Honorable mentions

- pgsql-http
- cstore
- pg_repack
- Madlib
- pg_cron
- ZomboDB

Further reading

https://www.citusdata.com/blog/2017/10/25/what-it-means-to-be-a-postgresql-extension/

https://pgxn.org/

http://big-elephants.com/2015-10/writing-postgres-extensions-part-i/

Thanks