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?

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>userid</td>
<td>16384</td>
</tr>
<tr>
<td>dbid</td>
<td>16388</td>
</tr>
<tr>
<td>query</td>
<td>select * from users where email = ?;</td>
</tr>
<tr>
<td>calls</td>
<td>2</td>
</tr>
<tr>
<td>total_time</td>
<td>0.000268</td>
</tr>
<tr>
<td>rows</td>
<td>2</td>
</tr>
<tr>
<td>shared_blks_hit</td>
<td>16</td>
</tr>
<tr>
<td>shared_blks_read</td>
<td>0</td>
</tr>
<tr>
<td>shared_blks_dirtied</td>
<td>0</td>
</tr>
<tr>
<td>shared_blks_written</td>
<td>0</td>
</tr>
<tr>
<td>local_blks_hit</td>
<td>0</td>
</tr>
<tr>
<td>local_blks_read</td>
<td>0</td>
</tr>
<tr>
<td>local_blks_dirtied</td>
<td>0</td>
</tr>
<tr>
<td>local_blks_written</td>
<td>0</td>
</tr>
</tbody>
</table>

...
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

<table>
<thead>
<tr>
<th>total</th>
<th>avg</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>295.76</td>
<td>10.13</td>
<td>SELECT id FROM users...</td>
</tr>
<tr>
<td>219.13</td>
<td>80.24</td>
<td>SELECT * FROM ...</td>
</tr>
</tbody>
</table>

(2 rows)
Small plug

citus_stat_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

```sql
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

- 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
);

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

```sql
SELECT users
FROM daily_uniques;
```

<table>
<thead>
<tr>
<th>users</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\x128b7f80c8b96a7321f6cec13b12ffb2242de7db3caddf2d017737de20d6599165344df333908c162802510d314fd30926f3b50f16075cba0b38e94becbcd8bc8625e56066c5e4cf119639775c69d027242eec7da2b22671431358</code></td>
</tr>
</tbody>
</table>
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

- \( \log_2 m \) - log base 2 of registers
  - Between 4 and 17
  - An increase of 1 doubles storage

- \( \text{regwidth} \) - bits per register

- \( \text{expthresh} \) - threshold for explicit vs. sparse

- \( \text{spareson} \) - 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

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
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

```sql
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}
```
SELECT (topn(topn_union_agg(topn), 10)).*
FROM aggregated_topns
WHERE day IN ('2018-01-02', '2018-01-03');

<table>
<thead>
<tr>
<th>item</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dipper-github-fra-sin-syd-nrt/test-ruby-sample</td>
<td>12489</td>
</tr>
<tr>
<td>wangshub/wechat_jump_game</td>
<td>6402</td>
</tr>
<tr>
<td>shenzhouzd/update</td>
<td>6170</td>
</tr>
<tr>
<td>SCons/scons</td>
<td>4593</td>
</tr>
<tr>
<td>TheDimPause/thedimpause.github.io</td>
<td>3964</td>
</tr>
<tr>
<td>nicopeters/sigrhtest</td>
<td>3740</td>
</tr>
<tr>
<td>curtclifton/curtclifton.github.io</td>
<td>3345</td>
</tr>
<tr>
<td>CreatorB/hackerdroid</td>
<td>3206</td>
</tr>
<tr>
<td>dipper-github-icn-bom-cdg/test-ruby-sample</td>
<td>3126</td>
</tr>
<tr>
<td>dotclear/dotclear</td>
<td>2992</td>
</tr>
</tbody>
</table>

(10 rows)
```sql
select day, (topn(topn, 2)).* from aggregated_topns where day IN ('2018-01-01', '2018-01-02', '2018-01-03');
```

<table>
<thead>
<tr>
<th>day</th>
<th>item</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-01</td>
<td>dipper-github-fra-sin-syd-nrt/test-ruby-sample</td>
<td>9179</td>
</tr>
<tr>
<td>2018-01-01</td>
<td>shenzhouzd/update</td>
<td>4543</td>
</tr>
<tr>
<td>2018-01-02</td>
<td>dipper-github-fra-sin-syd-nrt/test-ruby-sample</td>
<td>7151</td>
</tr>
<tr>
<td>2018-01-02</td>
<td>SCons/scons</td>
<td>4593</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>dipper-github-fra-sin-syd-nrt/test-ruby-sample</td>
<td>5338</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>CreatorB/hackerdroid</td>
<td>3206</td>
</tr>
</tbody>
</table>

(6 rows)
**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.
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');
```sql
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;
```

<table>
<thead>
<tr>
<th>day</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-01 00:00:00</td>
<td>13.3990821679715529</td>
</tr>
<tr>
<td>2016-01-02 00:00:00</td>
<td>13.0224687415181399</td>
</tr>
<tr>
<td>2016-01-03 00:00:00</td>
<td>13.5382068607068607</td>
</tr>
<tr>
<td>2016-01-04 00:00:00</td>
<td>12.9618895561740149</td>
</tr>
<tr>
<td>2016-01-05 00:00:00</td>
<td>12.6614611935518309</td>
</tr>
<tr>
<td>2016-01-06 00:00:00</td>
<td>12.5775245695086098</td>
</tr>
<tr>
<td>2016-01-07 00:00:00</td>
<td>12.5868802584437019</td>
</tr>
</tbody>
</table>
(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;

<table>
<thead>
<tr>
<th>five_min</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-01 00:00:00</td>
<td>703</td>
</tr>
<tr>
<td>2016-01-01 00:05:00</td>
<td>1482</td>
</tr>
<tr>
<td>2016-01-01 00:10:00</td>
<td>1959</td>
</tr>
<tr>
<td>2016-01-01 00:15:00</td>
<td>2200</td>
</tr>
<tr>
<td>2016-01-01 00:20:00</td>
<td>2285</td>
</tr>
</tbody>
</table>
pg_partman

Another time partitioning extension

Builds on native Postgres time partitioning
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;
<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>events</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_event_id_seq</td>
<td>sequence</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0900</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0905</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0910</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0915</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0920</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0925</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0930</td>
<td>table</td>
<td>citus</td>
</tr>
<tr>
<td>public</td>
<td>events_p2018_10_23_0935</td>
<td>table</td>
<td>citus</td>
</tr>
</tbody>
</table>
SELECT * from partman.part_config;

<table>
<thead>
<tr>
<th>RECORD 1</th>
<th>-----------------------------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent_table</td>
<td>public.events</td>
</tr>
<tr>
<td>control</td>
<td>event_time</td>
</tr>
<tr>
<td>partition_type</td>
<td>native</td>
</tr>
<tr>
<td>partition_interval</td>
<td>00:05:00</td>
</tr>
<tr>
<td>constraint_cols</td>
<td>#</td>
</tr>
<tr>
<td>premake</td>
<td>4</td>
</tr>
<tr>
<td>optimize_trigger</td>
<td>4</td>
</tr>
<tr>
<td>optimize_constraint</td>
<td>30</td>
</tr>
<tr>
<td>epoch</td>
<td>none</td>
</tr>
<tr>
<td>inherit_fk</td>
<td>t</td>
</tr>
<tr>
<td>retention</td>
<td>#</td>
</tr>
<tr>
<td>retention_schema</td>
<td>#</td>
</tr>
<tr>
<td>retention_keep_table</td>
<td>t</td>
</tr>
<tr>
<td>retention_keep_index</td>
<td>t</td>
</tr>
<tr>
<td>infinite_time_partitions</td>
<td>t</td>
</tr>
<tr>
<td>datetime_string</td>
<td>YYYY_MM_DD_HH24MI</td>
</tr>
<tr>
<td>automatic_maintenance</td>
<td>on</td>
</tr>
<tr>
<td>jobmon</td>
<td>t</td>
</tr>
<tr>
<td>sub_partition_set_full</td>
<td>f</td>
</tr>
<tr>
<td>undo_in_progress</td>
<td>f</td>
</tr>
<tr>
<td>trigger_exception_handling</td>
<td>f</td>
</tr>
<tr>
<td>upsert</td>
<td></td>
</tr>
<tr>
<td>trigger_return_null</td>
<td>t</td>
</tr>
<tr>
<td>template_table</td>
<td>partman.template_public_events</td>
</tr>
<tr>
<td>publications</td>
<td>#</td>
</tr>
</tbody>
</table>
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

```sql
SELECT create_distributed_table('github_events', 'user_id');
SELECT create_distributed_table('github_users', 'user_id');
```
SELECT count(*) from github_events;

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>126245</td>
</tr>
</tbody>
</table>

(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

Parallellizes 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 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

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>products</td>
<td>table</td>
<td>craig</td>
</tr>
<tr>
<td>public</td>
<td>purchases</td>
<td>table</td>
<td>craig</td>
</tr>
<tr>
<td>public</td>
<td>redis_db0</td>
<td>foreign table</td>
<td>craig</td>
</tr>
<tr>
<td>public</td>
<td>users</td>
<td>table</td>
<td>craig</td>
</tr>
</tbody>
</table>

(4 rows)
```sql
SELECT * 
FROM redis_db0 
LIMIT 5;
```

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_40</td>
<td>44</td>
</tr>
<tr>
<td>user_41</td>
<td>32</td>
</tr>
<tr>
<td>user_42</td>
<td>11</td>
</tr>
<tr>
<td>user_43</td>
<td>3</td>
</tr>
<tr>
<td>user_80</td>
<td>7</td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>id</th>
<th>email</th>
<th>visits</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td><a href="mailto:Cherryl.Crissman@gmail.com">Cherryl.Crissman@gmail.com</a></td>
<td>44</td>
</tr>
<tr>
<td>44</td>
<td><a href="mailto:Brady.Paramo@gmail.com">Brady.Paramo@gmail.com</a></td>
<td>44</td>
</tr>
<tr>
<td>46</td>
<td><a href="mailto:Laronda.Razor@yahoo.com">Laronda.Razor@yahoo.com</a></td>
<td>44</td>
</tr>
<tr>
<td>47</td>
<td><a href="mailto:Karole.Sosnowski@gmail.com">Karole.Sosnowski@gmail.com</a></td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td><a href="mailto:Jami.Jeon@yahoo.com">Jami.Jeon@yahoo.com</a></td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td><a href="mailto:Jenee.Morrissey@gmail.com">Jenee.Morrissey@gmail.com</a></td>
<td>47</td>
</tr>
</tbody>
</table>

(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://pgxn.org/

Thanks