

# Scaling out PostgreSQL for write-intensive workloads

Marco Slot <drmarco@citusdata.com>

Lead devs: jason@ sumedh@

**citusdata**

# Why not postgres?

Real-time, data-intensive applications  
require **horizontal scaling**

NoSQL provides **seamless** horizontal scaling

What if PostgreSQL could do that?

# Citus Data family

**cstore\_fdw** (github)

Columnar storage foreign data wrapper

**CitusDB** (citusdata.com)

Real-time analytics on sharded tables



**pg\_shard** (github)

Transparently shards tables for real-time reads & writes

# What is CitusDB?

CitusDB is a scalable analytics database that extends PostgreSQL

- CitusDB shards your data and automatically parallelizes your queries
- CitusDB hooks onto the planner and executor for distributed query execution.
- Always rebased to newest Postgres version
- Natively supports new data types and extensions

# #1 Requested Feature from Citus

Real-time analytics calls for **real-time data ingestion**

Some customers built their own real-time insert solutions

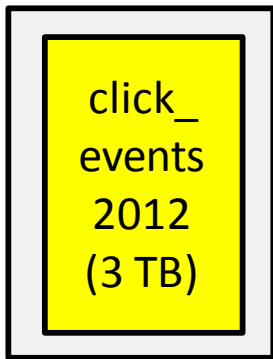
We also talked to PostgreSQL users. Some considered application level sharding or migrating to NoSQL solutions.

# Customer Interviews

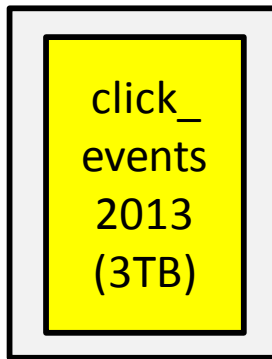
- Dynamically scale a cluster as new machines are added or old ones are retired.
- Handle node failures.
- Simple to set up and use. Works natively on PostgreSQL.
- Transactional semantics aren't as important.

# Technical challenges

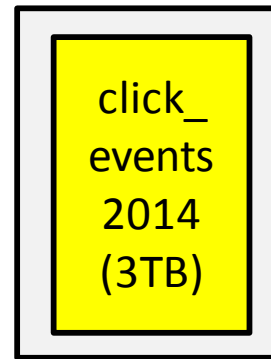
- How to shard data across cluster?
- What to do in case of failure?
- How to perform distributed query planning and execution?
- How to make it seamlessly work with postgres?



node #1 (PostgreSQL)



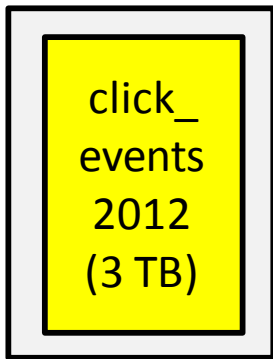
node #2



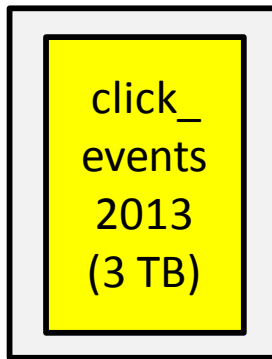
node #3

Standard sharding approach

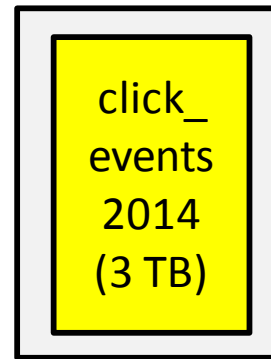




node #1 (PostgreSQL)



node #2

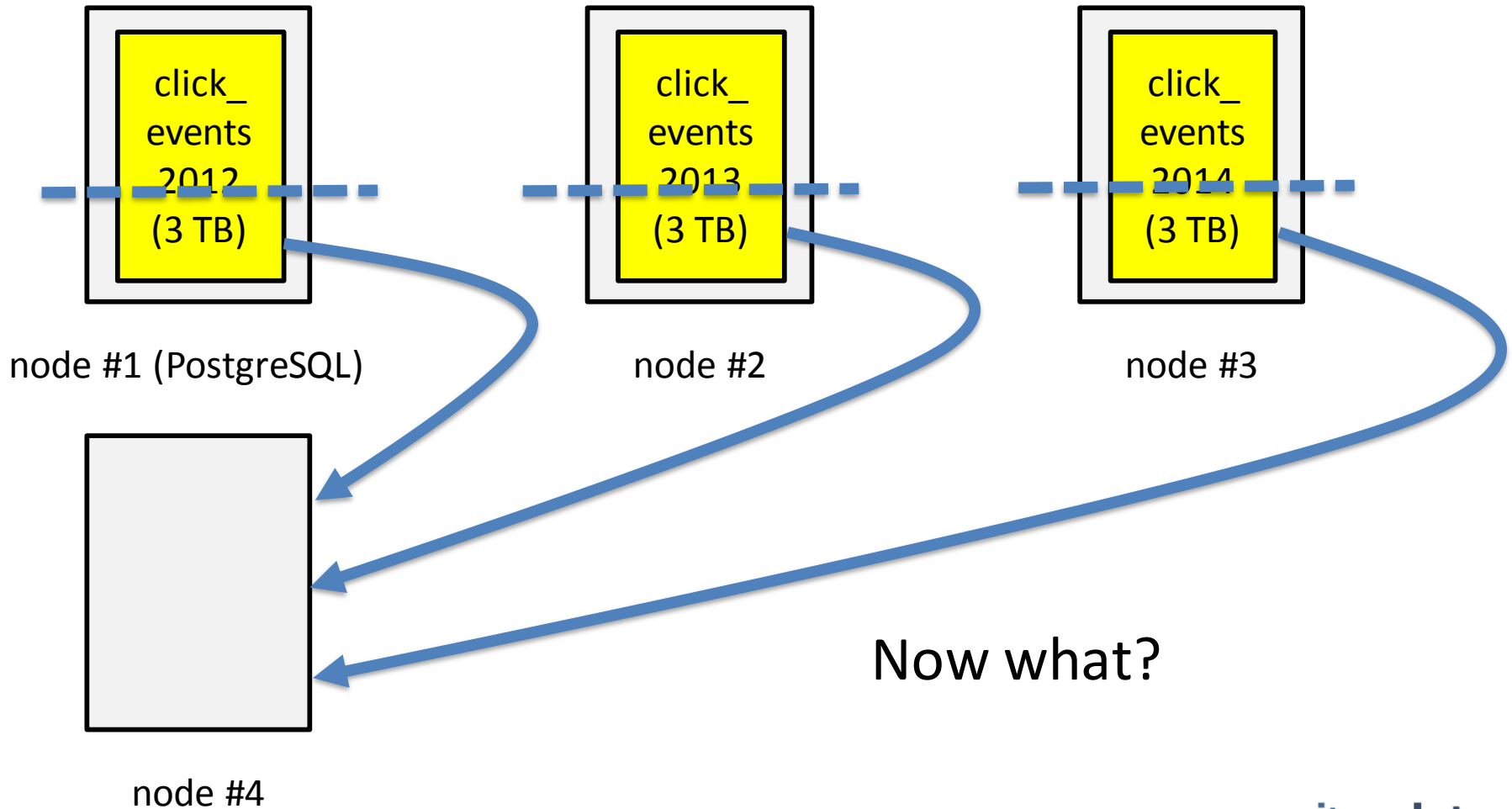


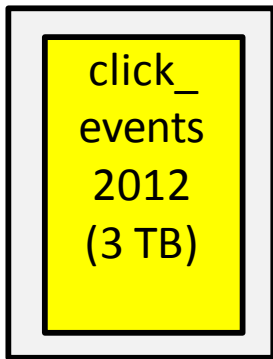
node #3



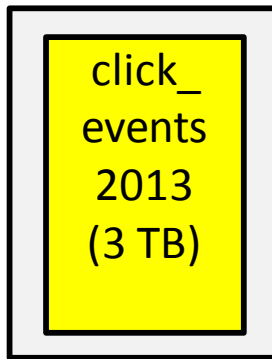
node #4

Let's add a node

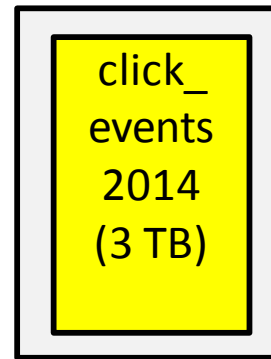




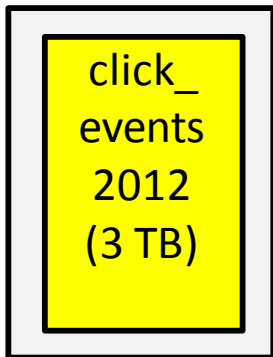
node #1 (PostgreSQL)



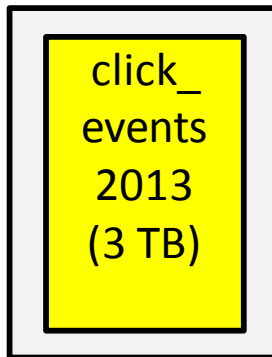
node #2



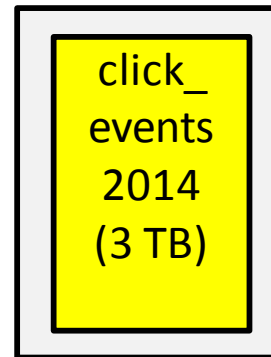
node #3



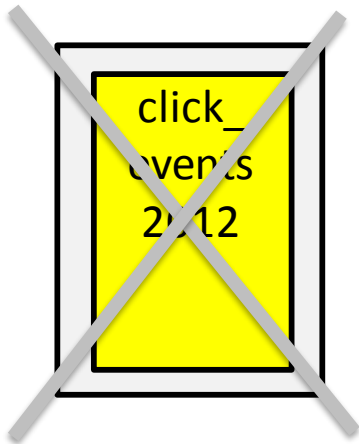
node #4 (PostgreSQL)



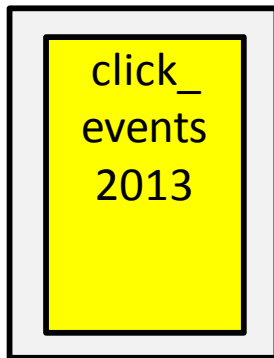
node #5



node #6



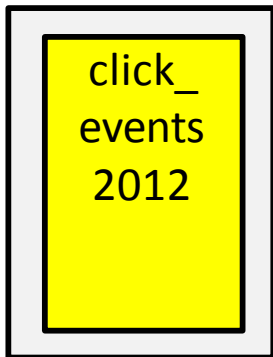
node #1 (PostgreSQL)



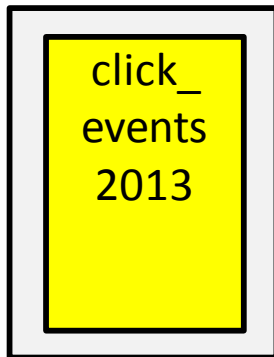
node #2



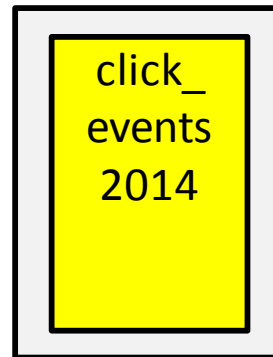
node #3



node #4 (PostgreSQL)



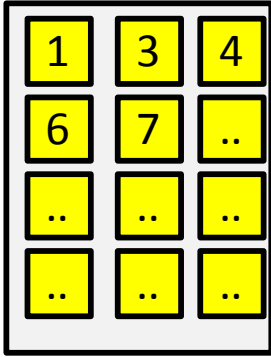
node #5



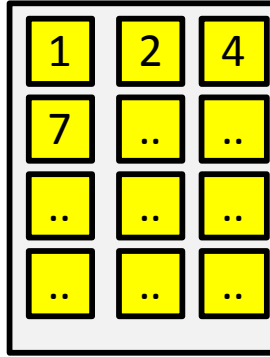
node #6

# Logical sharding

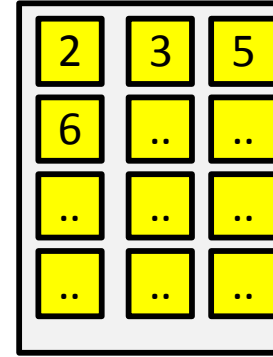
Use “logical shards” so that you can easily rebalance shards as cluster membership changes and handle failures



node #1 (PostgreSQL)



node #2



node #3

Logical sharding

1	3	4
6	7	..
..	..	..
..	..	..

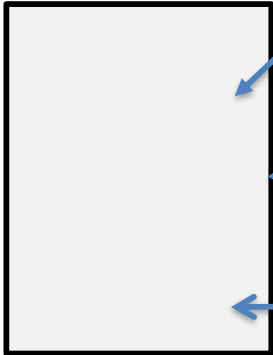
node #1 (PostgreSQL)

1	2	4
7	..	..
..	..	..
..	..	..

node #2

2	3	5
6	..	..
..	..	..
..	..	..

node #3



node #4

256 MB

1	4	7
6	..	..
..	..	..
..	..	..

node #1 (PostgreSQL)

1	2	7
8	..	..
..	..	..
..	..	..

node #2

2	3	1
8	9	..
..	..	..
..	..	..

node #3

3	4	9
10	..	..
..	..	..
..	..	..

node #4

4	5	9
10	11	..
..	..	..
..	..	..

node #5

5	6	9
11	..	..
..	..	..
..	..	..

node #6



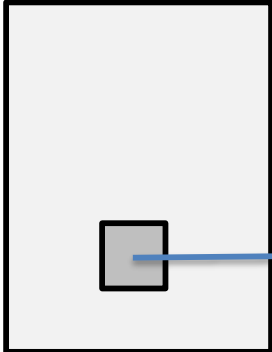
# Logical sharding using pg\_shard

Master node with pg\_shard extension keeps metadata on:

- distributed tables
- shards of a distributed table
- placements of a shard

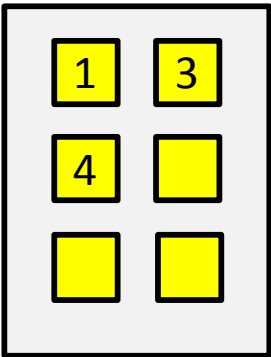
Shard placements are regular postgres tables on worker nodes named:

*<distributed table name>\_<shard id>, e.g. customers\_12*

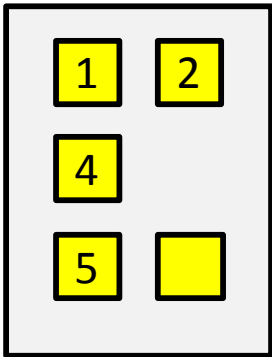


master node  
(PostgreSQL + pg\_shard)

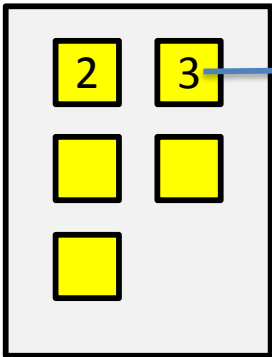
shard and shard  
placement metadata



worker node #1  
(PostgreSQL)



worker node #2  
(PostgreSQL)



worker node #3  
(PostgreSQL)

1 shard =  
1 Postgres  
table

....

# Master node failure

Several options:

1. Use streaming replication and fail-over
2. In the cloud, use EBS volumes (metadata size is small)
3. Reconstruct metadata from tables in the worker nodes
4. Back-ups

# Getting started using pg\_shard

```
CREATE EXTENSION pg_shard;
```

Create a regular postgres table:

```
CREATE TABLE customer_reviews (  
  customer_id TEXT NOT NULL,  
  review_date DATE,  
  ...  
);
```

Distribute the table on the given partition key:

```
SELECT master_create_distributed_table('customer_reviews',  
  customer_id);
```

Create 16 logical shards with 2 placements (replicas) on workers:

```
SELECT master_create_worker_shards('customer_reviews', 16, 2);
```

# Metadata and Hash Partitioning

```
postgres=# SELECT * FROM pgs_distribution_metadata.shard;
```

<b>id</b>	<b>relation_id</b>	<b>storage</b>	<b>min_value</b>	<b>max_value</b>
11	16790	t	-2147483648	-1879048194
12	16790	t	-1879048193	-1610612739
13	16790	t	-1610612738	-1342177284
14	16790	t	-1342177283	-1073741829
15	16790	t	-1073741828	-805306374
16	16790	t	-805306373	-536870919
..	..	t	..	..

# Query Execution using pg\_shard

- Queries on master intercepted via postgres planner, executor hooks
- Insert/update/delete/select on distributed tables are rewritten and forwarded to the right worker node(s)

# PostgreSQL Hooks

```
static planner_hook_type PreviousPlannerHook;  
static ExecutorStart_hook_type PreviousExecutorStartHook;  
  
void _PG_init(void)  
{  
    PreviousPlannerHook = planner_hook;  
    planner_hook = PgShardPlanner;  
    ...  
}  
  
static PlannedStmt * PgShardPlanner(Query *query, int cursorOptions,  
ParamListInfo boundParams)  
{  
    ...  
}
```

# Plan Distributed Query

```
INSERT INTO customer_reviews (customer_id, rating)
VALUES ('HN892', 5);
```

## 1. Find clauses on partition key:

```
customer_id = 'HN892'
```

## 2. Find shard ids in pgs\_distribution\_metadata.shard for which:

```
min_value <= hashtext('HN892') and hashtext('HN892') <= max_value
```

## 3. Rewrite query for shards:

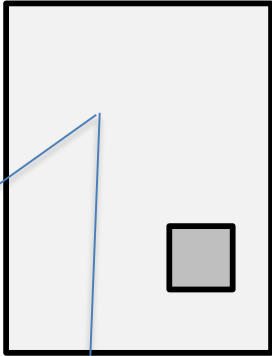
```
INSERT INTO customer_reviews_16 (customer_id, rating)
VALUES ('HN892', 5);
```



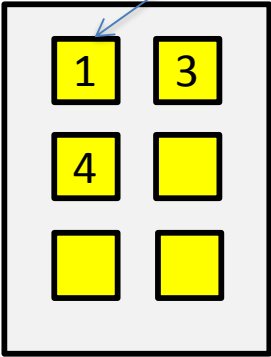
# Execute Distributed Insert

1. Acquire locks for shards, taking into account commutativity rules:
  - Select      No lock
  - Insert      Shared lock
  - Update     Exclusive lock
  - Delete     Exclusive lock
2. For each active placement:
  1. Get a connection to worker from pool
  2. Use libpq functions ([PQexec](#)) to send query to worker
  3. On failure, mark placement as inactive

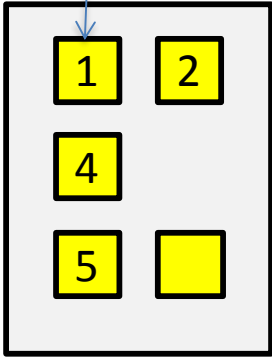
Insert on 2 placements



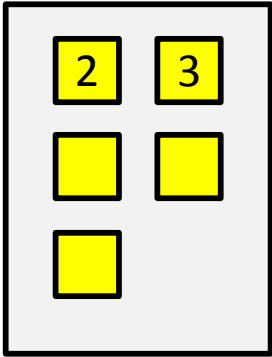
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



worker node #2  
(PostgreSQL)

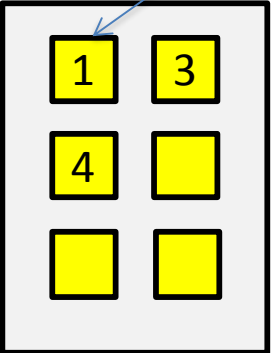


worker node #3  
(PostgreSQL)

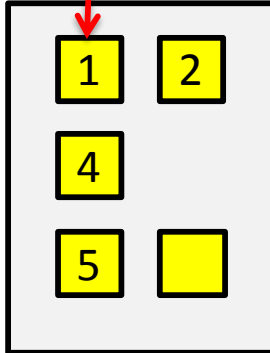
....

Insert failure

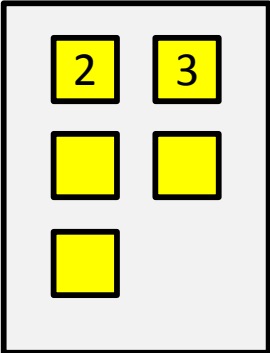
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



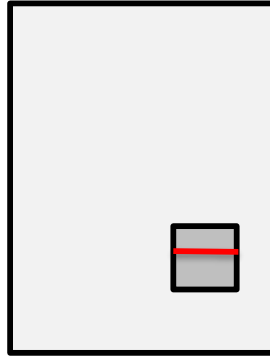
worker node #2  
(PostgreSQL)



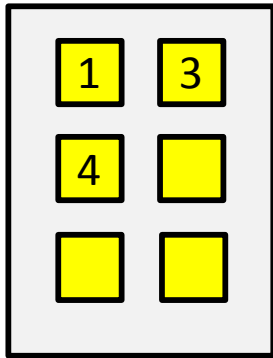
worker node #3  
(PostgreSQL)

....

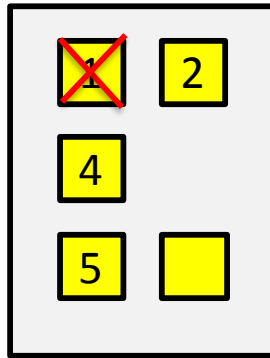
Mark placement as inactive



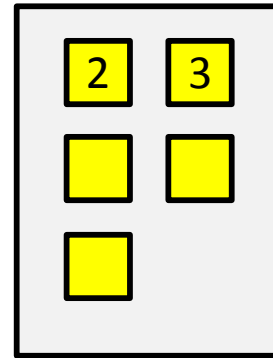
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



worker node #2  
(PostgreSQL)



worker node #3  
(PostgreSQL)

....

# Select on Single Shard

```
SELECT avg(rating)
FROM customer_reviews
WHERE customer_id = 'XD702';
```

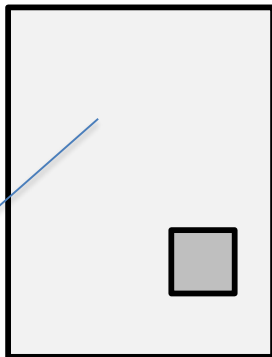
1. Send query to first placement using libpq:

```
SELECT avg(rating) FROM customer_reviews_3 WHERE customer_id = 'XD702';
```

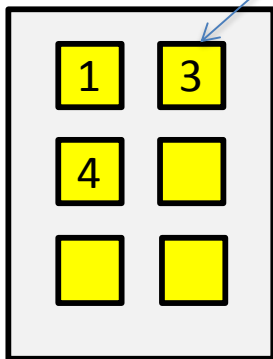
2. Collect results in memory from `PQgetResult`

3. Write results to user-defined destination

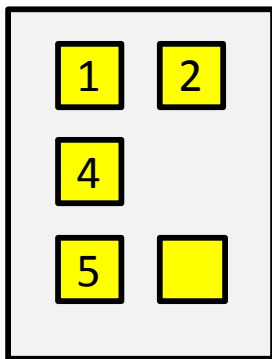
Select from 1st placement



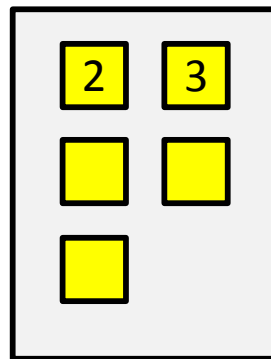
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



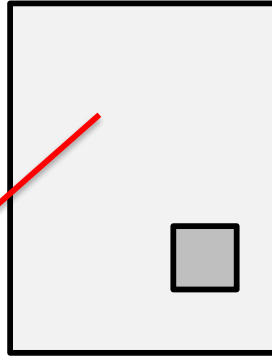
worker node #2  
(PostgreSQL)



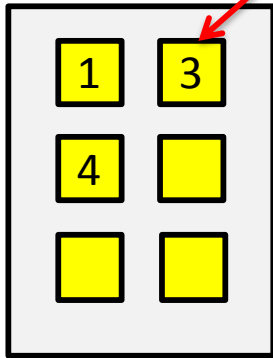
worker node #3  
(PostgreSQL)

....

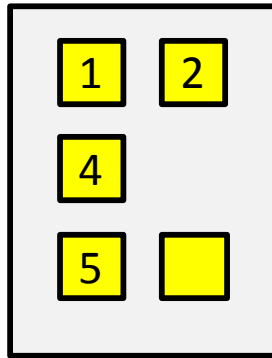
Select failure



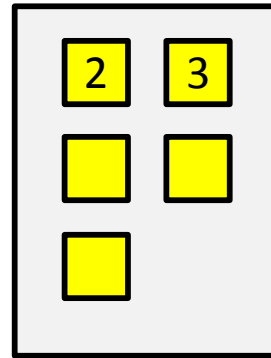
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



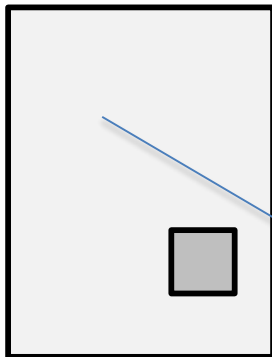
worker node #2  
(PostgreSQL)



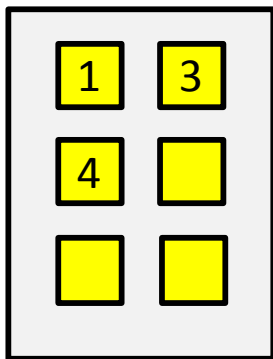
worker node #3  
(PostgreSQL)

....

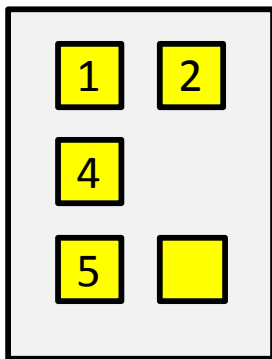
Select from 2nd placement



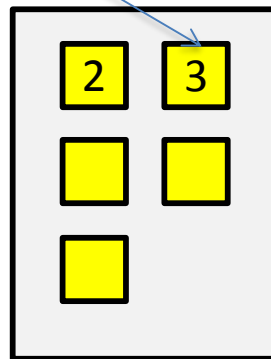
master node  
(PostgreSQL + pg\_shard)



worker node #1  
(PostgreSQL)



worker node #2  
(PostgreSQL)



worker node #3  
(PostgreSQL)

....



# Select on Multiple Shards

```
SELECT avg(rating)
FROM customer_reviews
WHERE review_date >= '2004-01-01';
```

**pg\_shard:** Pull relevant data to master and perform query locally

```
SELECT rating FROM customer_reviews_1 WHERE review_date >= '2004-01-01';
SELECT rating FROM customer_reviews_2 WHERE review_date >= '2004-01-01';
...
```

**CitusDB:** Compute average in distributed way

# Limitations

- No multi-shard transactions
- No multi-statement transactions
- No join support (upgrade to CitusDB)
- No unique constraints on columns other than the partition key

# Upcoming Features

- More complete SQL coverage?
- Re-balancing?
- Multi-master?
- Range partitioning?
- Auto-recovery?

What would make you use pg\_shard?

# Summary

## **pg\_shard: Sharding extension for PostgreSQL**

[https://github.com/citusdata/pg\\_shard](https://github.com/citusdata/pg_shard)

### Logical shards:

- 1 Add new machines and move shards to them
- 2 When a machine fails, evenly spread the load
- 3 pg\_shard could also be your sharding library

### Simple to use:

- 1 PostgreSQL hooks are magical
- 2 Load extension. Create Table. Distribute.
- 3 JSONB + pg\_shard instead of NoSQL?