Horizontally Scalable Relational Databases with Spark

Cody Koeninger
Kixer

What is Citus?

- Standard Postgres
- Sharded across multiple nodes
- CREATE EXTENSION citus; -- not a fork
- Good for live analytics, multi-tenant
- Open source, commercial support

Where does Citus fit with Spark?

1. Shove your data into Kafka
2. Munge it with Spark
3. ???
4. Profit!

Where does Citus fit with Spark?

1. Shove your data into Kafka
2. Munge it with Spark
3. Serve live traffic using
   - ML models or key-value stores
   - ? Spark SQL ?
   - ? Relational database ?
4. Profit!

Spark SQL + HDFS Pain Points

- Multi-user
- Query latency
- Mutable rows
- Co-locating related writes for joins

Relational Database Pain Points

- “Schemaless” data
- Scaling out, without giving up
  - Aggregations
  - Joins
  - Transactions

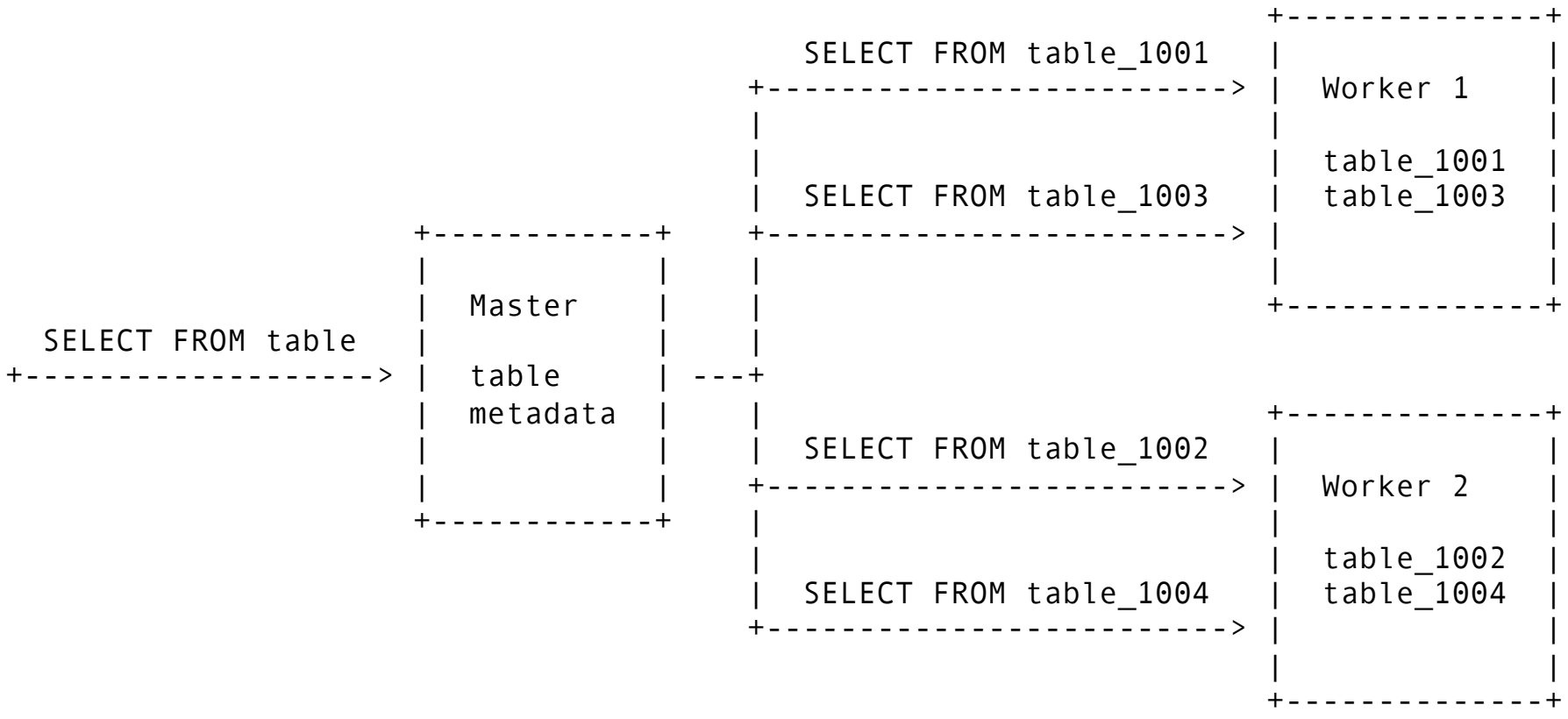
“Schemaless” Data

```
master# create table no_schema (
  data JSONB);
master# create index on no_schema using gin(data);
master# insert into no_schema values
 ('{"user":"cody","cart":["apples","peanut_butter"]}'),
 ('{"user":"jake","cart":["whiskey"],"drunk":true}'),
 ('{"user":"omar","cart":["fireball"],"drunk":true}');

master# select data->'user' from no_schema
where data->'drunk' = ‘true';

?column?
----------
"jake"
"omar"
(2 rows)
```

Scaling Out


SELECT FROM table
Master
table
metadata
SELECT FROM table_1001
SELECT FROM table_1003
SELECT FROM table_1002
SELECT FROM table_1004
Worker 1
table_1001
table_1003
Worker 2
table_1002
table_1004

Choosing a Distribution Key

- Commonly queried column (e.g. customer id)
  - 1 master query : 1 worker query
  - easy join on distribution key
  - possible hot spots if load is skewed
- Evenly distributed column (e.g. event GUID)
  - 1 master query : N shards worker queries
  - hard to join
  - no hot spots

Creating Distributed Tables

```
master# create table impressions(
  date date,
  ad_id integer,
  site_id integer,
  total integer);

master# set citus.shard_replication_factor = 1;
master# set citus.shard_count = 4;

master# select create_distributed_table('impressions', 'ad_id');
```

- Metadata in master tables:

```
master# select * from pg_dist_shard;

logicalrelid | shardid | shardminvalue | shardmaxvalue
--------------+---------+---------------+---------------
impressions  |  102008 | -2147483648   | -1073741825
impressions  |  102009 | -1073741824   | -1
impressions  |  102010 | 0             | 1073741823
impressions  |  102011 | 1073741824    | 2147483647
```

- Data in worker shard tables:

```
worker1# \d
List of relations
Schema |        Name         | Type  | Owner
--------+---------------------+-------+-------
public | impressions_102008 | table | cody
public | impressions_102010 | table | cody
```

Writing Data

```
master# insert into impressions values (now(), 23, 42, 1337);

master# update impressions set total = total + 1
  where ad_id = 23 and site_id = 42;

master# \copy impressions from ‘/var/tmp/bogus_impressions';
```

- Can also write directly to worker shards
  - as long as you hash correctly (at your own risk)

Aggregations

- Commutative and associative operations "just work":

```
master# select site_id, avg(total)
 from impressions group by 1 order by 2 desc limit 1;

site_id |          avg
---------+----------------------
5225 | 790503.061538461538
(1 row)
```

- In worker1's log:

```
COPY (SELECT site_id, sum(total) AS avg, count(total) AS avg
 FROM impressions_102010 impressions WHERE true GROUP BY site_id)
TO STDOUT

COPY (SELECT site_id, sum(total) AS avg, count(total) AS avg
 FROM impressions_102008 impressions WHERE true GROUP BY site_id)
TO STDOUT
```

- For non-associative operations
  - query a subset of data to temp table on master
  - then use arbitrary SQL

Joins: Distribution key = fast

```
master# create table clicks(
  date date,
  ad_id integer,
  site_id integer,
  price numeric,
  total integer);
master# select create_distributed_table('clicks', 'ad_id');

master# select i.ad_id, sum(c.total) / sum(i.total)::float as
clickthrough from impressions i
  inner join clicks c on i.ad_id = c.ad_id and i.date = c.date
  and i.site_id = c.site_id group by 1 order by 2 desc limit 1;

 ad_id | clickthrough
-------+-------------
 814 |         0.1
```

Joins: Non-distribution key = slow

```
master# create table discounts(
  date date,
  amt numeric);
master# select create_distributed_table('discounts', 'date');

master# select c.ad_id, max(c.price * d.amt) from clicks c
inner join discounts d on c.date = d.date group by 1 order by 2 desc limit 1;
ERROR:  cannot use real time executor with repartition jobs
HINT:  Set citus.task_executor_type to "task-tracker".

master# SET citus.task_executor_type = 'task-tracker';
master# select c.ad_id, max(c.price * d.amt) from clicks c
inner join discounts d on c.date = d.date group by 1 order by 2 desc limit 1;

ad_id |                  max
-------+-----------------------------------
587 | 67.8871491877362000000000000000000
(1 row)

Time: 3464.598 ms
```

Joins: Table on all workers = fast

```
-- Replication factor equal to number of nodes
master# SET citus.shard_replication_factor = 2;
master# select create_reference_table('discounts2');

master# select c.ad_id, max(c.price * d.amt) from clicks c
 inner join discounts2 d on c.date = d.date group by 1 order by 2
desc limit 1;

ad_id |                  max
-------+-----------------------------------
587 | 67.8871491877362000000000000000000
(1 row)

Time: 31.237 ms
```

Transactions

- No global transactions (Postgres-XL)
- Individual worker transactions still very useful
  - Spark output actions aren't exactly-once
  - Transactions allow exactly-once semantics
    - Even for non-idempotent updates
  - If sharded by user, each sees consistent world

Transactional writes: Spark to 1 DB

- Table of offset ranges
- foreachPartition, transaction on DB
  - insert or update results
  - update offsets table rows
  - roll back if offsets weren't as expected
- On failure
  - begin from minimum offset range
  - recalculate all results for that range (due to shuffle)

Transactional writes: Spark to Citus

- Partition Spark results to match Citus shards
- Table of offset ranges on each worker
- foreachPartition, transaction on corresponding worker
  - insert or update results
  - update offsets table rows for that shard
  - roll back if offsets weren't as expected
- On failure
  - begin from minimum offset range of all workers
  - recalculate all results for that range (due to shuffle)
  - skip writes for shards that already have that range

Spark Custom Partitioner

```
/** same number of Spark partitions as Citus shards */
override def numPartitions: Int

/** given a key from data, which Spark partition */
override def getPartition(key: Any): Int

/** given a Spark partition, which worker shard */
def shardPlacement(partitionId: Int): ShardPlacement
```

- Citus uses Postgres hash (hashfunc.c) based on Jenkins' 2006 hash
- Min / Max hash values for given shard are in pg_dist_shard table
- Worker nodes for a given shard are in pg_dist_shard_placement table
- Query once at partitioner creation time, build a lookup array

```
select
(ds.logicalrelid::regclass)::varchar as tablename,
ds.shardmaxvalue::integer as hmax,
ds.shardid::integer,
p.nodename::varchar,
p.nodeport::integer
from pg_dist_shard ds
left join pg_dist_shard_placement p on ds.shardid = p.shardid
where (ds.logicalrelid::regclass)::varchar in ('impressions')
order by tablename, hmax asc;

tablename  |    hmax     | shardid | nodename  | nodeport
-----------+-------------+---------+-----------+----------
impressions | -1073741825 |  102008 | host1     |     9701
impressions |          -1 |  102009 | host2     |     9702
impressions |  1073741823 |  102010 | host1     |     9701
impressions |  2147483647 |  102011 | host2     |     9702

-- key with hash of -2 would go into shard 102009
```

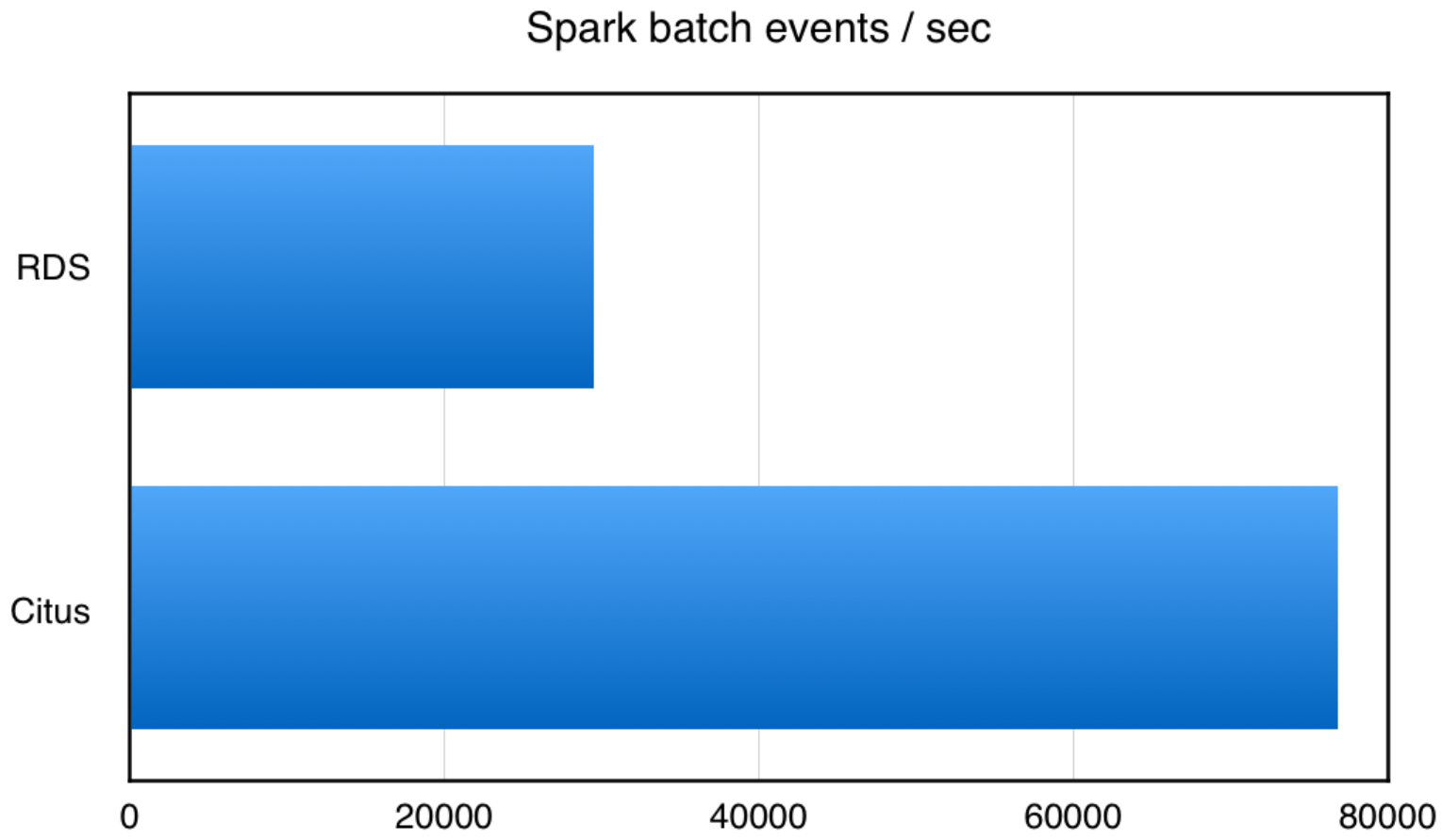
- To find Spark partition
  - hash key using Jenkins
  - walk array until hmax >= hashed value
- To find worker shard, index directly by partition #
- See github link at end of slides for working code

Offsets Table

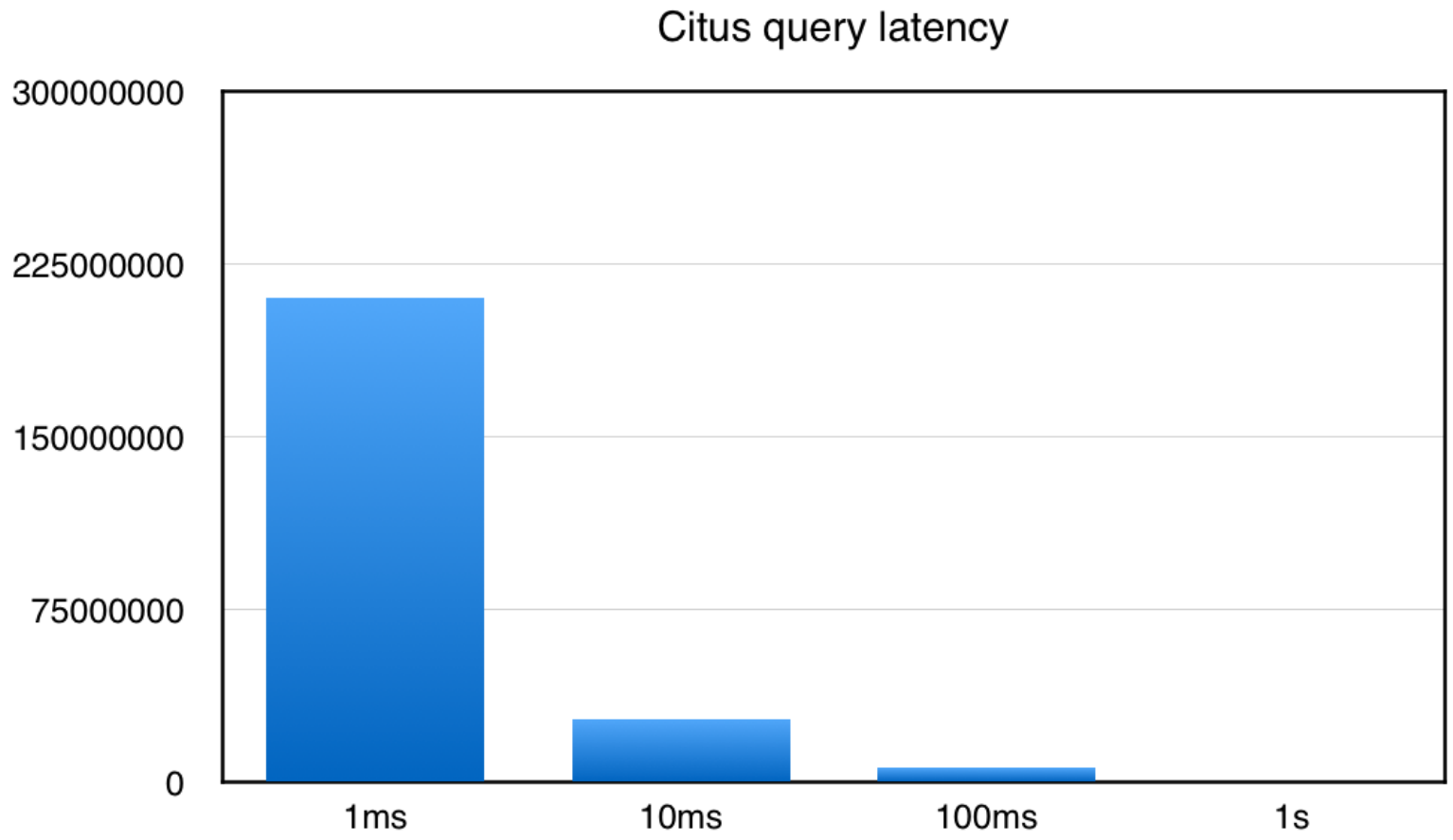
```
app   |     topic     | part |   shard_table_name   |    off
-------+---------------+------+----------------------+---------
myapp | impressions |     0 | impressions_102008 | 20000
myapp | impressions |     0 | impressions_102009 | 20000
myapp | impressions |     0 | impressions_102010 | 19000   -- behind
myapp | impressions |     0 | impressions_102011 | 20000
myapp | impressions |     1 | impressions_102008 | 20001
myapp | impressions |     1 | impressions_102009 | 20001
myapp | impressions |     1 | impressions_102010 | 18000   -- behind
myapp | impressions |     1 | impressions_102011 | 20001
```

- In this case, failure occurred before writes to impressions_102010 finished
- Restart Spark app from Kafka offset ranges
  - partition 0 offsets 19000 -> 20000
  - partition 1 offsets 18000 -> 20001
- Writes to impressions_102010 succeed, other shards are skipped

Lies, Damn Lies, and Bar Charts



Lies, Damn Lies, and Bar Charts



Lessons Learned

- When moving to sharding, avoid other changes
- PgBouncer is necessary in front of workers too
- Some cognitive overhead about what SQL works
  - still better than “SQL” on different data model
- Want hash distribution on top of date partitions
  - can be done manually, easy way on roadmap

Questions?

https://github.com/koeninger/spark-citus

cody@koeninger.org