



# Scaling the Relational Database for the Cloud Age

Sumedh Pathak, Co-Founder & VP Engineering, Citus Data

DataEngConf San Francisco 2018

@moss\_toss | @citusdata

# About Me

---

- Co-Founder & VP Engineering at **Citus Data**
- Amazon Shopping Cart  
(former)
- Amazon Supply Chain & Order Fulfillment (former)
- Stanford Computer Science



This repository

Search

Pull requests

Issues

Marketplace

Explore



 [citusdata / citus](#)

Watch

133

Unstar

2,641

Fork

175

Code

Issues 483

Pull requests 16

Projects 0

Wiki

Insights

Settings

Scalable PostgreSQL for multi-tenant and real-time workloads <https://www.citusdata.com>

Edit

database

citus

multi-tenant

postgresql

scale

sharding

sql

distributed-database

postgres

Manage topics

1,734 commits

88 branches

36 releases

29 contributors

AGPL-3.0

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download



**onderkalaci** Merge pull request #2031 from citusdata/fix\_immediate\_shut\_down\_issue

Latest commit e7b28dd 3 days ago

config

Add citus\_version(), analogous to PG's version()

5 months ago

src

Handle failures during I/O

3 days ago

.codecov.yml

Remove obsolete lines

5 months ago

.editorconfig

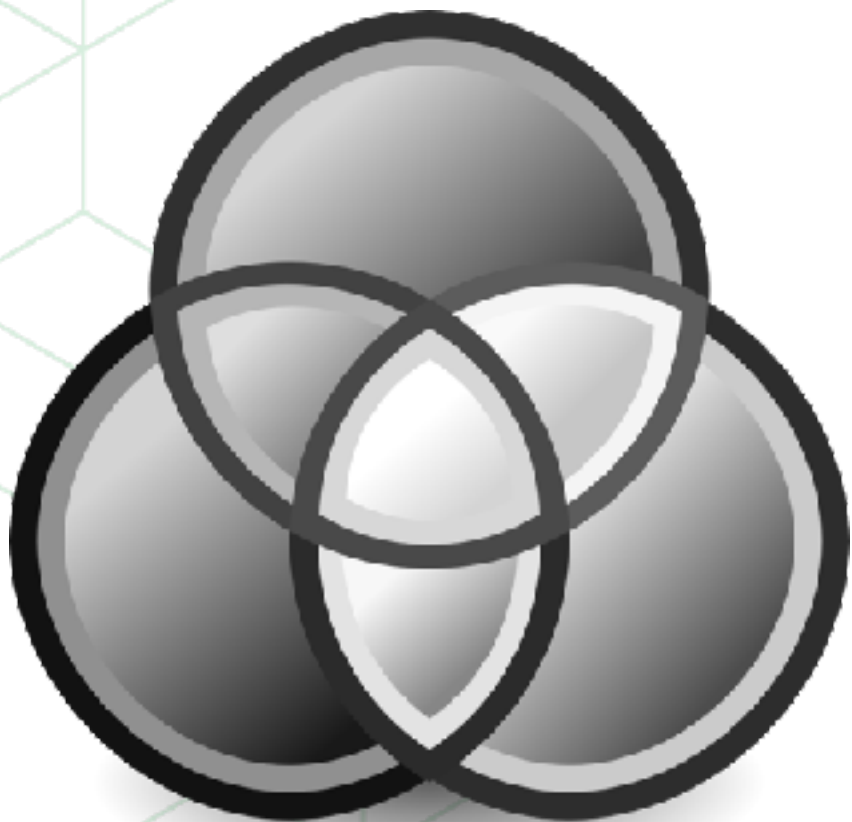
Set tab size for GitHub display

a year ago

.gitattributes

Add ruleutils file for PostgreSQL 11

5 months ago



# Why RDBMS?

# Because your architecture could be simpler



# An RDBMS is a general-purpose data platform

**Fast writes**

**High throughput**

**Data consistency**

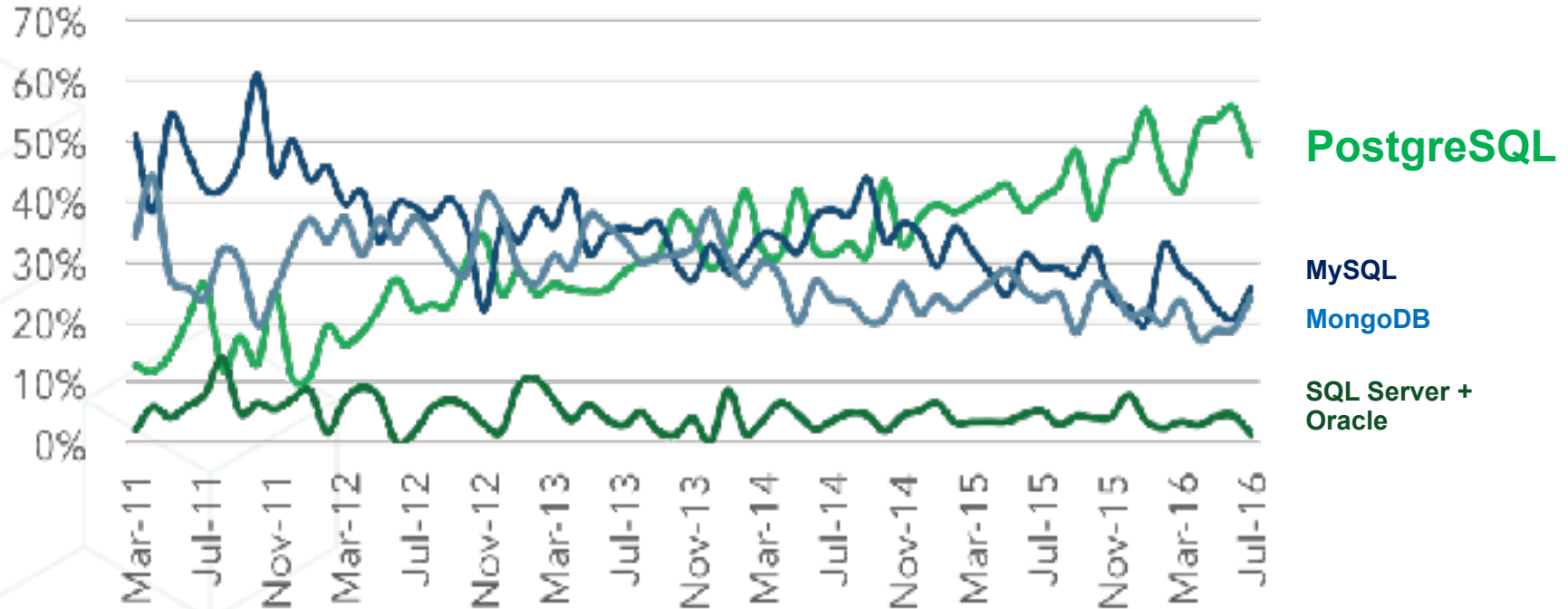
**Real-time & bulk**

**High concurrency**

**Query optimizers**

# Startups Are Choosing Postgres

% database job posts mentioning each database, across 20K+ job posts



Source: Hacker News, <https://news.ycombinator.com>

# Data Trends in the “Cloud Age”

---

- Explosion of Data
- Higher Volume
- Higher Velocity

**\*\* SCALE \*\***





but RDBMS's  
don't scale!

~~but RDBMS's don't scale~~

***RDBMS's are hard to scale***

# What exactly needs to Scale?

1

## Tables (Data)

- Partitioning, Co-location, Reference Tables

2

## SQL (Reads)

- How do we express and optimize distributed SQL

3

## Transactions (Writes)

- Cross Shard updates/deletes, Global Atomic Transactions



Sumeeth Pathak | Citrus Data | DataFestConf 2018

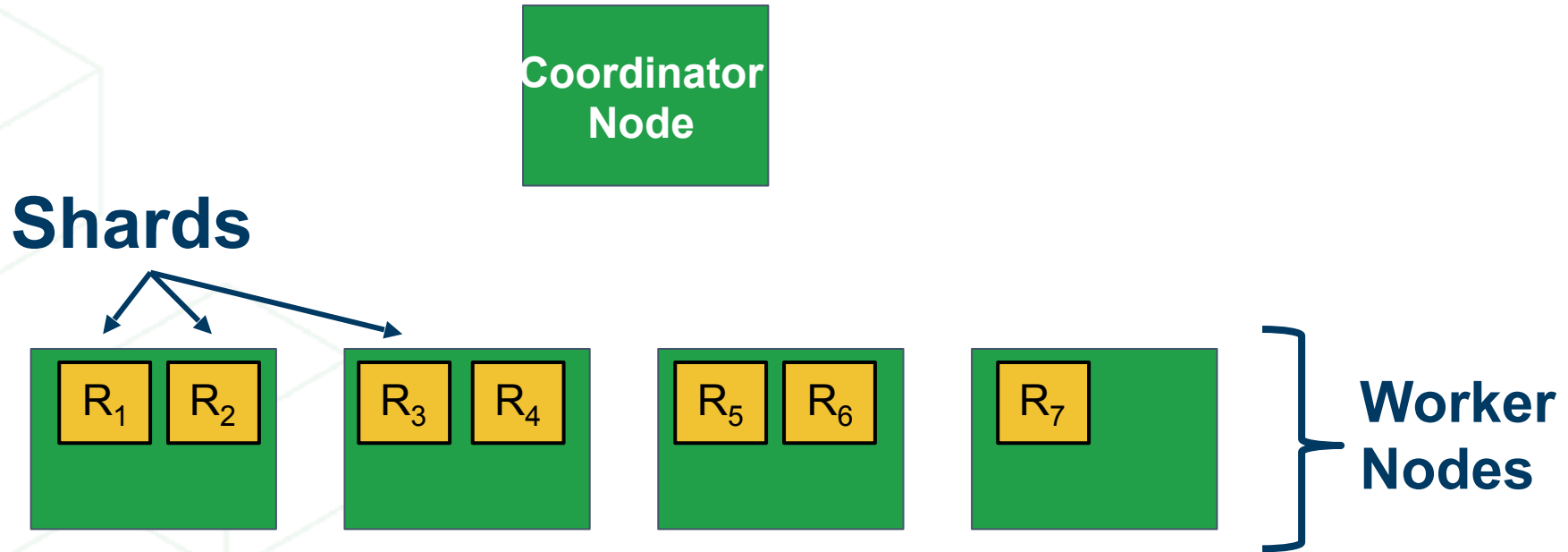
# Scaling Tables

# Data Partitioning

---

- Pick a column
  - Date
  - Id (customer\_id, cart\_id)
  
- Pick a method
  - Hash
  - Range

# Partition data across nodes



# Worker → RDBMS, Shard → Table

```
[postgres=# \d campaigns_102040
```

```
Table "public.campaigns_102040"
```

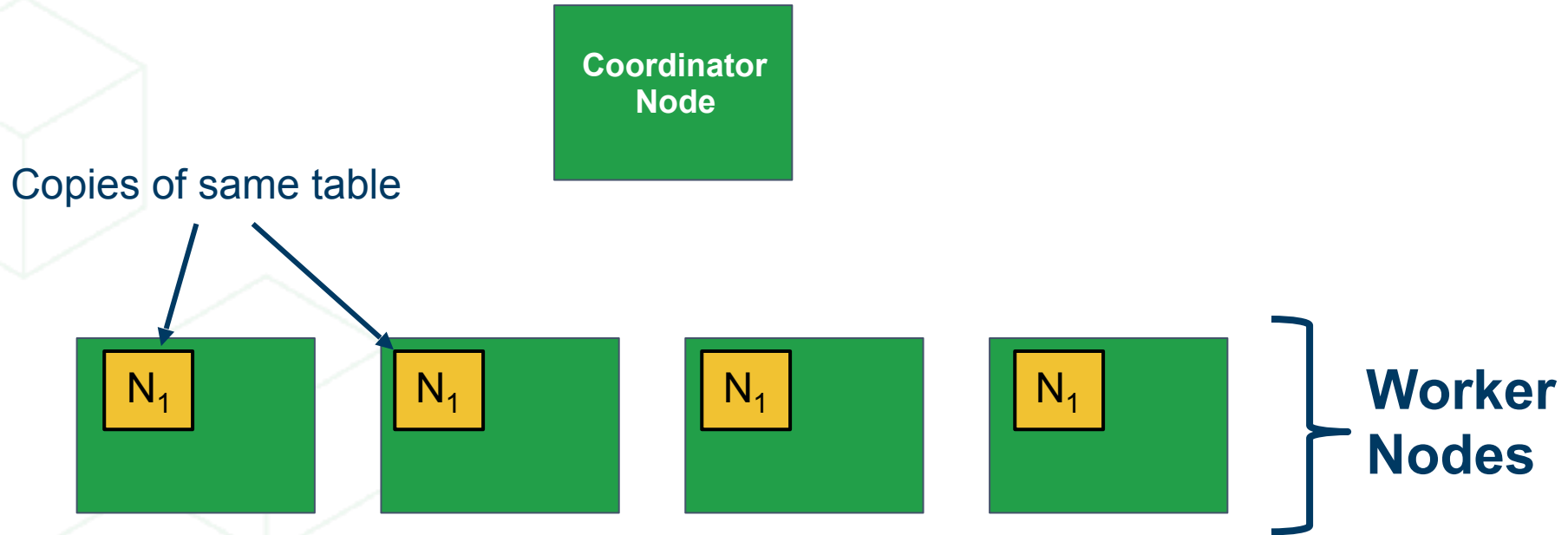
Column	Type	Collation	Nullable	Default
id	bigint		not null	
company_id	bigint		not null	
name	text		not null	
cost_model	text		not null	
state	text		not null	
monthly_budget	bigint			
blacklisted_site_urls	text[]			
created_at	timestamp without time zone		not null	
updated_at	timestamp without time zone		not null	

```
Indexes:
```

```
"campaigns_pkey_102040" PRIMARY KEY, btree (id, company_id)
```

```
"campaigns_name_102040" btree (name)
```

# Reference Tables

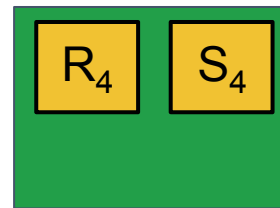
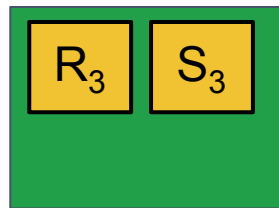
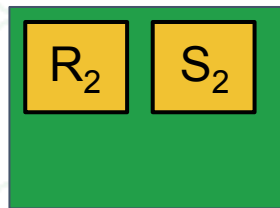
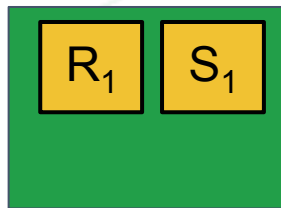




# Co-Location

Coordinator  
Node

Explicit Co-Location API.  
E.g. Partition by Tenant



Worker  
Nodes

# What about Foreign Keys?

```
CREATE TABLE campaigns (  
  id bigserial PRIMARY KEY,  
  company_id bigint REFERENCES companies (id),  
  name text NOT NULL,  
  cost_model text NOT NULL,  
  state text NOT NULL,  
  monthly_budget bigint,  
  blacklisted_site_urls text[],  
  created_at timestamp without time zone NOT NULL,  
  updated_at timestamp without time zone NOT NULL  
);
```

```
CREATE TABLE ads (  
  id bigserial PRIMARY KEY,  
  campaign_id bigint REFERENCES campaigns (id),  
  name text NOT NULL,  
  image_url text,  
  target_url text,  
  impressions_count bigint DEFAULT 0,  
  clicks_count bigint DEFAULT 0,  
  created_at timestamp without time zone NOT NULL,  
  updated_at timestamp without time zone NOT NULL  
);
```

```
CREATE TABLE campaigns (  
  id bigserial, -- was: PRIMARY KEY  
  company_id bigint REFERENCES companies (id),  
  name text NOT NULL,  
  cost_model text NOT NULL,  
  state text NOT NULL,  
  monthly_budget bigint,  
  blacklisted_site_urls text[],  
  created_at timestamp without time zone NOT NULL,  
  updated_at timestamp without time zone NOT NULL,  
  PRIMARY KEY (company_id, id) -- added  
);
```

```
CREATE TABLE ads (  
  id bigserial, -- was: PRIMARY KEY  
  company_id bigint, -- added  
  campaign_id bigint, -- was: REFERENCES campaigns (id)  
  name text NOT NULL,  
  image_url text,  
  target_url text,  
  impressions_count bigint DEFAULT 0,  
  clicks_count bigint DEFAULT 0,  
  created_at timestamp without time zone NOT NULL,  
  updated_at timestamp without time zone NOT NULL,  
  PRIMARY KEY (company_id, id), -- added  
  FOREIGN KEY (company_id, campaign_id) -- added  
  REFERENCES campaigns (company_id, id)  
);
```

# The key to scaling tables...

---

- Use relational databases as a building block
- Understand semantics of application—to be smart about partitioning
  - Multi-tenant applications

```
1 SELECT list_id, list_name, num_items
2 FROM
3   todo_lists lists
4   JOIN
5   ( SELECT user_id, list_id, count(*)
6     USING (user_id, list_id)
7   WHERE user_id = 1
8   ORDER BY num_items DESC;
9
10  list_id | list_name | num_ite
11  -----+-----+-----
12         1 | work things |
13         2 | personal things |
14 (2 rows)
15
16 Time: 2.014 ms|
```

# Scaling SQL

# SQL $\leftrightarrow$ Relational Algebra

**FROM** table

$R$

**SELECT** x

$\text{Project}_x(R) \rightarrow R'$

**WHERE**  $f(x)$

$\text{Filter}_{f(x)}(R) \rightarrow R'$

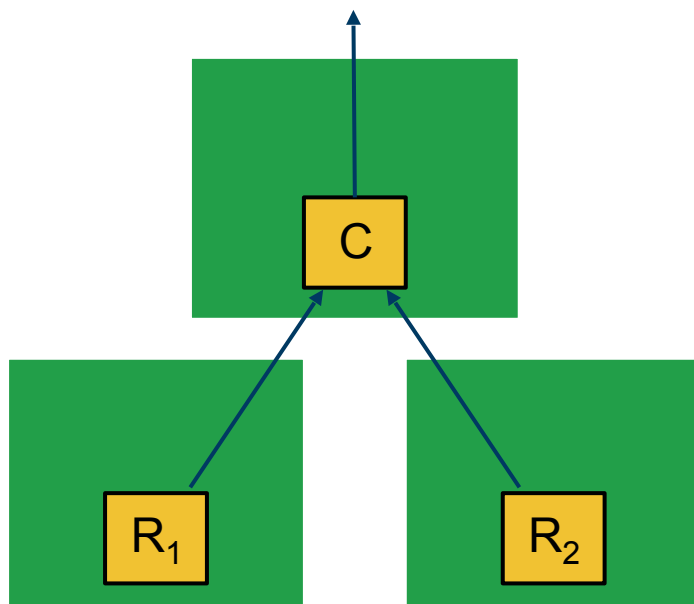
**... JOIN ...**

$R \times S \rightarrow R'$

# Distributed Relational Algebra

FROM sharded\_table

Collect( $R_1, R_2, \dots$ )  $\rightarrow$  R



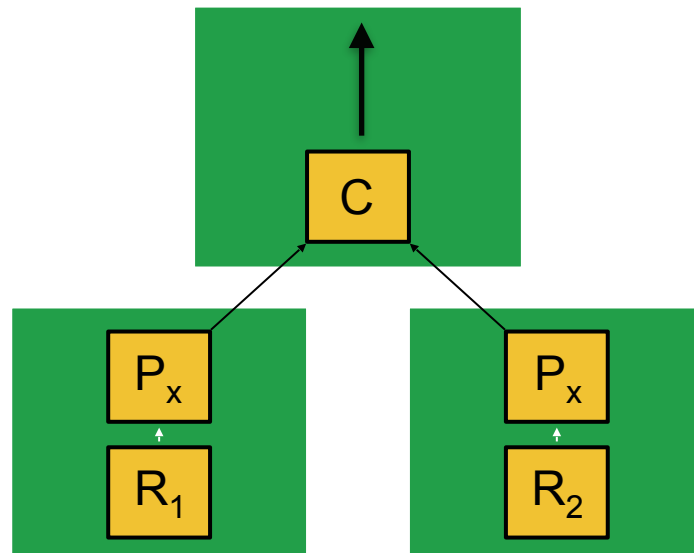
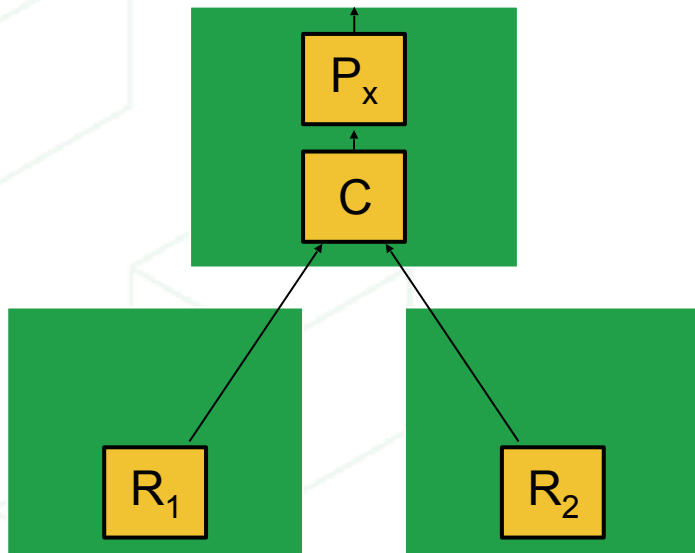
# Commutativity

---

$$A + B = B + A$$

# Commutative property

$$\text{Project}_x(\text{Collect}(R_1, R_2, \dots)) = \text{Collect}(\text{Project}_x(R_1), \text{Project}_x(R_2), \dots)$$





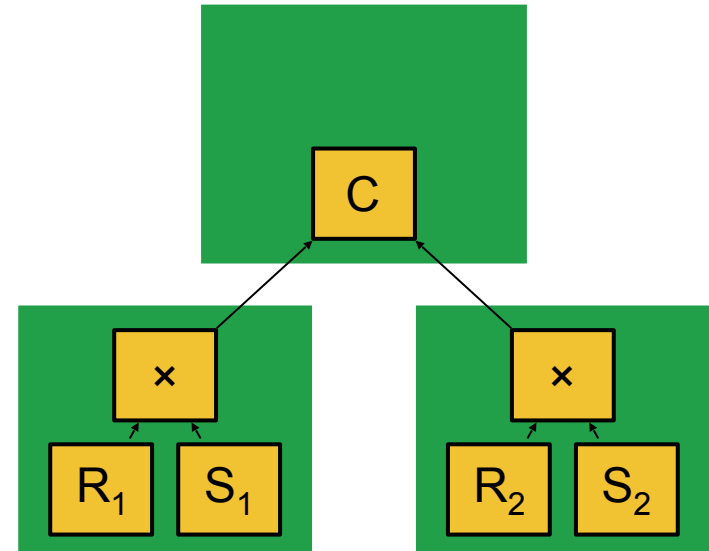
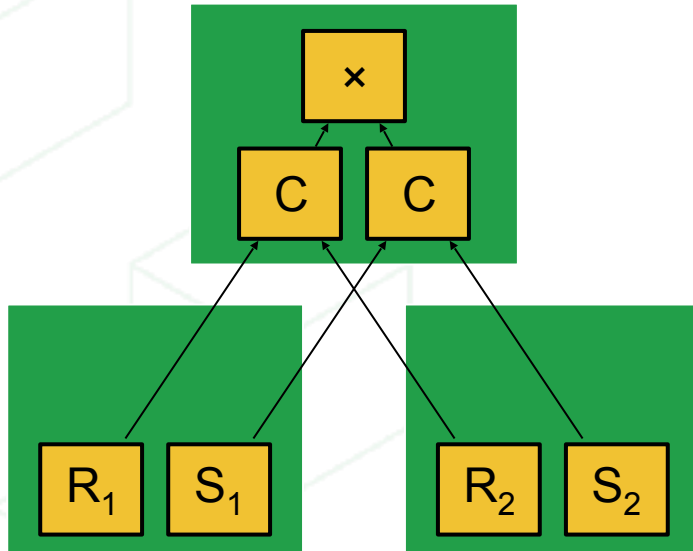
# Distributivity

---

$$A^*(B + C) = A^*B + A^*C$$

# Distributive property

$$\text{Collect}(R_1, R_2, \dots) \times \text{Collect}(S_1, S_2, \dots) = \text{Collect}(R_1 \times S_1, R_2 \times S_2, \dots)$$



X = Join Operator

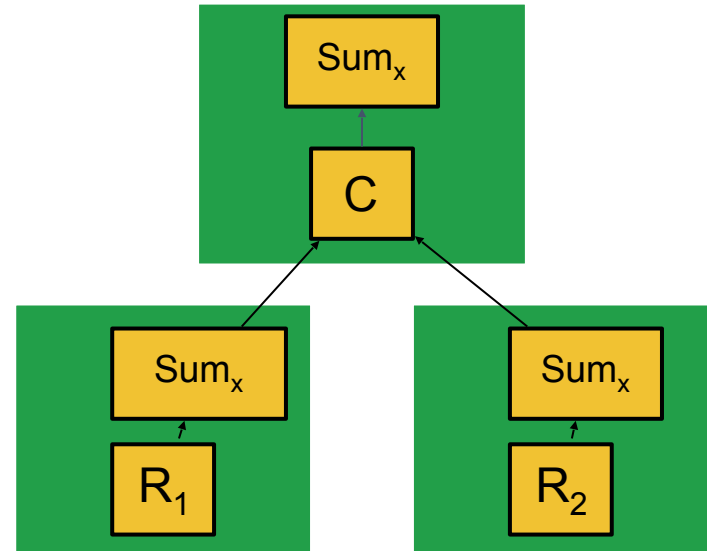
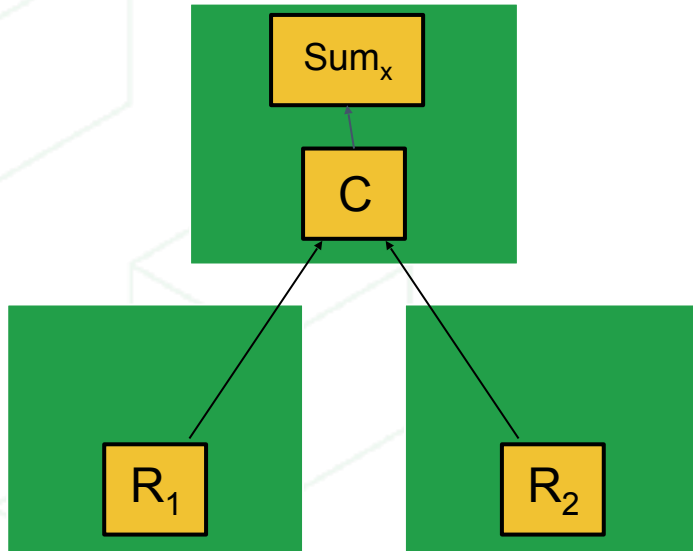
# Associativity

---

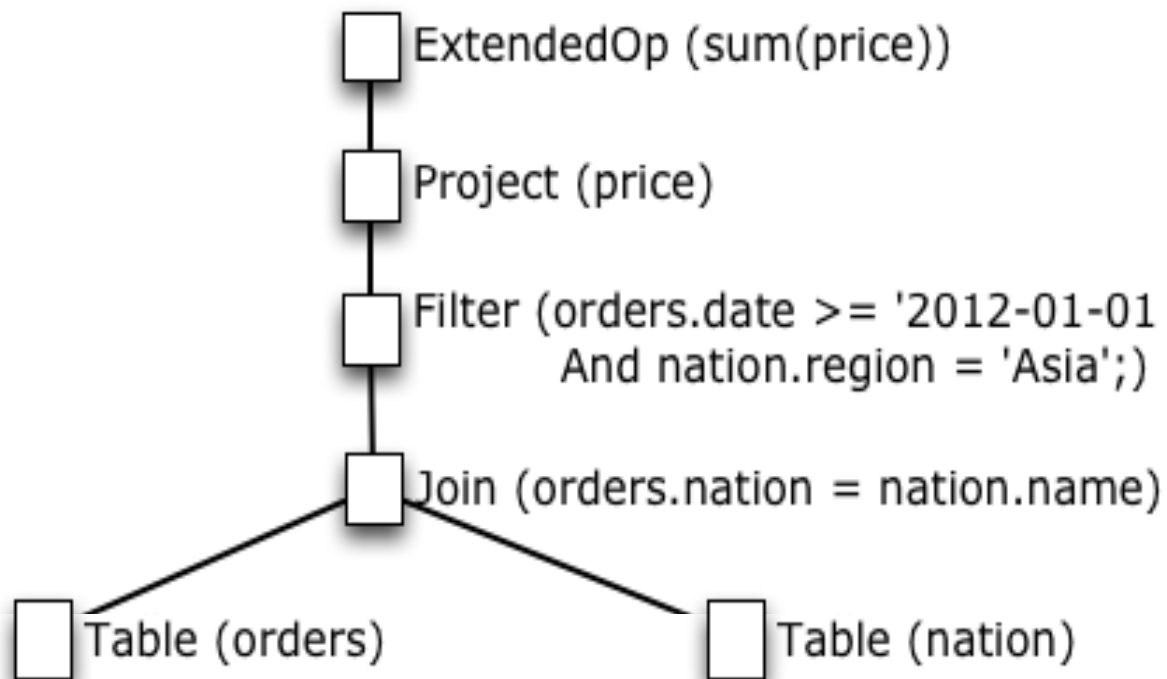
$$\begin{aligned} A + B + C &= \\ (A + B) + C &= \\ A + (B + C) & \end{aligned}$$

# Associative property

$$\text{SUM}(x)(\text{Collect}(R_1, R_2, \dots)) = \text{SUM}(\text{Collect}(\text{SUM}(R_1), \text{SUM}(R_2), \dots))$$

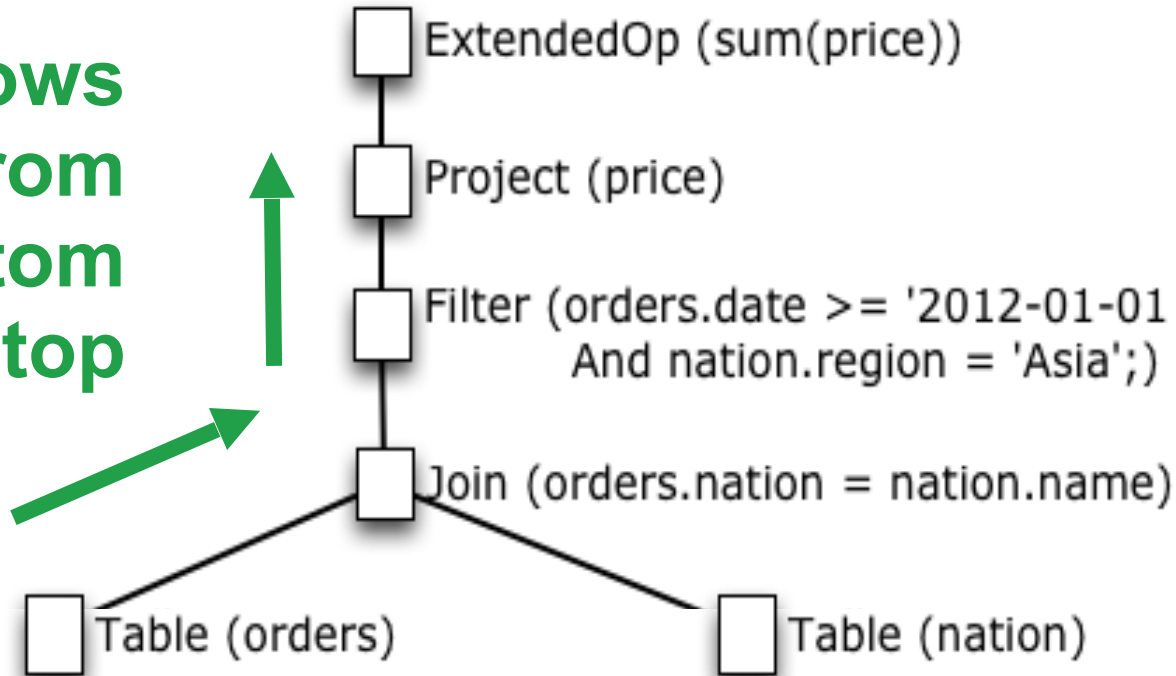


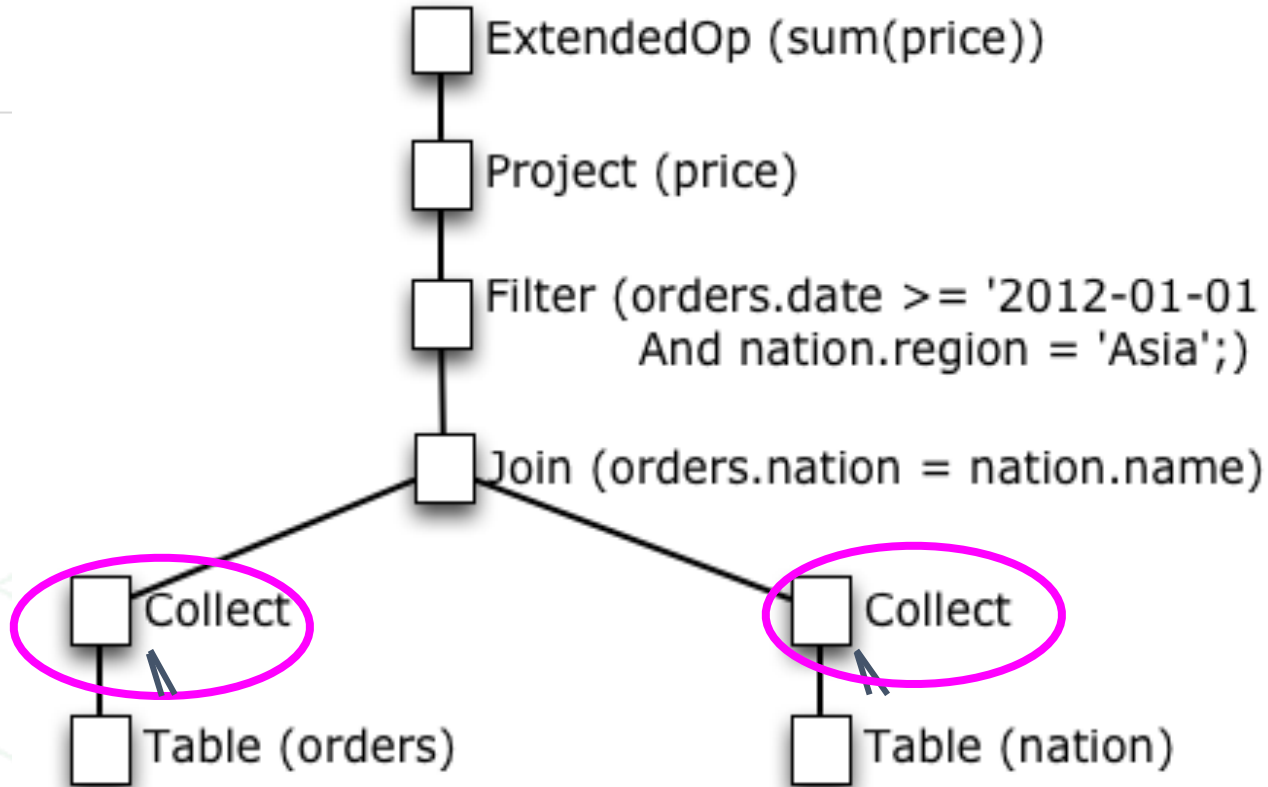
```
SELECT sum(price) FROM orders,  
nation  
WHERE orders.nation = nation.name  
AND  
      orders.date >= '2012-01-01'  
AND  
      nation.region = 'Asia';
```



# Volcano style processing

**Data flows  
from  
bottom  
to top**



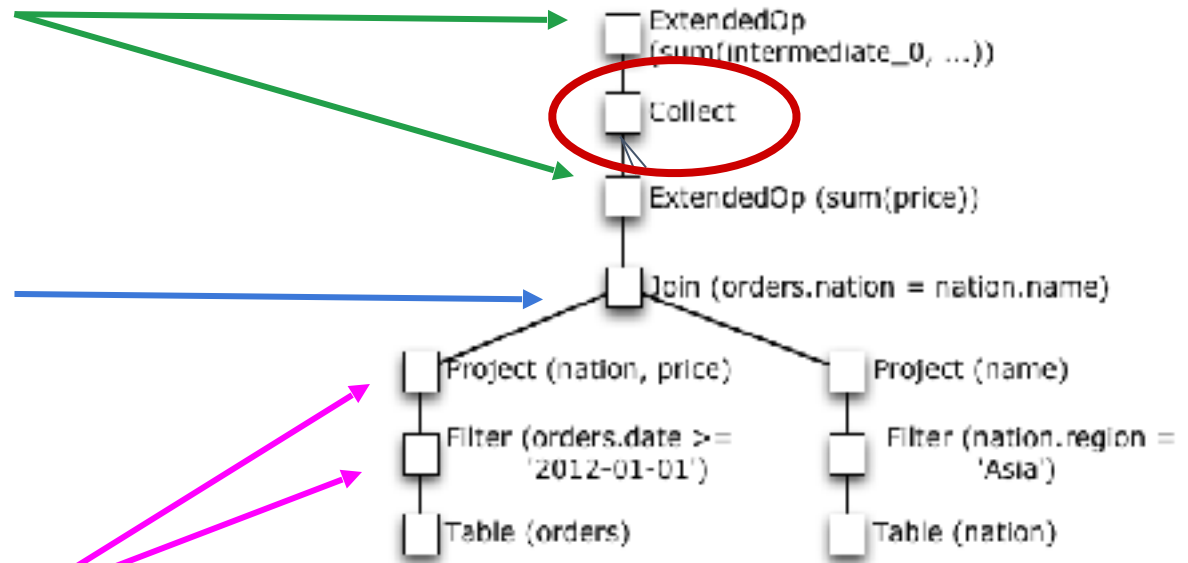


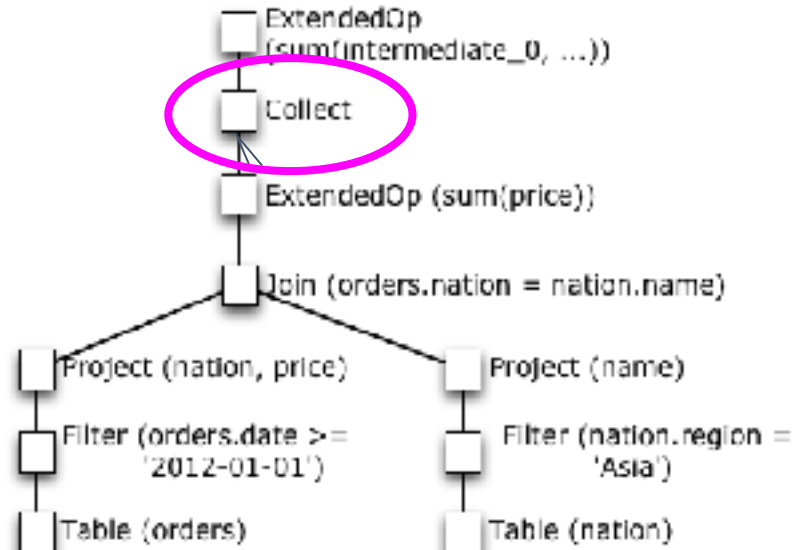
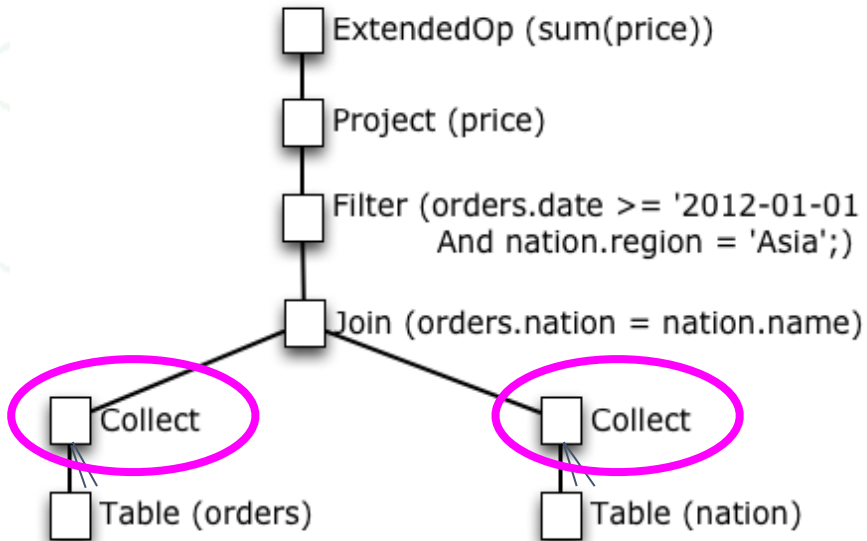


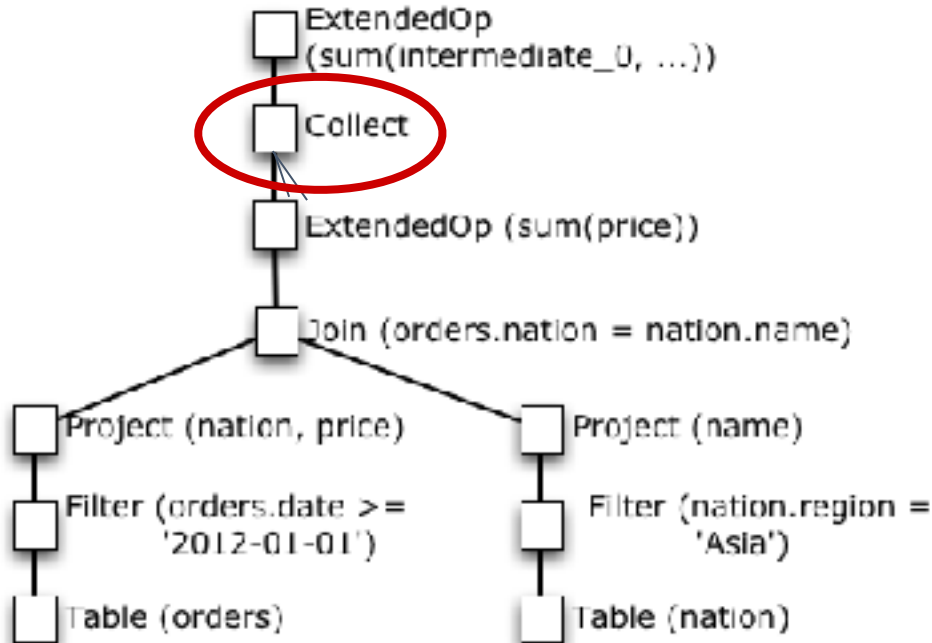
Parallelize  
Aggregate

Push Joins & Filters  
below collect. Run in  
parallel across all  
nodes

Filters & Projections  
done before Join







```
SELECT sum(intermediate_col)
FROM <concatenated results>;
```

```
SELECT sum(price)
FROM orders_2 JOIN nation_1
ON (orders_2.name =
nation_1.name)
WHERE
    orders_2.date >= '2017-01-01'
AND
    nation_2.region = 'Asia';
```

# Executing Distributed SQL

```
SELECT sum(price)
FROM orders_1 o JOIN nation_1
ON (o.name = n.name)
WHERE o.date >= '2017-01-01'
AND n.region = 'Asia'
```

orders

nation

```
SELECT sum(price) FROM
<results>;
```

```
SELECT sum(price)
FROM orders_2 o JOIN nation_1
ON (o.name = n.name)
WHERE o.date >= '2017-01-01'
AND n.region = 'Asia';
```

orders\_1

nation\_1

SQL database

orders\_2

nation\_1

SQL database

# The key to scaling SQL...

---

- New relational algebra operators for distributed processing
- Relational Algebra Properties to optimize tree: **Commutativity, Associativity, & Distributivity**
- Map / Reduce operators



# Scaling Transactions

# Money Transfer, as an example

---

```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance -      WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance +      WHERE id = 'BOB';
```

```
COMMIT;
```

```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```



Coordinator

A<sub>1</sub>

A<sub>3</sub>

A<sub>2</sub>

A<sub>4</sub>



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

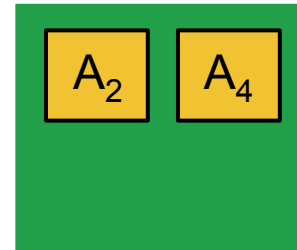
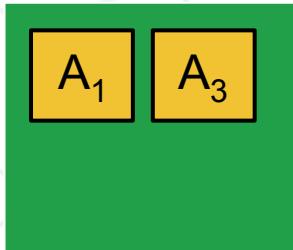
```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

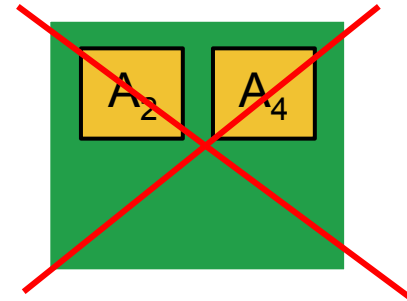
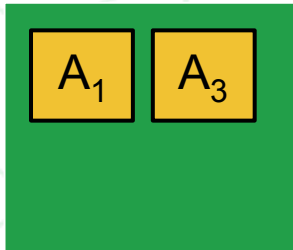
```
COMMIT;
```



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
COMMIT;
```

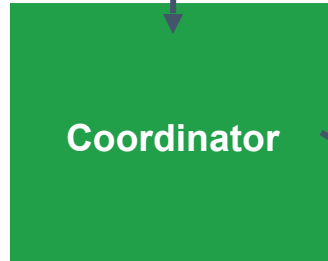


```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

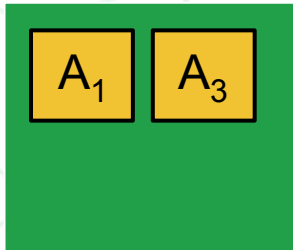
```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```



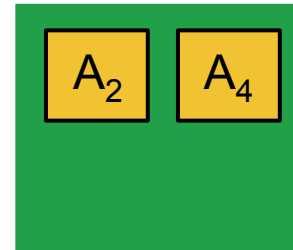
```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

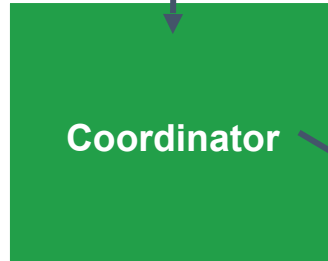


```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```

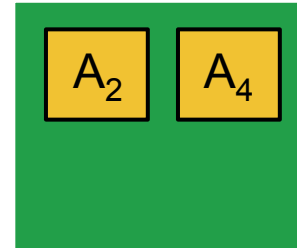


Coordinator

```
BEGIN;
```

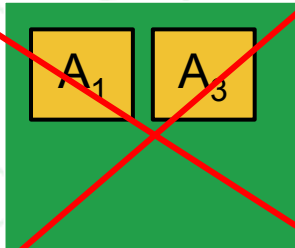
```
UPDATE accounts SET balance =  
balance + 100 WHERE id =  
'BOB';
```

```
COMMIT
```



```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

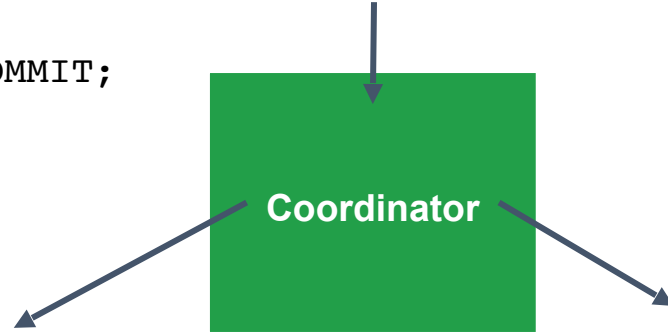


```
BEGIN;
```

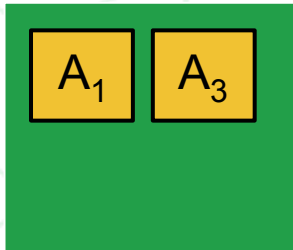
```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

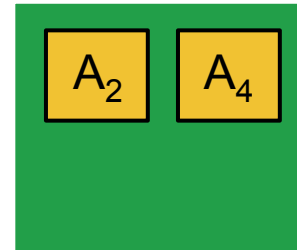
```
COMMIT;
```



```
PREPARE TRANSACTION 'citus_...98';
```

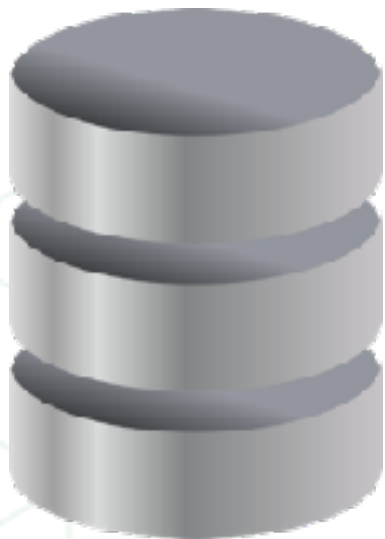


```
PREPARE TRANSACTION 'citus_...98';
```



# What happens during PREPARE?

**State of transaction stored  
on a durable store**



**&**

**Locks are  
maintained**

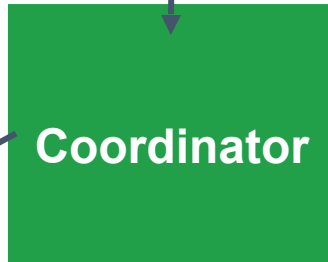


```
BEGIN;
```

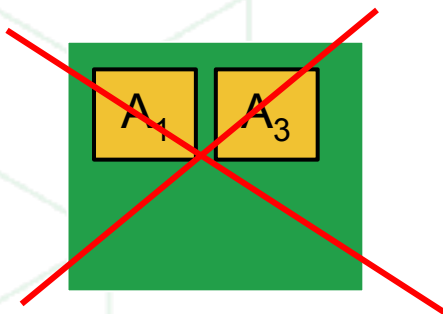
```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

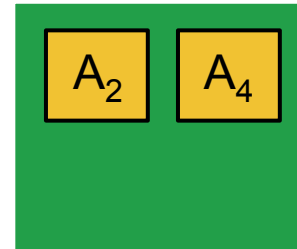
```
COMMIT;
```



```
PREPARE TRANSACTION 'citus_...98';
```



```
ROLLBACK TRANSACTION 'citus_...  
98';
```



```
BEGIN;
```

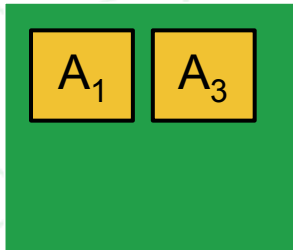
```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

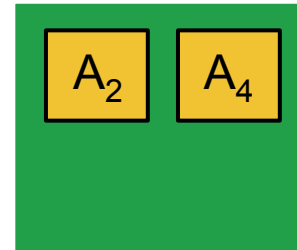
```
COMMIT;
```



```
PREPARE TRANSACTION 'citus_...98';
```



```
PREPARE TRANSACTION 'citus_...98';
```



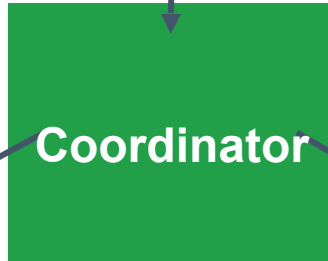


```
BEGIN;
```

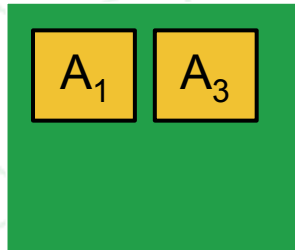
```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

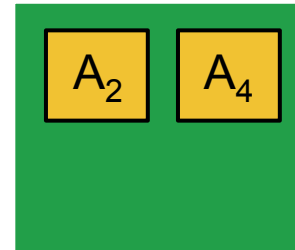
```
COMMIT;
```



```
COMMIT PREPARED 'citus_...  
98';
```



```
COMMIT PREPARED 'citus_...  
98';
```

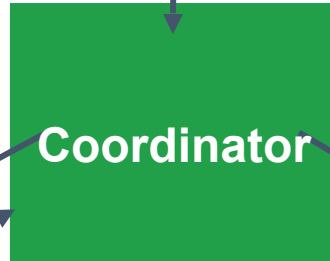


```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

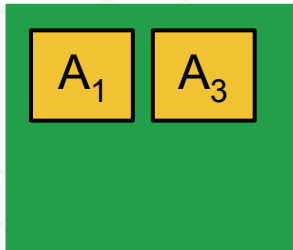
```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```

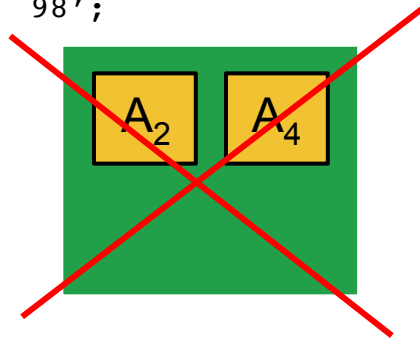


Coordinator

```
COMMIT PREPARED 'citus_...  
98';
```



```
COMMIT PREPARED 'citus_...  
98';
```

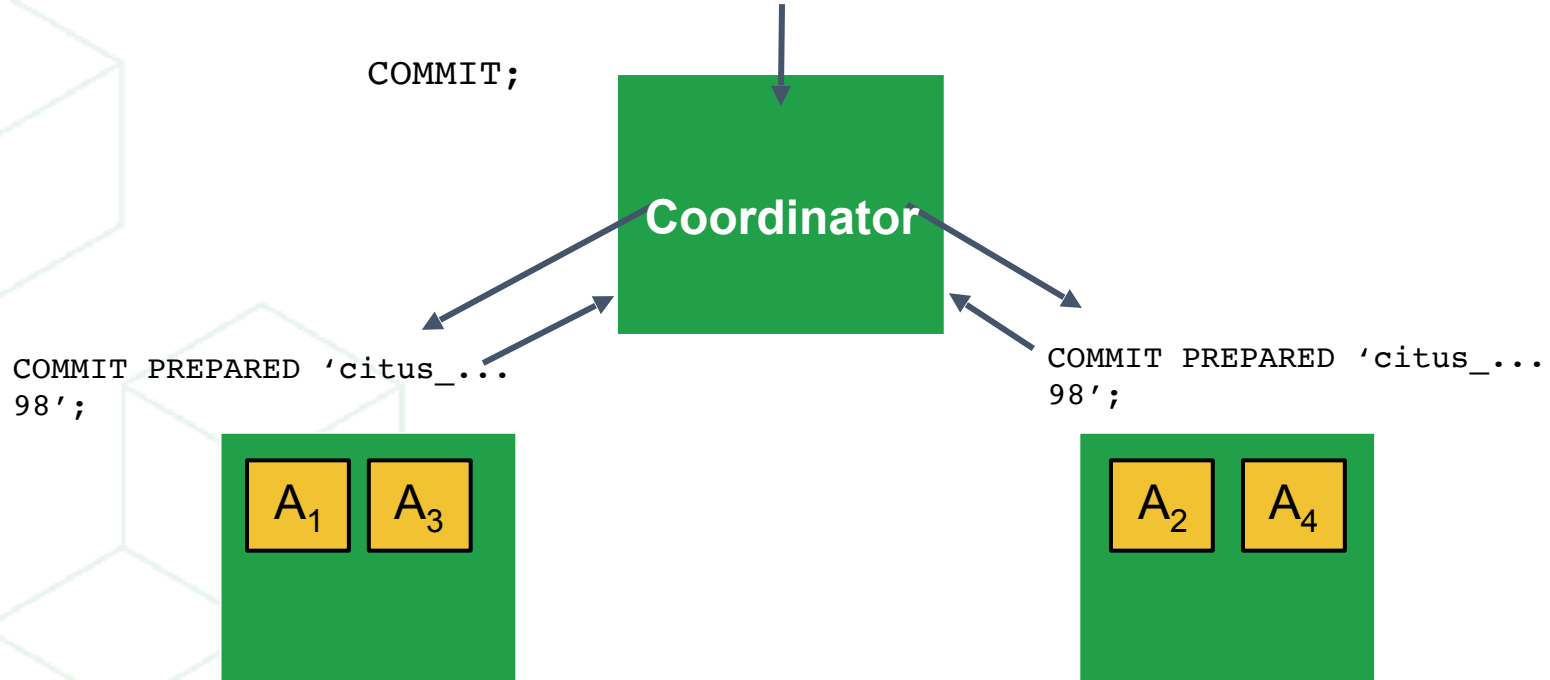


```
BEGIN;
```

```
UPDATE accounts SET balance =  
balance - 100 WHERE id = 'ALICE';
```

```
UPDATE accounts SET balance =  
balance + 100 WHERE id = 'BOB';
```

```
COMMIT;
```





But....  
Deadlocks!

# Example

```
// SESSION 1
```

```
BEGIN;  
UPDATE accounts SET  
balance = balance - 100  
WHERE id = 'ALICE';  
(LOCK on ROW with id  
'ALICE')
```

```
// SESSION 2
```

# Example

## // SESSION 1

```
BEGIN;  
UPDATE accounts SET balance = balance - 100  
WHERE id = 'ALICE';  
(LOCK on ROW with id 'ALICE')
```

## // SESSION 2

```
BEGIN;  
UPDATE accounts SET  
balance = balance + 100  
WHERE id = 'BOB';  
(LOCK on ROW with id  
'BOB')
```

# Example

## // SESSION 1

```
BEGIN;  
UPDATE accounts SET balance = balance - 100  
WHERE id = 'ALICE';  
(LOCK on ROW with id 'ALICE')
```

```
UPDATE accounts SET  
balance = balance + 100  
WHERE id = 'BOB';
```

## // SESSION 2

```
BEGIN;  
UPDATE accounts SET balance = balance + 100  
WHERE id = 'BOB';  
(LOCK on ROW with id 'BOB')
```

# Example

## // SESSION 1

```
BEGIN;  
UPDATE accounts SET balance = balance - 100  
WHERE id = 'ALICE';  
(LOCK on ROW with id 'ALICE')
```

```
UPDATE accounts SET balance = balance +  
100 WHERE id = 'BOB';
```

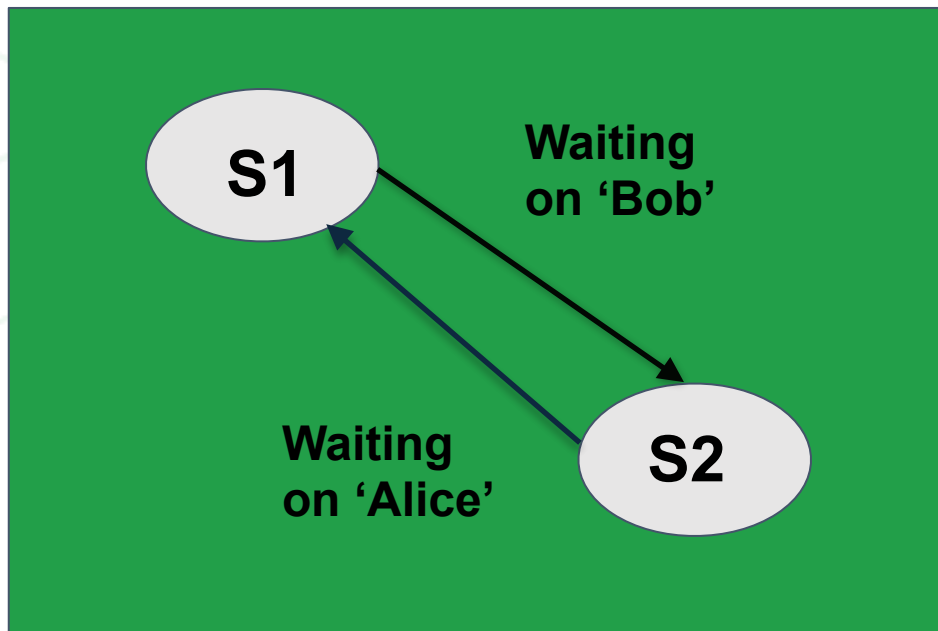
## // SESSION 2

```
BEGIN;  
UPDATE accounts SET balance = balance + 100  
WHERE id = 'BOB';  
(LOCK on ROW with id 'BOB')
```

```
UPDATE accounts SET  
balance = balance - 100  
WHERE id = 'ALICE'
```

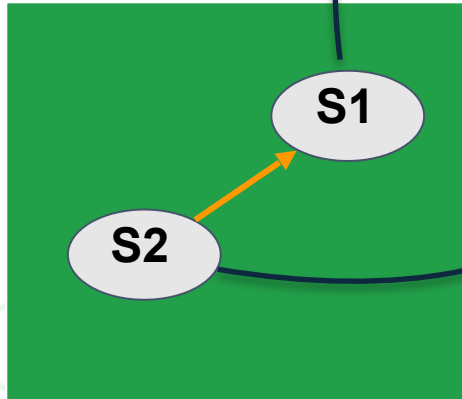


# How do Relational DB's solve this?

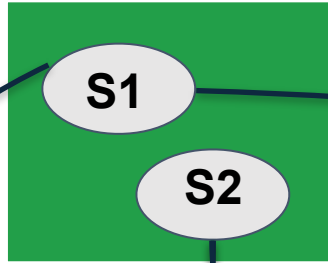
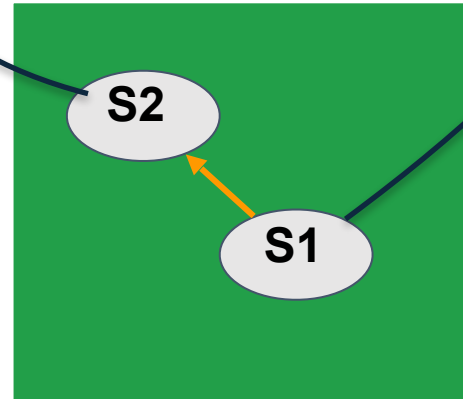


- Construct a Directed Graph
- Each node is a session/transaction
- Edge represents a wait on a lock

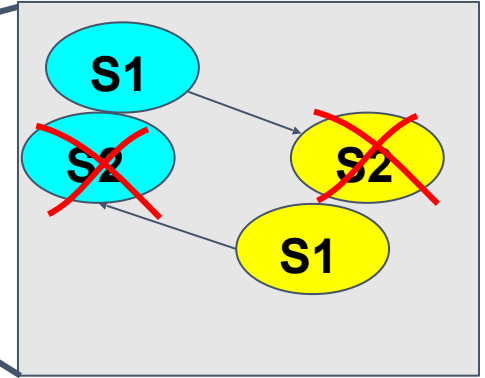
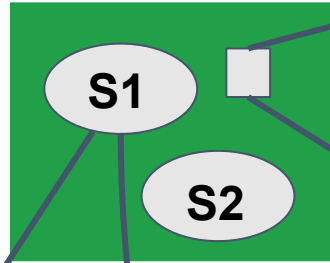
**S2 Waits on S1**



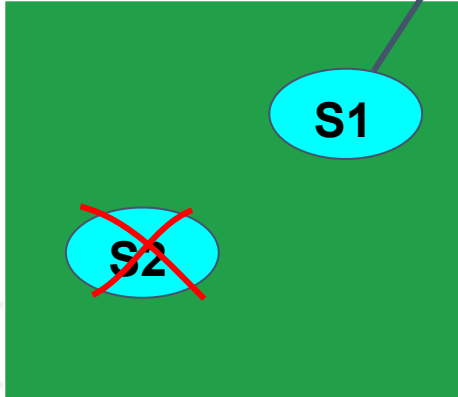
**S1 Waits on S2**



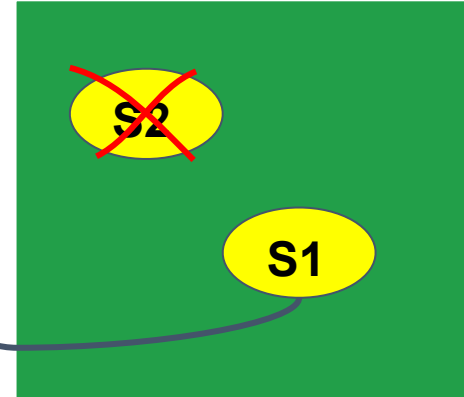
# Distributed Deadlock Detector



S2 Waits on S1



S1 Waits on S2



# 4

## keys to scaling transactions

---

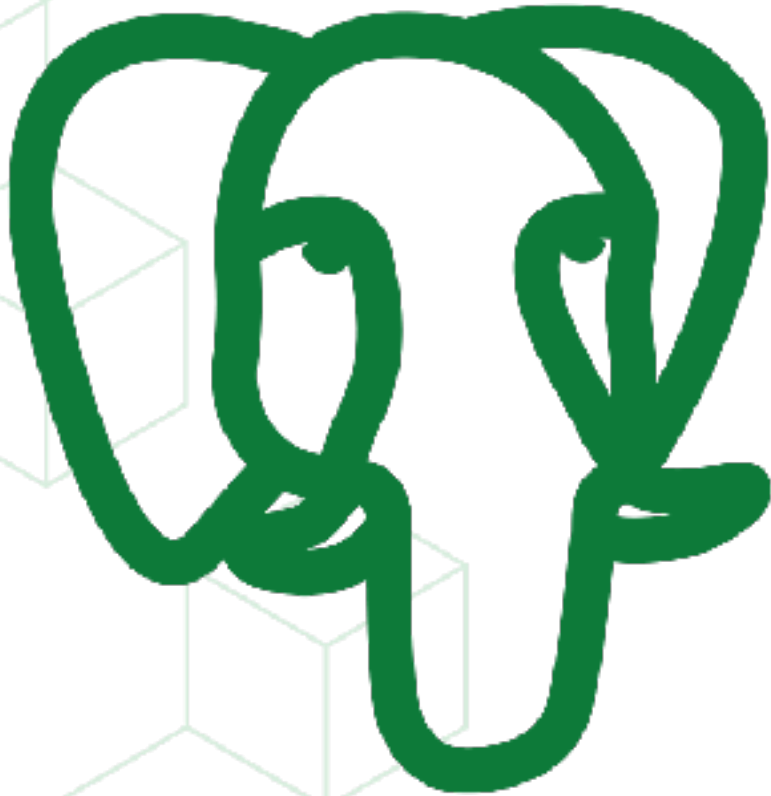
- 2PC to ensure atomic transactions across nodes
- Deadlock Detection—to scale complex & concurrent transaction workloads
- MVCC
- Failure Handling

# *It's 2018. Distributed can be Relational*

---

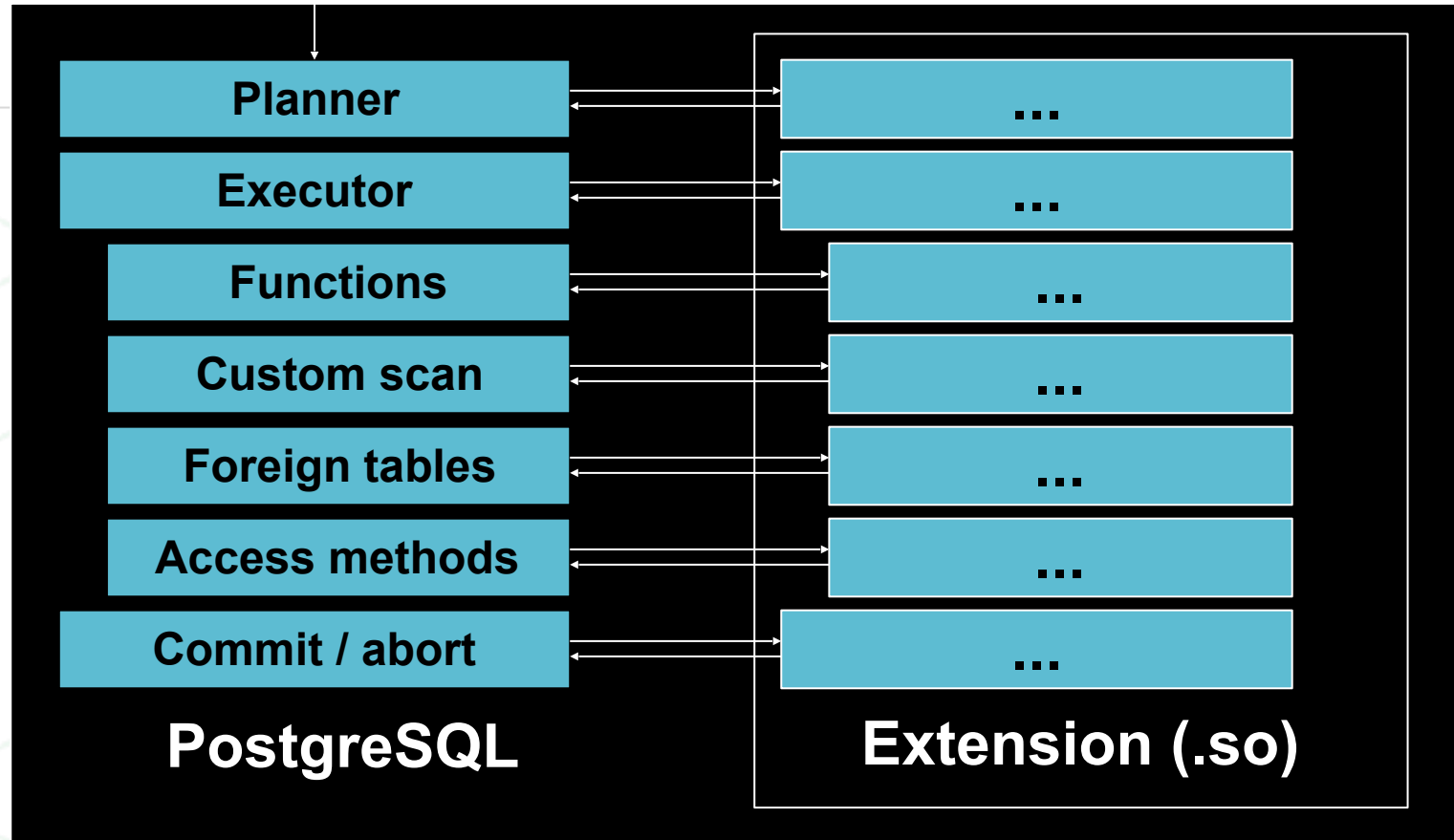
- Scale tables—**via sharding**
- Scale SQL—**via distributed relational algebra**
- Scale transactions—**via 2PC & Deadlock Detection**

 *Now, how do we implement all of this?*

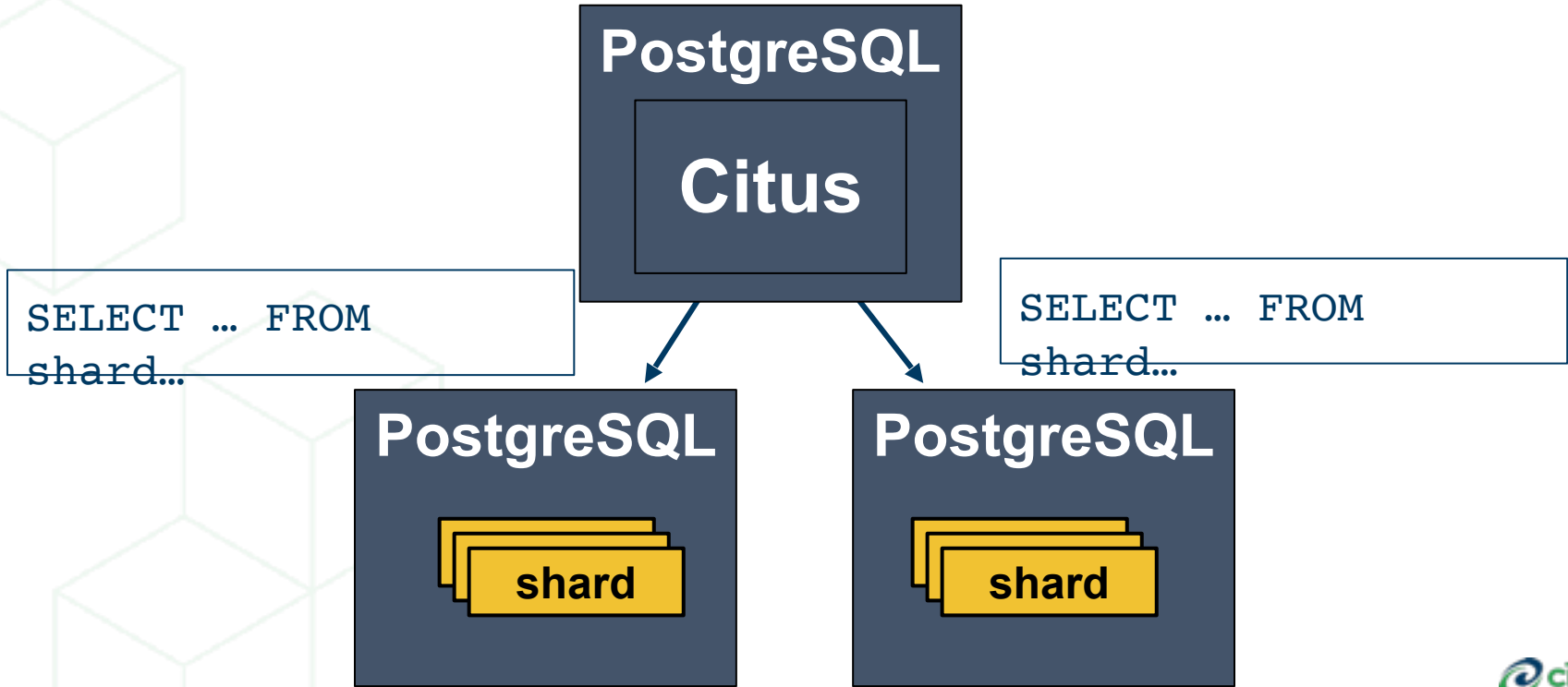


# Postgres

# CREATE EXTENSION ...

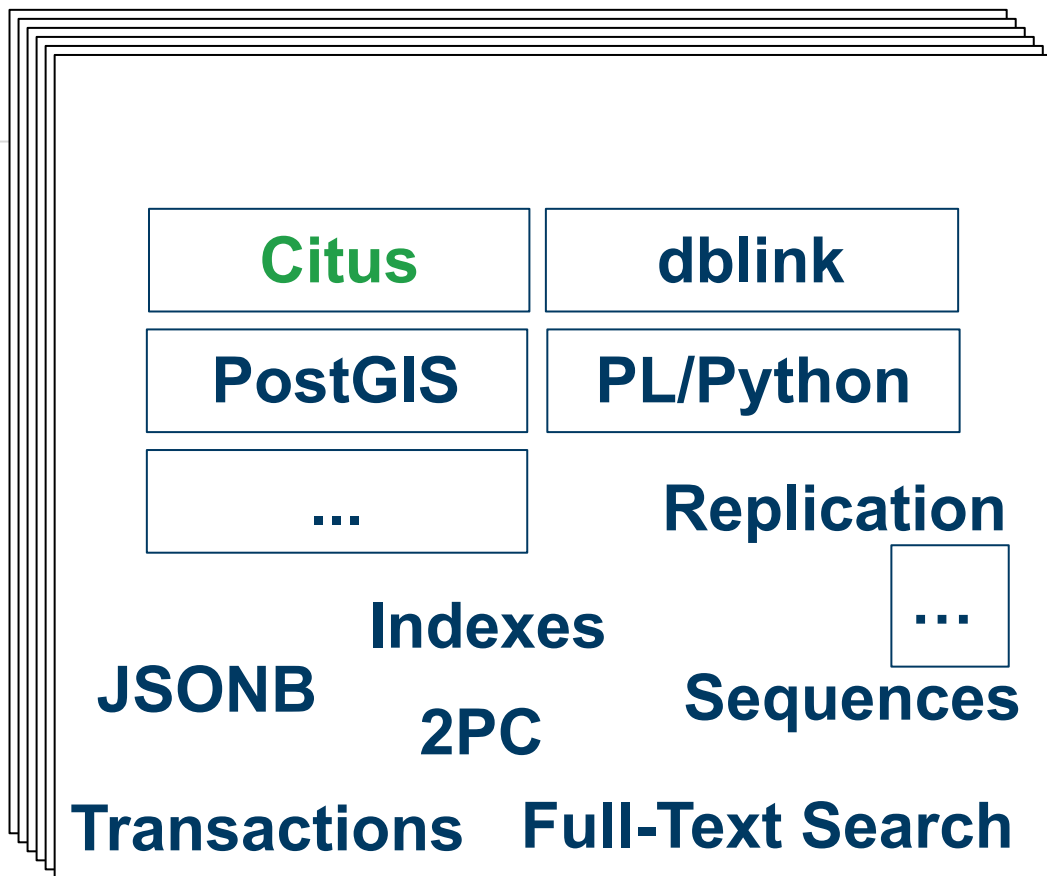


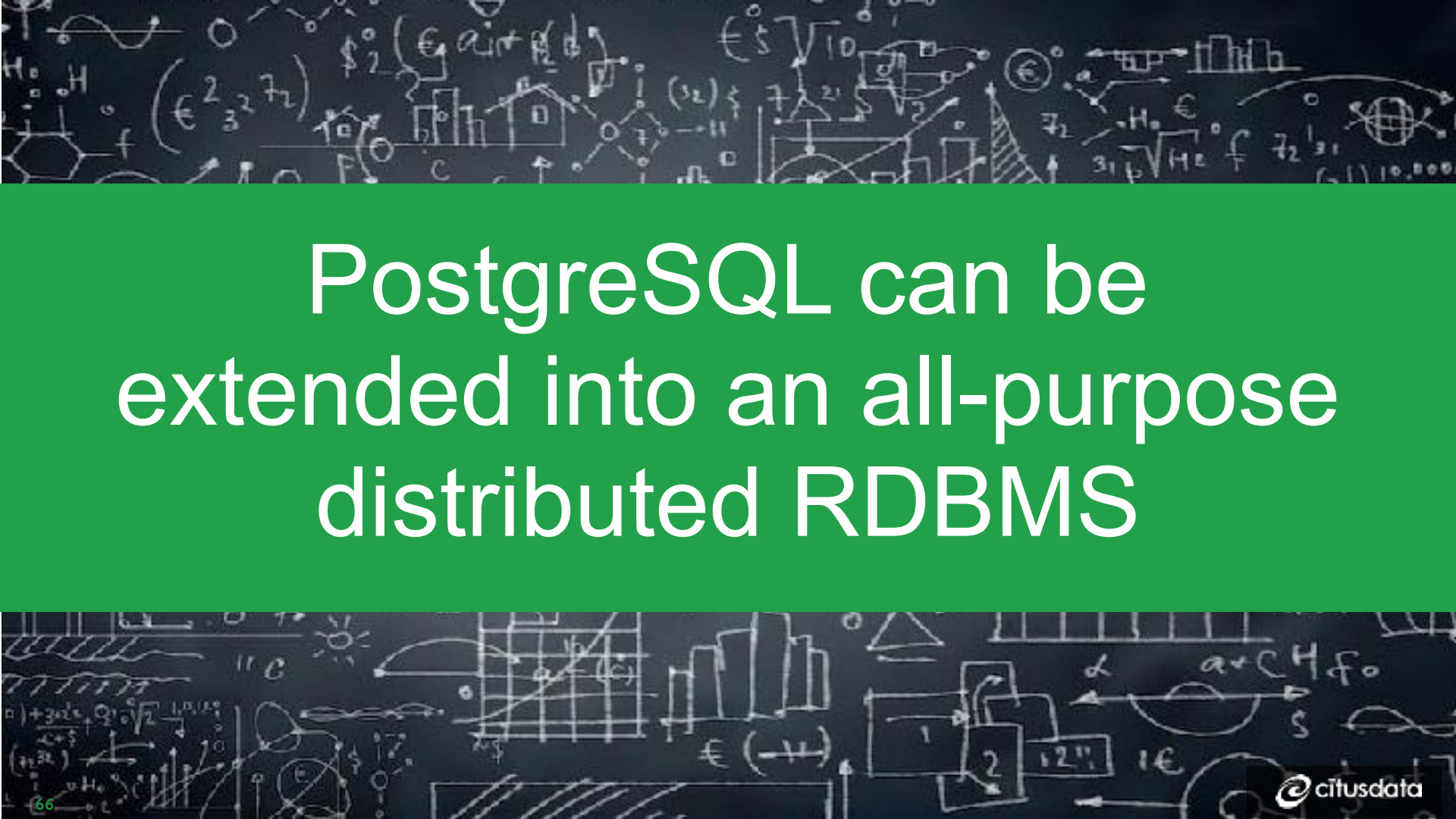
```
SELECT ... FROM distributed_table ...
```





# Rich ecosystem of tooling, data types, & extensions in PostgreSQL





PostgreSQL can be  
extended into an all-purpose  
distributed RDBMS



I believe the future of distributed databases *is* relational.





*Thank you!*

Sumedh Pathak

sumedh@citusdata.com

@moss\_toss | @citusdata | citusdata.com

DataEngConf 2018