Distributed Databases

Marco Slot <marco.slot@microsoft.com>

My career so far

amazon web services

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

citusdata

Microsoft

2009

2014

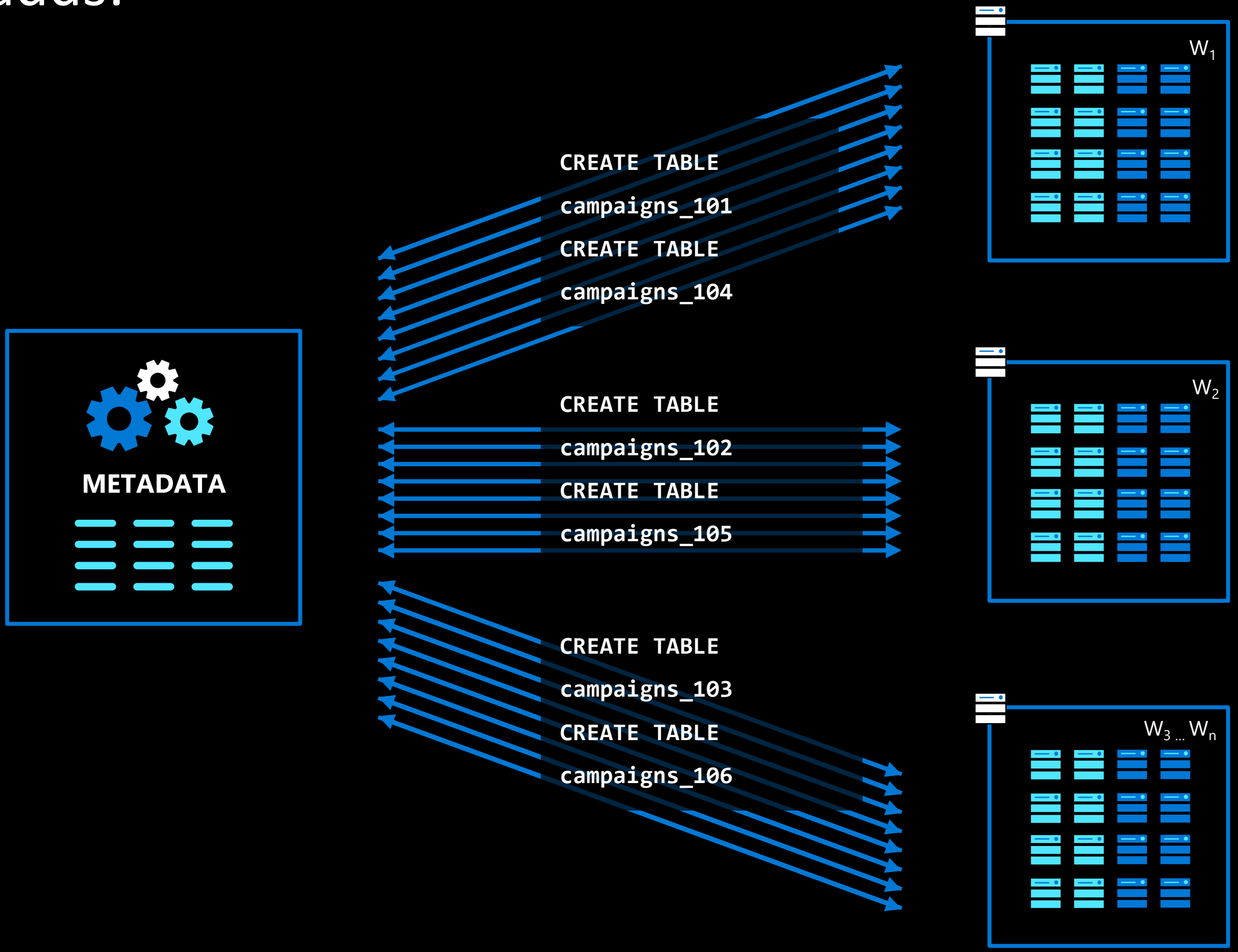
2019

Citus: Distributed PostgreSQL as an extension

Open source PostgreSQL extension that adds:

- Distributed tables
- Reference tables
- Columnar storage
- Query routing
- Parallel, distributed query
- Distributed transactions

https://github.com/citusdata/citus



Distributed database management systems **distribute** and **replicate** data over multiple machines to **try** to meet the **availability**, **durability**, **performance**, **regulatory**, and **scale** requirements of large organizations, subject to physics.

A brief history of distributed databases

2000s: Distributed databases were mostly document stores (NoSQL).

Effectively, a document store is a distributed key -> JSON/BSON/… map with a custom query language. Easy to scale by buying more servers.

2010s: Relational databases added JSON support, Postgres/MySQL caught up to Oracle/MSSQL, and cloud providers made it easy to resize hardware.

Newer distributed databases (e.g. Citus, CockroachDB, Spanner, TiDB, Vitess, Yugabyte) are mostly relational / PostgreSQL- or MySQL-based.

A distributed database does two things

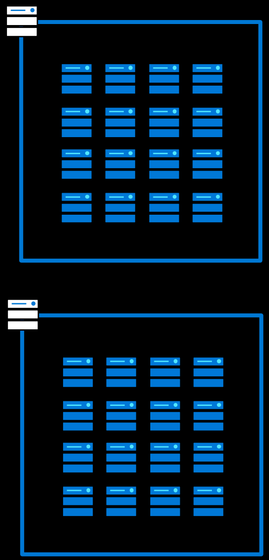
Distribution	- Place partitions of data on different machines
Replication	- Place copies of (a partition of) data on different machines

Goal: Offer same functionality and transactional semantics as an RDBMS

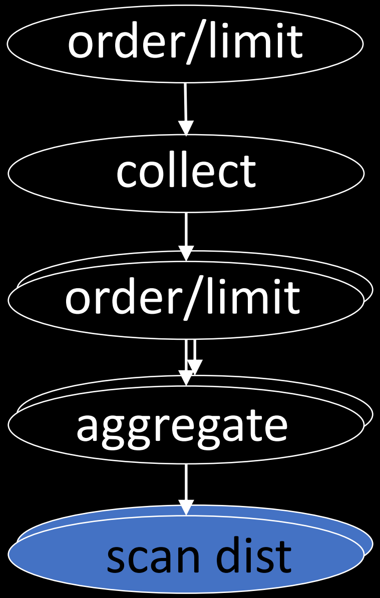
Reality: Concessions in terms of functionality, transactional semantics, and performance

Distribution challenges

Data distribution



Data access (SQL)



Transactions

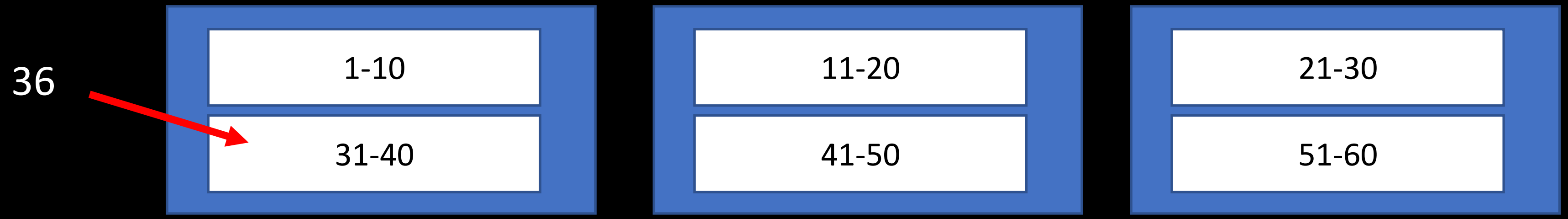
```
BEGIN;
UPDATE account SET amount += 20
WHERE account_id = 1149274;
UPDATE account SET amount -= 20
WHERE account_id = 8523861;
END;
```

Data distribution: Range-distribution

Tables are partitioned by a “distribution key” (part of primary key)

INSERT INTO dist_table (dist_key, other_key) VALUES (**36**, 12);

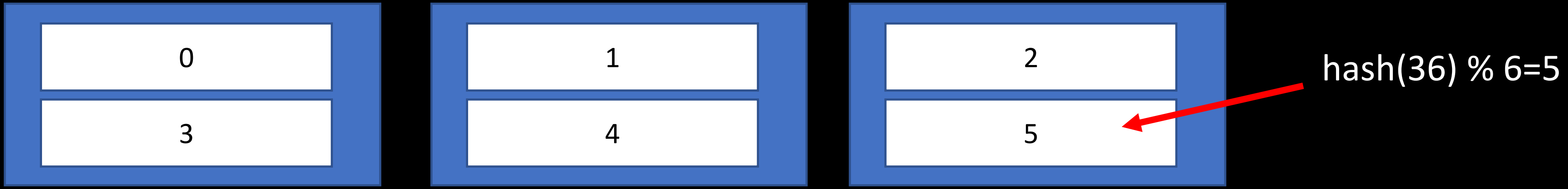
Each “shard” contains a range of values



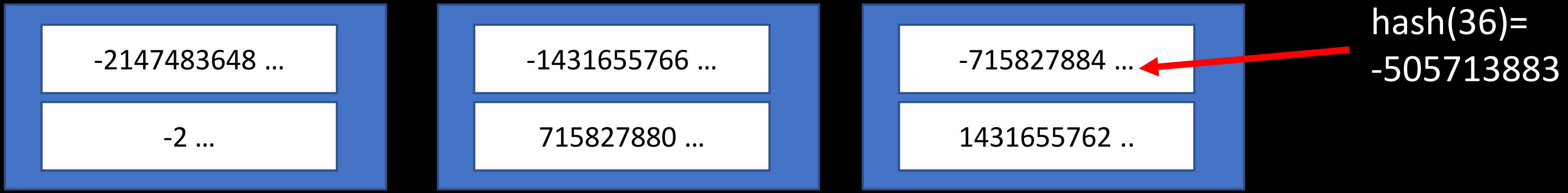
Data distribution: Hash-distribution

```
INSERT INTO dist_tables (dist_key, other_key) VALUES (36, 12);
```

Each shard contains a modulo of a hash value (bad idea)

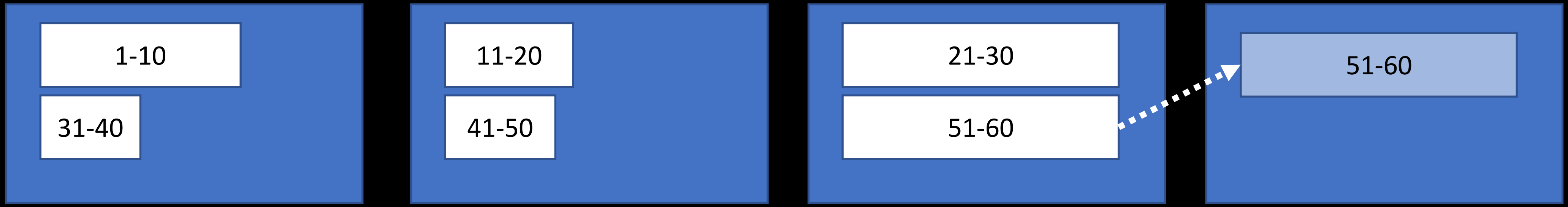


Each shard contains a range of *hash* values (good idea)

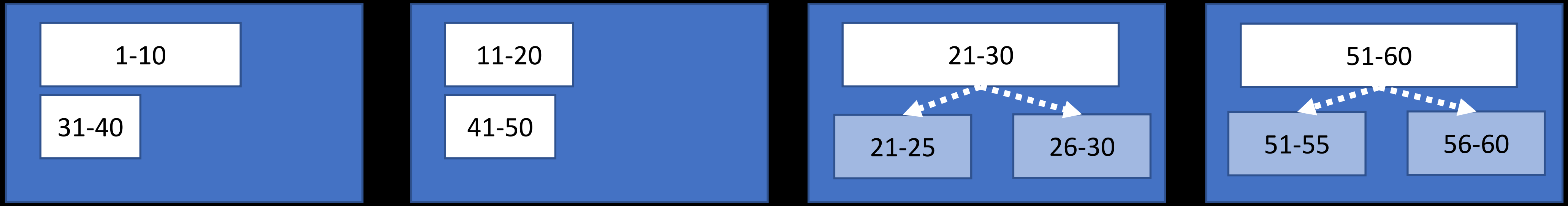


Data distribution: Rebalancing

Move shards to achieve better data distribution across nodes

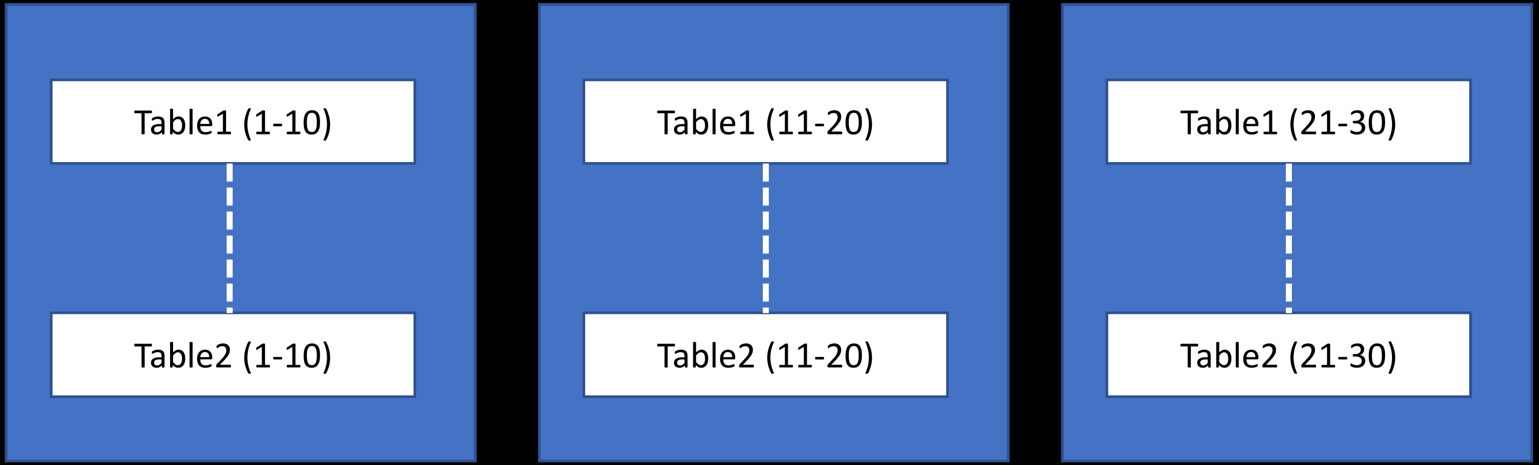


Split shards to achieve better data distribution across shards



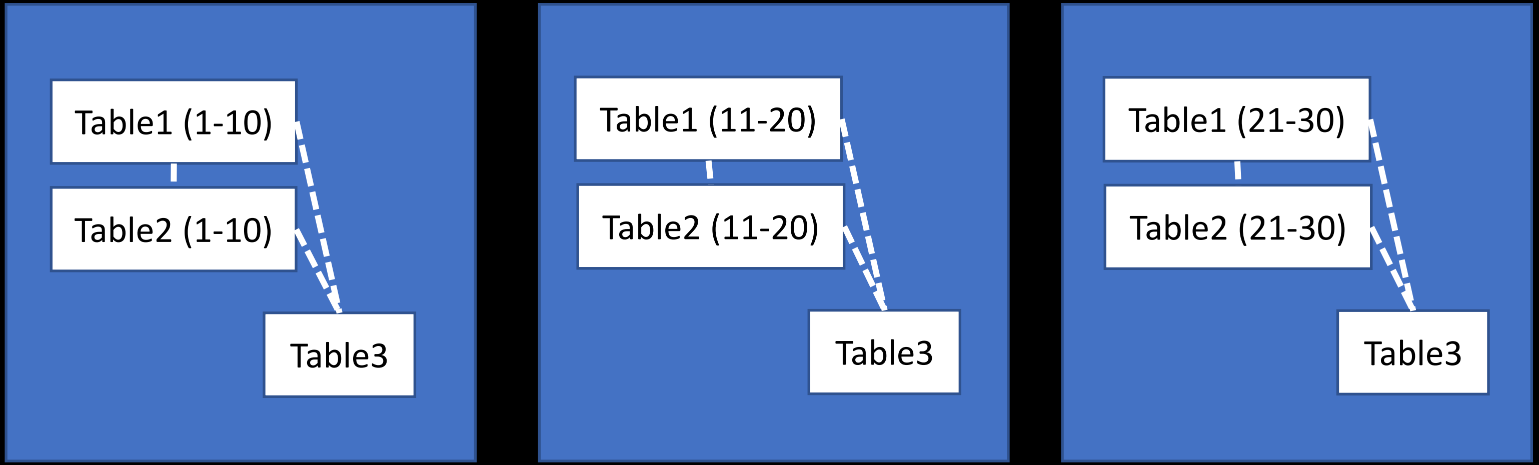
Data distribution: Co-location

Ensure same range is on same node across different tables to enable fast joins, foreign keys, and other operations on distribution key.



Data distribution: Reference tables

Replicate a small table to all nodes to enable fast joins, foreign keys, and other operations on any column.



Data distribution: Other forms

Some other varieties:

• Append distribution	- Write to any partition, read from all
• Random distribution	- Write to random partition, read from all
• List distribution	- Assign “BE” “NL” “UK” to specific partitions
• Spatial distribution	- Assign areas to partitions
• …	

Can combine variants with efficient INSERT..SELECT operations.

Routing queries

To scale query throughput linearly with the number of nodes, queries should only access one node.

```
INSERT INTO dist1 VALUES (36, 11);
SELECT * FROM dist1 WHERE dist_key = 36 AND value < 11;
UPDATE dist1 SET value = 3 WHERE dist_key = 36 AND value < 11;
```

Co-location and reference table enable relatively complex router queries, e.g.:

```
SELECT * FROM dist1 JOIN dist2 USING (dist_key) WHERE dist1.dist_key = 36 AND dist1.value < 11;
UPDATE dist1 d1 SET value = 3 WHERE d1.other_key IN (SELECT other_key FROM ref_table) AND
dist1.dist_key = 36;
DELETE FROM dist1 d1 USING dist2 d2 WHERE d1.dist_key = d2.dist_key AND d1.dist_key = 36;
```

Distributed SQL

SQL	≈ Relational algebra
Distributed SQL	≈ Multi-relational algebra

Relational algebra:

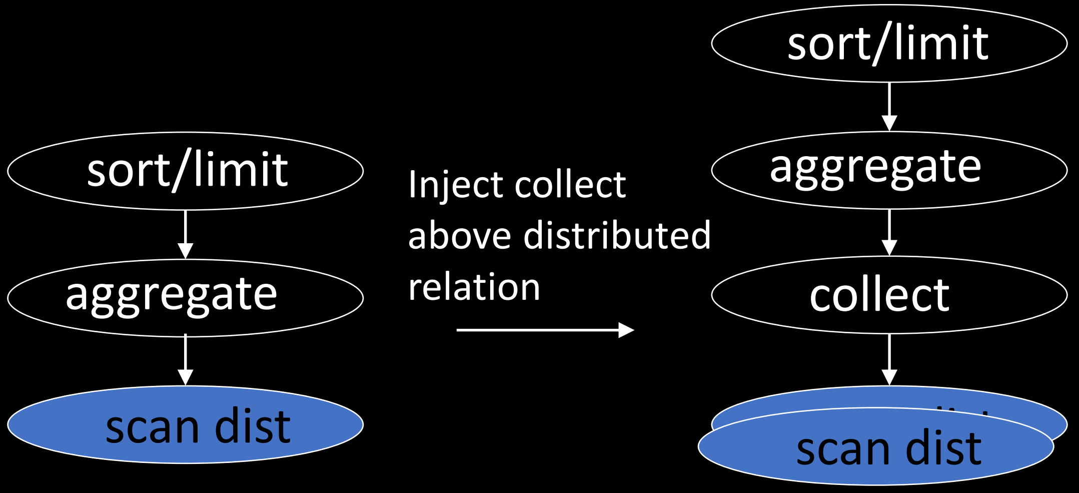
- Scan, Filter, Project, Join, (Aggregate, Order, Limit)

Multi-relational algebra:

- Collect, Repartition, Broadcast + Relational algebra

Distributed SQL: Logical planning

SELECT dist_key, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;

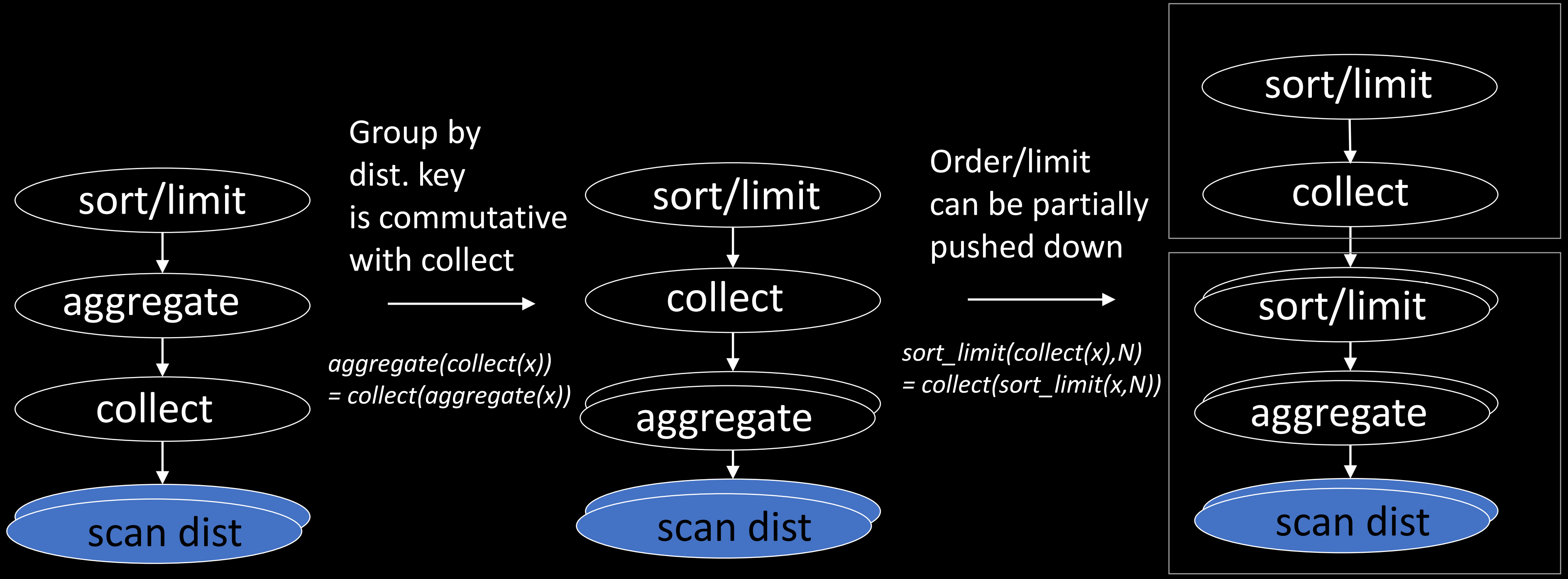


Regular table plan

Distributed table plan

Distributed SQL: Logical optimization

SELECT dist_key, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;



Merge plan

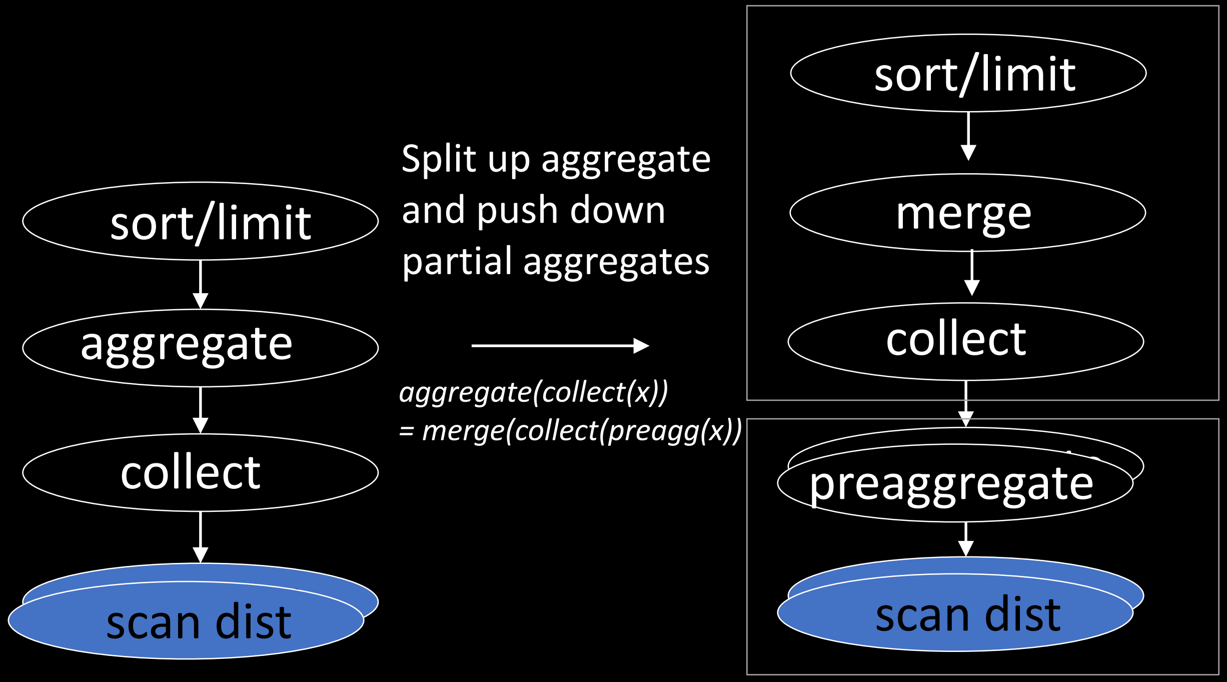
SELECT dist_key, count
FROM <results>
ORDER BY 2 LIMIT 10;

Shard plan (can run in parallel)

SELECT dist_key, count(*)
FROM dist_table_*
GROUP BY 1
ORDER BY 2 LIMIT 10;

Distributed SQL: Logical optimization

SELECT **other_key**, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;


sort/limit
aggregate
collect
scan dist
Split up aggregate and push down partial aggregates
aggregate(collect(x)) = merge(collect(preagg(x))
sort/limit
merge
collect
preaggregate
scan dist

Merge plan

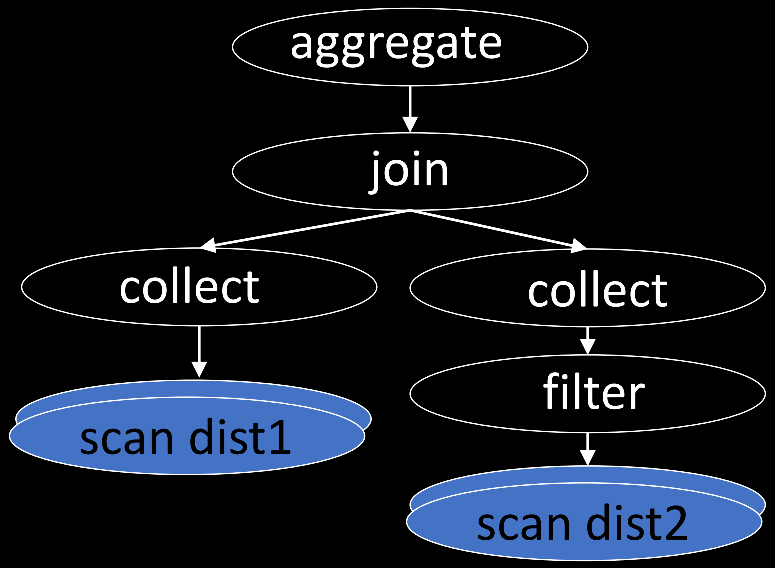
SELECT other_key, sum(count)
FROM <results>
GROUP BY other_key
ORDER BY 2 LIMIT 10;

Shard plan

SELECT other_key, count(*)
FROM dist_table_*
GROUP BY other_key

Distributed SQL: Co-located joins

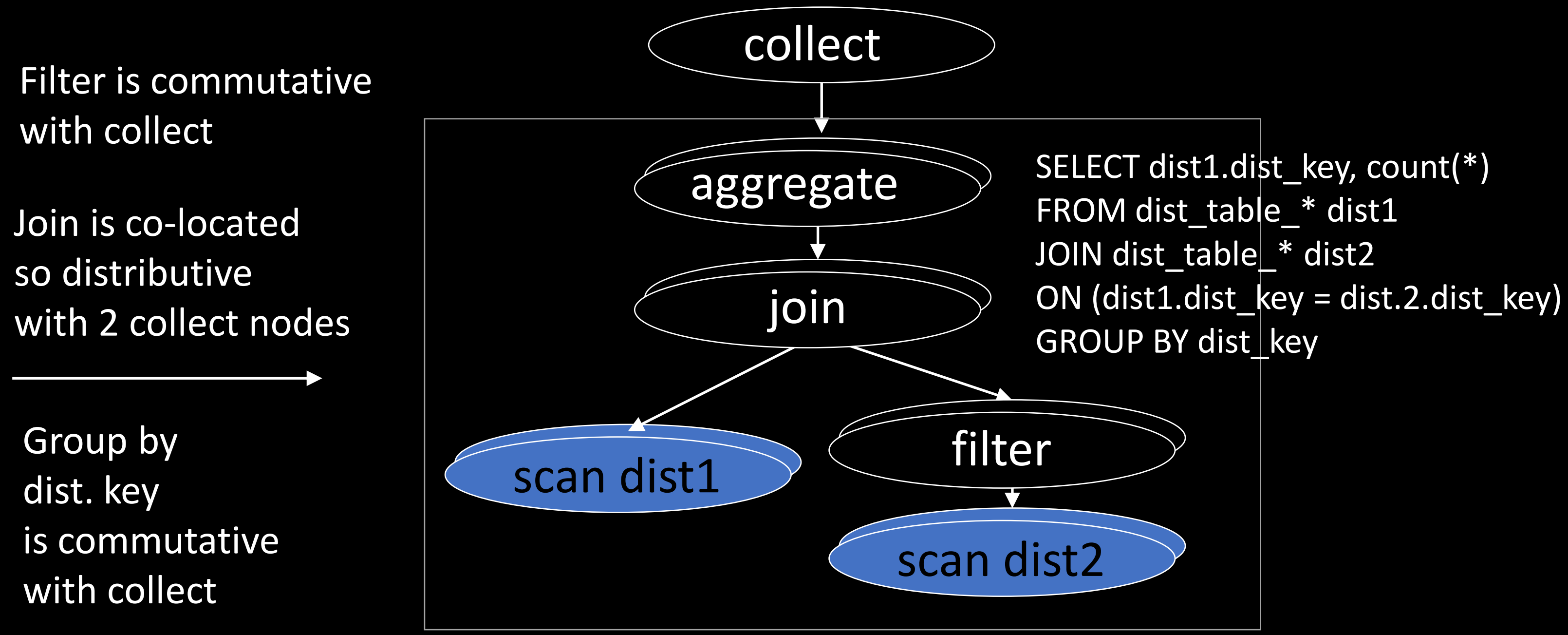
SELECT dist1.dist_key, count(*)
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.dist_key)
WHERE dist2.value < 44 GROUP BY dist1. dist_ key;



Filter is commutative
with collect

Join is co-located
so distributive
with 2 collect nodes

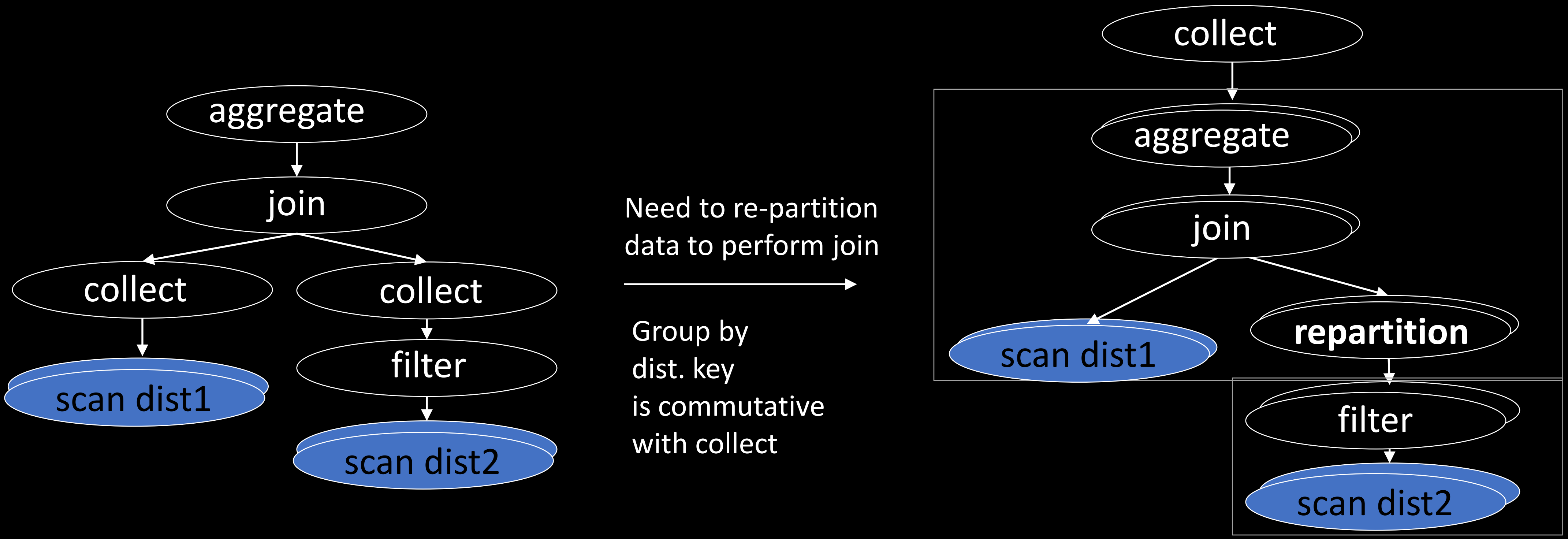
Group by
dist. key
is commutative
with collect



SELECT dist1.dist_key, count(*)
FROM dist_table_* dist1
JOIN dist_table_* dist2
ON (dist1.dist_key = dist.2.dist_key)
GROUP BY dist_key

Distributed SQL: Re-partition joins

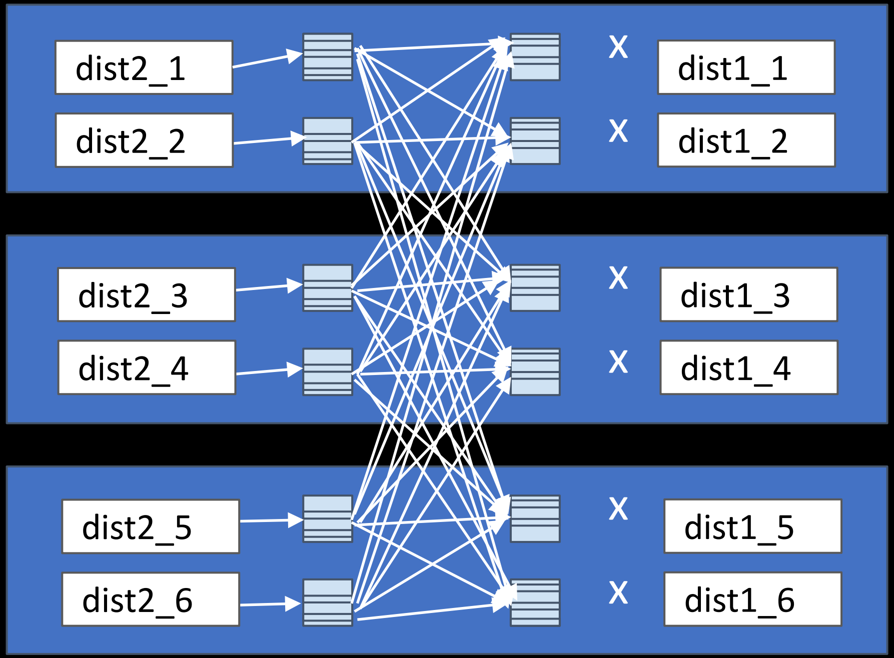
SELECT dist1.dist_key, count(*)
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.**other_key**)
WHERE dist2.value < 44 GROUP BY dist1.dist_key;



Distributed SQL: Re-partition operations

```
SELECT dist1.dist_key, count(*)
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.other_key)
WHERE dist2.value < 44 GROUP BY dist1.dist_key;
```

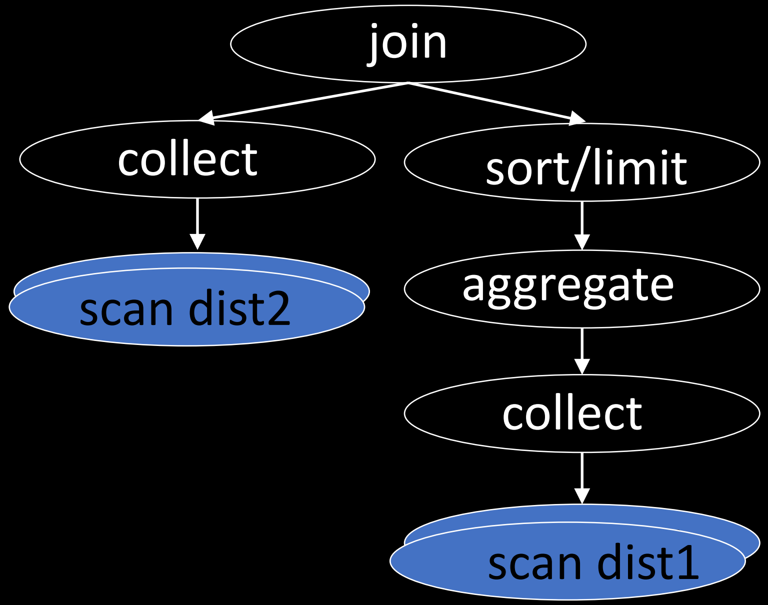
```
SELECT other_key
FROM dist2_*
WHERE value < 44;
```



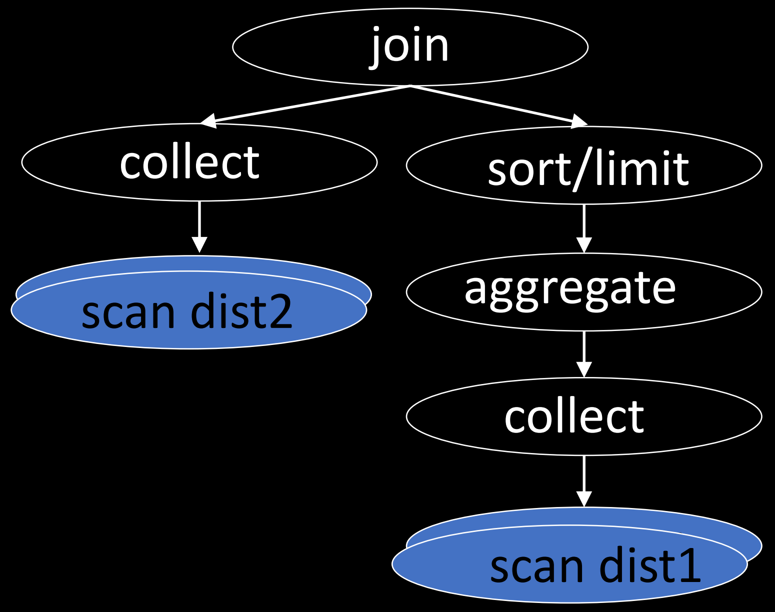
```
SELECT dist1.dist_key, count(*)
FROM dist1_* JOIN <results>
ON (dist1_*.dist_key = <results>.other_key)
GROUP by dist1.dist_key;
```

Distributed SQL: Broadcast joins

```
WITH top10 AS (
  SELECT other_key, count(*) FROM dist1 GROUP BY 1 ORDER BY 2 LIMIT 10
)
SELECT * FROM dist2 WHERE other_key IN (SELECT dist_key FROM top10);
```

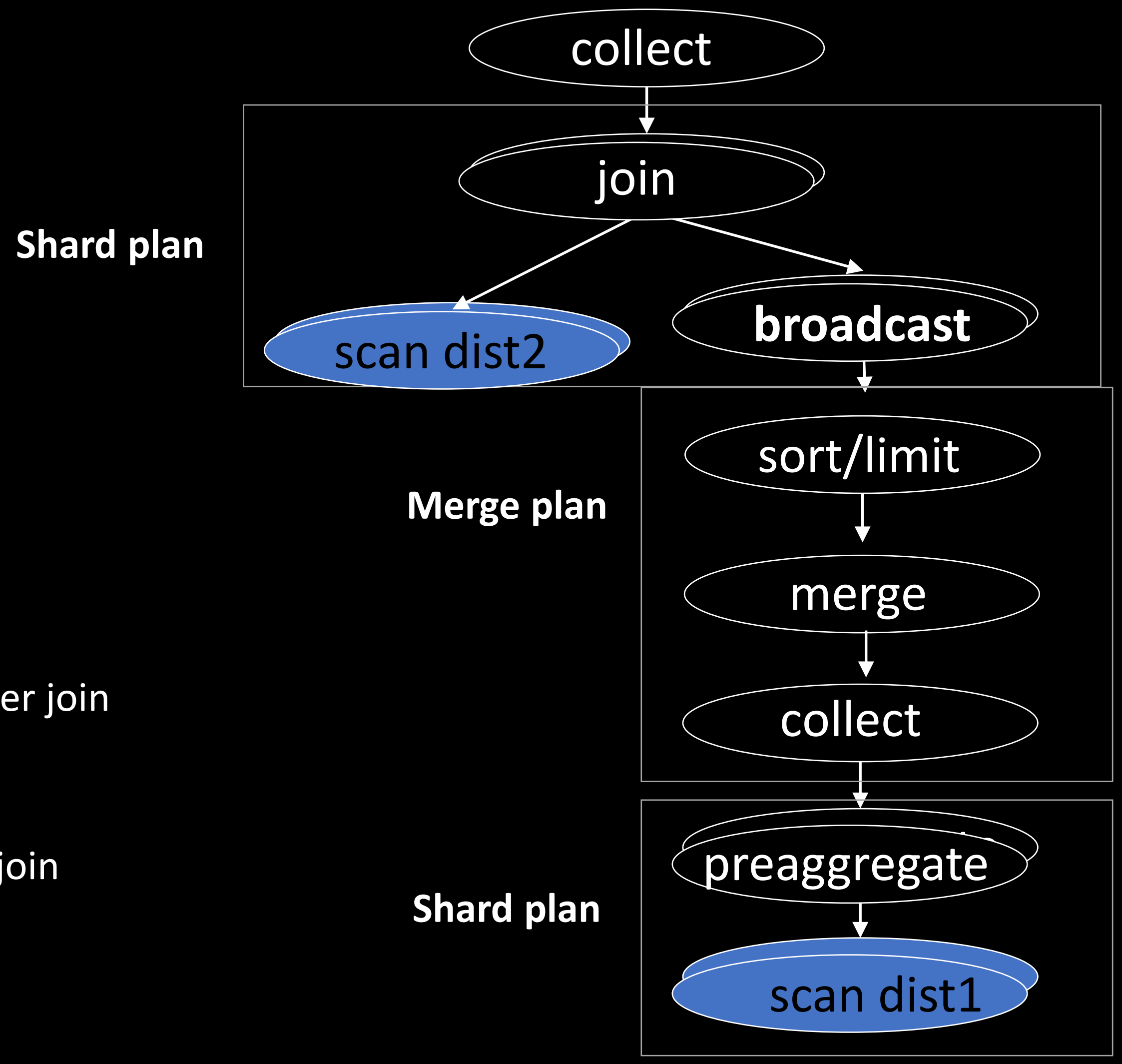


Distributed SQL: Broadcast joins



Create subplan to
handle order/limit under join

Broadcast subplan to
pull collect above the join



Distributed SQL: Observations

Query plans depend heavily on the distribution key.

Runtime also depends on query, data, data size (big in distributed databases), network speed, cluster size, ...

Distributed databases require adjusting your distribution keys & queries to each other to achieve high performance.

Distributed Transactions

Ideally, we have:

Atomicity, Consistency, Isolation, Durability (ACID)

Main distribution challenges:

Atomicity - Commit on all nodes or none

Isolation - See other distributed transactions as committed/aborted

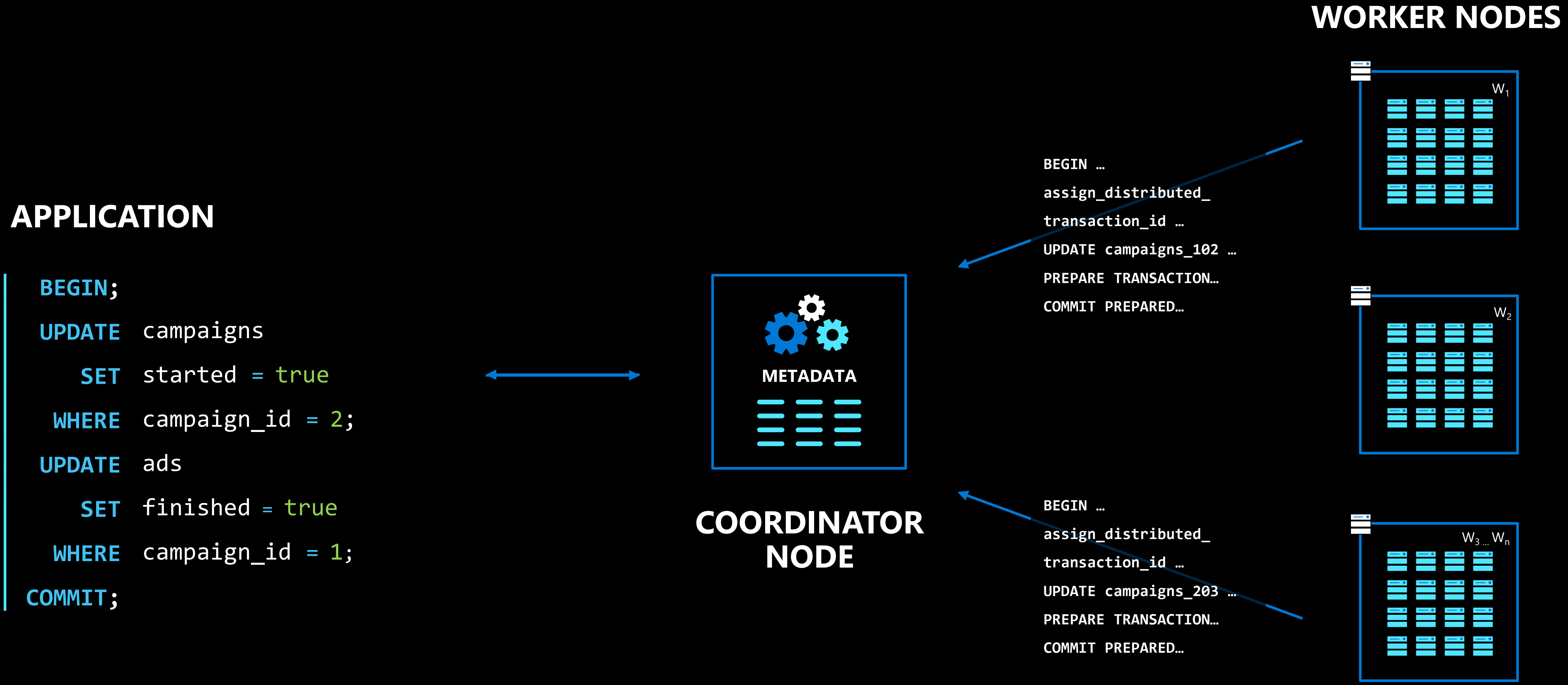
Additionally:

Distributed deadlock detection

Distributed Transactions: Atomicity

Atomicity is generally achieved through 2PC = 2-Phase Commit

Phase 1:	Store ("prepare") transactions on all nodes
Phase 2:	Store final commit decision and …
If success,	Commit all prepared transactions
If error,	Abort all prepared transactions
Secret phase 3:	Commit/abort prepared transactions after failure

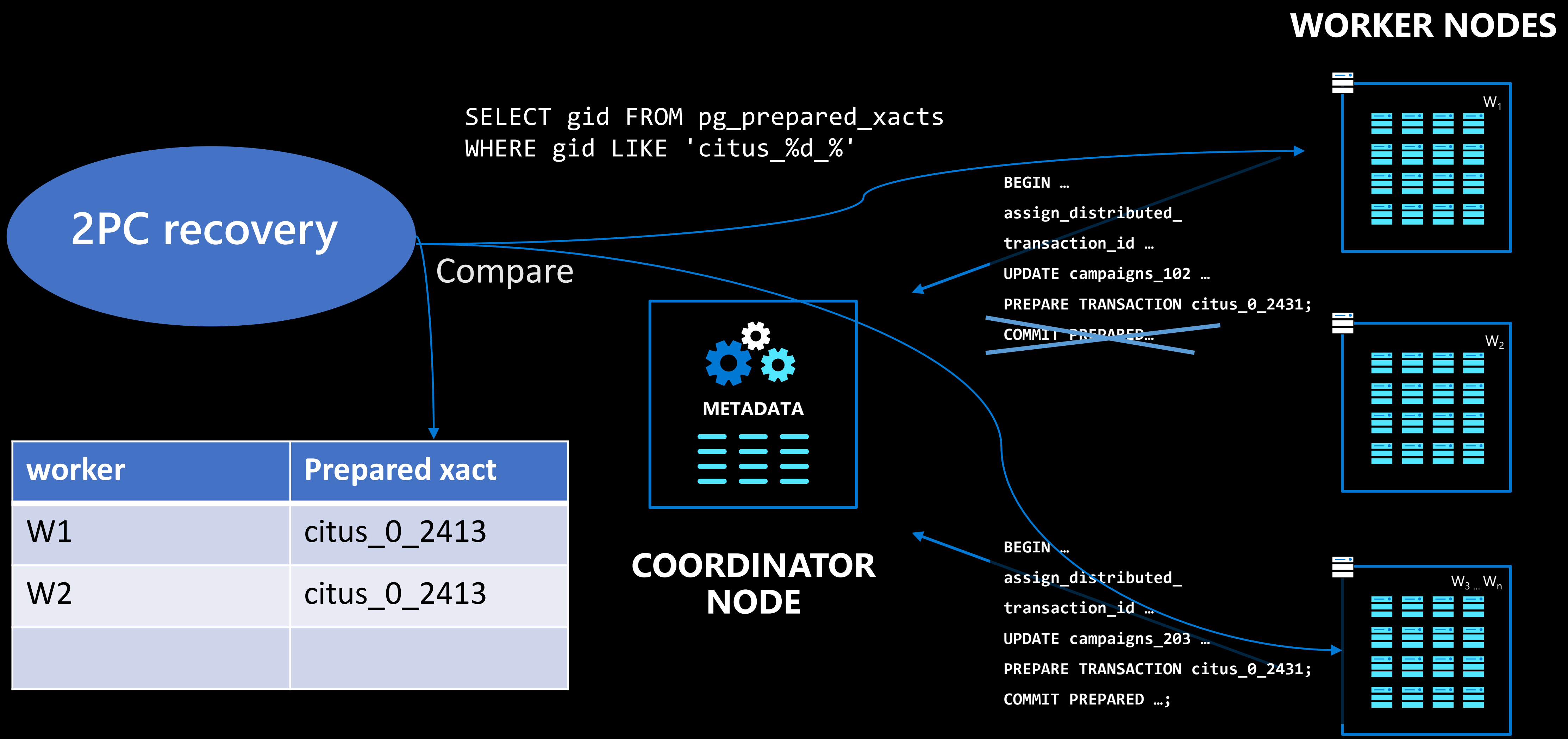


```
  BEGIN;
  UPDATE  campaigns
     SET  started = true
   WHERE  campaign_id = 2;
  UPDATE  ads
     SET  finished = true
   WHERE  campaign_id = 1;
COMMIT;
```

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_102 …
PREPARE TRANSACTION…
COMMIT PREPARED…
```

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_203 …
PREPARE TRANSACTION…
COMMIT PREPARED…
```

How Citus distributes transactions in a multi-node cluster

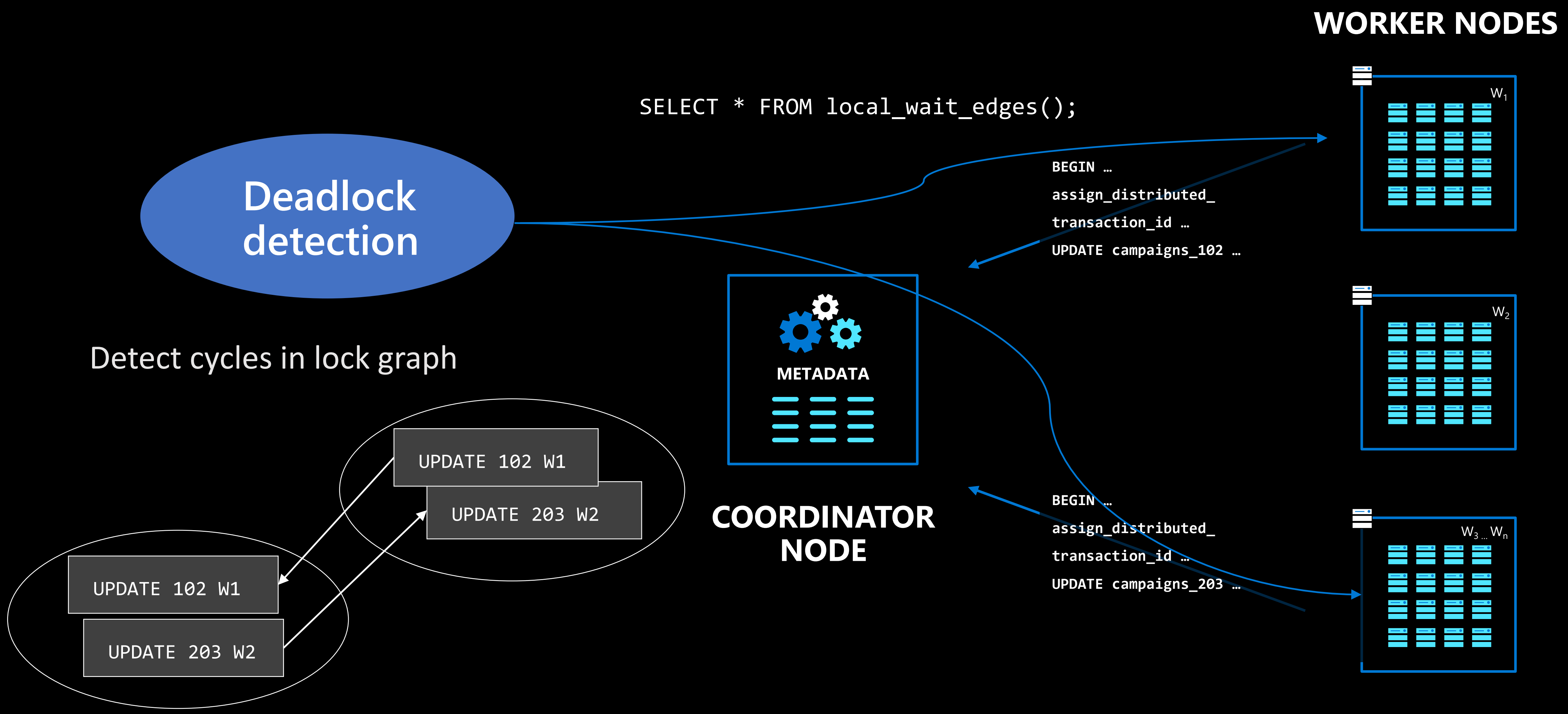


```
SELECT gid FROM pg_prepared_xacts
WHERE gid LIKE 'citus_%d_%'
```

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_102 …
PREPARE TRANSACTION citus_0_2431;
```
~~COMMIT PREPARED…~~

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_203 …
PREPARE TRANSACTION citus_0_2431;
COMMIT PREPARED …;
```

worker	Prepared xact
W1	citus_0_2413
W2	citus_0_2413



```
SELECT * FROM local_wait_edges();
```

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_102 …
```

```
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_203 …
```

Distributed Transactions: Isolation

If we query different nodes at different times, we may see a concurrent transaction as committed on one node, but not yet committed on another.

Distributed snapshot isolation means we have the same of view of what is committed and not committed on all the nodes.

Must also ensure *consistency*: Any preceding write is seen as committed.

Distributed Transactions: Isolation

Each query has a timestamp, should see all commits with lower timestamps.

Different ways of dealing with clock synchronization:

TrueTime:	Used in Google Spanner, synchronize clocks using GPS/atomic clocks. Commits pause until all clocks move past commit time.
Clock-SI:	Queries collect current time from all nodes involved, pick the highest timestamp and wait for it to pass.
HLC:	Hybrid logical clocks are increased whenever an event occurs or a message from another node is received with a higher timestamp

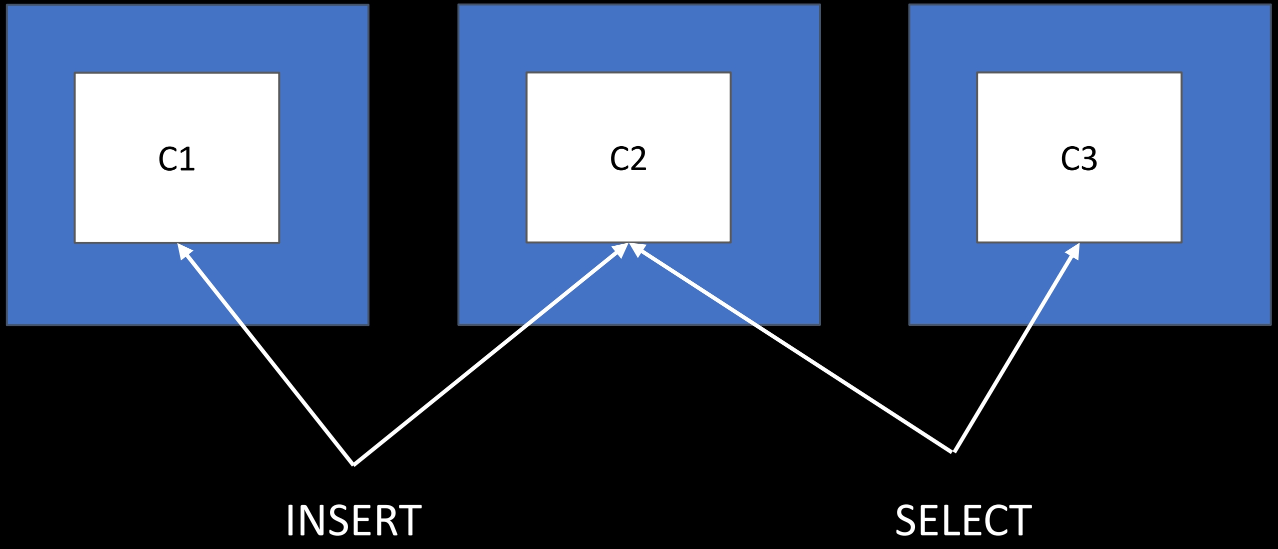
Replication

Why replication?

for availability	- resume from replica in case of node failure
for durability	- restore from replica in case of disk failure
for read throughput	- divide reads across read replicas
for read latency	- local/nearby replica gives lower read latency
for write latency	- local/ nearby replica gives lower write latency

Replication: Quorums

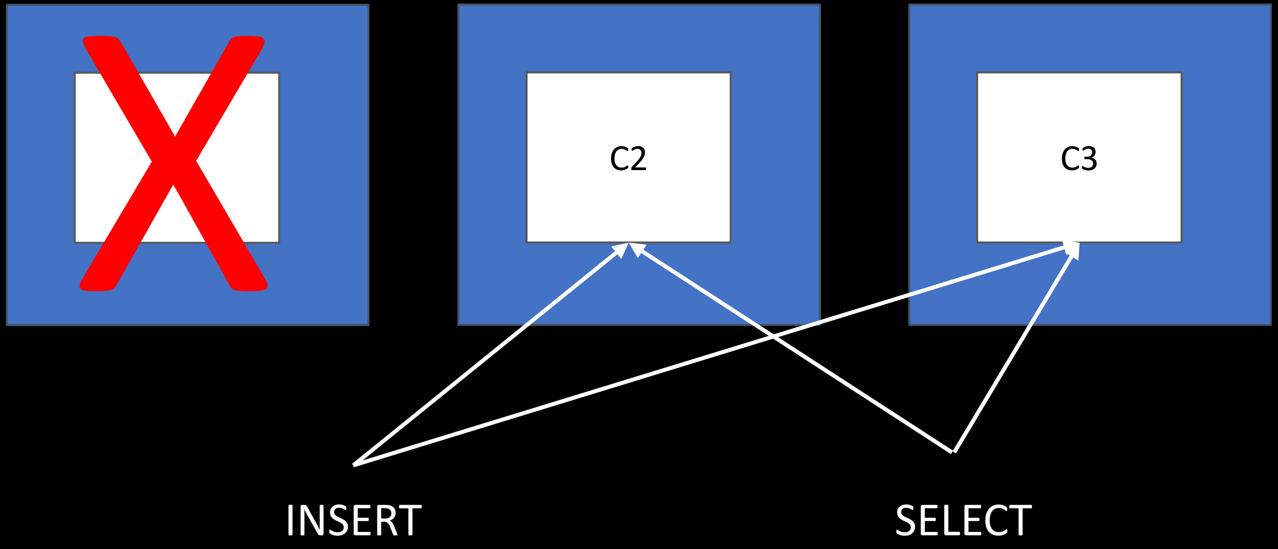
Basic idea: Read from R nodes, Write to W nodes, R +W > N



Challenge: Applying events in same order everywhere

Replication: Quorums

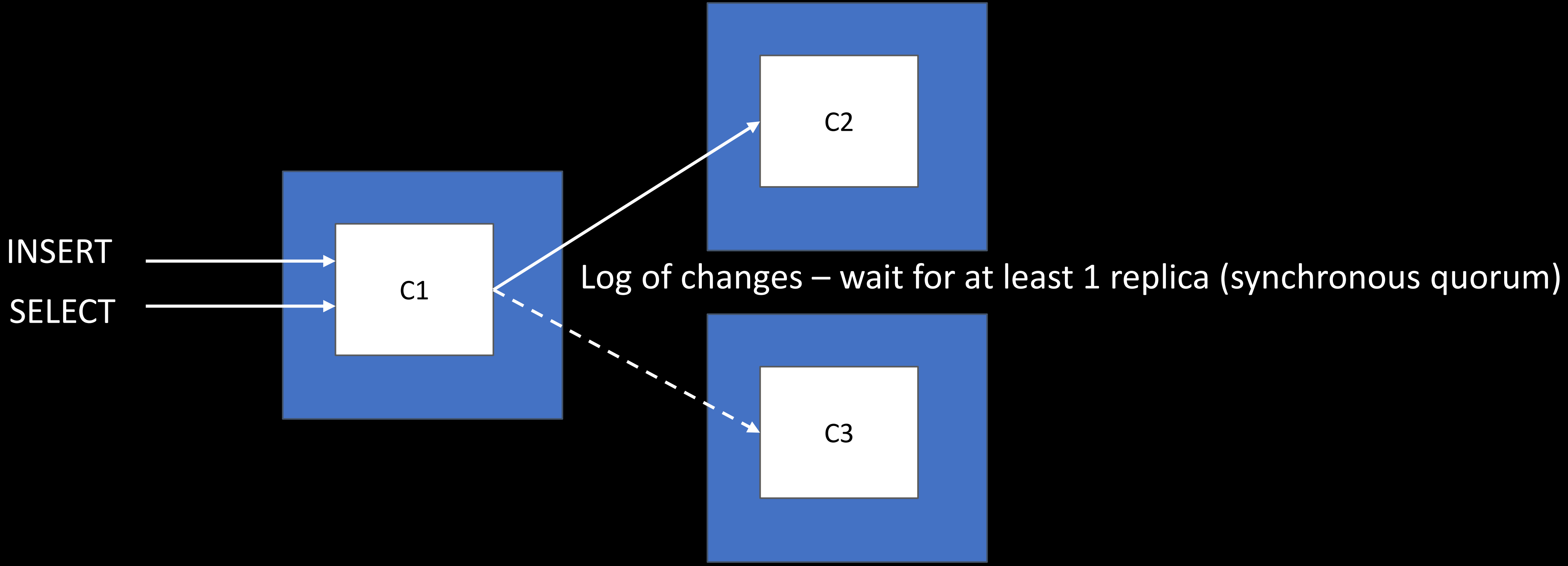
Basic idea: Read from R nodes, Write to W nodes, R +W > N



Challenge: Applying events in same order everywhere

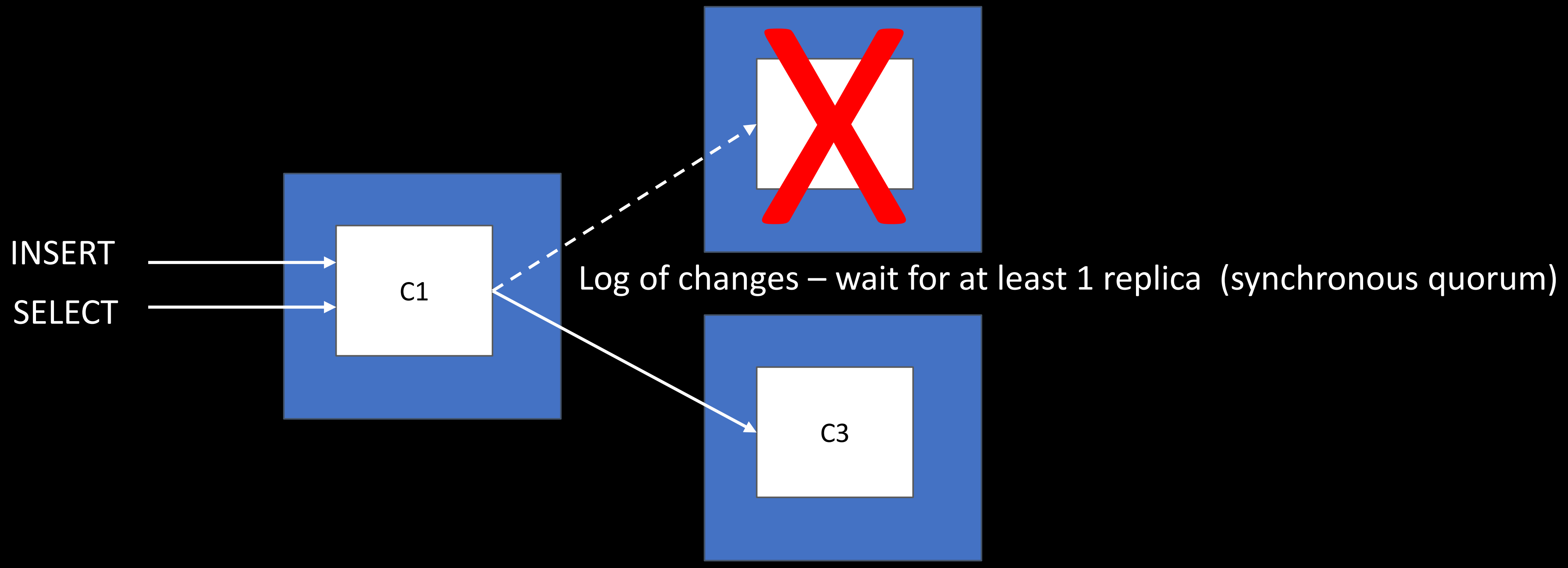
Replication: Follow the leader

Assign temporary leader to serialize writes efficiently


INSERT
SELECT
C1
C2
C3
Log of changes – wait for at least 1 replica (synchronous quorum)

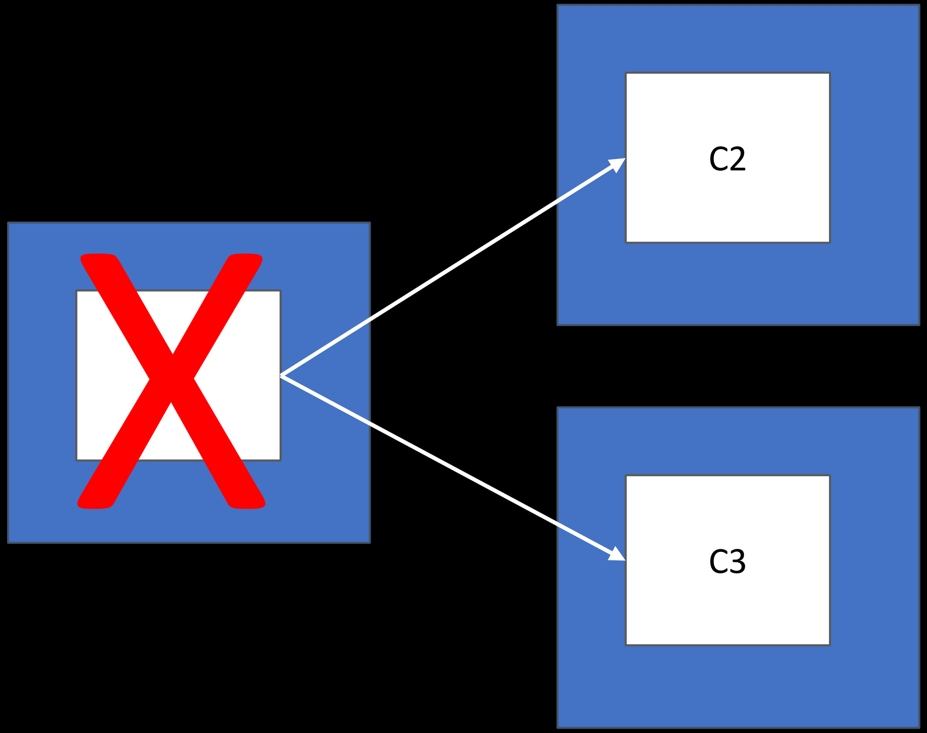
Replication: Follow the leader

Standby fails: Continue writing to other replica


INSERT
SELECT
C1
Log of changes – wait for at least 1 replica (synchronous quorum)
C3

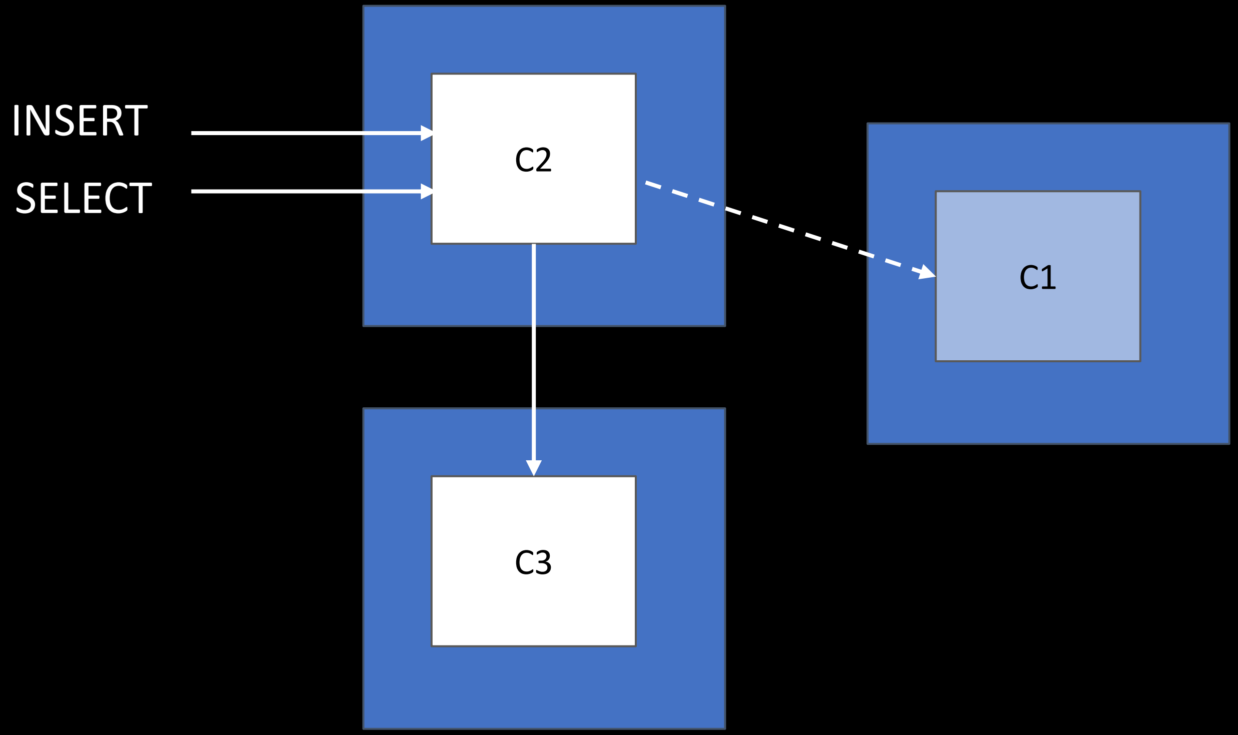
Replication: Follow the leader

Primary fails: Initiate a failover



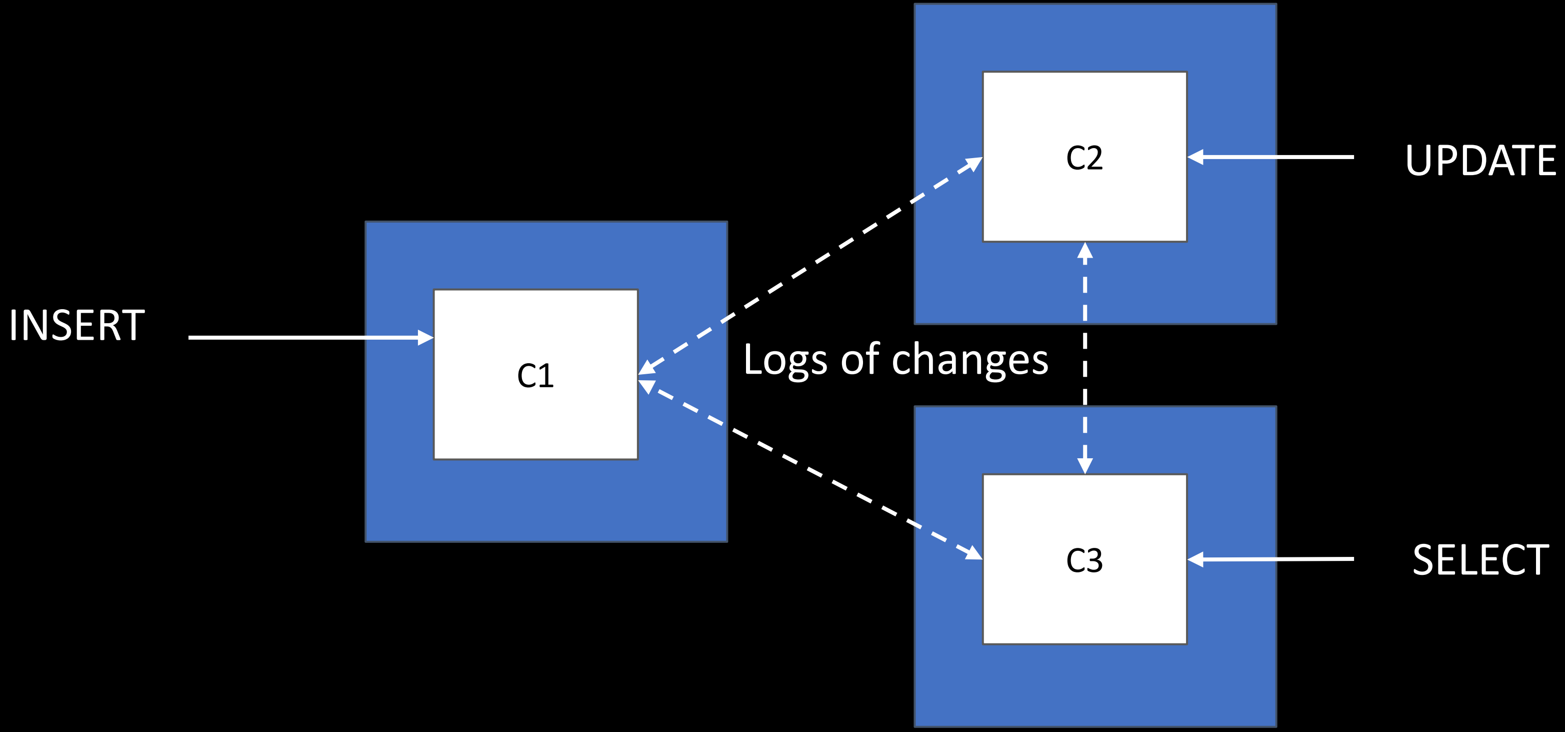
Replication: Follow the leader

Replica is promoted to leader, other replicas follow new leader.



Replication: N-directional

All nodes accept writes, somehow reconcile conflicting changes.

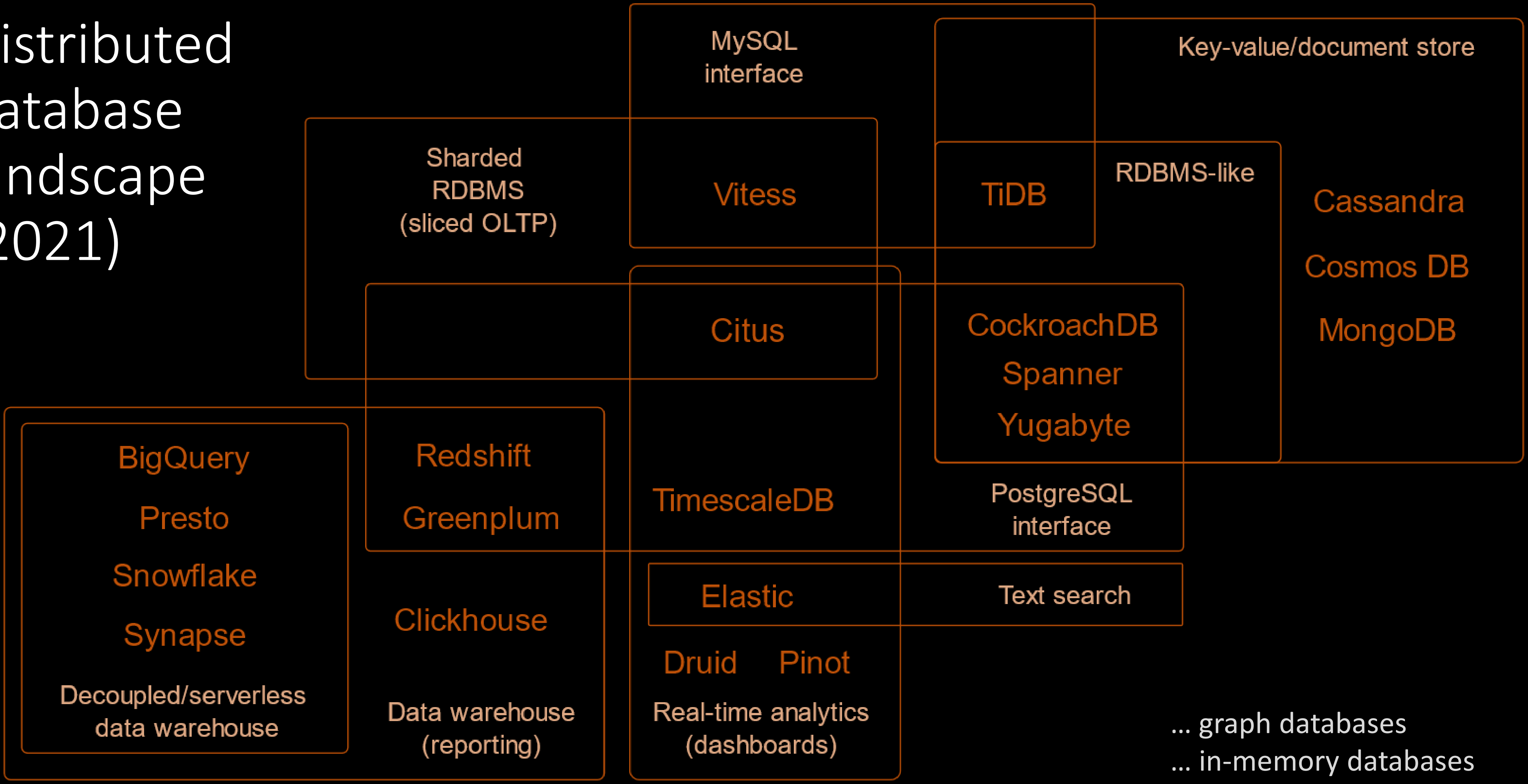


Replication

Which form of replication?

for availability	- follow the leader, synchronous to a quorum
for durability	- any
for read throughput	- follow the leader, synchronous/asynchronous
for read latency	- follow the leader, asynchronous
for write latency	- n-directional

Distributed database landscape (2021)



… graph databases
… in-memory databases

Questions?

marco.slot@microsoft.com