Distributed Databases

Marco Slot - marco.slot@microsoft.com

Principal Software Engineer at Microsoft

My career timeline

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

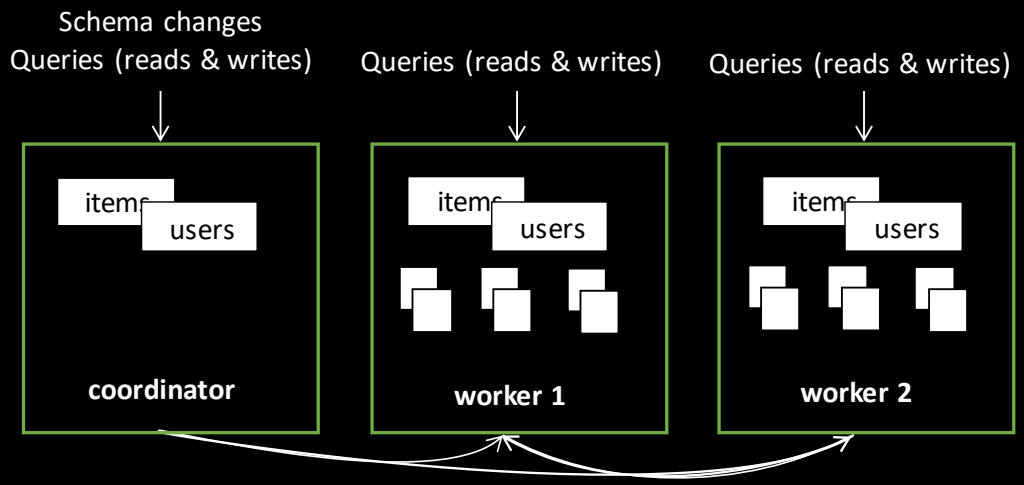
2009

2014

2019

Citus: Distributed PostgreSQL as an extension

Citus is a PostgreSQL *extension* that adds the ability to transparently distribute or replicate tables across a cluster of PostgreSQL servers.



GitHub: https://github.com/citusdata/citus

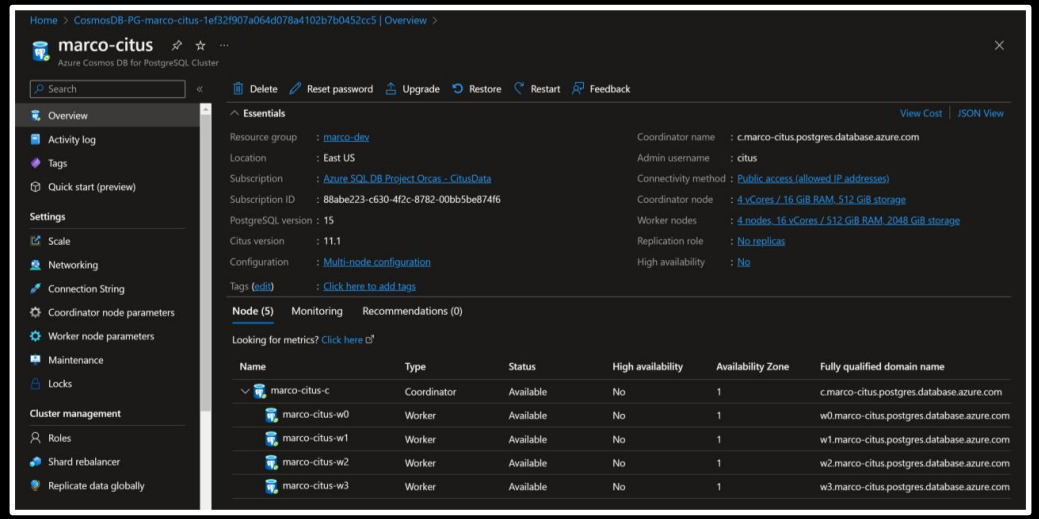
SIGMOD '21 paper: "Citus: Distributed PostgreSQL for Data-Intensive Applications"

Azure Cosmos DB for PostgreSQL

Managed service for **Citus** clusters:

- Deploys a cluster in a few clicks
- Hot standby and automatic failover
- Point-in-time-recovery
- Read replicas
- Logging & Monitoring
- *We carry the pager*

(managed service is like a contractor vs. DIY)



Azure Portal

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.

SQL vs. NoSQL then and now

SQL / RDBMS – powerful API, but historically rigid, fixed set of types
NoSQL – limited API, store unstructured documents (JSON) by key

A new database: Distributed databases were initially document stores.
People liked document stores due to schema flexibility, scalability.

SQL strikes back: PostgreSQL added good JSON support in 2012, others followed.
Open-source RDBMS became good enough for enterprises.
Managed services made running a SQL database & scaling up easy.

Return of distributed: Most new distributed databases have SQL and distributed transactions
NoSQL still has specialized use cases (ultra high performance, availability)

✓ A distributed database does two things

have data distributed over different nodes ⇒ not everything on the same node → no need to replace a previous node with a more powerful one, we just add a node to take part of the job

Distribution - Place partitions of data on different machines

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

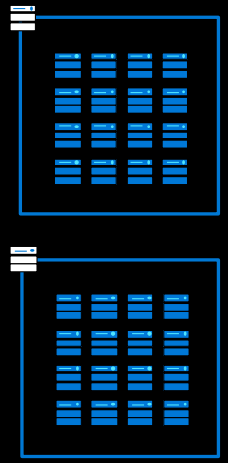
↳ data is not saved once, it is saved multiple times in shards stored in different nodes
⇒ a node is not the only one having the data it store → can fail without the whole system losing access to the data it saved

Goal: Offer same functionality and transactional semantics as an RDBMS

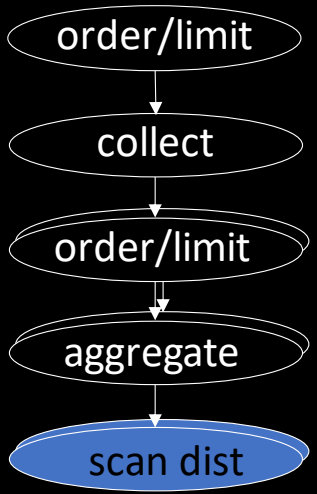
Reality: Concessions in terms of functionality, transactional semantics, and/or performance

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

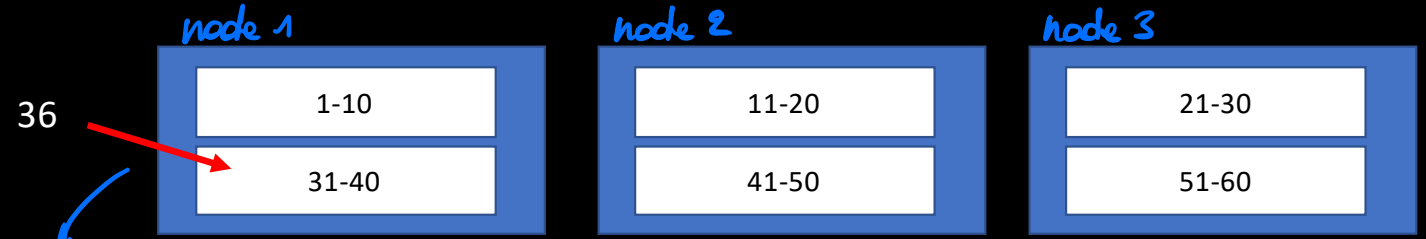
additionnal key used to split the database in shards that can be distributed over different nodes

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

↳ allows for referencing of a shard of the database that is located on an other node of the distributed database

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

Each “shard” contains a range of values



data is sorted by range of distribution keys in one shard for range

! ranges are not easy to define for all data types → not trivial / impossible for some objects

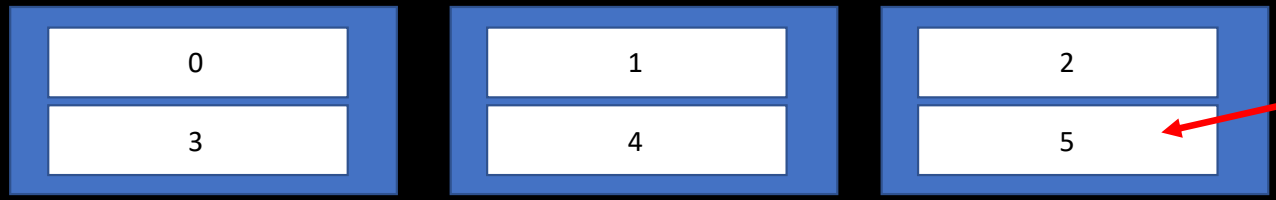
Data distribution: Hash-distribution

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

would allow to have range-distribution

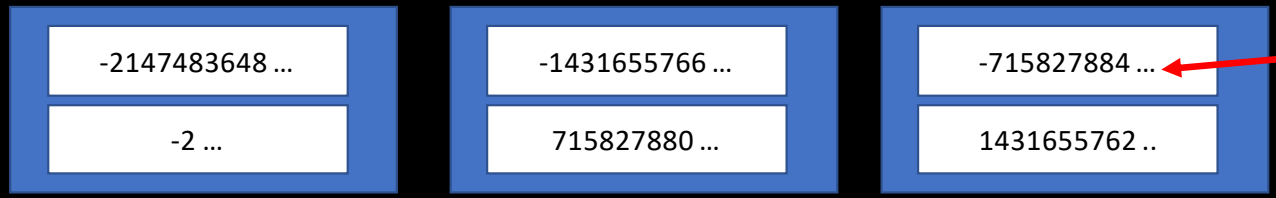
Each shard contains a modulo of a hash value (bad idea)

because if the modulo changes, all the data could need to be moved



but not able to sort in ranges anymore

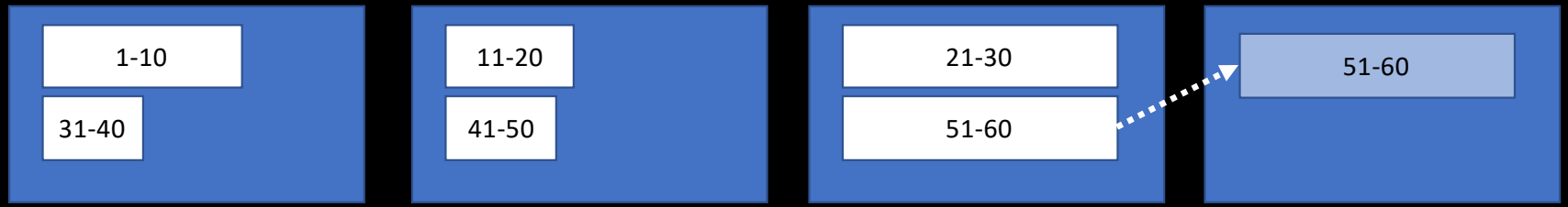
Each shard contains a range of *hash* values (good idea)



Data distribution: Rebalancing

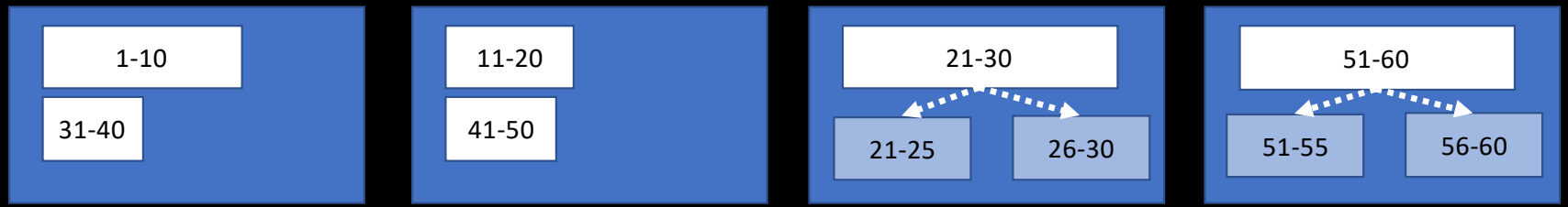
One of the main concerns of distributed databases, keep the data in such a way that the database is still efficient and that each node is not overcharged

Move shards to achieve better data distribution across nodes



+ the data must still be queryable during the rebalancing ⇒ keep track of modification that happened during the rebalancing and update after completely done → could have new changes during update ⇒ cycles until up to date

Split shards to achieve better data distribution across shards

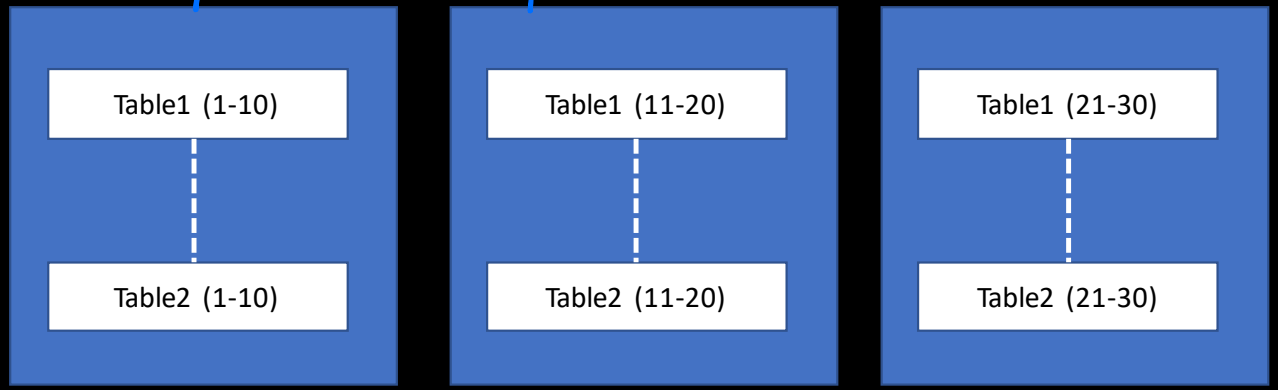


Data distribution: Co-location

we store similar parts of the different tables having the same columns (same ranges of the column in common)

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

→ also true for several other operations



⟹ enable faster joins on those columns since the distributed database won't need to transfer a part of data from node to node to compute this part of the join

↳ more efficient than doing it over the network

Note: Network latency is the enemy compared to operations on the same machine

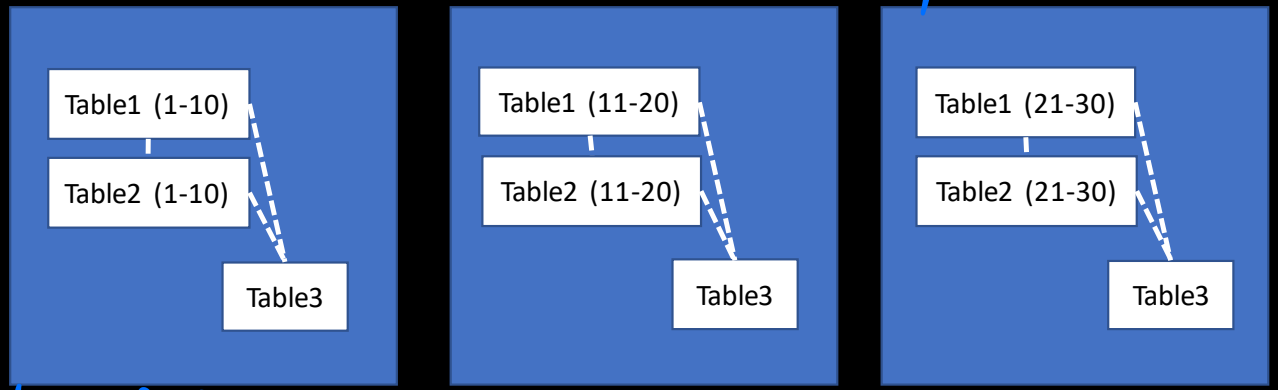
Co-location only works when we are joining on the distribution key

Data distribution: Reference tables

some smaller tables are replicated in all the nodes

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

each node compute the join on its own replicate table and then return the result



⇒ only works for relatively small machines

+ we don't want very high write throughput to be shared on all the nodes

↳ if we don't have that

"pointer to the reference table"

it is a great way to increase scalability since the distributed table can store an ID that will allow to search for details in reference table (ex: customer info)

doesn't need to store all the details in the gigantic distributed tables (ex: order table)

Data distribution: Other forms

Some other varieties:

- 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.

Distributed SQL

Sometimes faster than regular SQL

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 → add operators

Relational algebra:

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

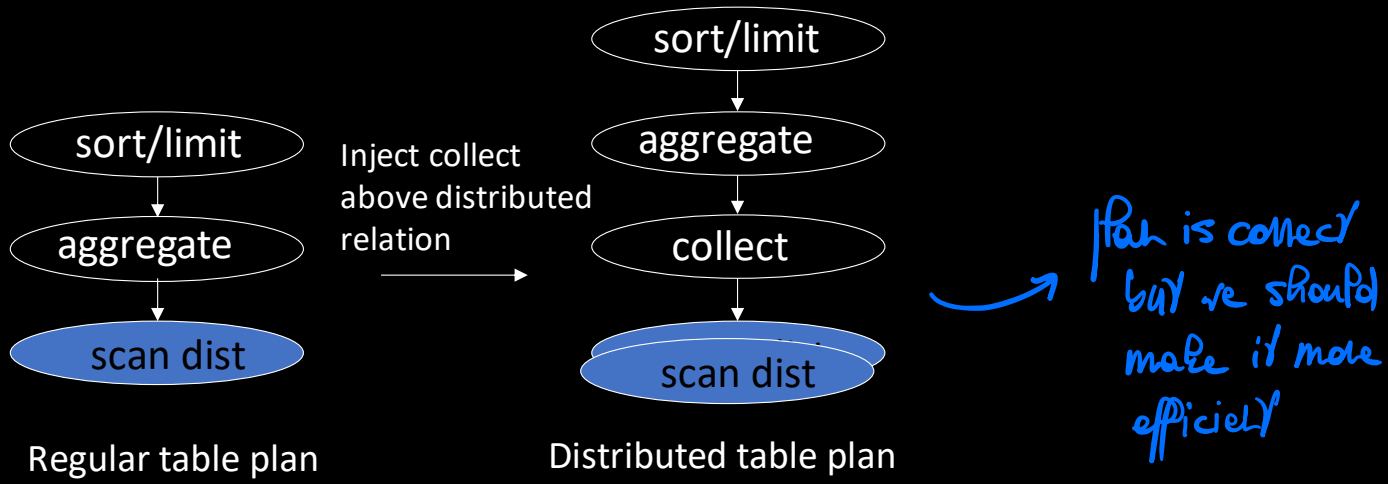
Multi-relational algebra:

- Collect, Repartition, Broadcast + Relational algebra

gather all information in one place

Distributed SQL: Logical planning

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



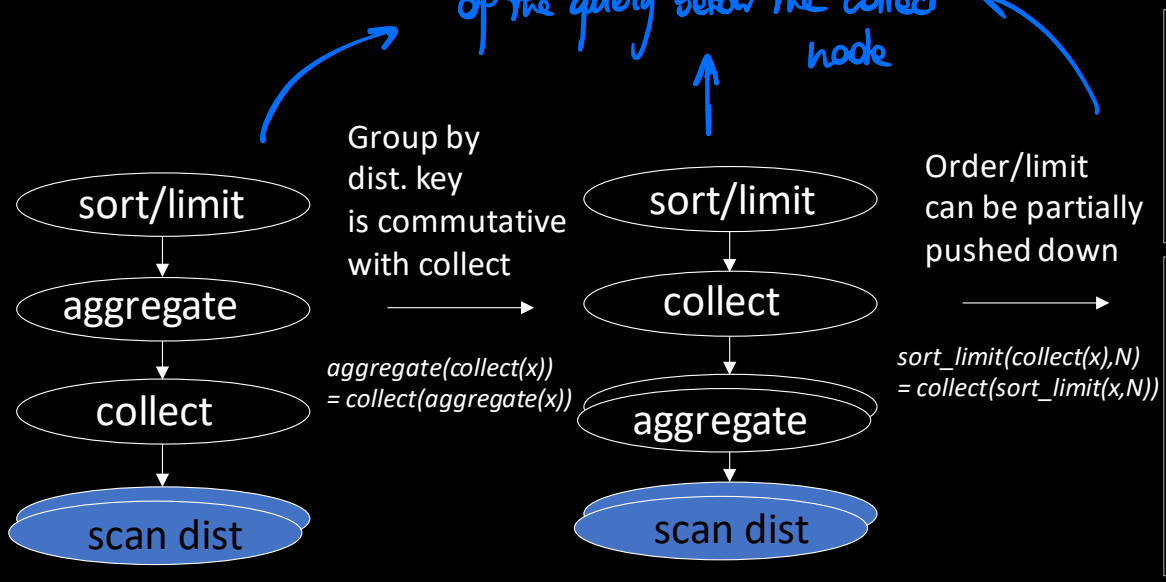
Distributed SQL: Logical optimization

as much parallelism as possible

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

all of them are math. correct but we want most of the query below the collect node

top 10 from every shard, i collect the top 10's and i get the overall top 10



Merge plan

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

Shard plan (can run in parallel)

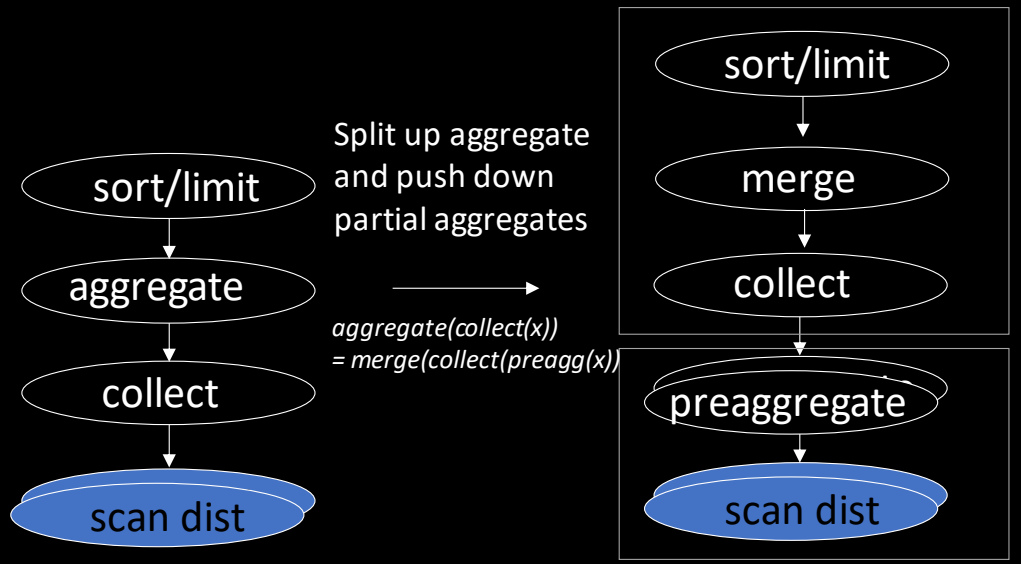
SELECT dist_key, count(*)
FROM dist_table_*
GROUP BY 1
ORDER BY 2 LIMIT 10;

Distributed SQL: Logical optimization

same value of other key may appear in many different shards

do not group by distribution key ⟹ collect and aggregate are not commutatives anymore

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



Merge plan

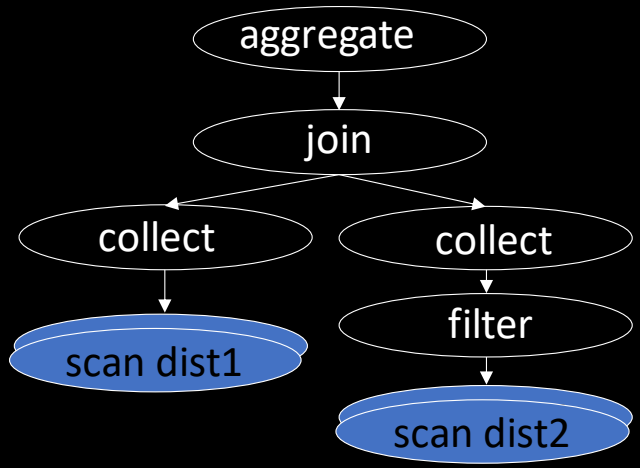
SELECT other_key, sum(count)
FROM <results>
GROUP BY other_key
ORDER BY 2 LIMIT 10;

Shard plan

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

Distributed SQL: Co-located joins

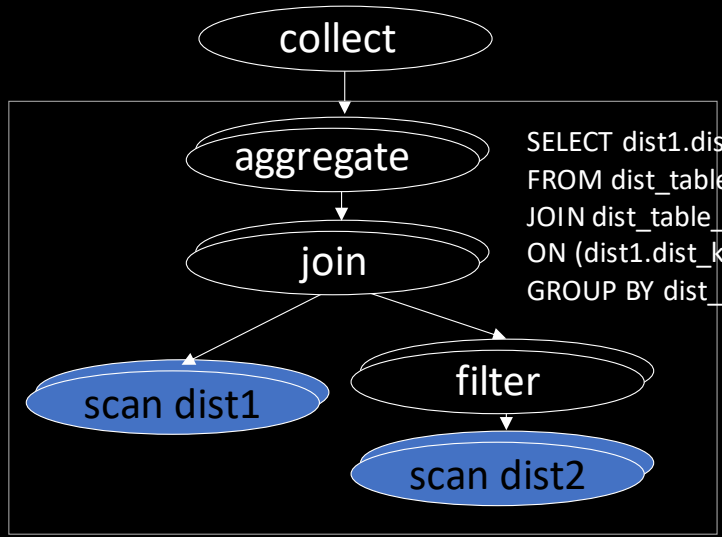
SELECT dist1.dist_key, count(*)
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.dist_key)
WHERE dist2.value < 44 GROUP BY dist1. dist_ key;



Filter is commutative with collect

Join is co-located so distributive with 2 collect nodes

Group by dist. key is commutative with collect

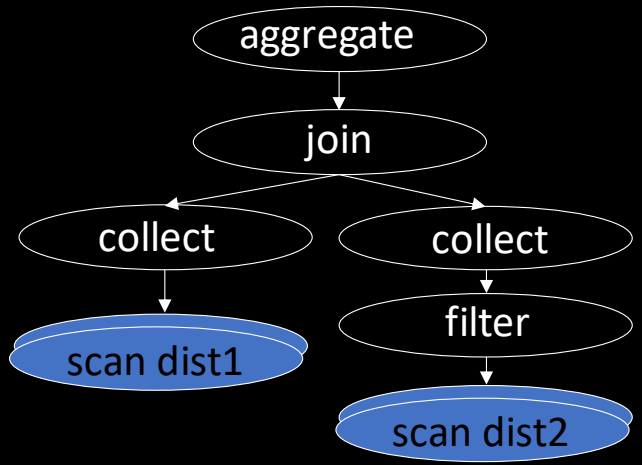


SELECT dist1.dist_key, count(*)
FROM dist_table_* dist1
JOIN dist_table_* dist2
ON (dist1.dist_key = dist.2.dist_key)
GROUP BY dist_key

Distributed SQL: Re-partition joins

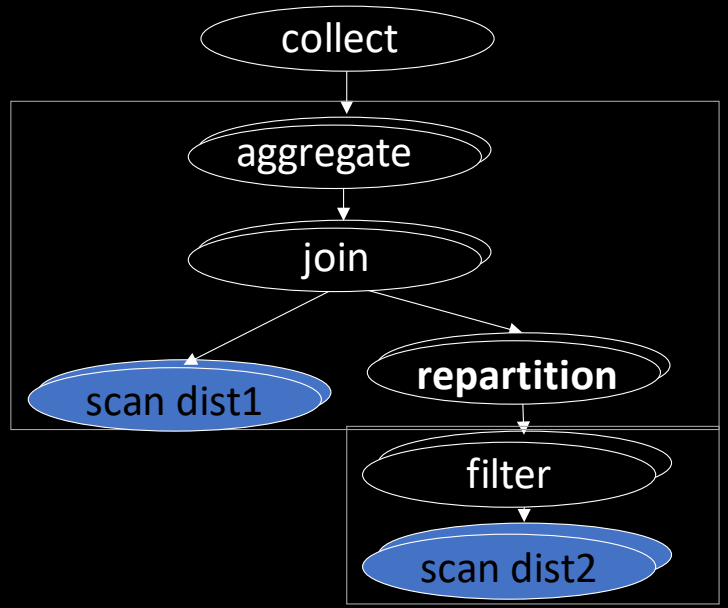
not joining on the distribution key anymore

```
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;
```



Need to re-partition data to perform join

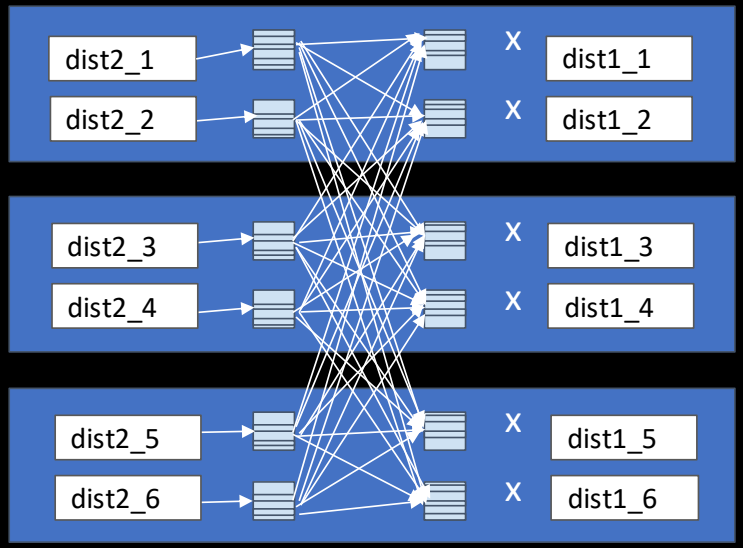
Group by dist. key is commutative with collect



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;
```

relatively expensive to shuffle between the different nodes



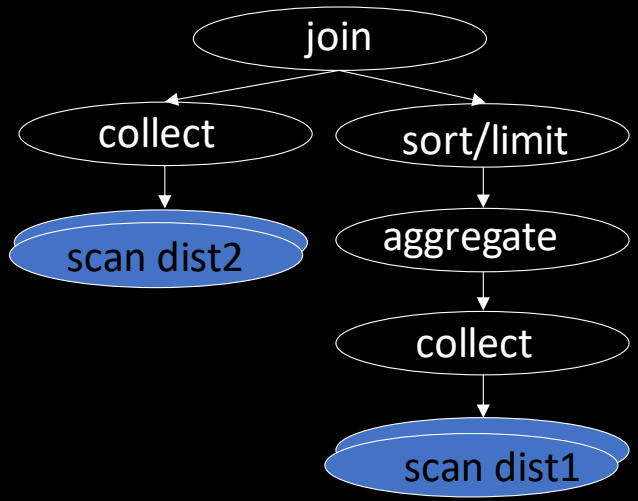

```
SELECT other_key
FROM dist2_*
WHERE value < 44;
```

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

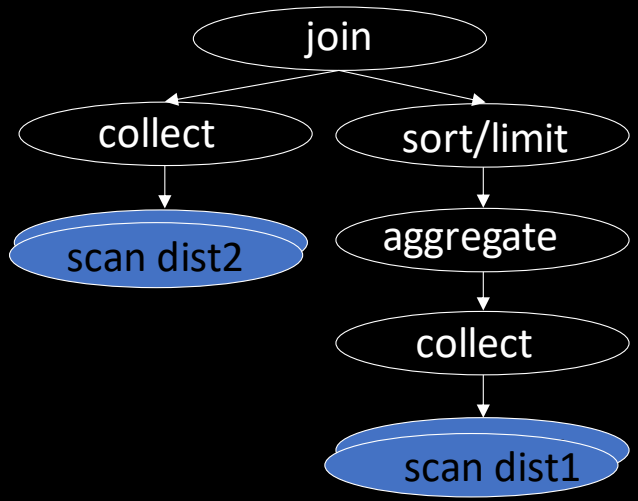
⇒ small change in the query can have quiet a lot of impact in the

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);
```

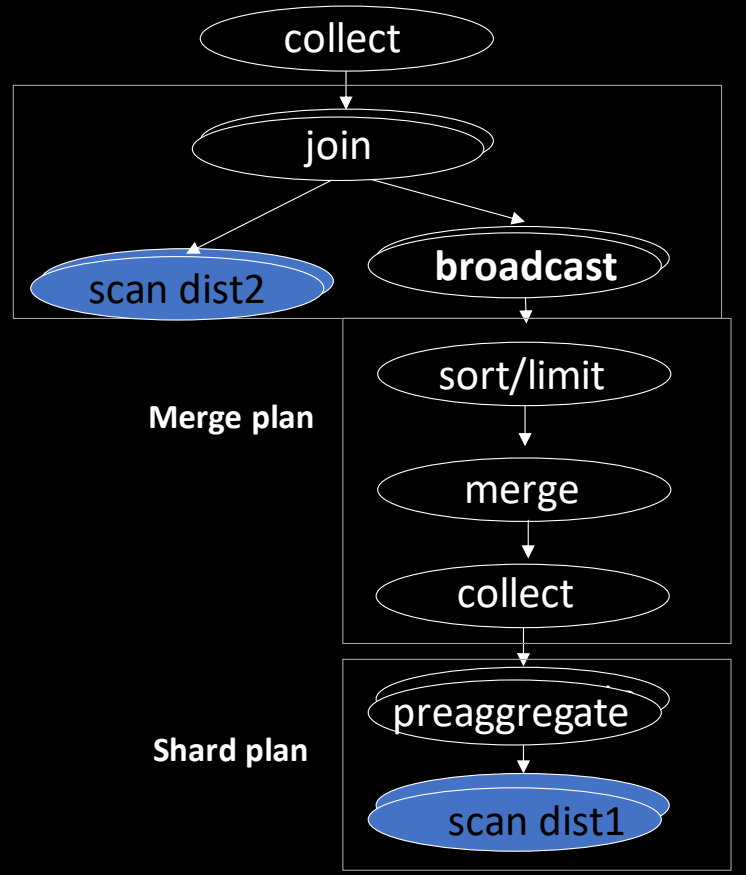


Distributed SQL: Broadcast joins



Create subplan to handle order/limit under join

Broadcast subplan to pull collect above the join



Distributed SQL: Observations

the choice is really important

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

REST OF THE LECTURE IS NOT PART OF THE EXAM SUBJECTS

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

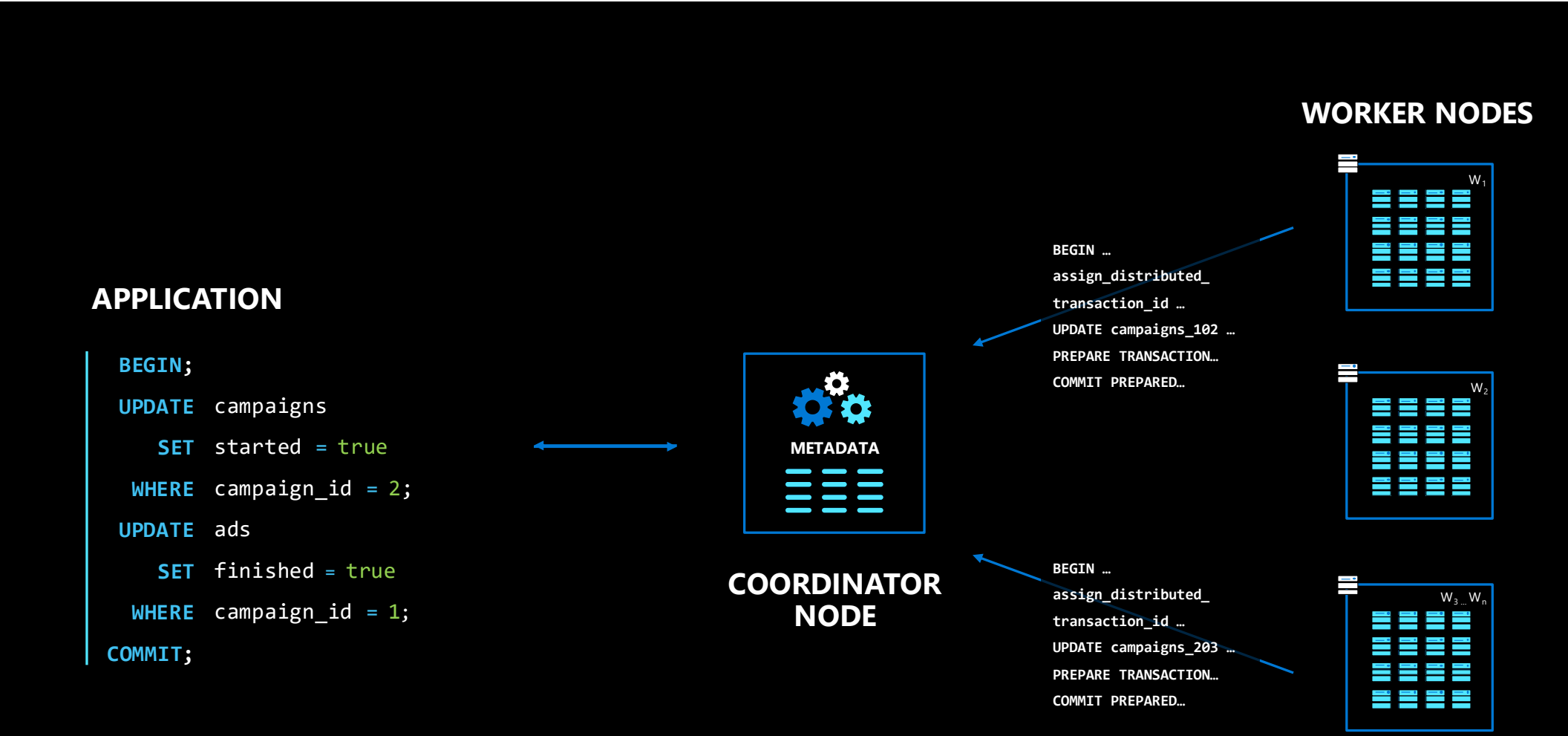
Additionally:

Distributed deadlock detection

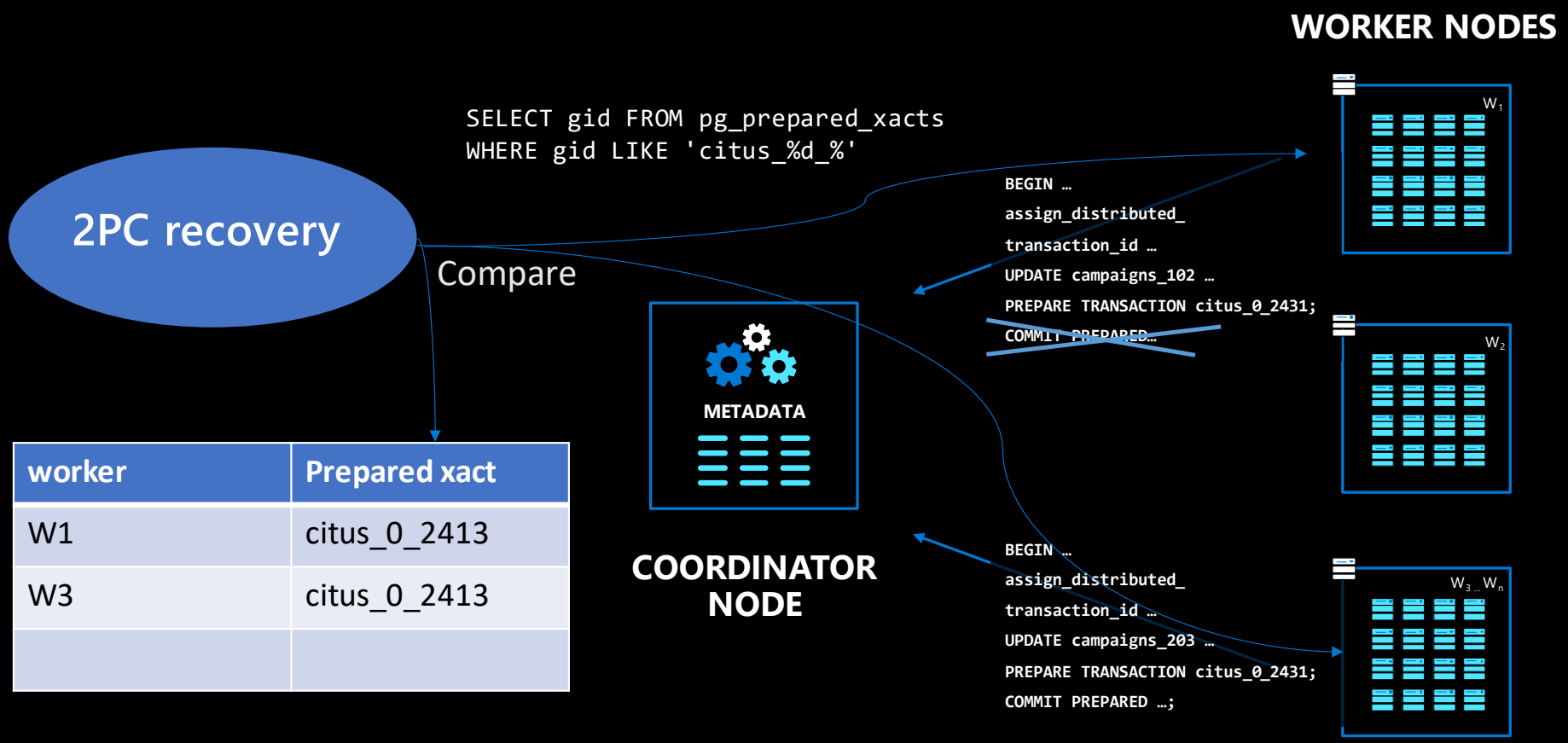
Distributed Transactions: Atomicity

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

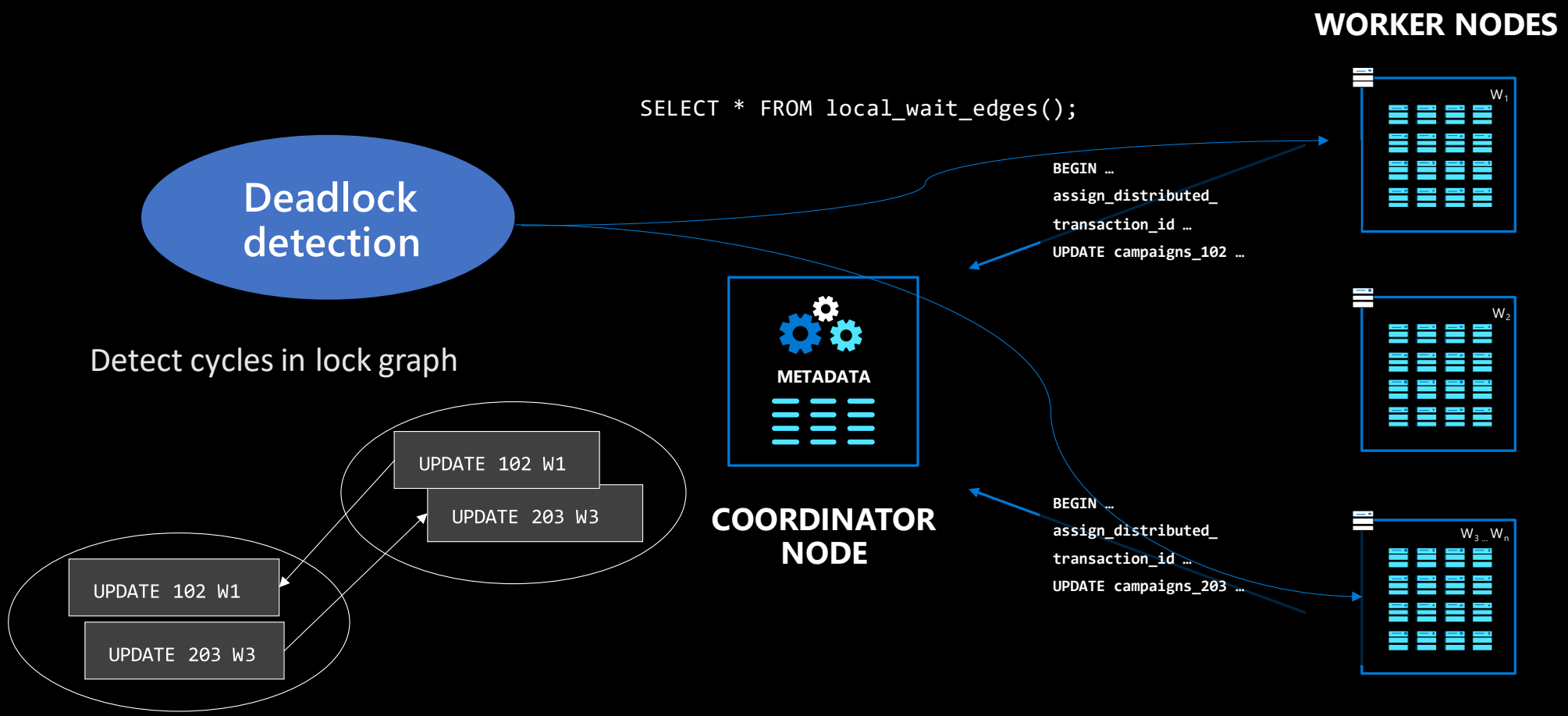
Phase 1:	Store (“prepare”) transactions on all nodes
Phase 2:	Store final commit decision and …
If success,	Commit all prepared transactions
If error,	Abort all prepared transactions
Secret phase 3:	Commit/abort prepared transactions after failure



How Citus distributes transactions in a multi-node cluster



worker	Prepared xact
W1	citus_0_2413
W3	citus_0_2413


WORKER NODES
SELECT * FROM local_wait_edges();
Deadlock detection
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_102 …
W1
W2
Detect cycles in lock graph
METADATA
UPDATE 102 W1
UPDATE 203 W3
COORDINATOR NODE
BEGIN …
assign_distributed_
transaction_id …
UPDATE campaigns_203 …
W3 .. Wn
UPDATE 102 W1
UPDATE 203 W3

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.

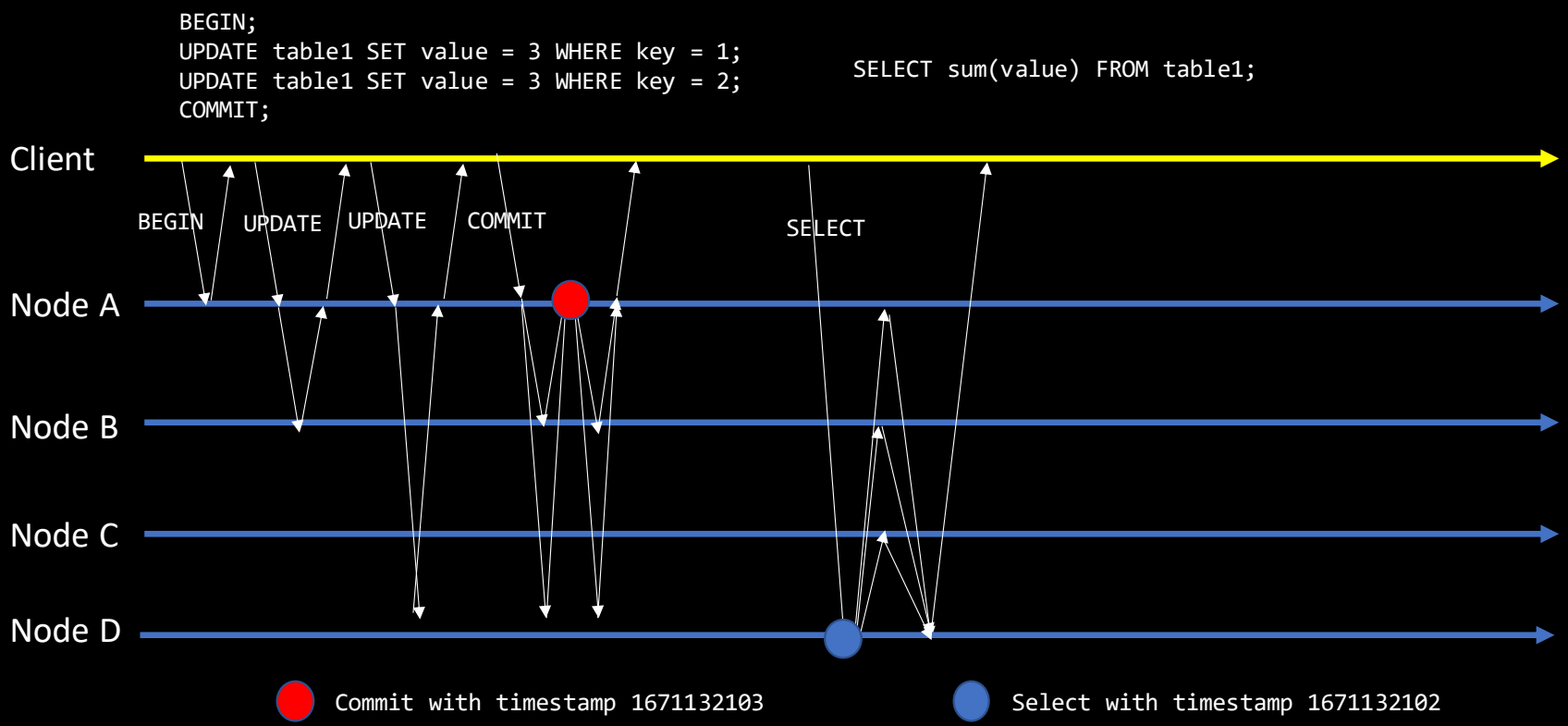
Distributed snapshot isolation means we have the same of view of what is committed and not committed on all the nodes.

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

Common approach:

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

Distributed snapshot isolation



Distributed Transactions: Isolation

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

Distributed transactions: Considerations

Fully resolving a 2PC might take time in case of failure.

Distributed deadlock detection is essential to stability, but not always implemented.

Snapshot isolation avoids seeing partially committed transactions, but at a cost, and read-your-writes consistency can be at risk.

Replication

Trade-offs all the way down

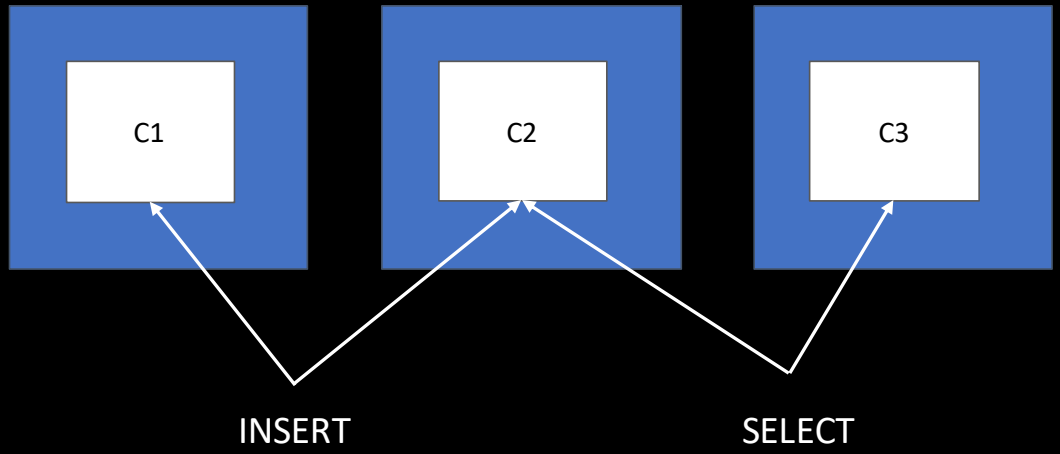
Replication

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

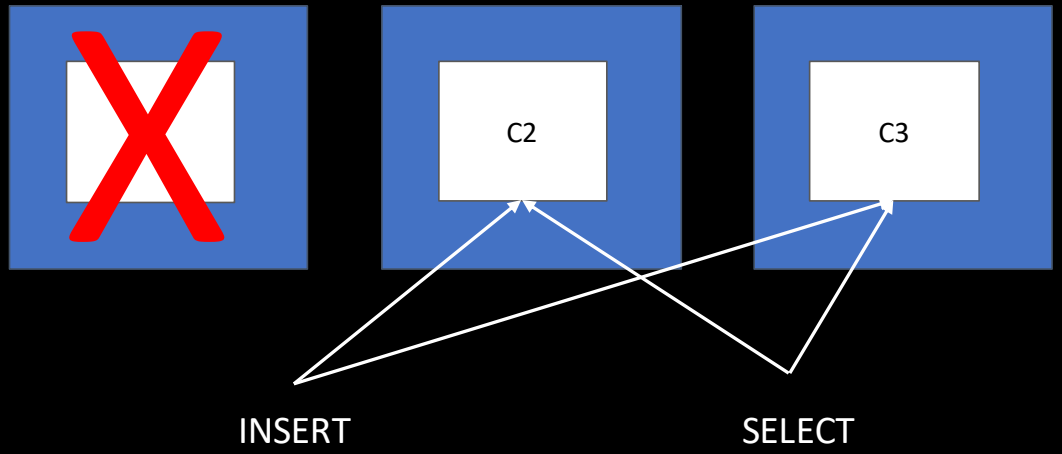
Basic idea: Read from R nodes, Write to W nodes, R +W > N



Challenge: Applying events in same order everywhere

Replication: Quorums

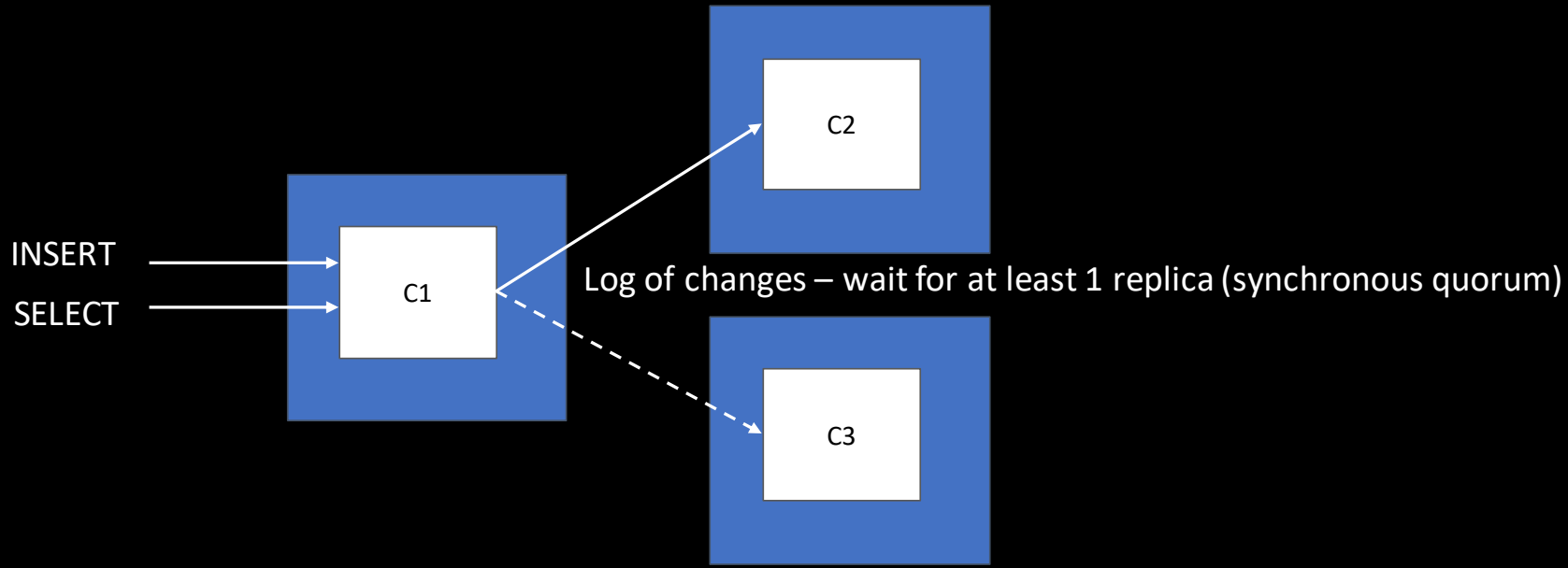
Basic idea: Read from R nodes, Write to W nodes, R +W > N



Challenge: Applying events in same order everywhere

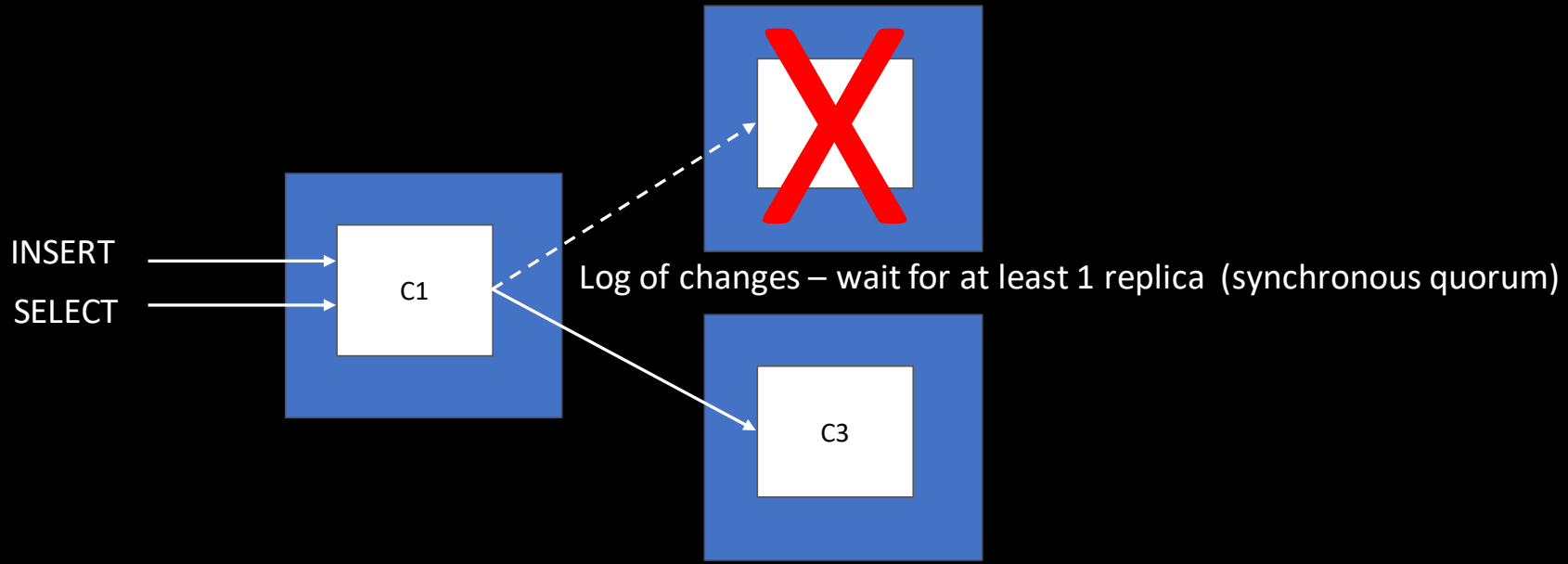
Replication: Follow the leader

Assign temporary leader to serialize writes efficiently



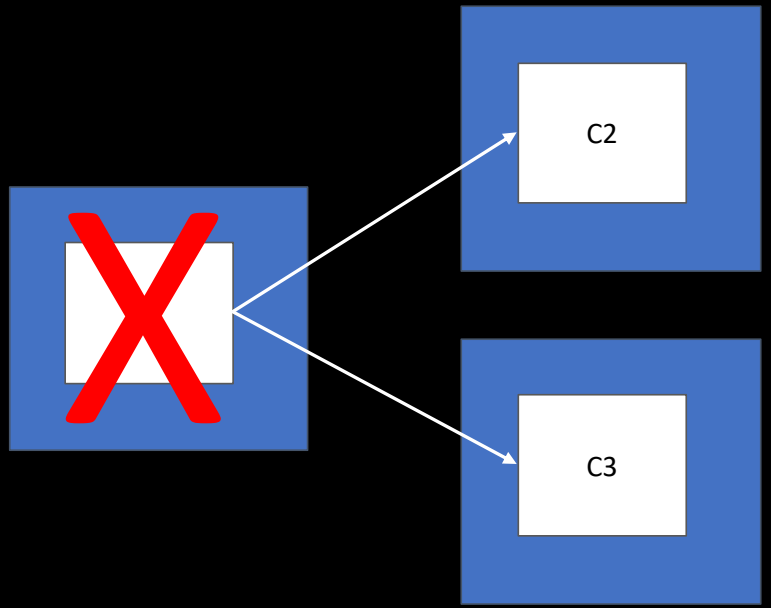
Replication: Follow the leader

Standby fails: Continue writing to other replica



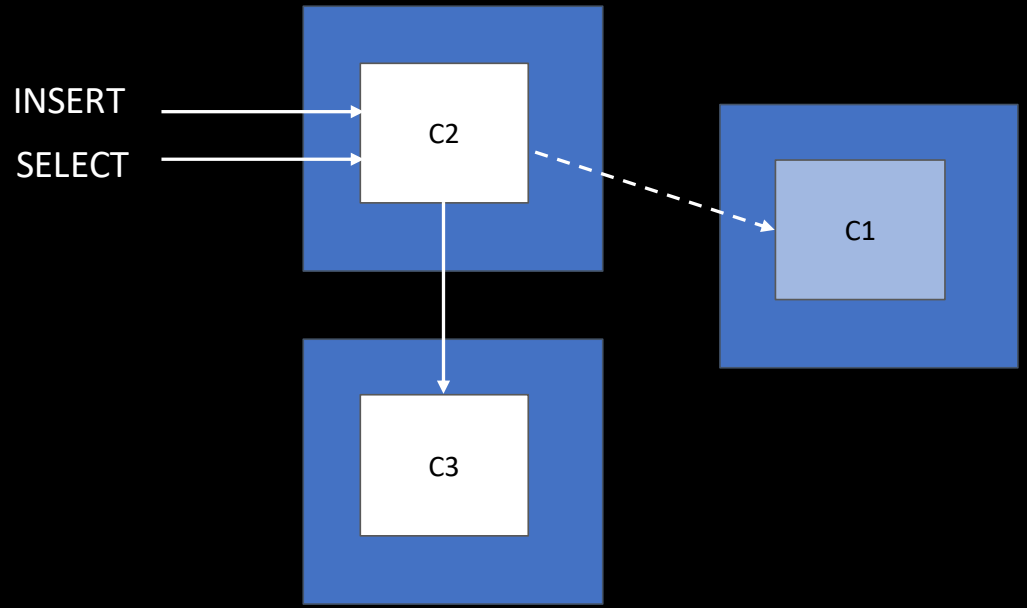
Replication: Follow the leader

Primary fails: Initiate a failover



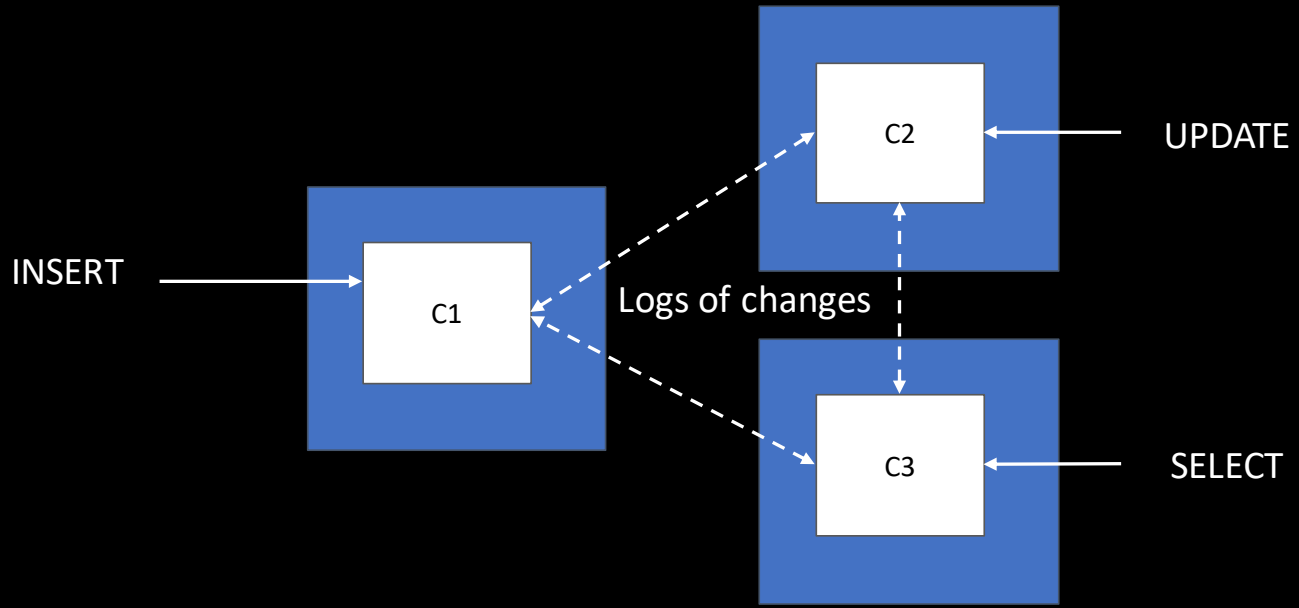
Replication: Follow the leader

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



Replication: N-directional

All nodes accept writes, somehow reconcile conflicting changes.



Replication

Things you could get using replication:

~99.999% read availability if using geo-replication without read-your-writes consistency.

~99.999% write availability if using geo-replication with active-active.

~99.95% read/write availability with consistency by assigning a new primary.

~500k reads/sec per (large) replica

CAP theorem

Choose **C**onsistency vs **A**vailability when in a (minority) network **P**artition

Availability (AP) = Keep writing to minority of nodes, majority does not see it

Consistency (CP) = Writes/reads unavailable, consistency must be preserved

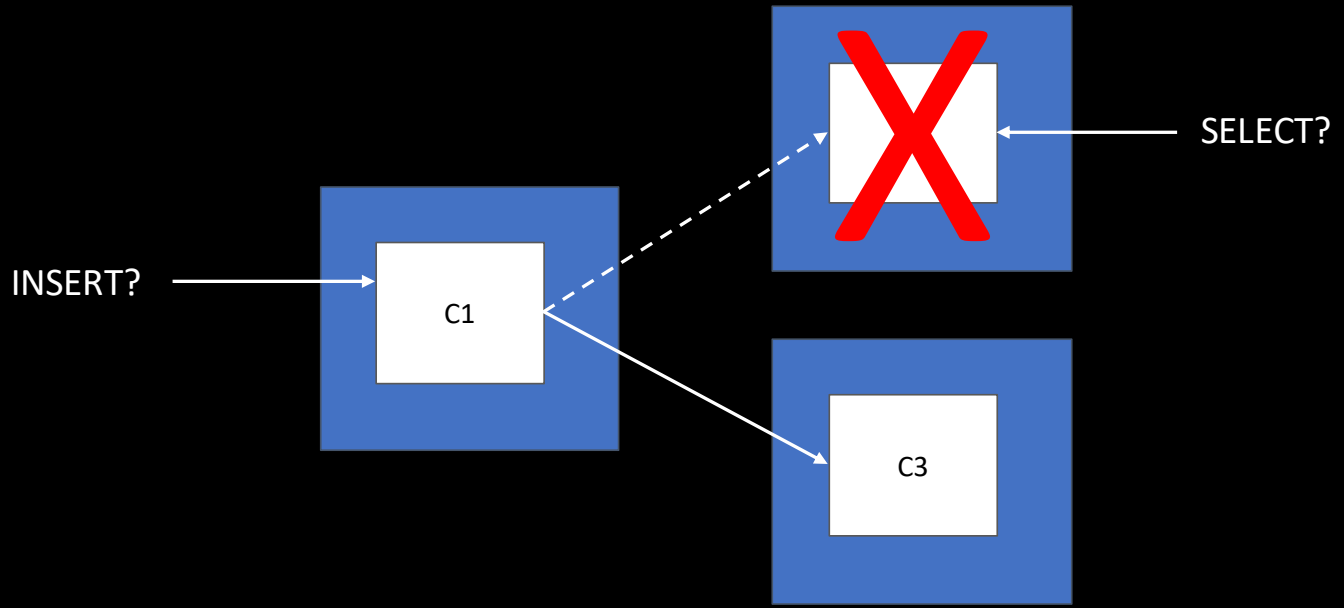
Incomplete picture of distributed database trade-offs

PACELC theorem

Slightly better, but still oversimplified:

If Network **P**artition: choose **A**vailability vs. **C**onsistency
Else: choose **L**atency vs. **C**onsistency

Replication: Follow the leader



Other distributed database trade-offs

Consistency
- Read-your-writes
- No lost updates
- Linearizability

Availability
- For Reads
- For Writes
- Handle availability zone failure

Partition-tolerance
- For Reads
- For Writes

Durability
- Node failure does not result in data loss
- Writes are archived in a timely manner

Low latency
- Low read latency
- Low write latency
- Global vs. local

Complexity
- Dependencies on other systems
- Multiple node types
- Many optimizations

Distributed database gotchas

Watch out for:

- Latency means response times are always higher than on a single node
- Throughput of synchronous protocols is bounded by #sessions/avg. response time
- Higher response times => longer lock waits => lower concurrency
- Evaluating relationships (foreign keys, joins, indexes, aggregation, ..) requires many sequential data accesses => higher response times
- Most benchmarks are lies

Distributed database tips

General advice:

- Do not expect magic
- Keep densely connected data (or <100GB) on a single node
- Understand your access pattern (what causes the load?)
- Pick your distribution keys carefully
- Use batching to reduce latency-sensitivity
- Be creative with database features

Questions?

marco.slot@microsoft.com