Parallel and Distributed Databases

Parallel vs. Distributed

Parallel DBMSs:
→ Nodes are physically close to each other.
→ Nodes connected with high-speed LAN.
→ Communication cost is assumed to be small.

Distributed DBMSs:
→ Nodes can be far from each other.
→ Nodes connected using public network.
→ Communication cost and problems cannot be ignored.

Parallel and Distributed DBMS

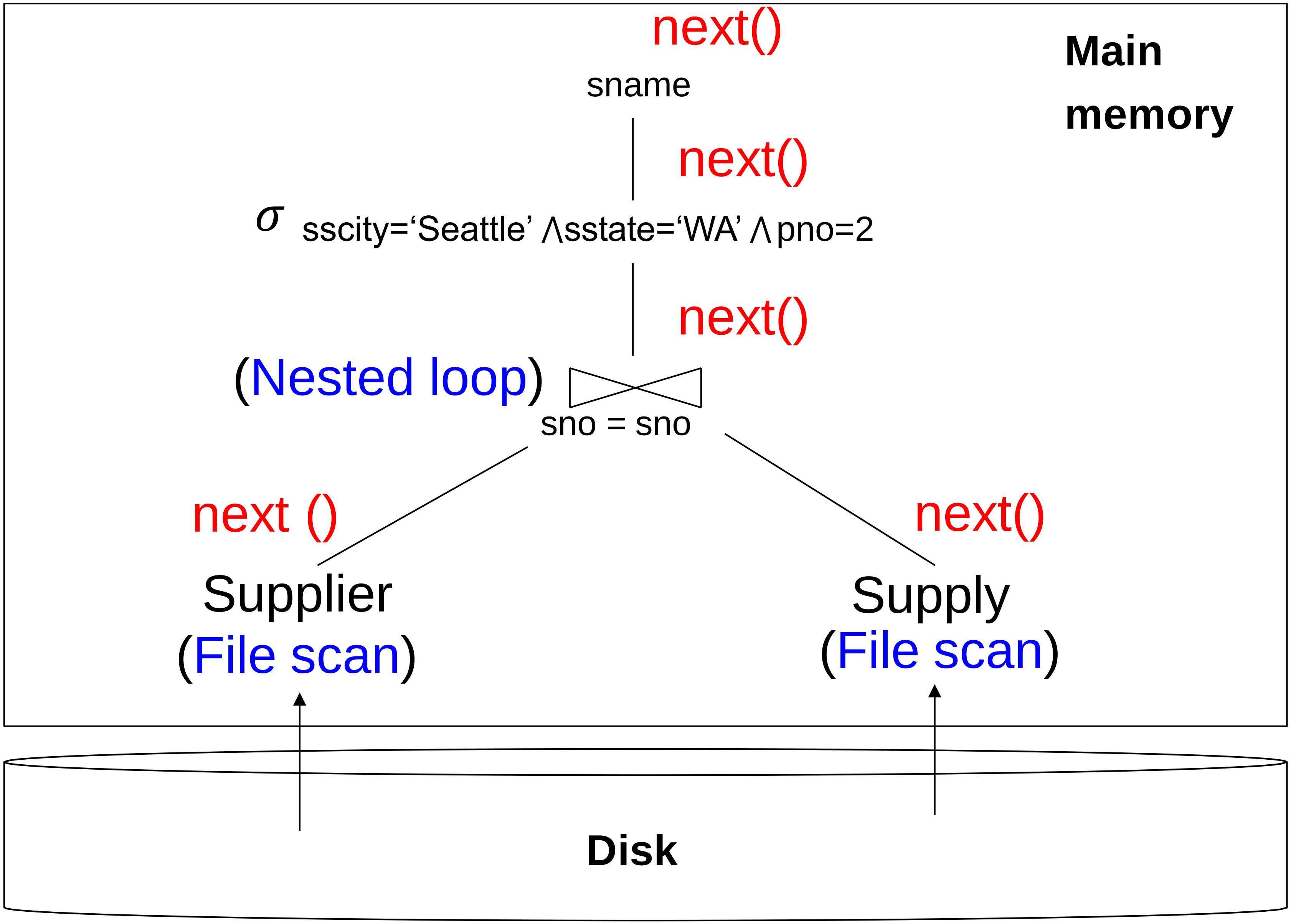
Use the building blocks that we covered in single- node DBMSs to now support transaction processing and query execution in parallel and distributed environments.

→ Optimization & Planning

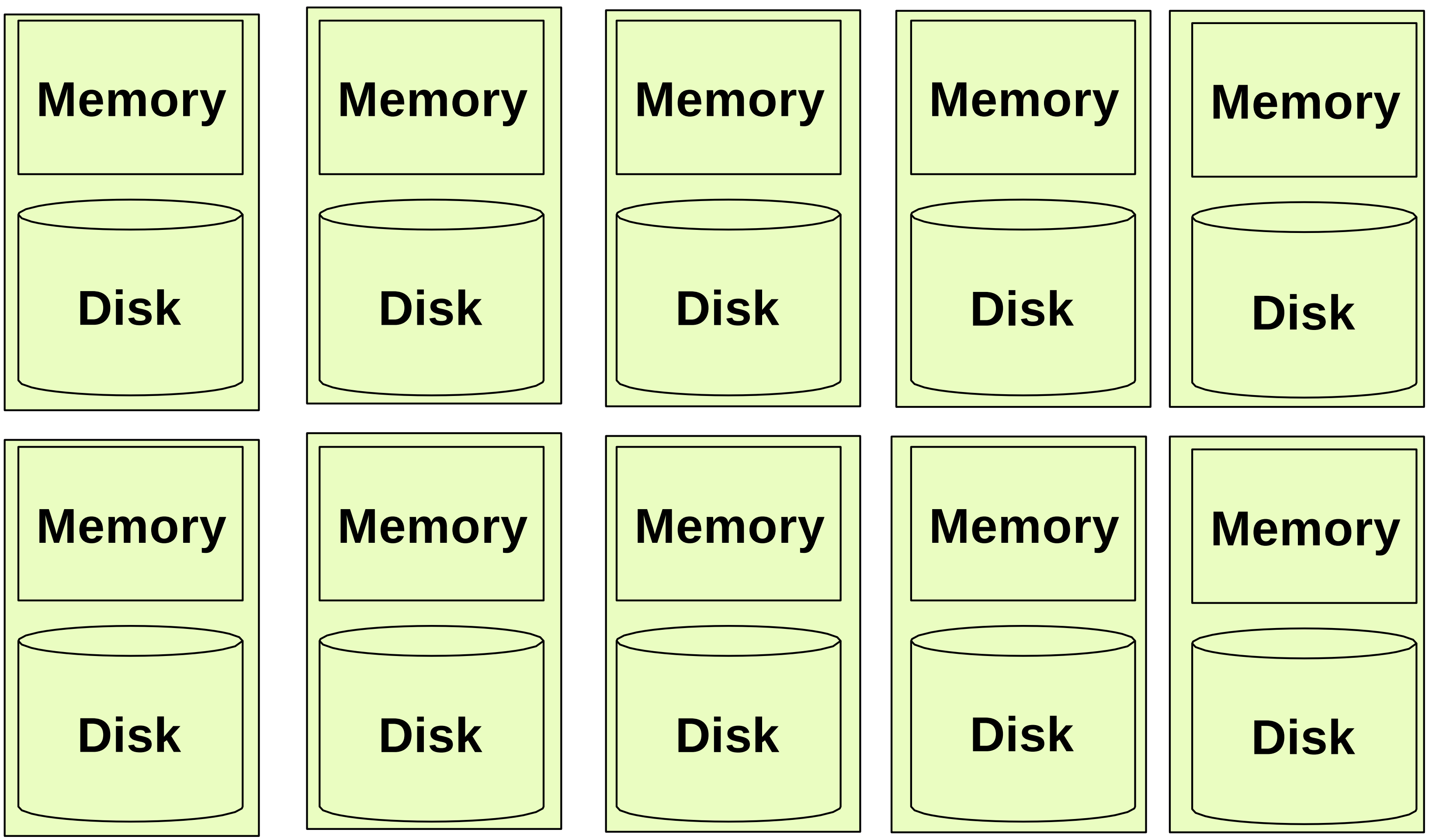
→ Concurrency Control

→ Logging & Recovery

Serial Query Execution



What if we Have a Cluster and a Large Amount of Data?



Serial Query Execution Algorithms

Basic query processing on one node (one node = one process)

- Serial execution: One machine with one process and one thread

Given relations R(A,B) and S(B, C), no indexes, how do we compute:

- Selection: $\sigma_{A=123}(R)$
  - Scan file R, select records with A=123
- Group-by: $\gamma_{A,sum(B)}(R)$
  - Scan file R, insert into a hash table using attribute A as key
  - When a new key is equal to an existing one, add B to the value
- Join: $R \bowtie S$
  - Scan file S, insert into a hash table using join attribute as key
  - Scan file R, probe the hash table using join attribute to look up matches

Parallel Query Execution Algorithms?

How do we compute these operations on a shared-nothing parallel db?

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

Before we answer that: how do we store R (and S) on a shared-nothing parallel db?

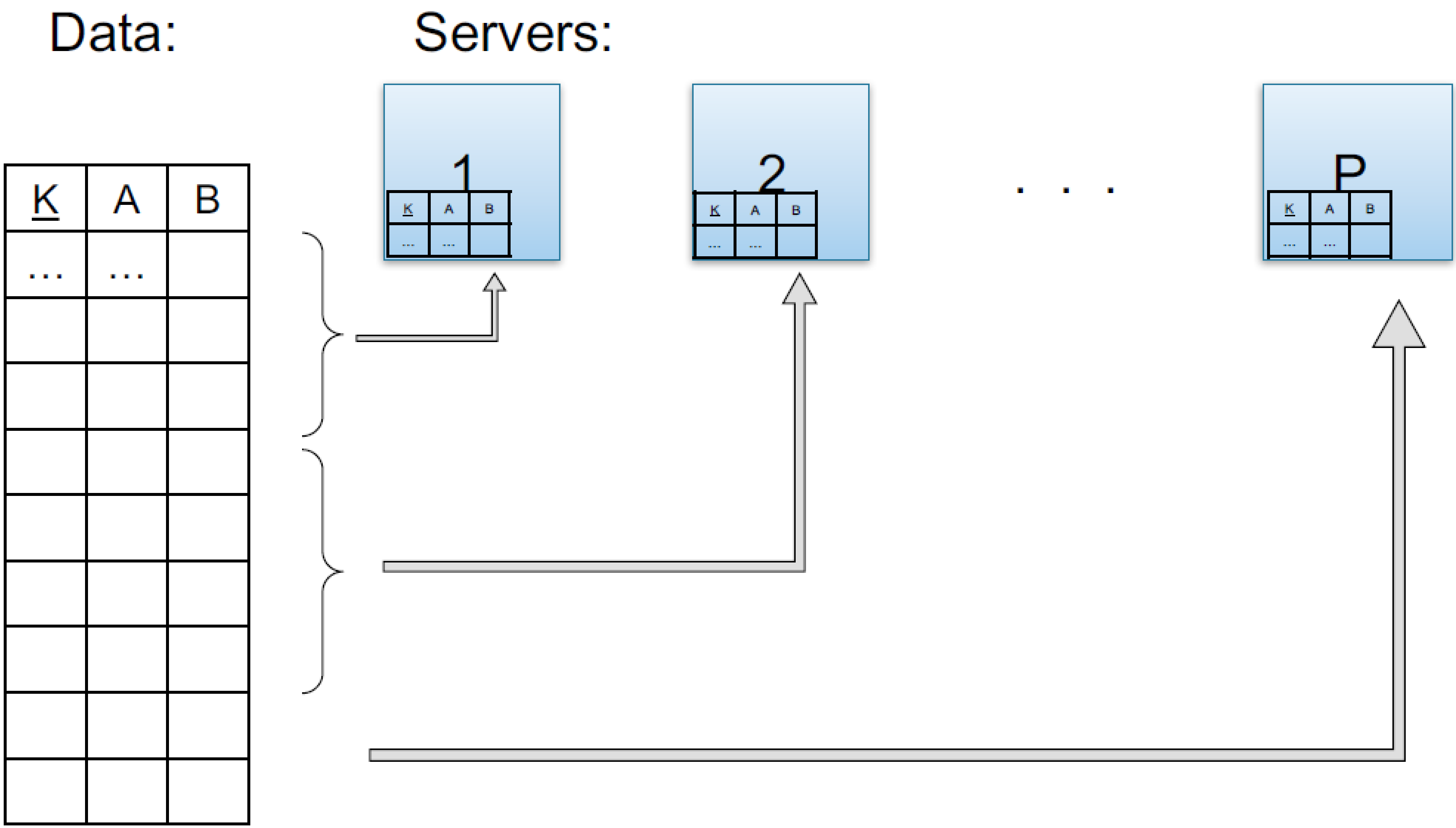
Horizontal Data Partitioning

Data:

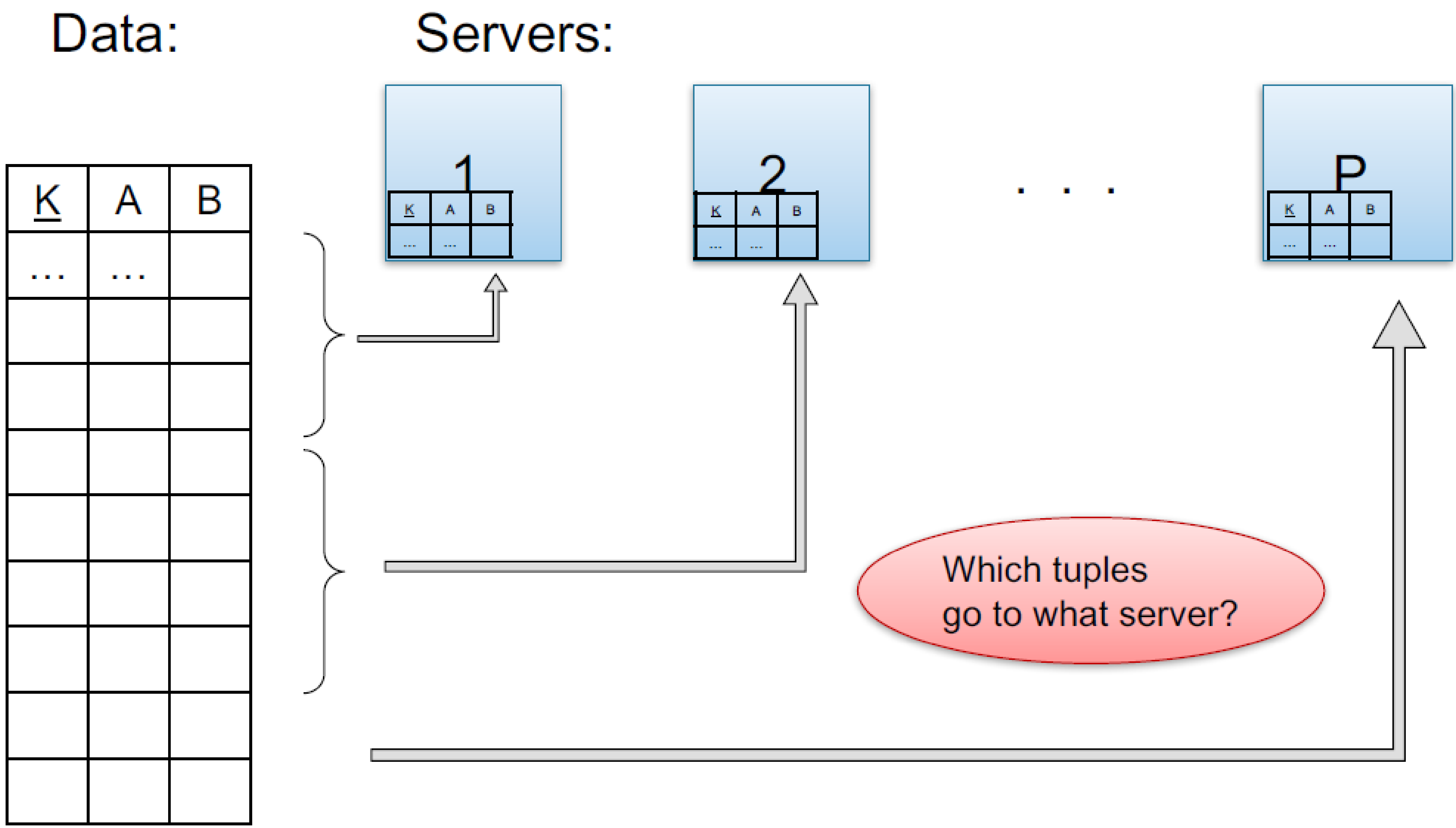
K	A	B
...	...	

Servers:

Horizontal Data Partitioning



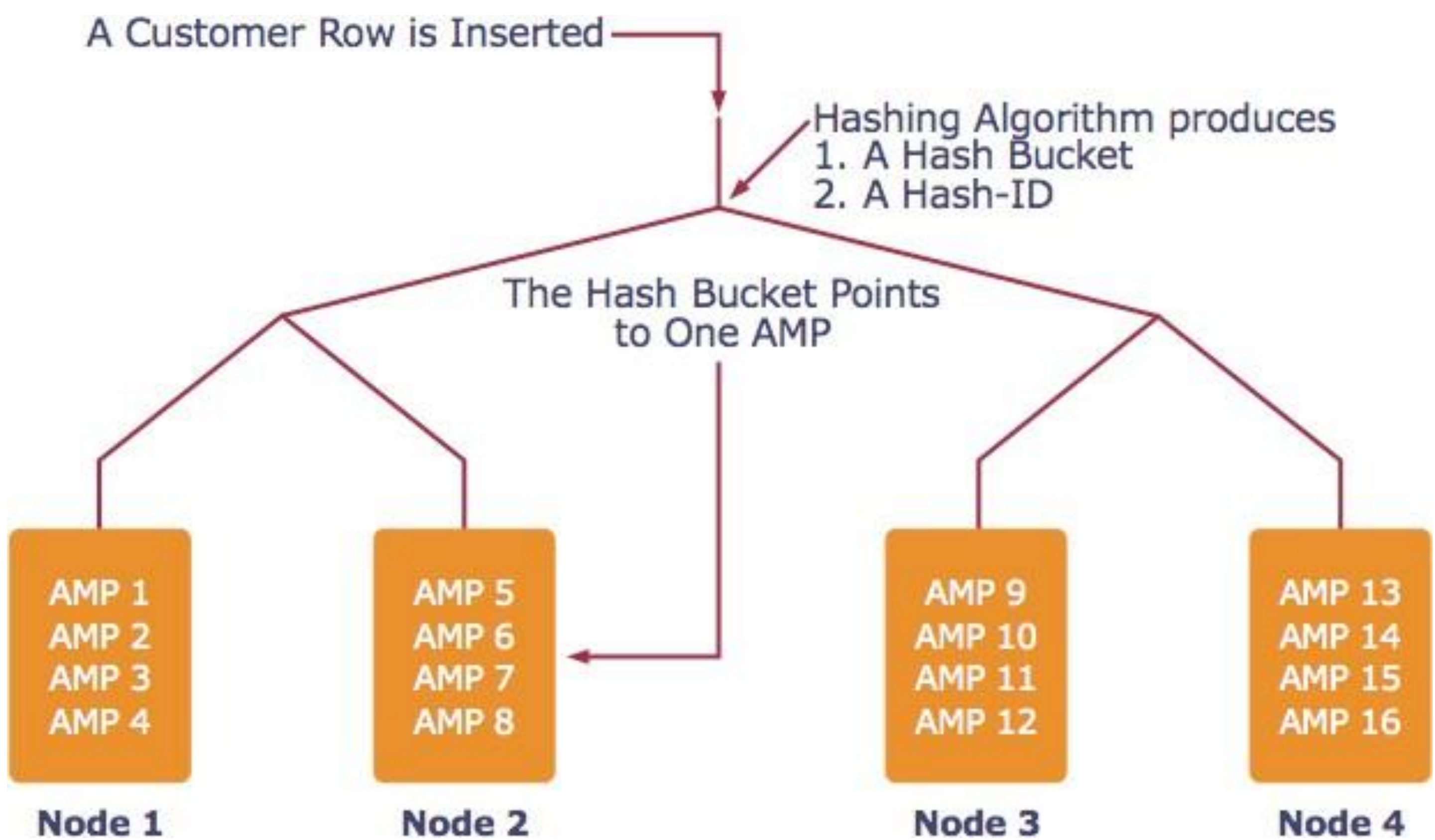
Horizontal Data Partitioning



Horizontal Data Partitioning

- Block Partition:
  - Partition tuples arbitrarily s.t. $size(R_1) \approx \ldots \approx size(R_P)$
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where $i = h(t.A) \bmod P + 1$
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Ingesting Data – Teradata Example



AMP = "Access Module Processor" = unit of parallelism

Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
  - All servers must scan and filter the data
- Hash partitioned:
  - Can have all servers scan and filter the data
  - Or can optimize and only have some servers do work
- Range partitioned
  - Also only some servers need to do the work

Parallel GroupBy

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

Discuss in class how to compute in each case:

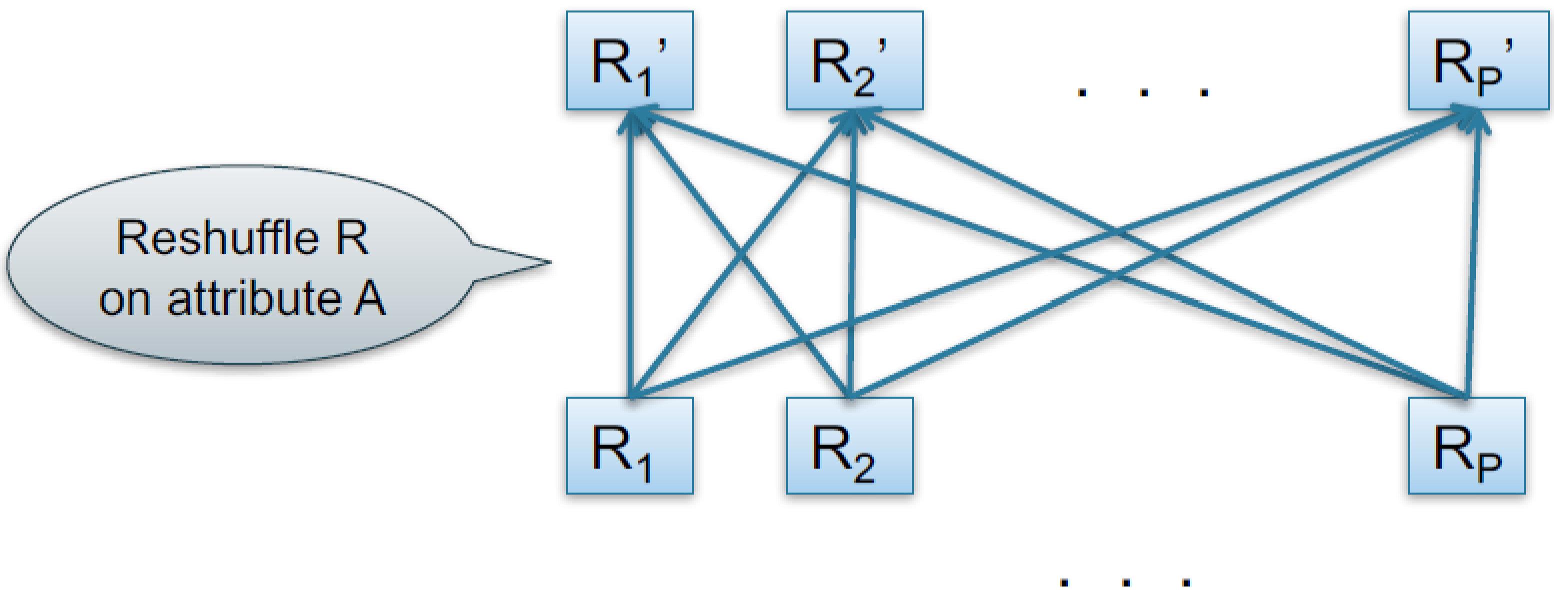
- R is hash-partitioned on A
- R is block-partitioned
- R is hash-partitioned on K

Basic Parallel GroupBy

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K



Basic Parallel GroupBy

- Step 1: each server i partitions tuples in its chunk R_i using a hash function h(t.A) mod P: $R_{i,0}, R_{i,1}, \ldots, R_{i,P-1}$

- Step 2: server j computes $\gamma_{A,\ sum(B)}$ on $R_{0,j}, R_{1,j}, \ldots, R_{P-1,j}$

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?

- If we double both P and the size of R, what is the new running time?

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?
  - Half (each server holds ½ as many chunks)
- If we double both P and the size of R, what is the new running time?
  - Same (each server holds the same # of chunks)

Basic Parallel GroupBy

Can we do better?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Basic Parallel GroupBy

Can we do better?

- Sum?
- Count?
- Avg?
- Max?

•Median?

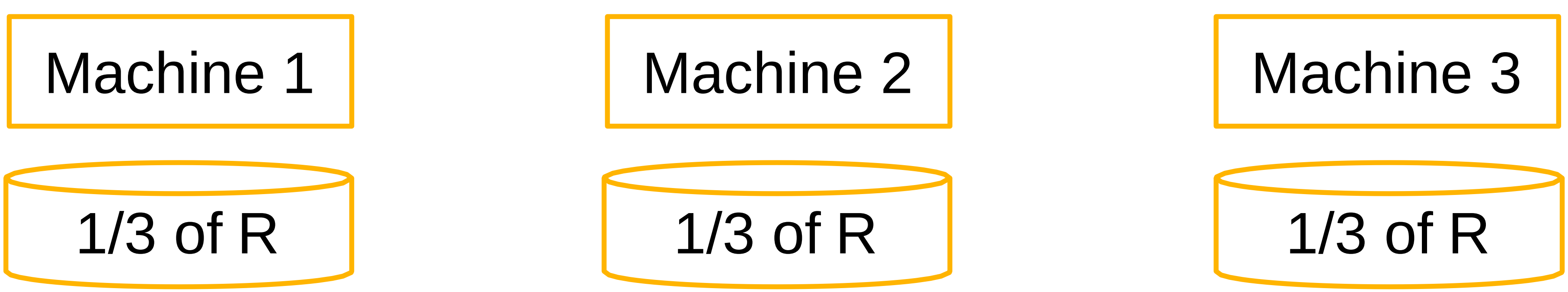
YES

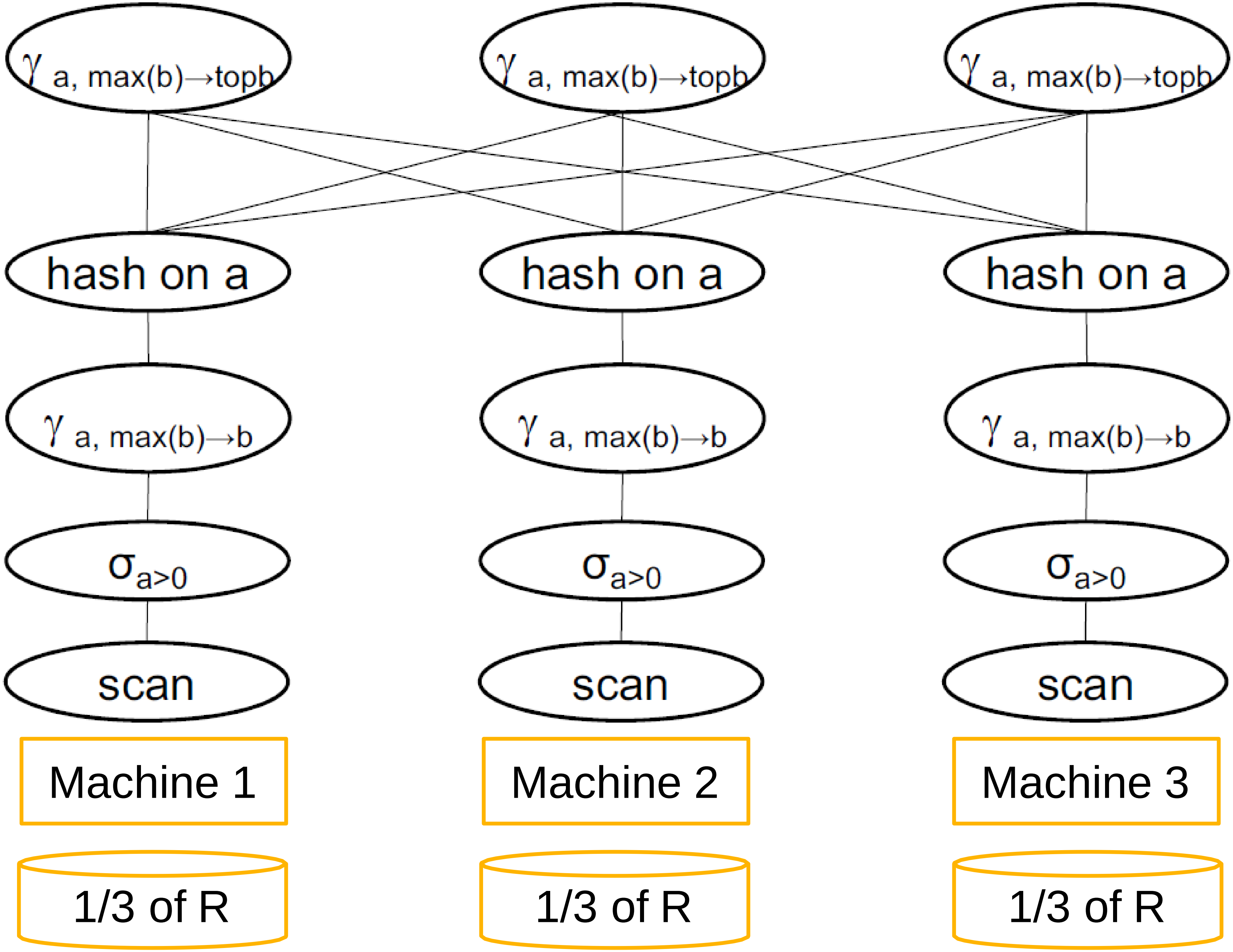
- Compute partial aggregates before shuffling

Distributive	Algebraic	Holistic
$sum(a_1+a_2+\ldots+a_9)=$ $sum(sum(a_1+a_2+a_3)+$ $sum(a_4+a_5+a_6)+$ $sum(a_7+a_8+a_9))$	avg(B) = sum(B)/count(B)	median(B)

Example Query with Group By

```
SELECT a, max(b) as topb
FROM R WHERE a > 0
GROUP BY a
```





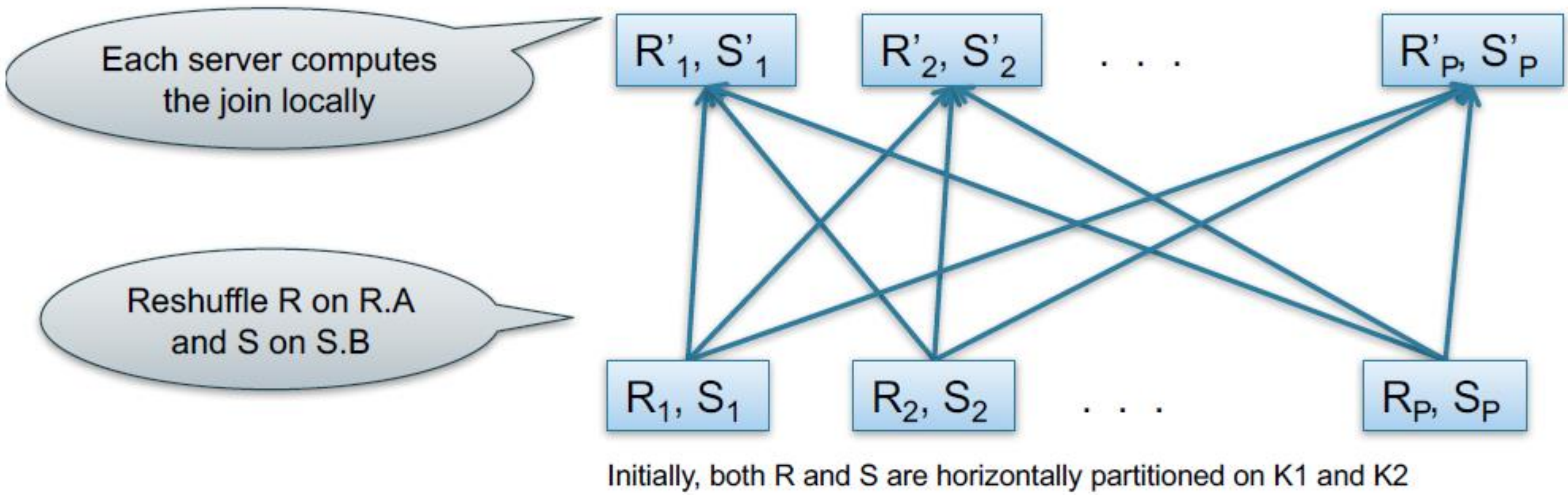
γ a, max(b)→topb
γ a, max(b)→topb
γ a, max(b)→topb
hash on a
hash on a
hash on a
γ a, max(b)→b
γ a, max(b)→b
γ a, max(b)→b
σa>0
σa>0
σa>0
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R

Parallel Join: R $\bowtie_{A=B}$ S

- Data: R(<u>K1</u>,A, C), S(<u>K2</u>, B, D)
- Query: R(<u>K1</u>,A,C) $\bowtie$ S(<u>K2</u>,B,D)

Parallel Join: R ⋈$_{A=B}$ S

- Data: R(<u>K1</u>,A, C), S(<u>K2</u>, B, D)
- Query: R(<u>K1</u>,A,C) ⋈ S(<u>K2</u>,B,D)



Parallel Join: $R \bowtie_{A=B} S$

- Step 1
  - Every server holding any chunk of R partitions its chunk using a hash function h(t.A) mod P
  - Every server holding any chunk of S partitions its chunk using a hash function h(t.B) mod P
- Step 2:
  - Each server computes the join of its local fragment of R with its local fragment of S

Data: R(K1,A, B), S(K2, B, C)

Query: R(K1,A,B) $\bowtie$ S(K2,B,C)

Join on R.B = S.B

Local Join

M1: R1' $\bowtie$ S1'

R1'

K1	B
1	20
3	20
4	20

S1'

K2	B
201	20

M2: R2' $\bowtie$ S2'

R2'

K1	B
2	50

S2'

K2	B
101	50
102	50
202	50

Shuffle

Partition

M1

R1

K1	B
1	20
2	50

S1

K2	B
101	50
102	50

M2

R2

K1	B
3	20
4	20

S2

K2	B
201	20
202	50

Optimization for Small Relations

When joining R and S

- If |R| >> |S|
  - Leave R where it is
  - Replicate entire S relation across nodes

- Also called a small join or a broadcast join

Other Interesting Parallel Join Implementation

Skew:

- Some partitions get more input tuples than others
  Reasons:
  - Range-partition instead of hash
  - Some values are very popular:
    - Heavy hitters values
  - Selection before join with different selectivities

- Some partitions generate more output tuples than others

Some Skew Handling Techniques

If using range partition:

- Ensure each range gets same number of tuples
- E.g.: {1, 1, 1, 2, 3, 4, 5, 6 } → [1,2] and [3,6]
- Eq-depth v.s. eq-width histograms

Some Skew Handling Techniques

Create more partitions than nodes

- And be smart about scheduling the partitions

- Note: MapReduce uses this technique
  - We will talk about MapReduce later

Some Skew Handling Techniques

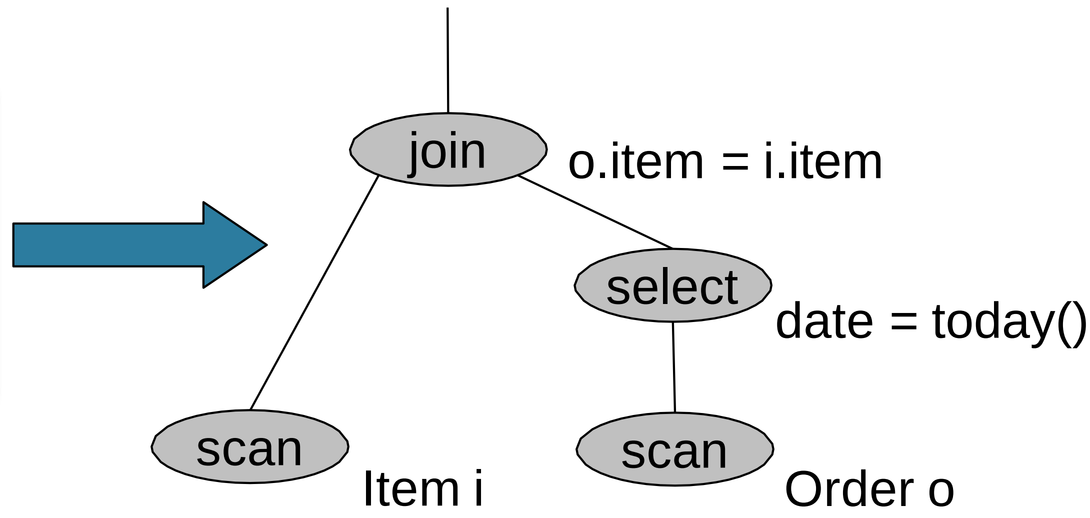
Use subset-replicate (a.k.a. “skewedJoin”)

- Given $R \bowtie_{A=B} S$
- Given a heavy hitter value R.A = ‘v’
  (i.e. ‘v’ occurs very many times in R)
- Partition R tuples with value ‘v’ across all nodes
  e.g. block-partition, or hash on other attributes
- Replicate S tuples with value ‘v’ to all nodes
- R = the build relation
- S = the probe relation

Example Query Execution

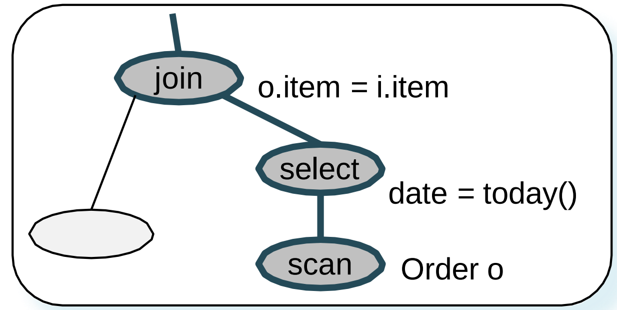
Find all orders from today, along with the items ordered

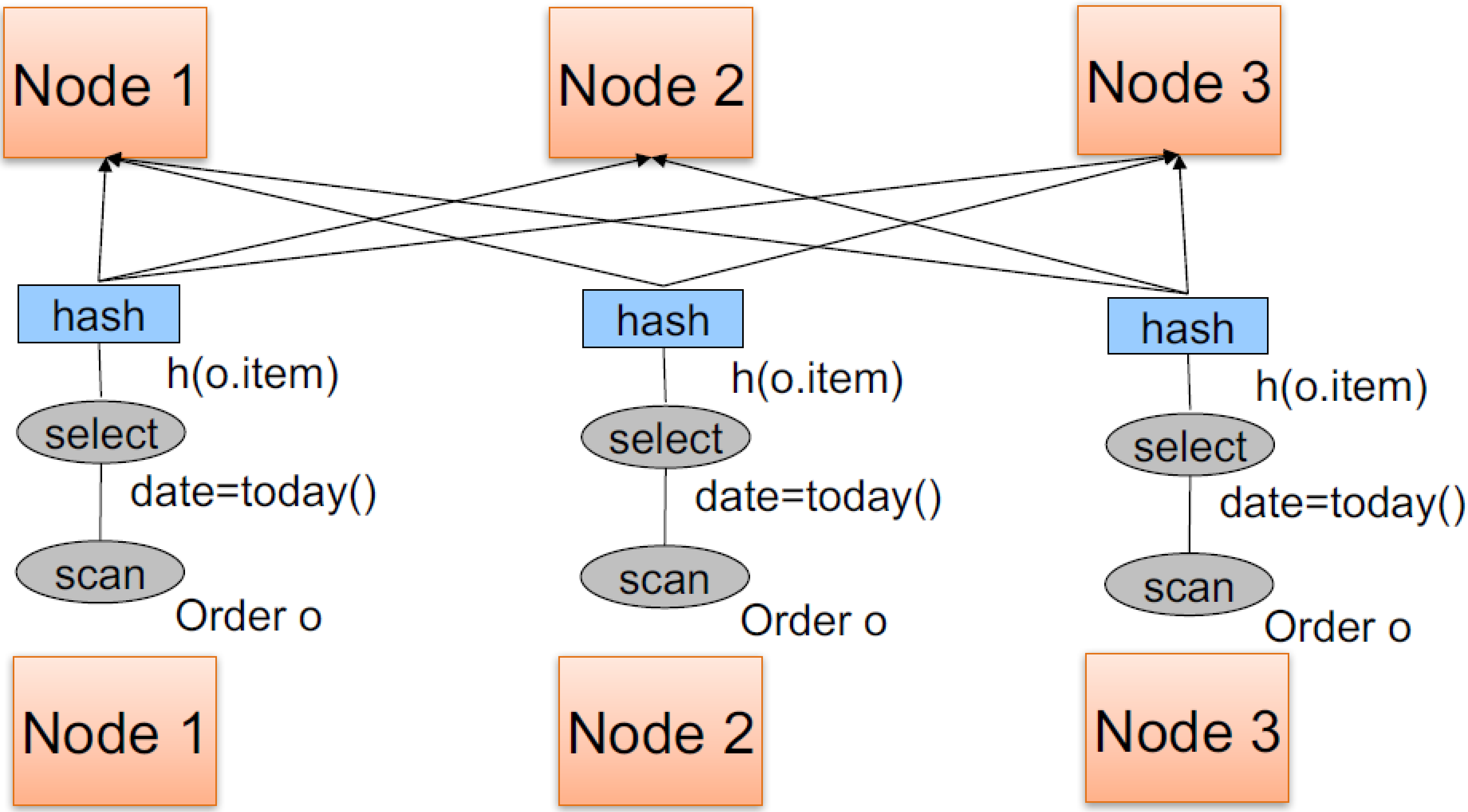
```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

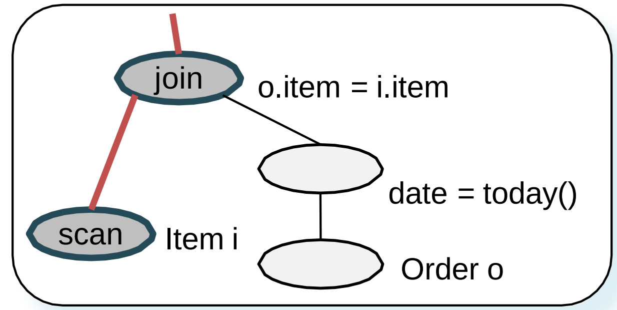
Query Execution

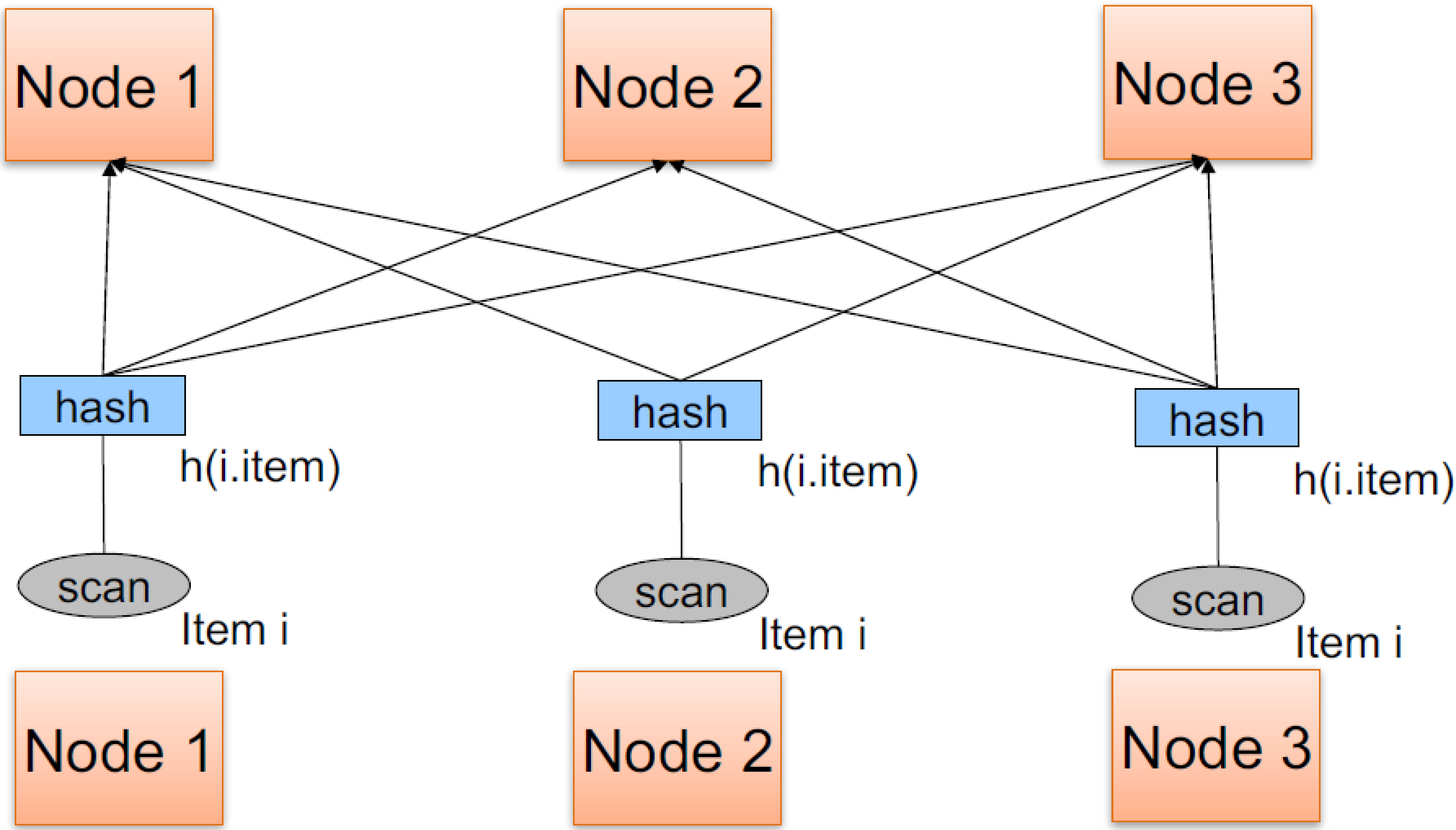




Order(oid, item, date), Line(item, …)

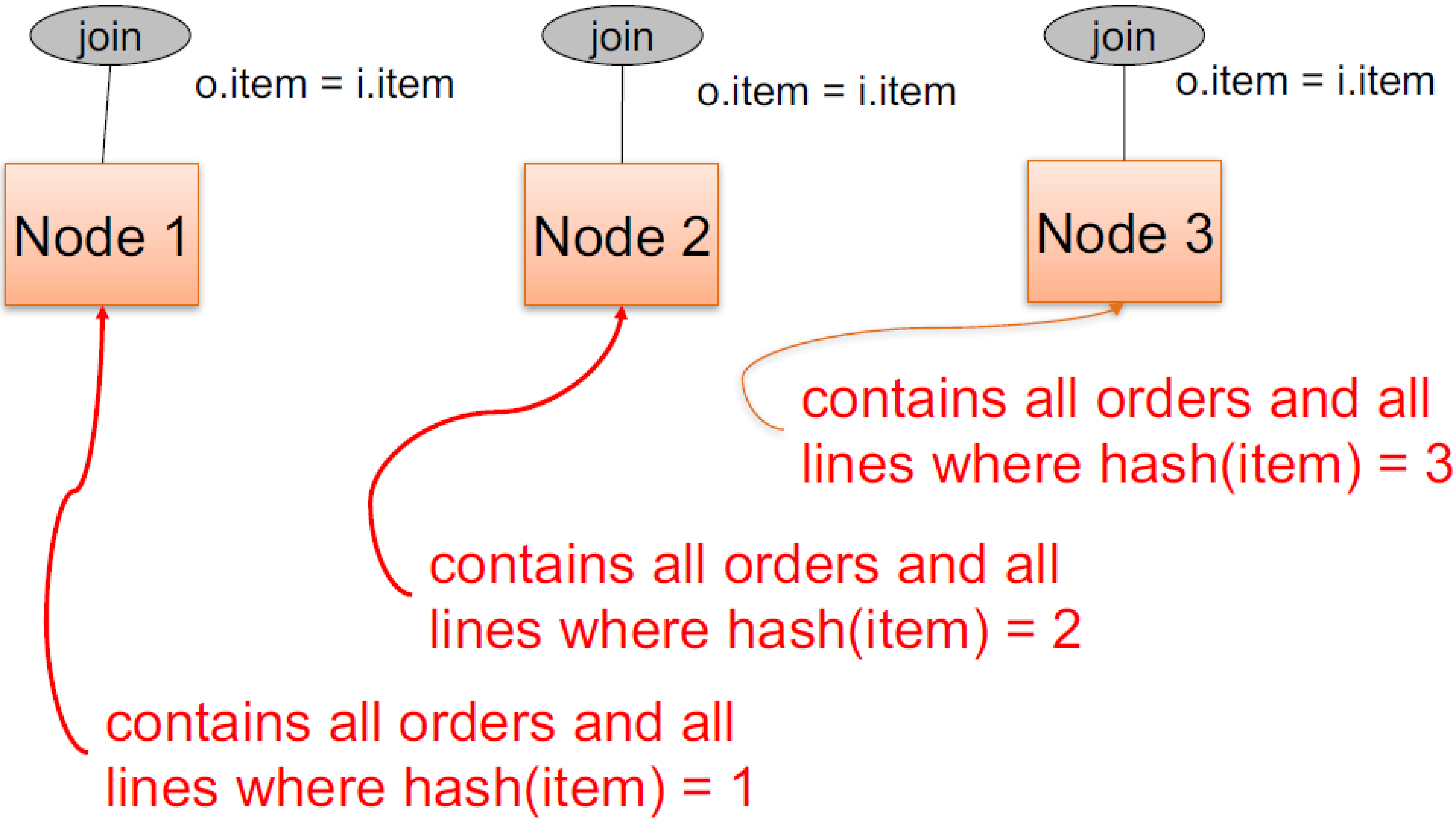
Query Execution





Order(oid, item, date), Line(item, …)

Query Execution



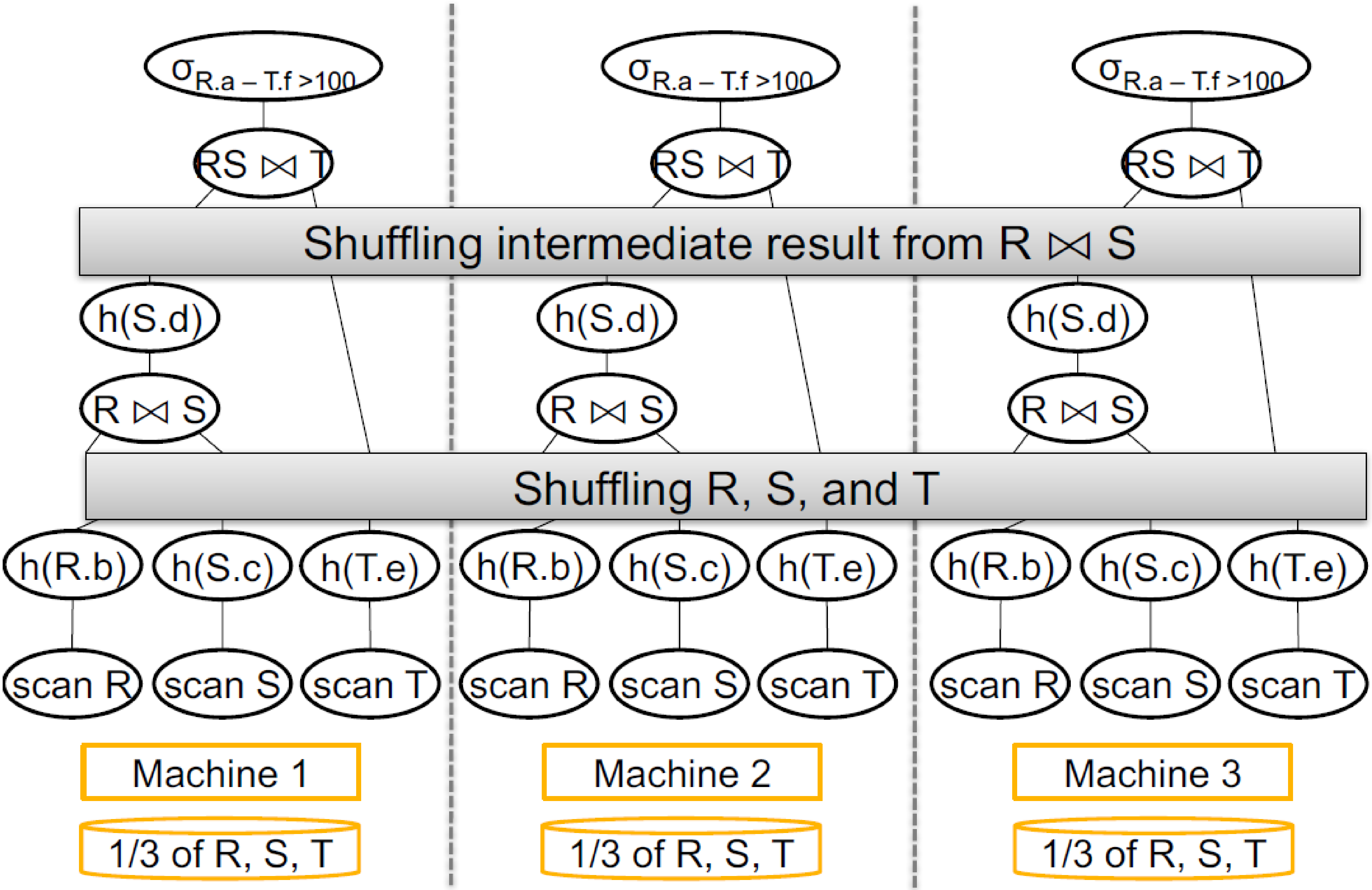
Example 2

SELECT *

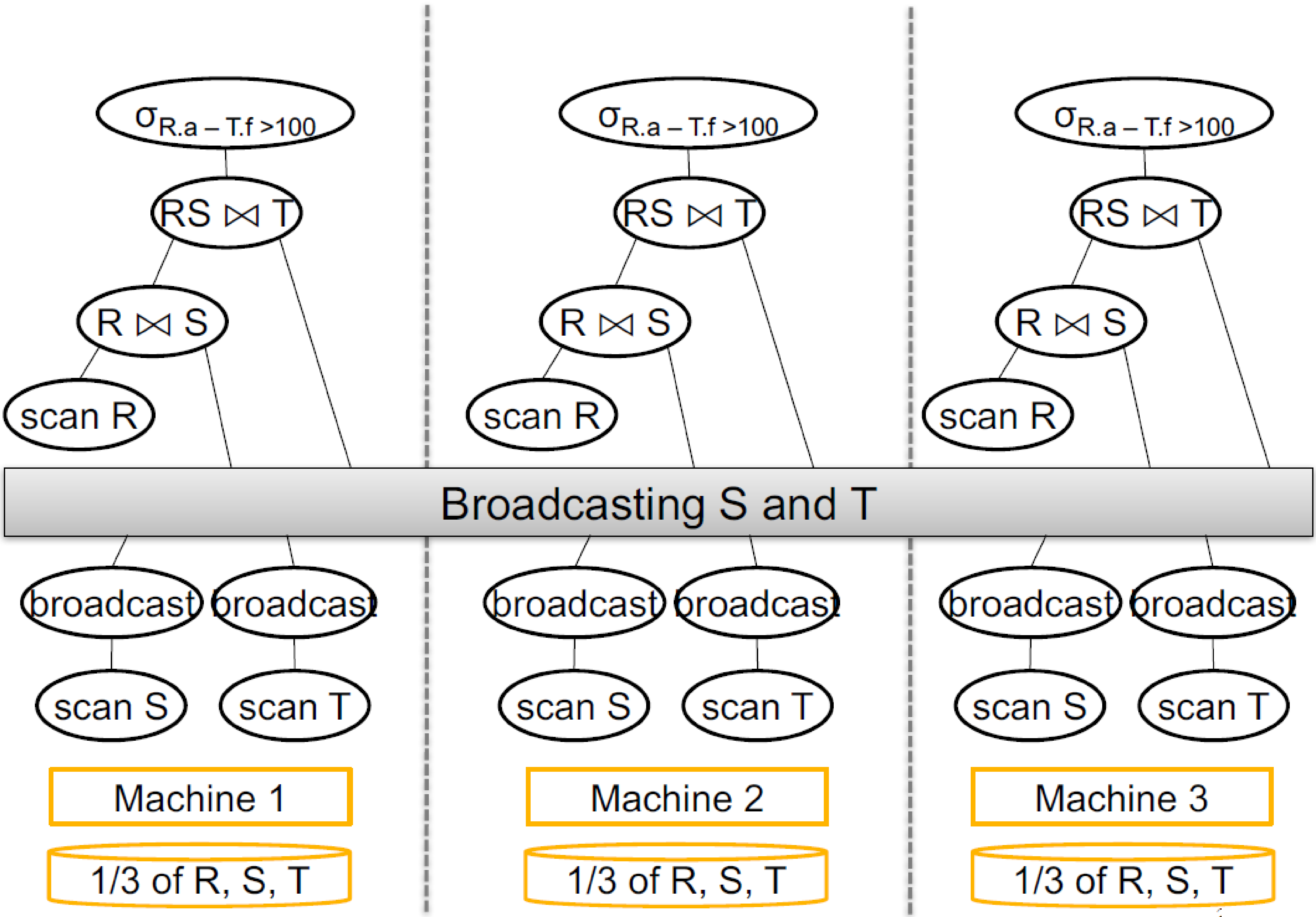
FROM R, S, T

WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100

Machine 1	Machine 2	Machine 3
1/3 of R, S, T	1/3 of R, S, T	1/3 of R,S, T



$\sigma_{R.a - T.f > 100}$
RS ⋈ T
Shuffling intermediate result from R ⋈ S
h(S.d)
R ⋈ S
Shuffling R, S, and T
h(R.b)
h(S.c)
h(T.e)
scan R
scan S
scan T
Machine 1
Machine 2
Machine 3
1/3 of R, S, T



$\sigma_{R.a - T.f > 100}$
RS ⋈ T
R ⋈ S
scan R
Broadcasting S and T
broadcast
broadcast
scan S
scan T
Machine 1
1/3 of R, S, T
Machine 2
1/3 of R, S, T
Machine 3
1/3 of R, S, T

Atomic Commitment

Informally: either all participants commit a transaction, or none do

“participants” = partitions involved in a given transaction

So, What’s Hard?

All the problems of consensus…

…plus, if *any* node votes to *abort*, all must decide to *abort*

» In consensus, simply need agreement on “some” value

Two-Phase Commit

Canonical protocol for atomic commitment (developed 1976-1978)

Basis for most fancier protocols

Widely used in practice

Use a transaction *coordinator*

» Usually client – not always!

Two Phase Commit (2PC)

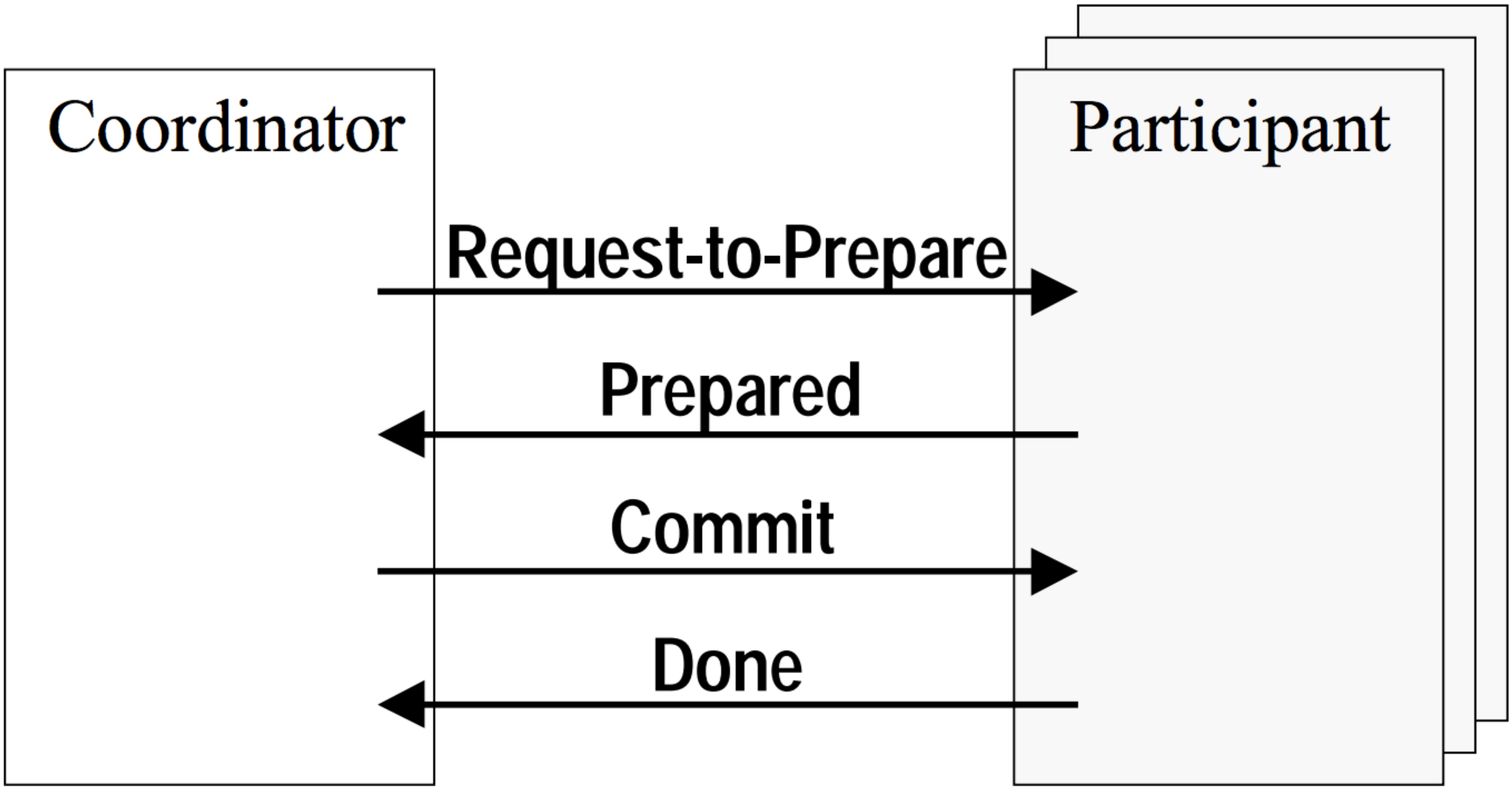
1. Transaction coordinator sends *prepare* message to each participating node

2. Each participating node responds to coordinator with *prepared* or *no*

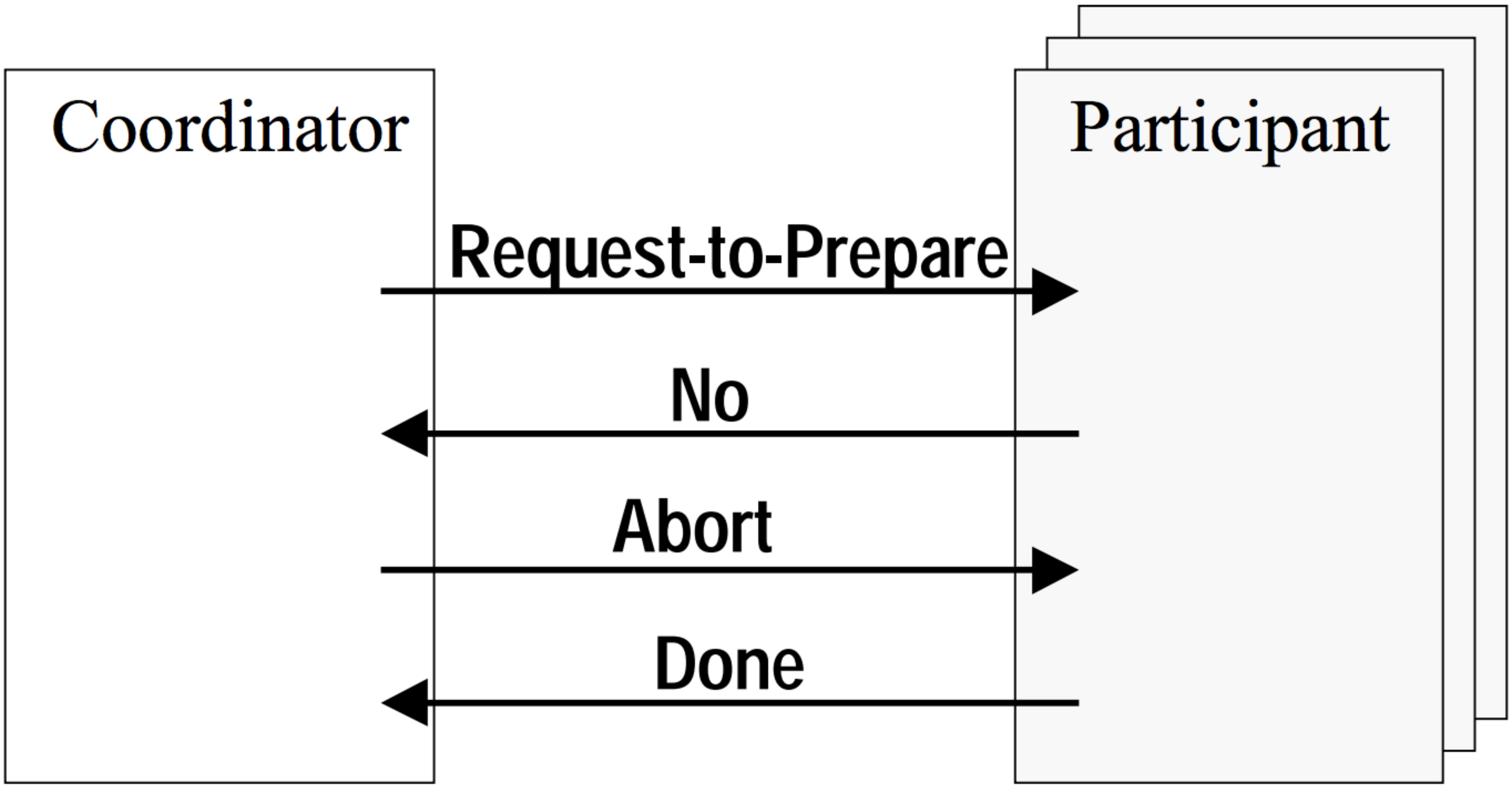
3. If coordinator receives all *prepared*:
   - » Broadcast *commit*

4. If coordinator receives any *no:*
   - » Broadcast *abort*

Case 1: Commit



Case 2: Abort



2PC + 2PL

Traditionally: run 2PC at commit time

» i.e., perform locking as usual, then run 2PC to have all participants agree that the transaction will commit

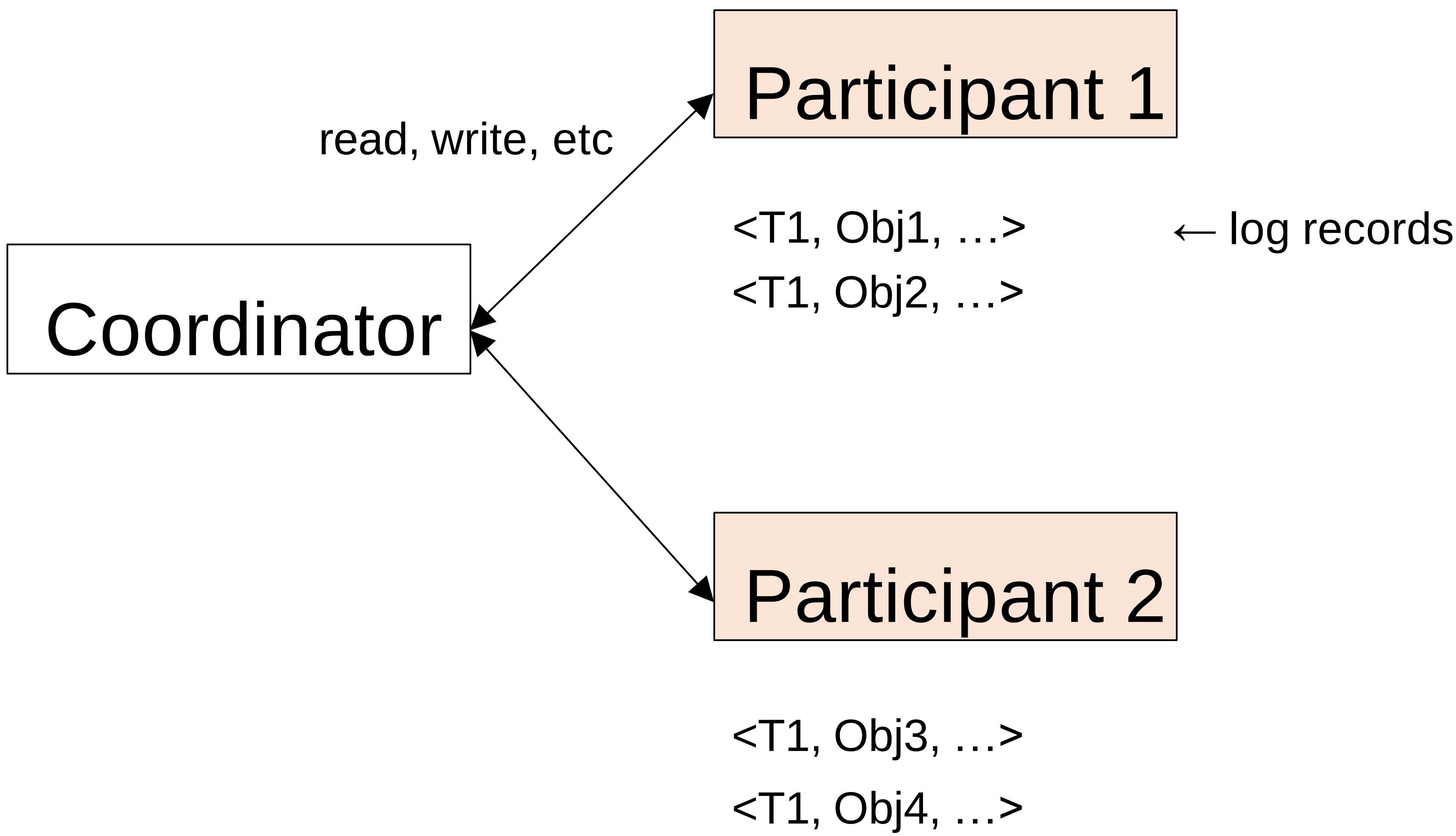
Under strict 2PL, run 2PC before unlocking the write locks

2PC + Logging

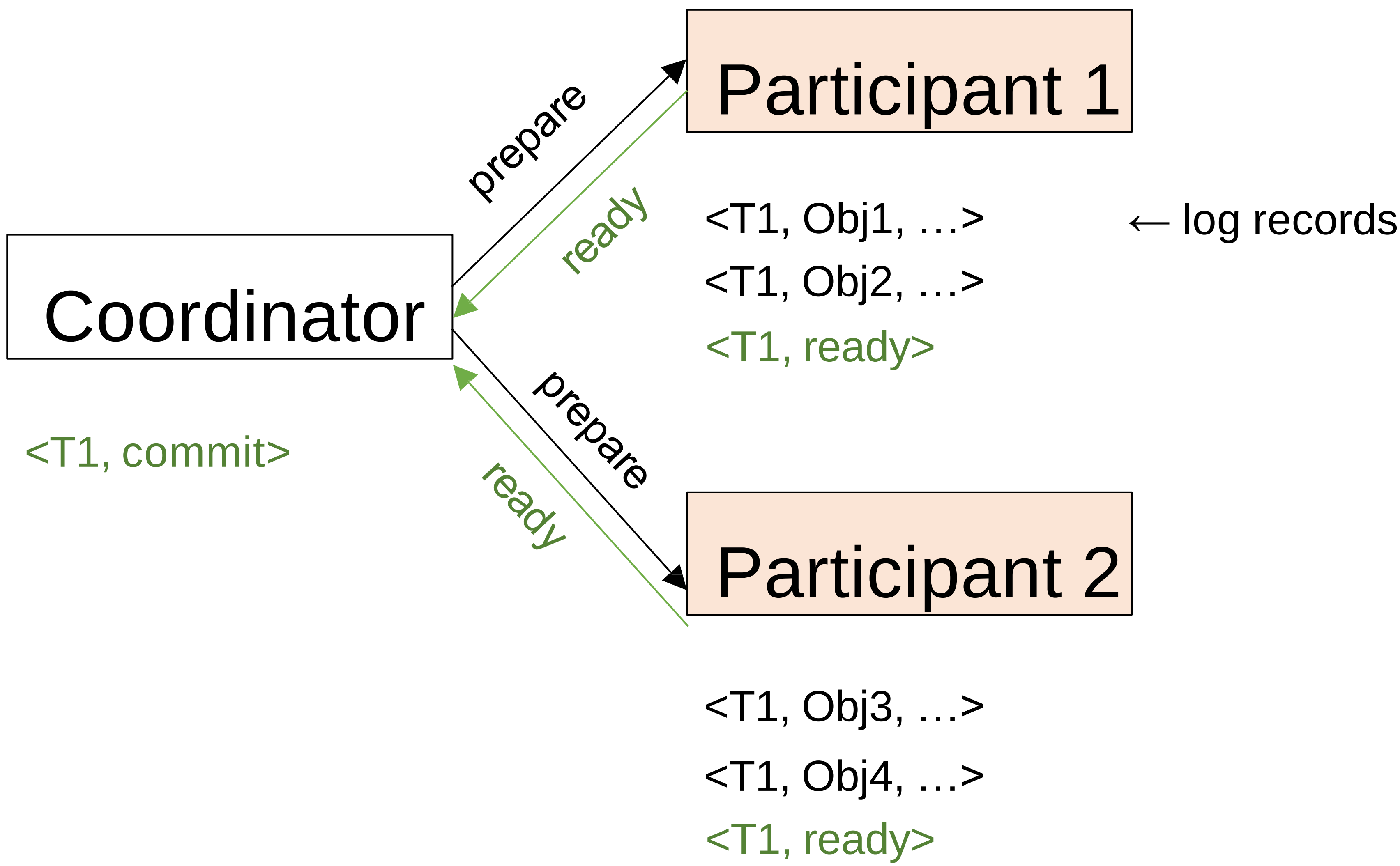
Log records must be flushed to disk on each participant before it replies to *prepare*

» The participant should log how it wants to respond + data needed if it wants to commit

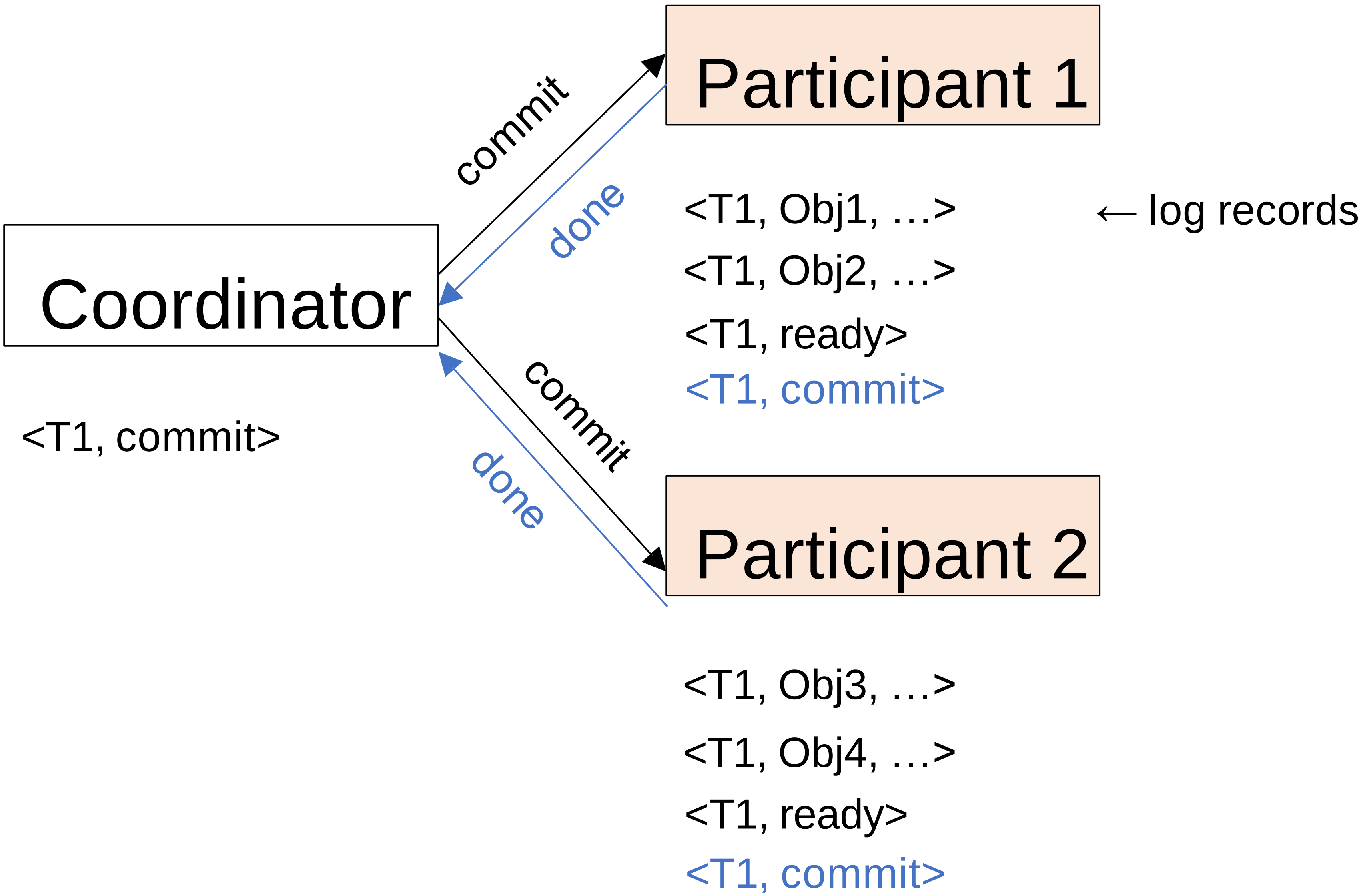
2PC + Logging Example



2PC + Logging Example



2PC + Logging Example



Optimizations Galore

Participants can send *prepared* messages to each other:

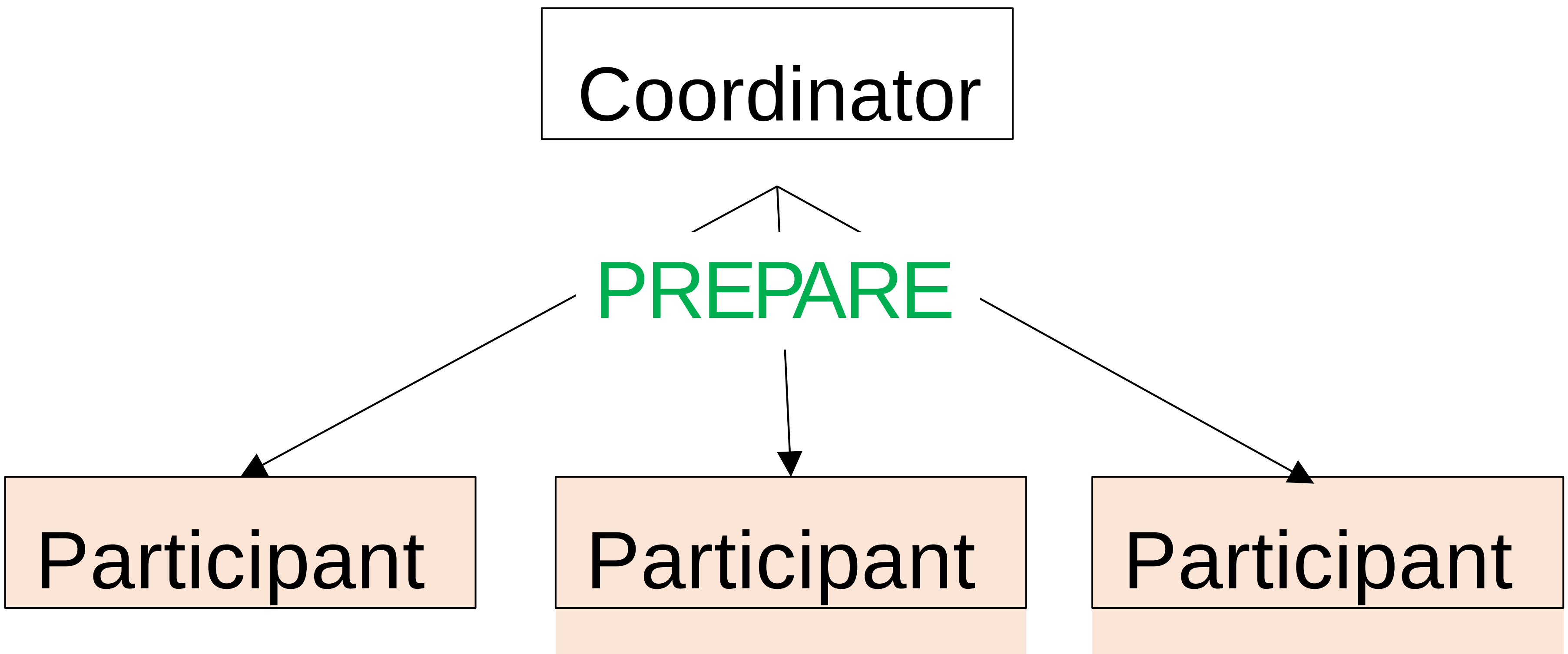
» Can commit without the client

» Requires $O(P^2)$ messages

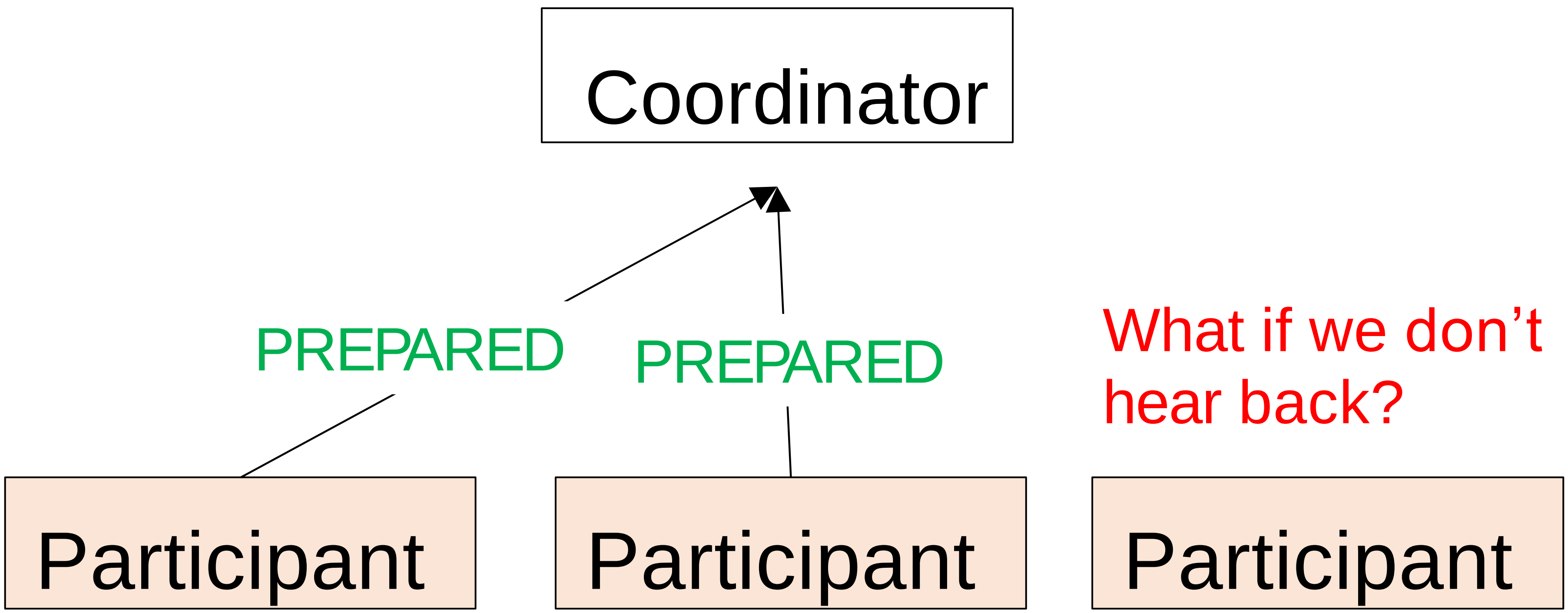
Piggyback transaction's last command on *prepare* message

2PL: piggyback lock "unlock" commands on *commit*/*abort* message

What Could Go Wrong?



What Could Go Wrong?



Case 1: Participant Unavailable

We don’t hear back from a participant

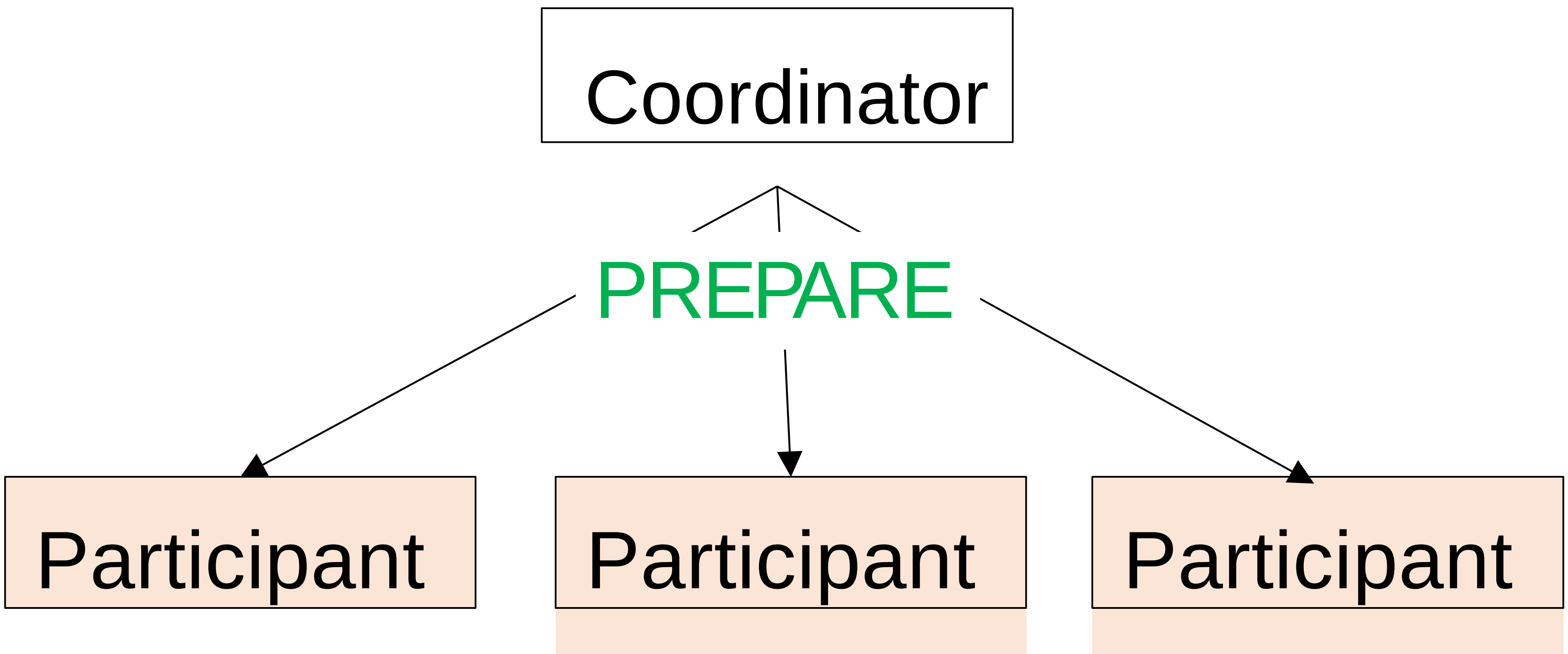
Coordinator can still decide to *abort*

» Coordinator makes the final call!

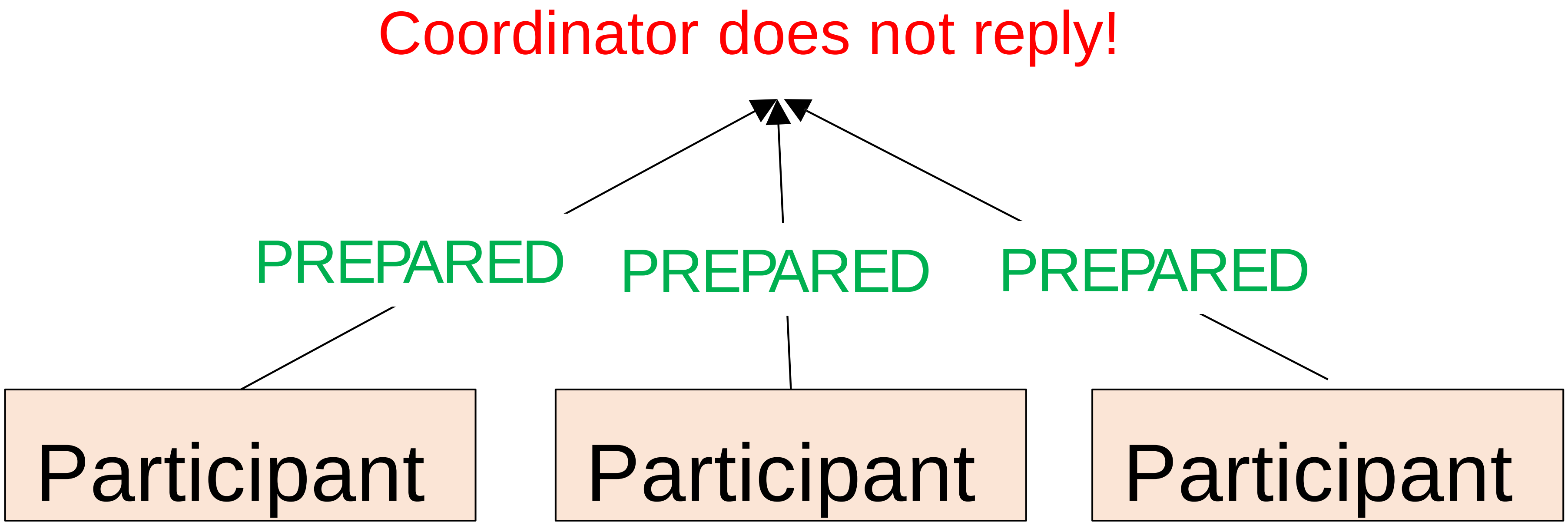
Participant comes back online?

» Will receive the *abort* message

What Could Go Wrong?



What Could Go Wrong?



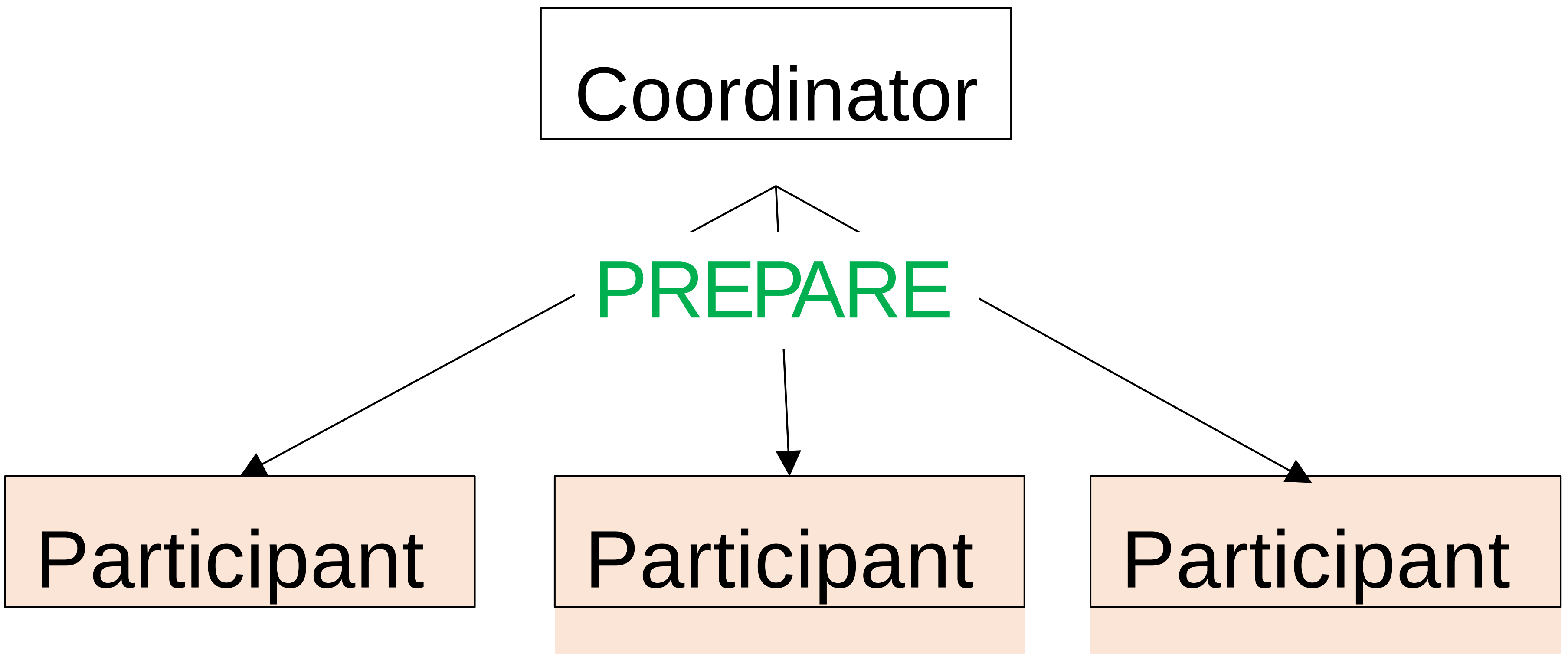
Case 2: Coordinator Unavailable

Participants cannot make progress

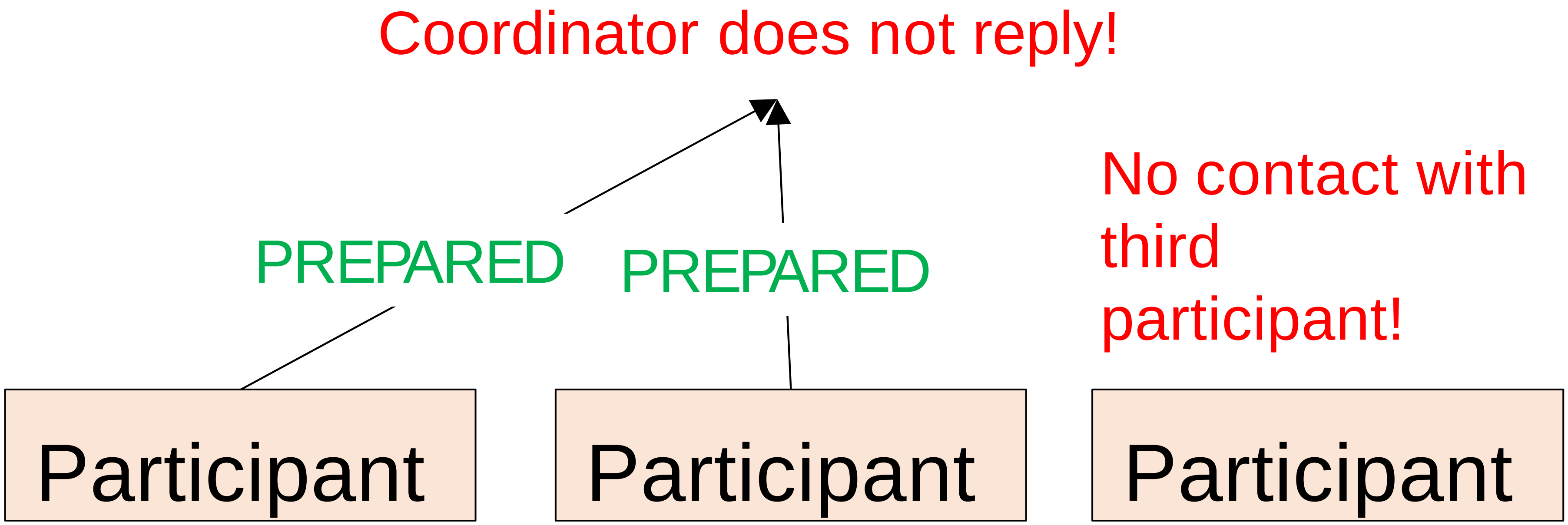
But: can agree to elect a *new* coordinator, never listen to the old one (using consensus)

» Old coordinator comes back? Overruled by participants, who reject its messages

What Could Go Wrong?



What Could Go Wrong?



Case 3: Coordinator and Participant Unavailable

Worst-case scenario:

- » Unavailable/unreachable participant voted to *prepare*
- » Coordinator hears back all *prepare*, broadcasts *commit*
- » Unavailable/unreachable participant *commits*

Rest of participants *must* wait!!!

Other Applications of 2PC

The “participants” can be any entities with distinct failure modes; for example:

- » Add a new user to database and queue a request to validate their email
- » Book a flight from SFO -> JFK on United and a flight from JFK -> LON on British Airways
- » Check whether Bob is in town, cancel my hotel room, and ask Bob to stay at his place

Coordination is Bad News

Every atomic commitment protocol is *blocking* (i.e., may stall) in the presence of:

» Asynchronous network behavior (e.g., unbounded delays)
  - Cannot distinguish between delay and failure

» Failing nodes
  - If nodes never failed, could just wait

CAP Theorem

In an asynchronous network, a distributed database can either:

- » guarantee a response from any replica in a finite amount of time ("availability") **OR**
- » guarantee arbitrary "consistency" criteria/constraints about data

but not both

CAP Theorem

Choose either:

» Consistency and “Partition tolerance” (CP)

» Availability and “Partition tolerance” (AP)

Example consistency criteria:

» Exactly one key can have value “Matei”

CAP is a reminder: no free lunch for distributed systems

Let’s Talk About Coordination

If we’re “AP”, then we don’t have to talk even when we can!

If we’re “CP”, then we have to talk all the time

Avoiding Coordination

Serializability has a provable cost to latency, availability, scalability (if there are conflicts)

“coordination-free execution”:

» Must look at application semantics

» Can be hard to get right!

» Strategy: start coordinated, then relax

Avoiding Coordination

Several techniques, e.g. the “BASE” ideas

» BASE = “Basically Available, Soft State, Eventual Consistency”

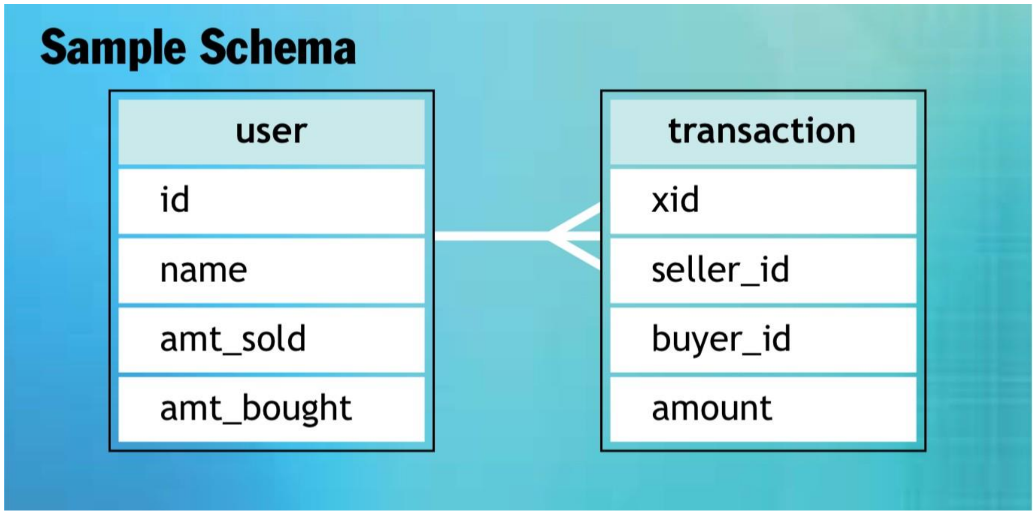
Key techniques for BASE:

» Partition data so that most transactions are local to one partition

» Tolerate stale data (eventual consistency):

- Caches
- Weaker isolation levels
- Helpful ideas: idempotence, commutativity

BASE Example



Constraint: each user's amt_sold and amt_bought is sum of their transactions

ACID Approach: to add a transaction, use 2PC to update transactions table + records for buyer, seller

One BASE approach: write new transactions to the transactions table and use a periodic batch job to fill in the users table