THE UNIVERSITY of EDINBURGH

Advanced Database Systems

Spring 2025

Lecture #21:

Distributed Transactions

R&G: Chapter 22

Parallel / Distributed DBMSs

Why do we need parallel / distributed DBMSs?

- Increased performance (throughput and latency)
- Increased availability

Database is spread out across multiple resources to improve parallelism

Appears as a single database instance to the application

- SQL query on a single-node DBMS must generate same result on a parallel or dist. DBMS
- Due to principle of **data independence**

Parallel vs. Distributed DBMSs

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

Observation

A distributed transaction can access data located on multiple nodes

- The DBMS must guarantee the ACID properties

We have not discussed how to ensure that all nodes agree to commit a transaction and then to make sure it does commit if we decide that it should

- What happens if a node fails?
- What happens if our messages show up late?
- What happens if we don't wait for every node to agree?

Outline

Distributed Locking

Distributed Deadlock Detection

Distributed Two-Phase Commit (2PC)

Recovery and 2PC

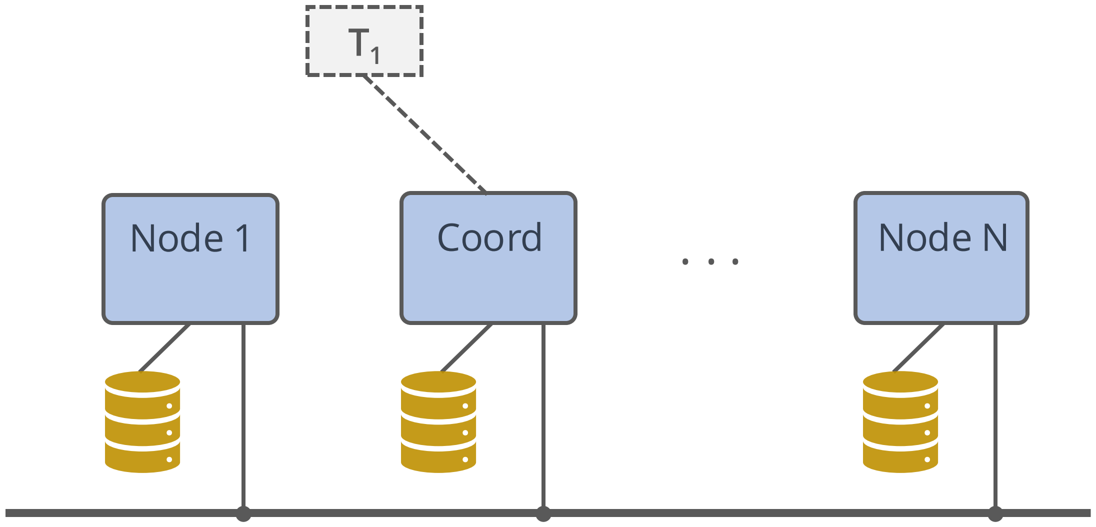
DISTRIBUTED CONCURRENCY CONTROL

Consider a shared-nothing distributed DBMS

For today, assume partitioning but no replication of data

Each transaction arrives at some node:

- The "coordinator" for the transaction



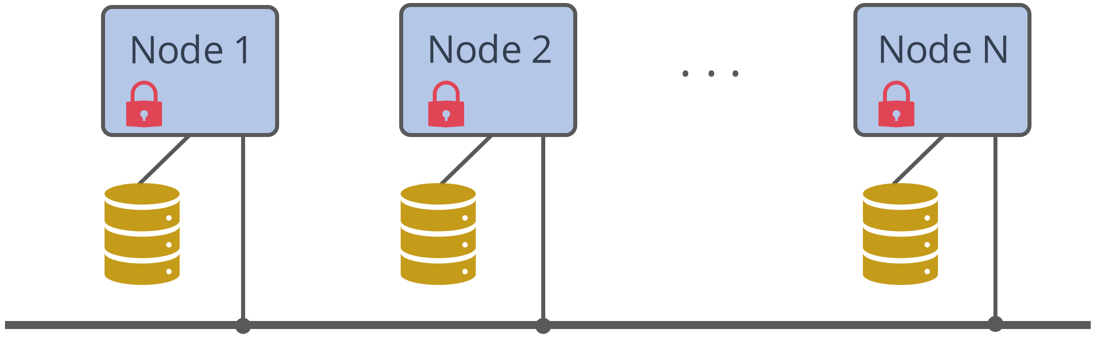
Where is the Lock Table?

Typical design: Locks partitioned with the data

- Independent: each node manages "its own" lock table
- Works for objects that fit on one node (pages, tuples)

For coarser-grained locks, assign a "home" node

- Object being locked (table, DB) exists across nodes



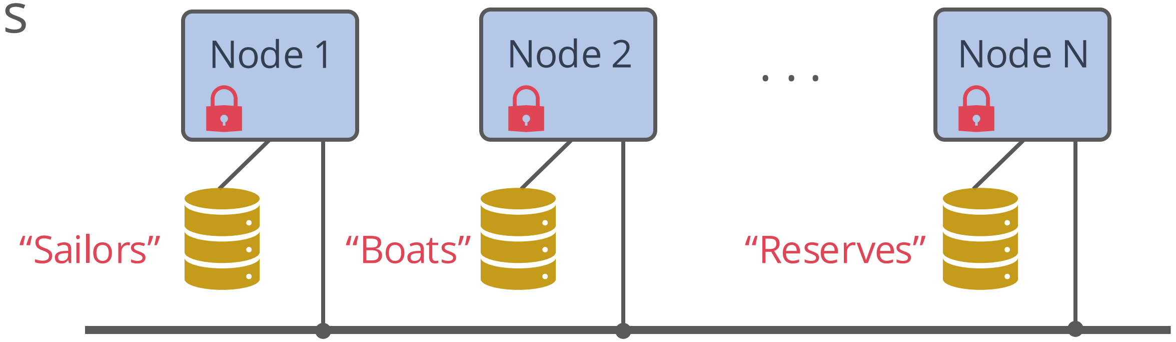
Where is the Lock Table?, Part 2

Typical design: Locks partitioned with the data

- Independent: each node manages "its own" lock table
- Works for objects that fit on one node (pages, tuples)

For coarser-grained locks, assign a "home" node

- Object being locked (table, DB) exists across nodes
- These locks can be partitioned across nodes



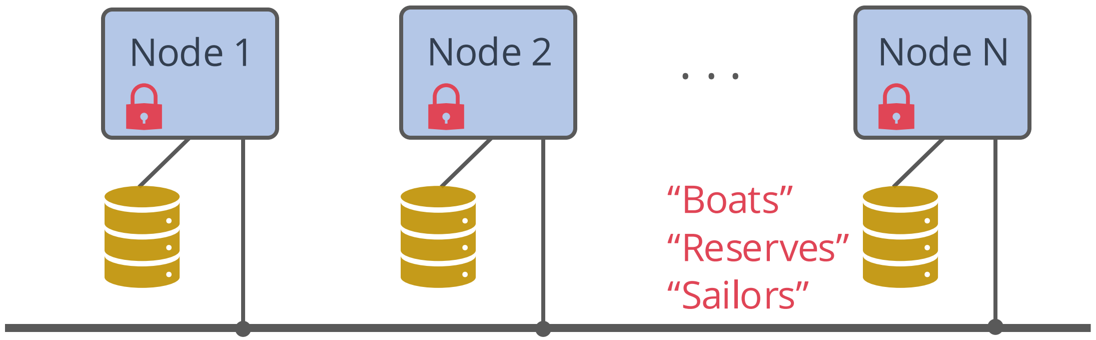
Where is the Lock Table?, Part 3

Typical design: Locks partitioned with the data

- Independent: each node manages "its own" lock table
- Works for objects that fit on one node (pages, tuples)

For coarser-grained locks, assign a "home" node

- Object being locked (table, DB) exists across nodes
- These locks can be partitioned across nodes
- Or centralized at one node



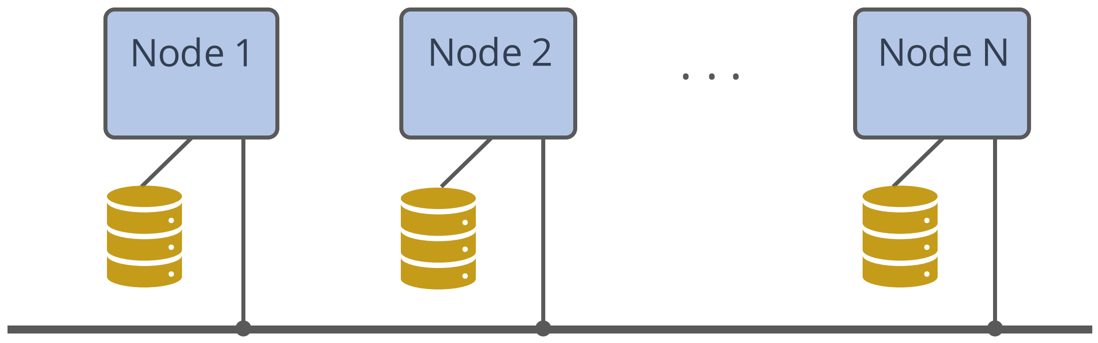
Ignore Global Locks For a Moment...

Every node does its own locking

Clean and efficient

"Global" issues remain:

- Deadlock
- Commit/Abort



OUTLINE

Distributed Locking

Distributed Deadlock Detection

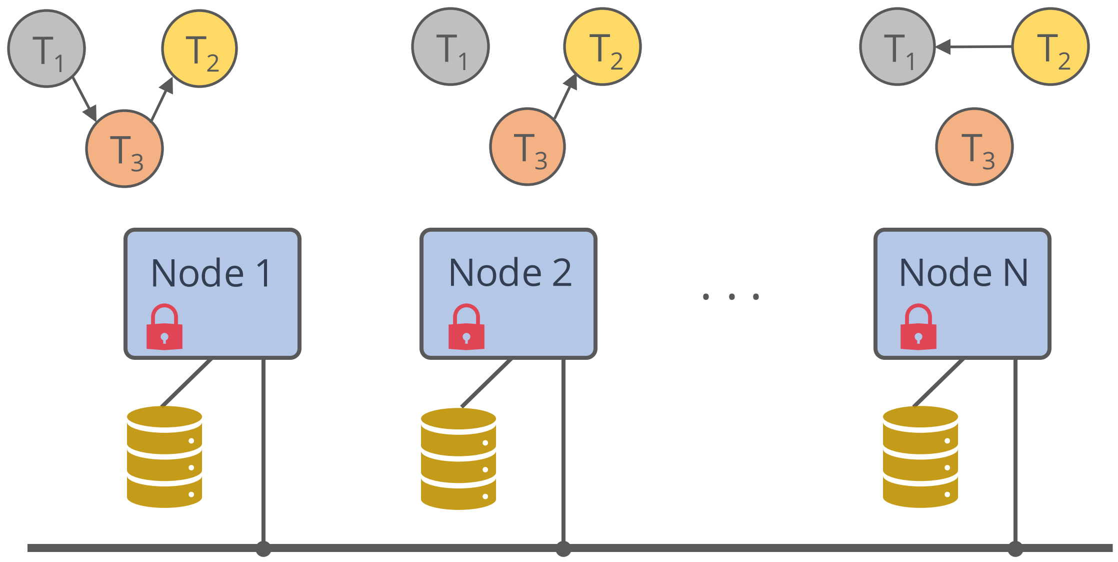
Distributed Two-Phase Commit (2PC)

Recovery and 2PC

What Could Go Wrong? #1

Deadlock detection

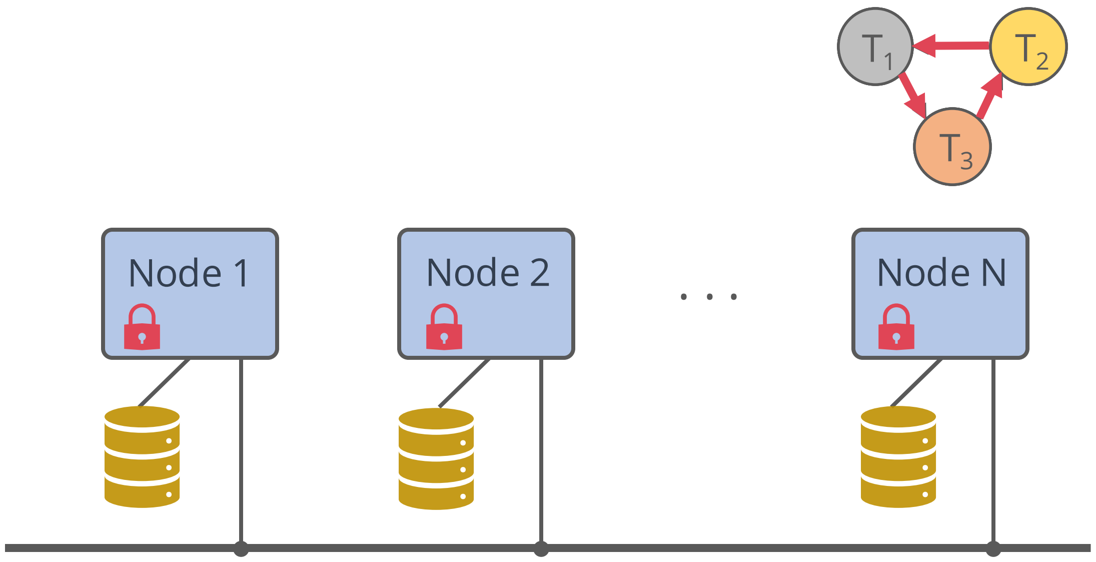
No cycles in local waits-for graphs, but there's a cycle in global waits-for graph



WHAT COULD GO WRONG? #1, PART 2

Deadlock detection

Easy fix: periodically union at designated node. If a cycle is detected, abort one txn



Outline

Distributed Locking

Distributed Deadlock Detection

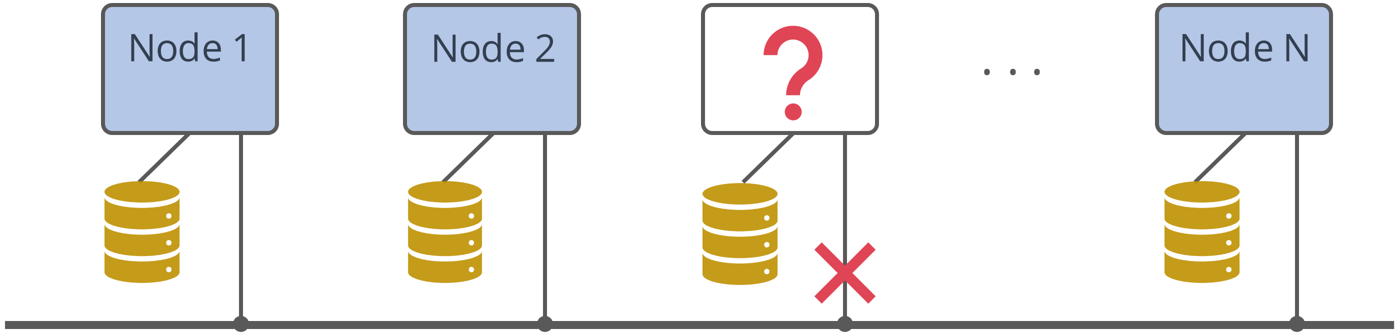
Distributed Two-Phase Commit (2PC)

Recovery and 2PC

What Could Go Wrong? #2

Failures/Delays: Nodes

- Commit? Abort?
- When the node comes back, how does it recover in a world that moved forward?

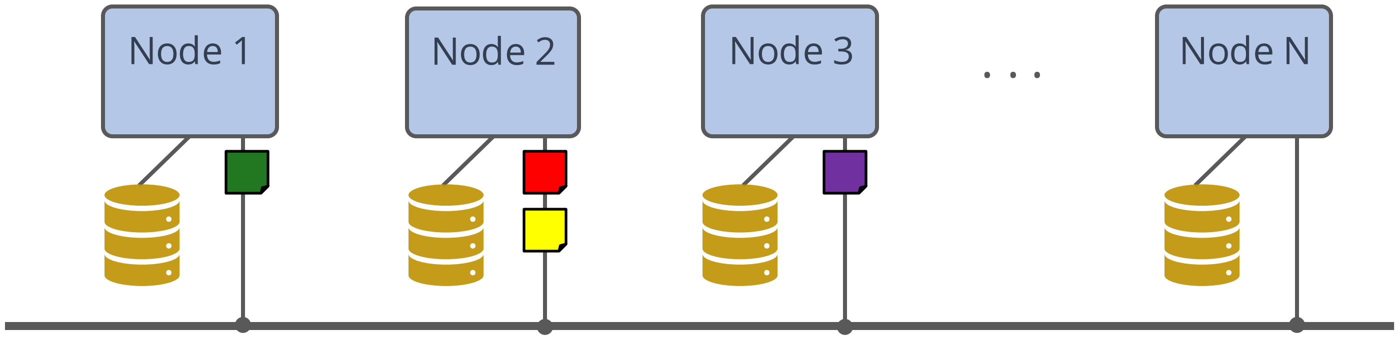


What Could Go Wrong? #2, Part 2

Failures/Delays: Nodes

Failures/Delays: Messages

- Non-deterministic reordering per channel, interleaving across channels
- “Lost” (very delayed) messages

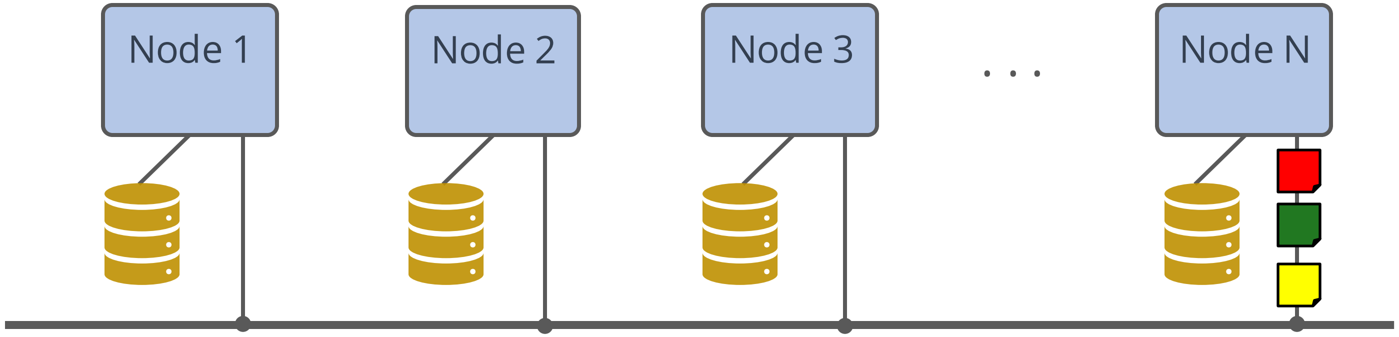


What Could Go Wrong? #2, Part 3

Failures/Delays: Nodes

Failures/Delays: Messages

- Non-deterministic reordering per channel, interleaving across channels
- “Lost” (very delayed) messages



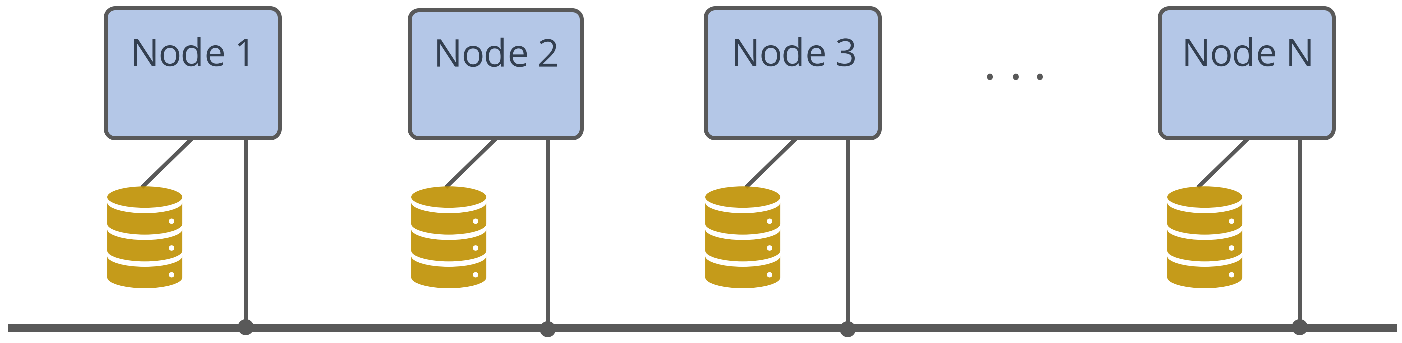
What Could Go Wrong? #2, Part 4

Failures/Delays: Nodes

Failures/Delays: Messages

- Non-deterministic reordering per channel, interleaving across channels
- “Lost” (very delayed) messages

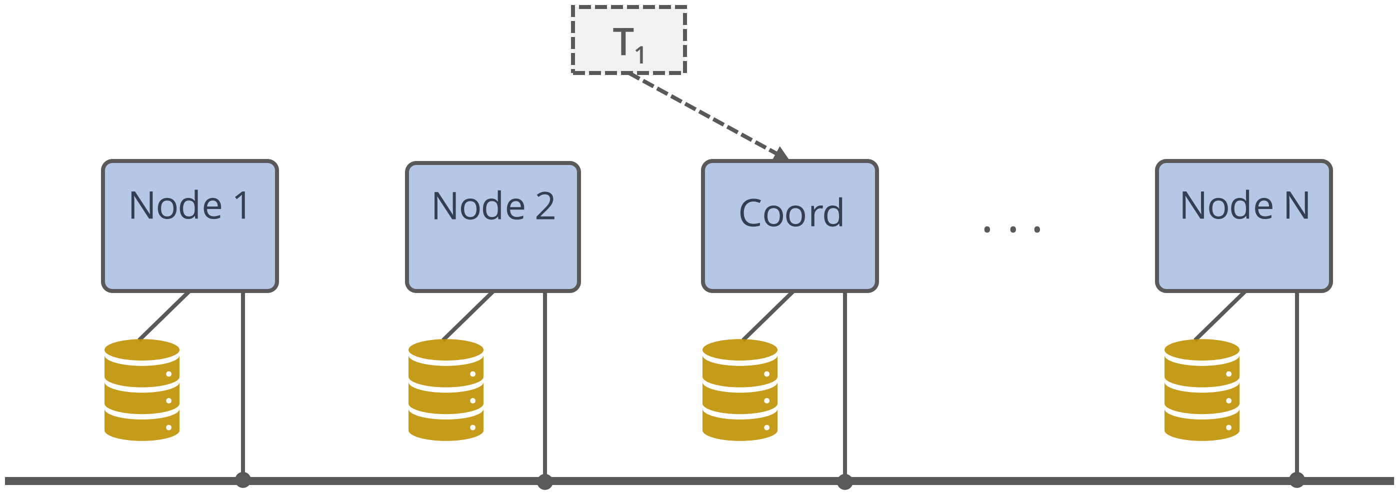
How do all nodes agree on Commit vs. Abort?



Basic Idea: Distributed Voting

Vote for commitment

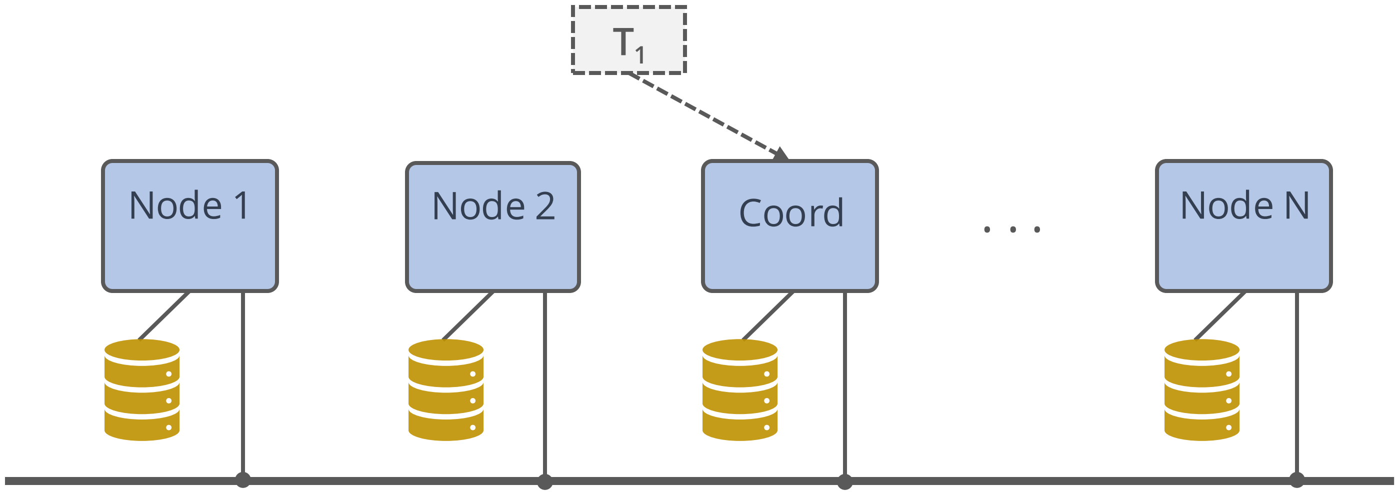
- How many votes does a commit need to win?
- Any single node could observe a problem (e.g., deadlock, constraint violation)
- Hence must be unanimous



Distributed voting? How?

How do we implement distributed voting?!

In the face of message/node failure/delay?



2-Phase Commit

A.k.a. 2PC. (Not to be confused with 2PL!)

Phase 1: Voting phase

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for yes!

Phase 2: Commit phase

- Coordinator disseminates result of the vote

Need to do some logging for failure handling....

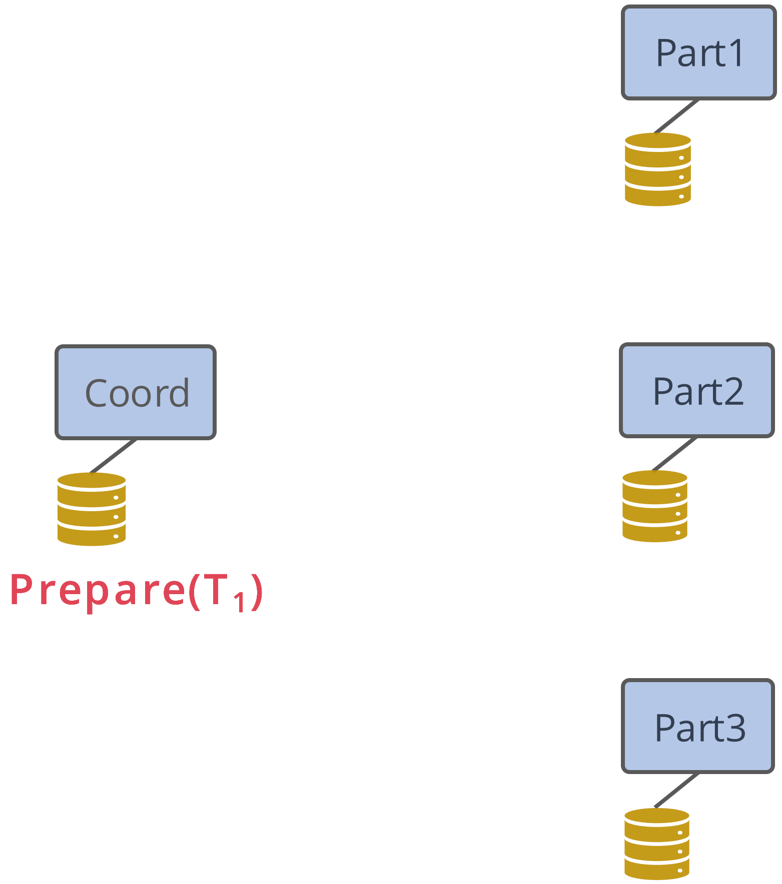
2-Phase Commit, Part 1

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- Coordinator disseminates result of the vote
- Participants respond with Ack



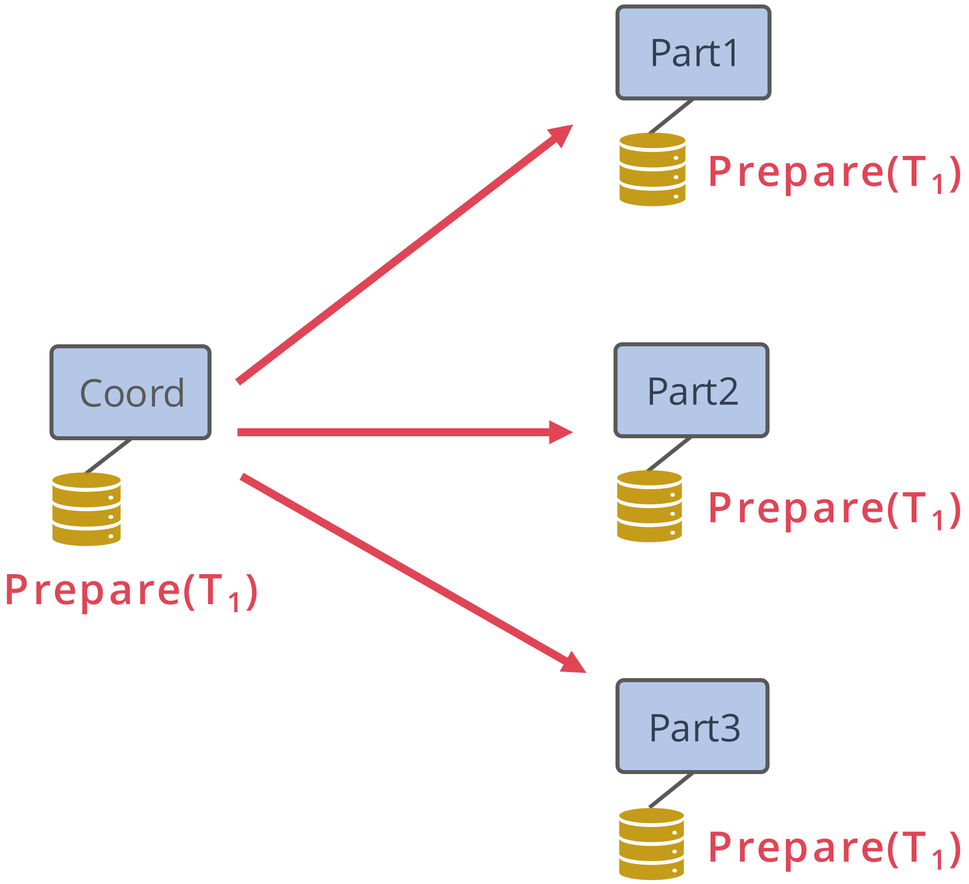
2-Phase Commit, Part 2

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- Coordinator disseminates result of the vote
- Participants respond with Ack



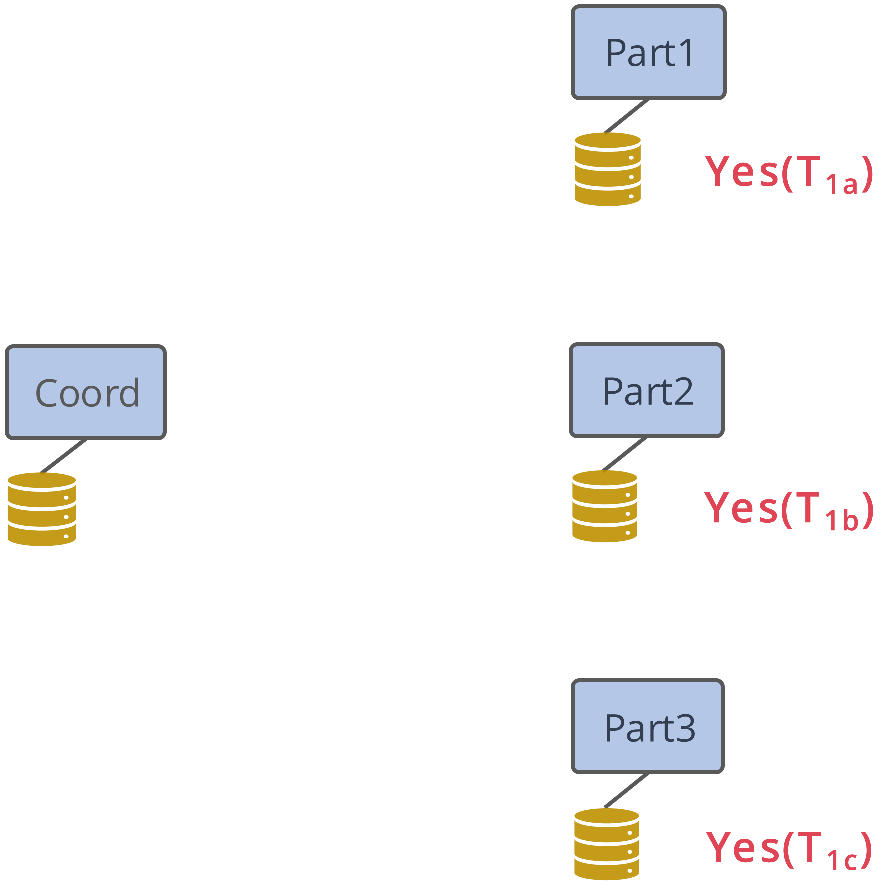
2-Phase Commit, Part 3

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- Coordinator disseminates result of the vote
- Participants respond with Ack



2-Phase Commit, Part 4

Phase 1:

Coordinator tells participants to "prepare"

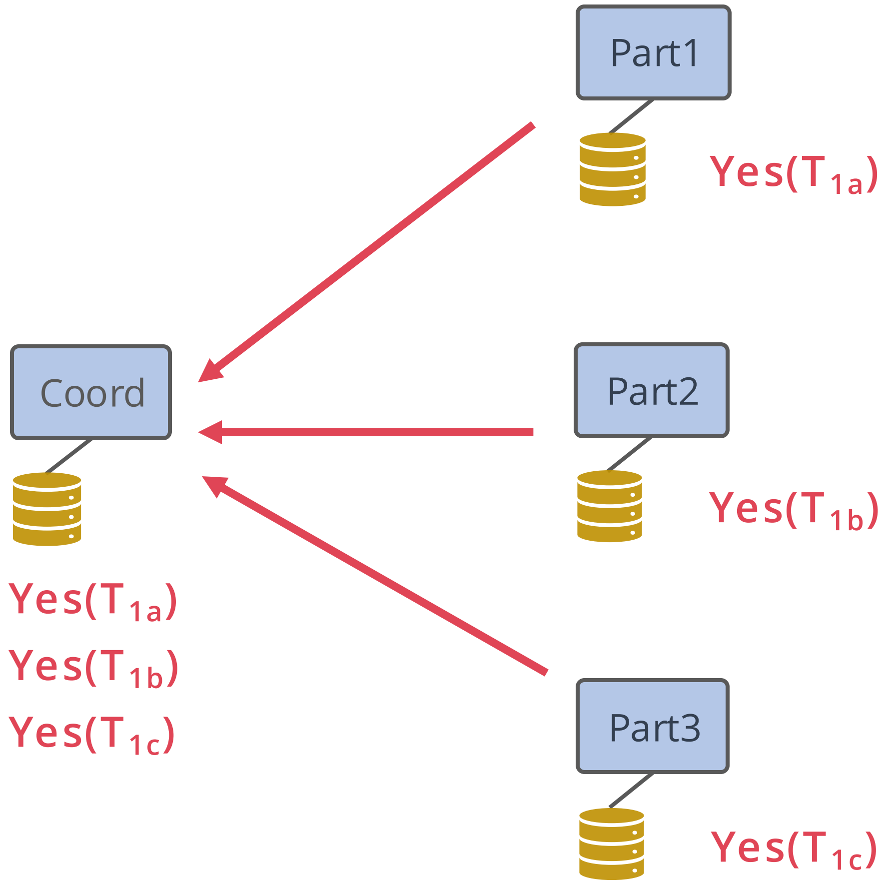
Participants respond with yes/no votes

Unanimity required for commit!

Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



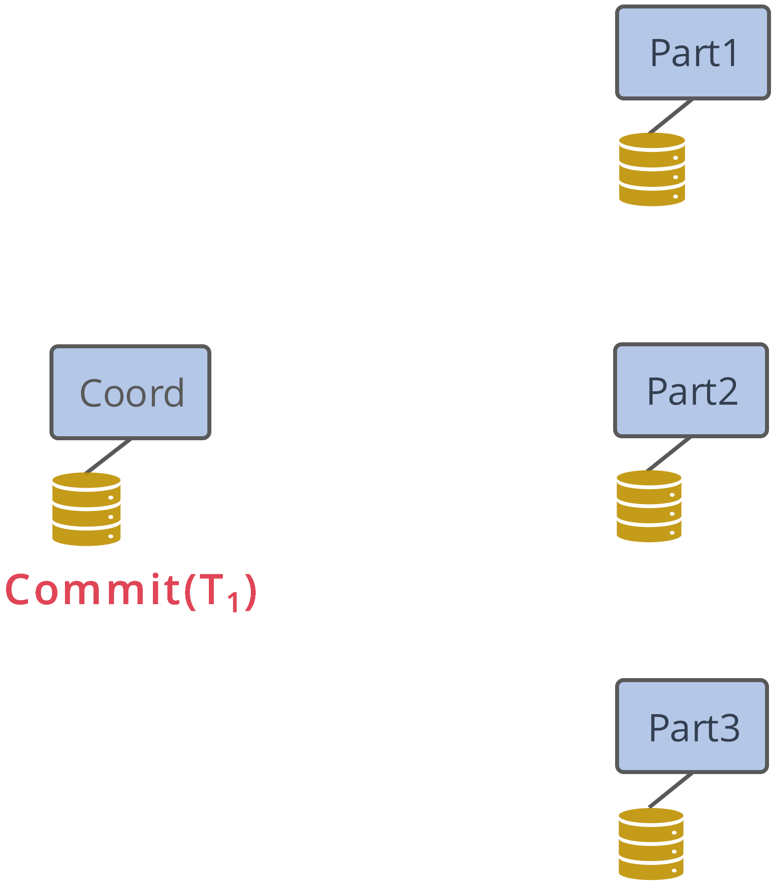
2-PHASE COMMIT, PART 5

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- **Coordinator disseminates result of the vote**
- Participants respond with Ack



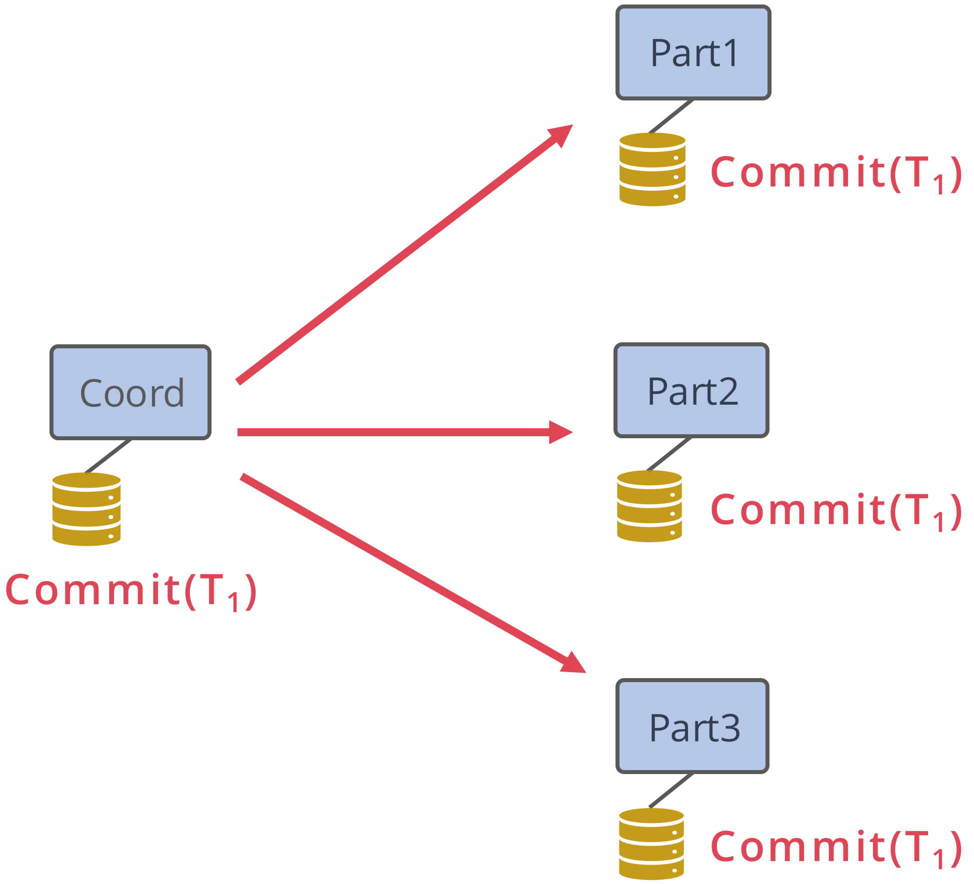
2-Phase Commit, Part 6

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- **Coordinator disseminates result of the vote**
- Participants respond with Ack



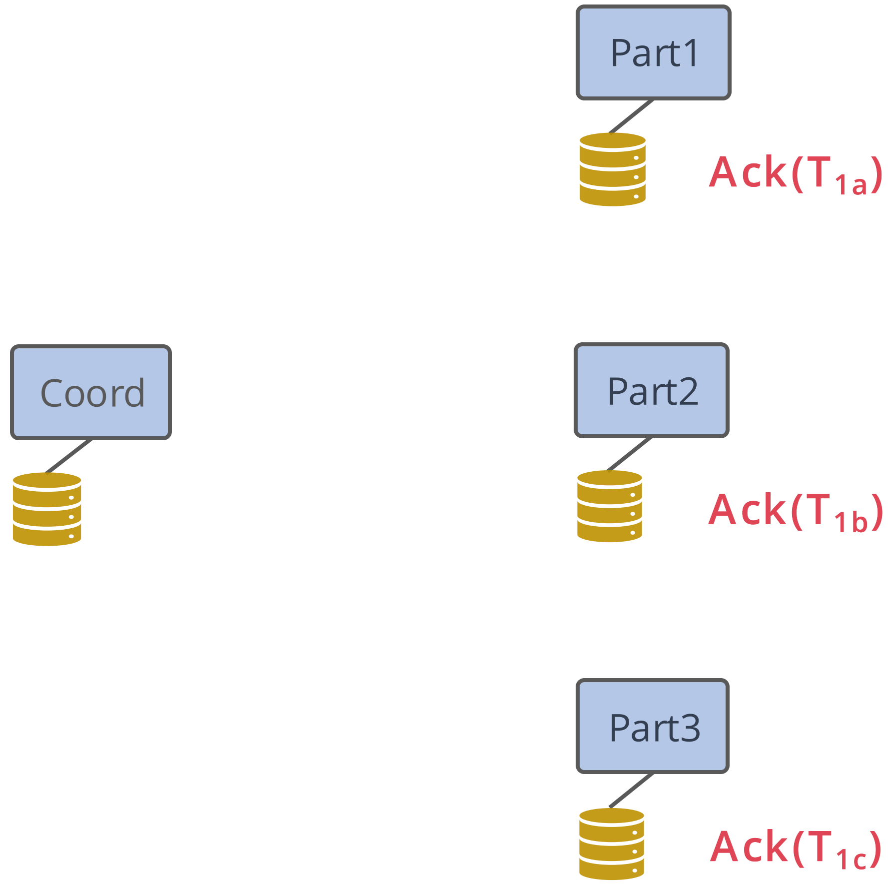
2-Phase Commit, Part 7

Phase 1:

- Coordinator tells participants to “prepare”
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

- Coordinator disseminates result of the vote
- Participants respond with Ack



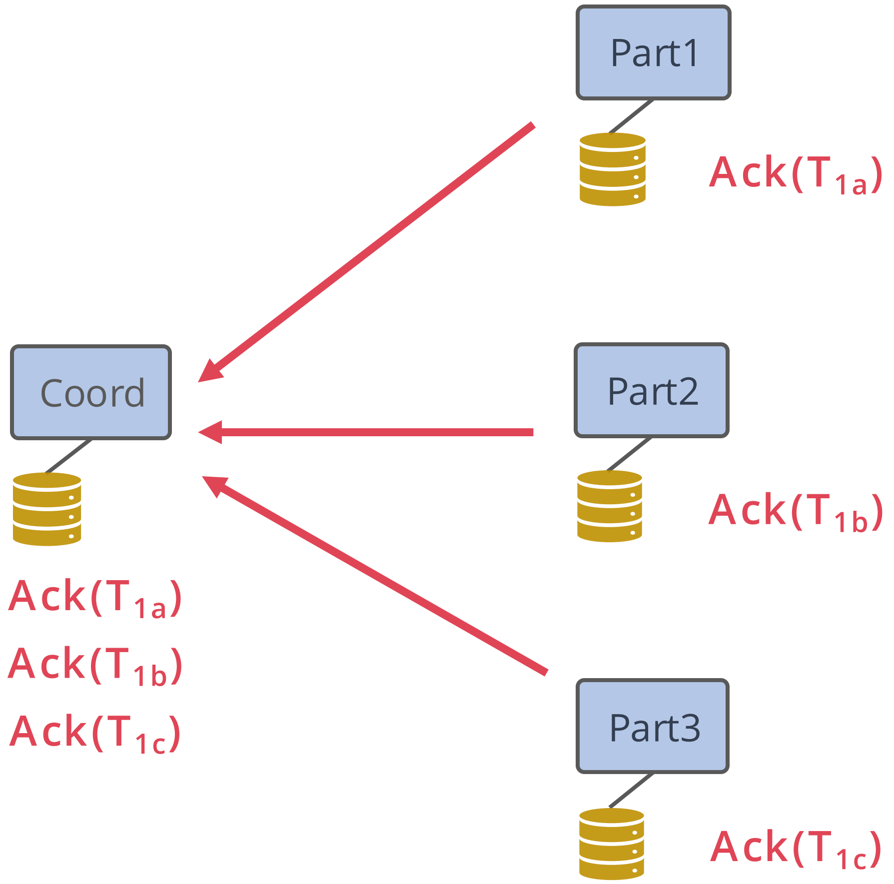
2-Phase Commit, Part 8

Phase 1:

- Coordinator tells participants to "prepare"
- Participants respond with yes/no votes
  - Unanimity required for commit!

Phase 2:

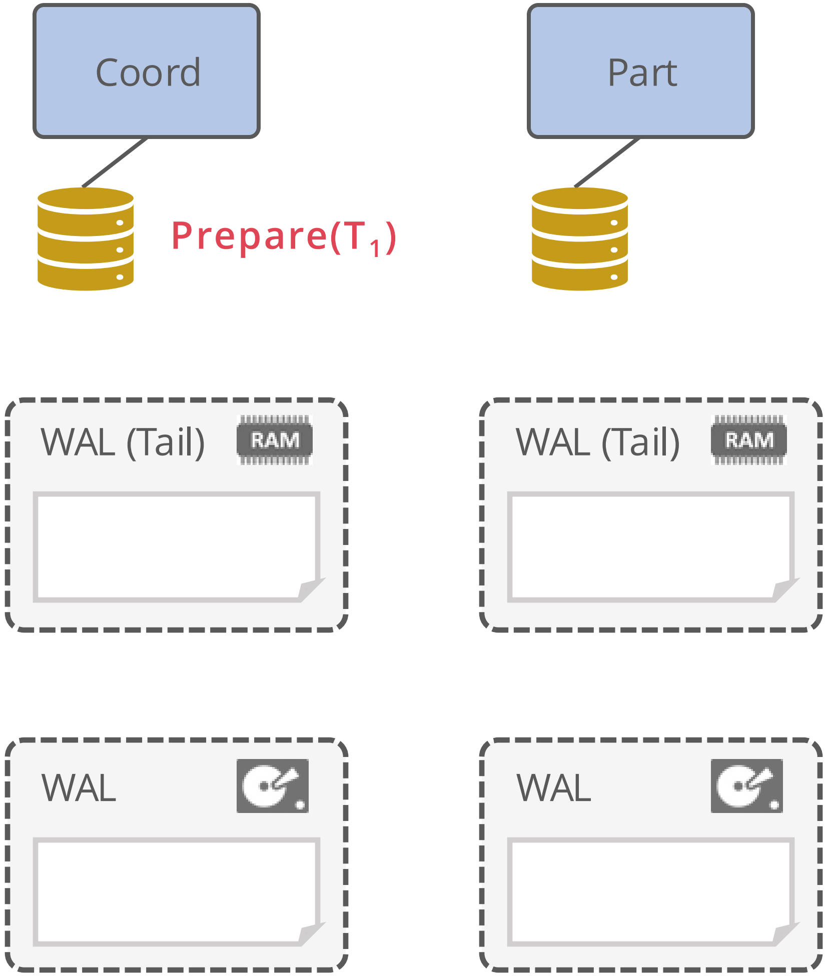
- Coordinator disseminates result of the vote
- Participants respond with Ack



One More Time, with Logging

Phase 1:

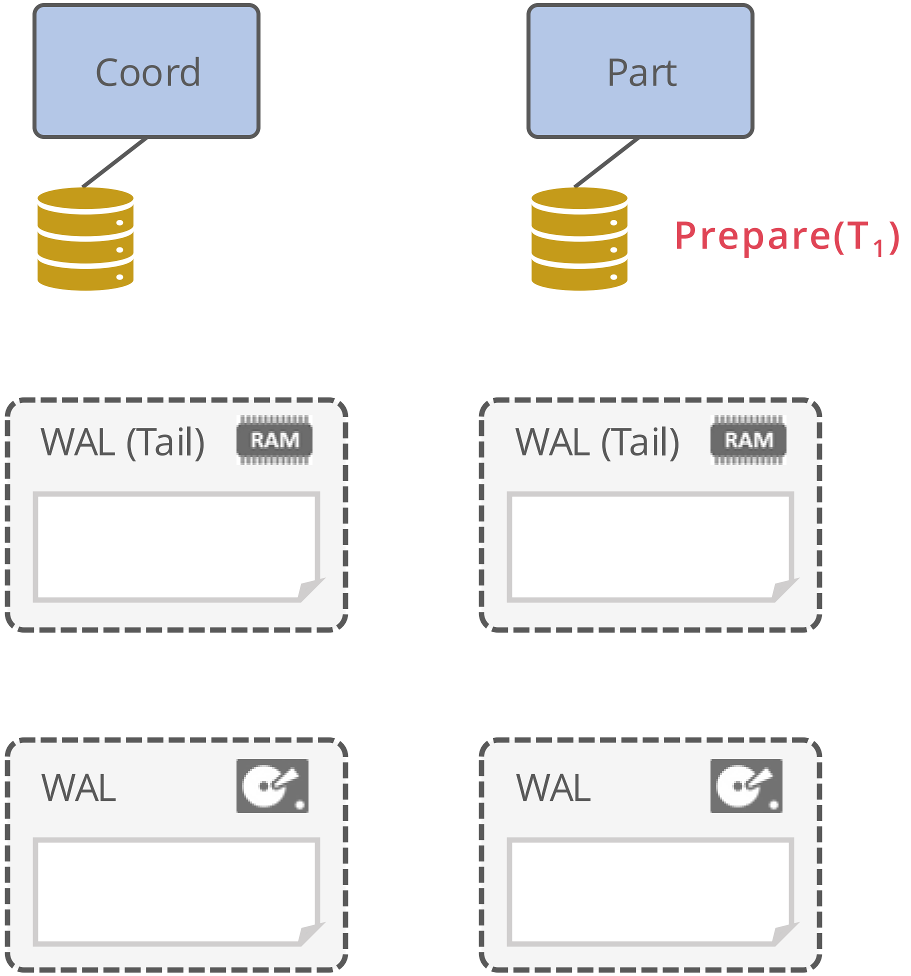
- Coordinator tells participants to "prepare"
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 2

Phase 1:

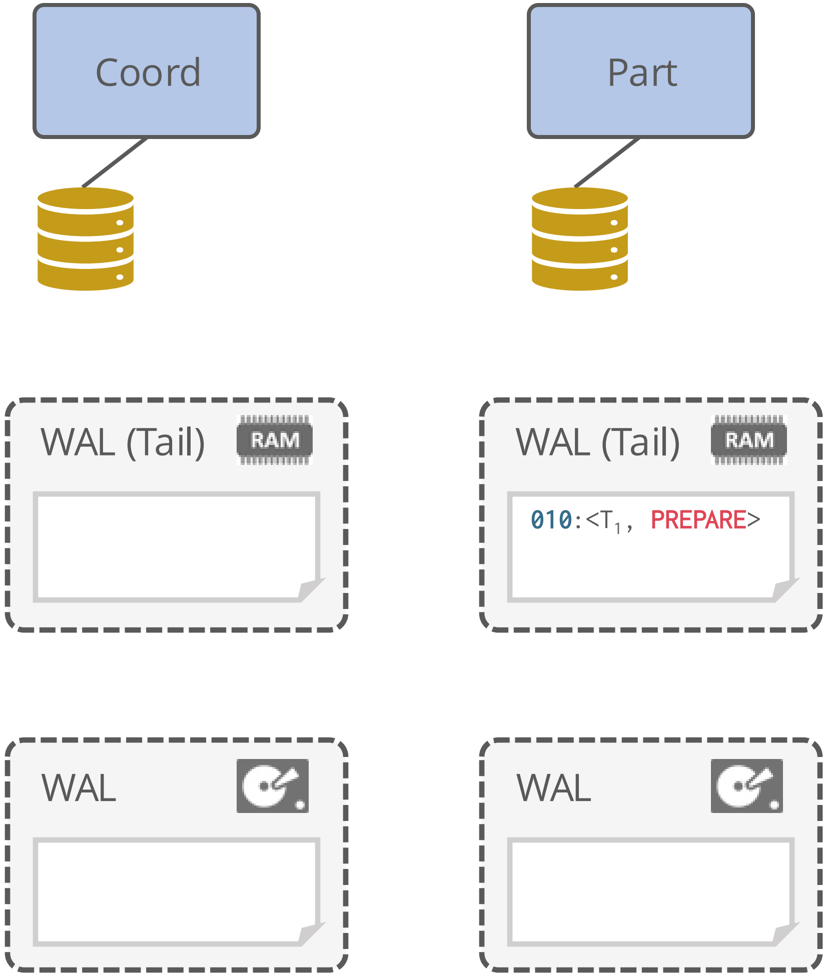
- Coordinator tells participants to "prepare"
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 3

Phase 1:

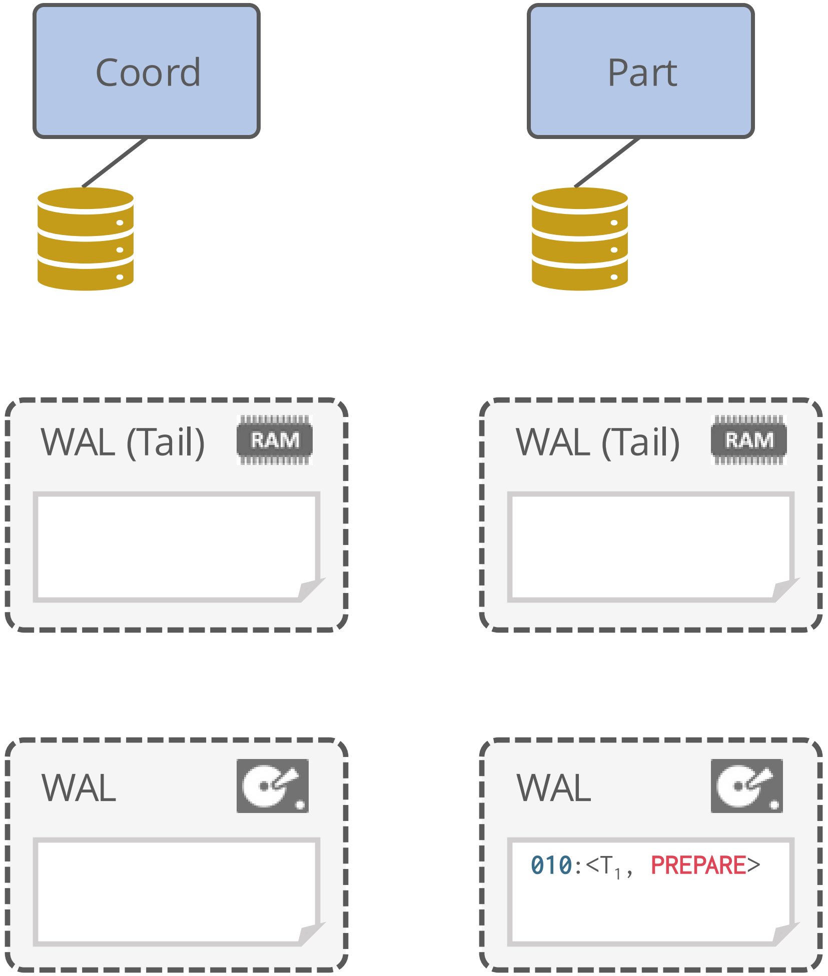
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 4

Phase 1:

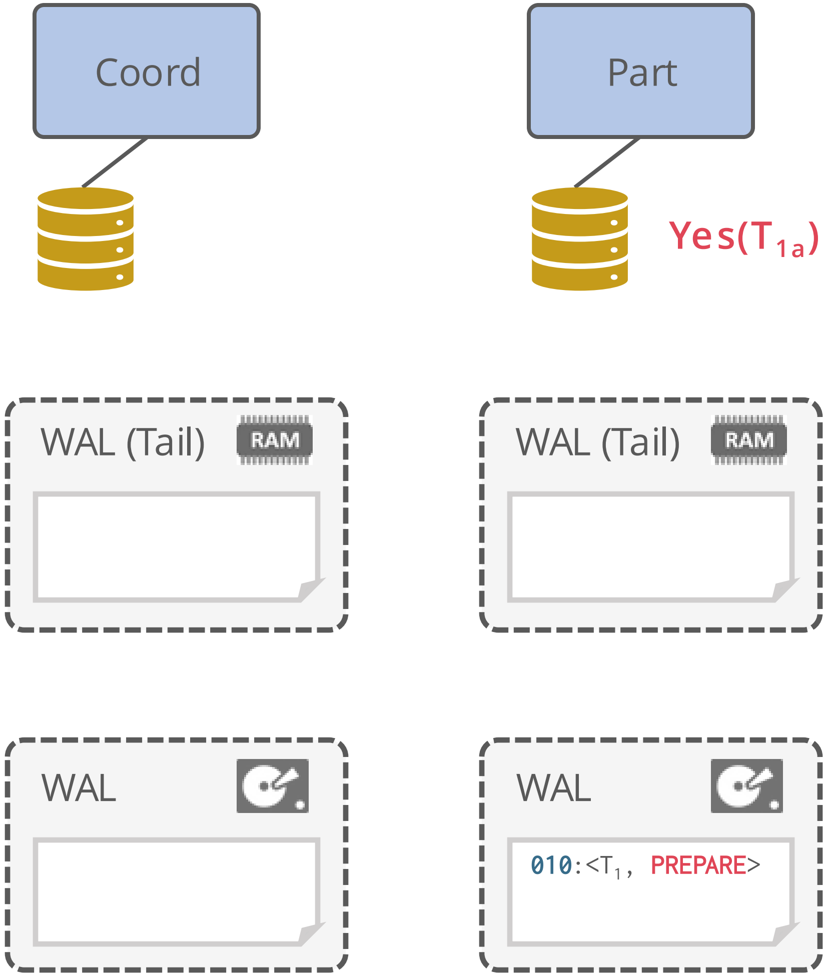
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- **Participants flush prepare/abort record**
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 5

Phase 1:

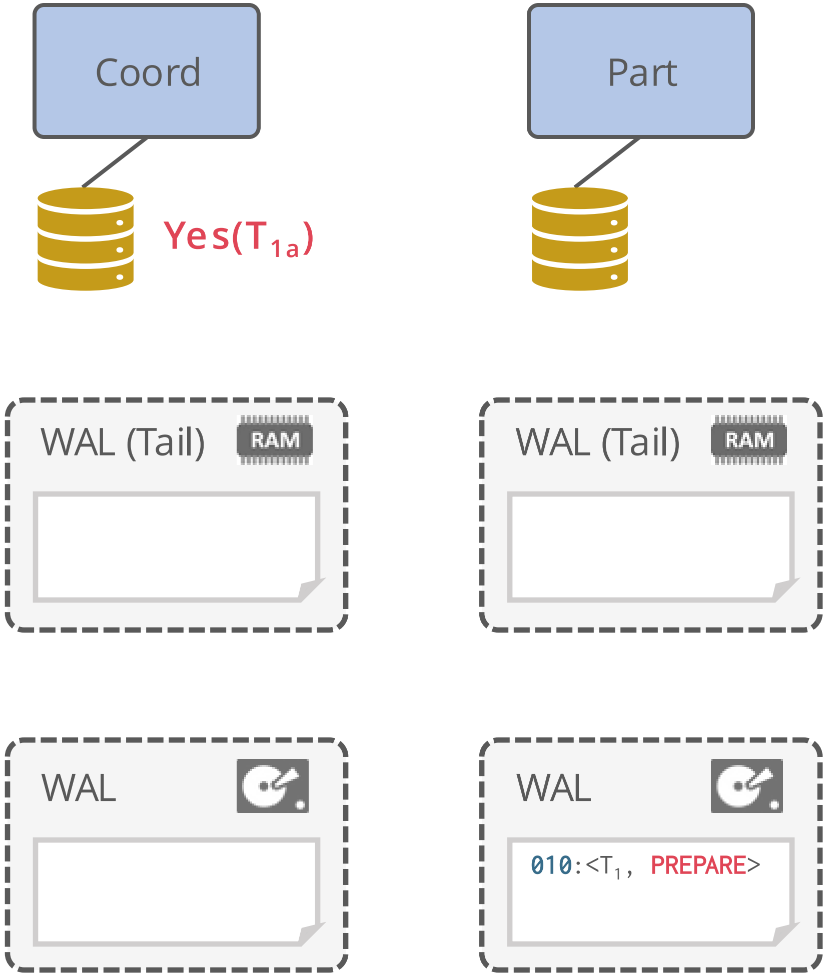
- Coordinator tells participants to "prepare"
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- **Participants respond with yes/no votes**
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 6

Phase 1:

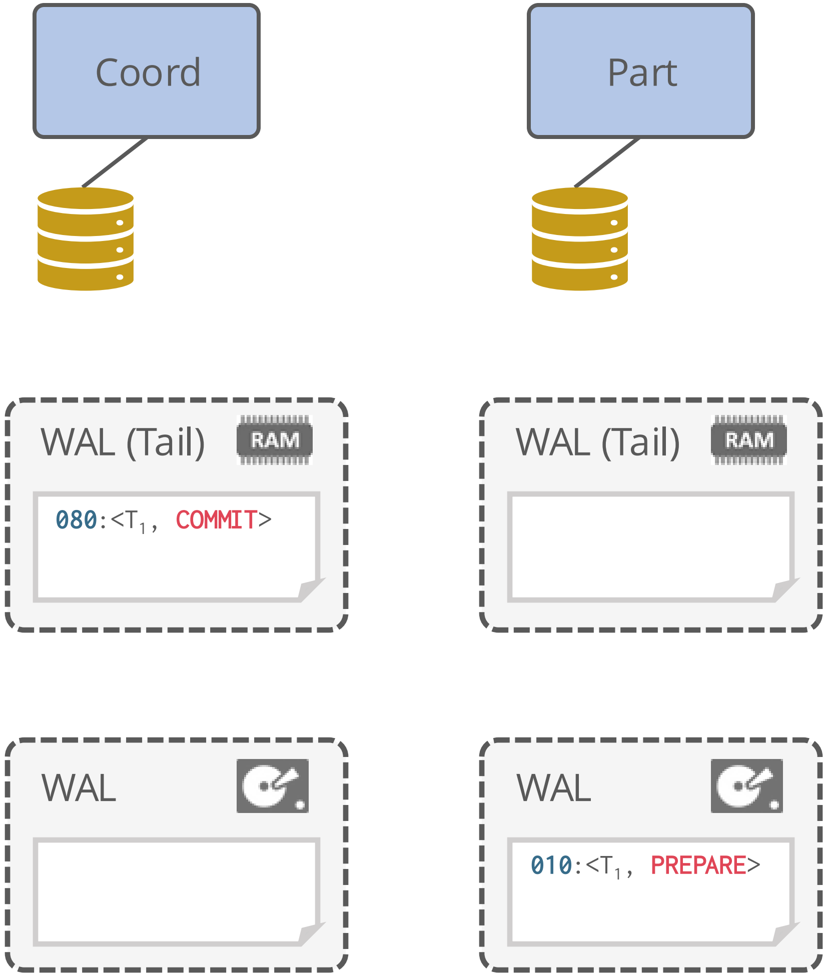
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- **Participants respond with yes/no votes**
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 7

Phase 1:

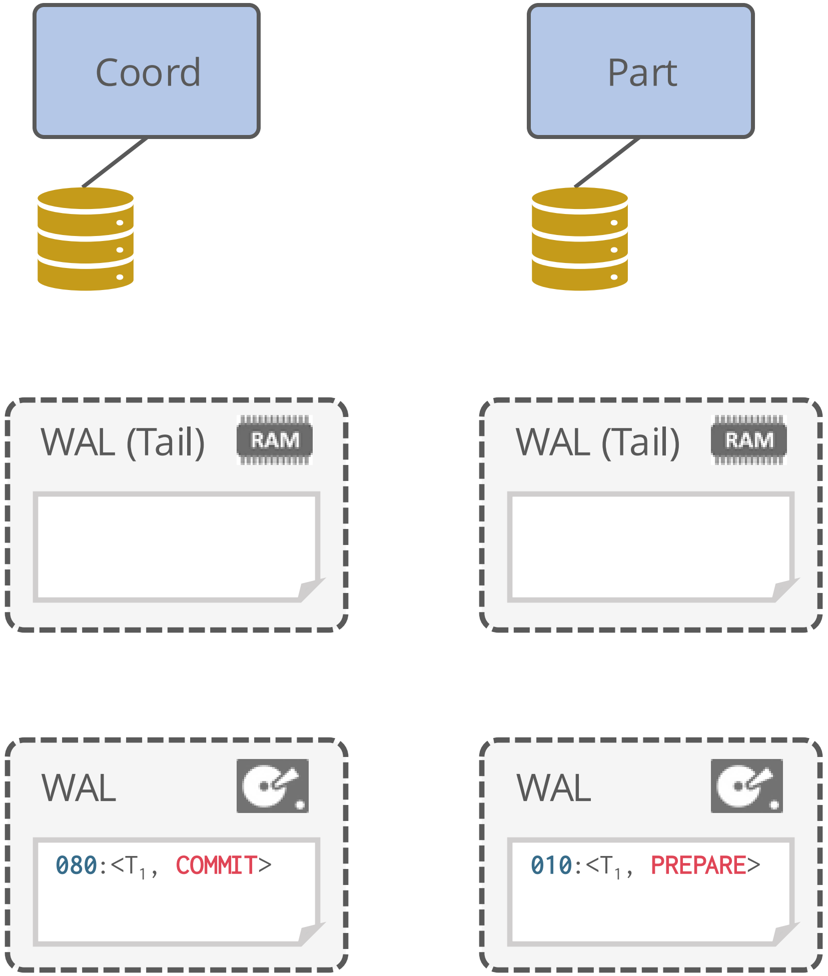
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



One More Time, with Logging, Part 8

Phase 1:

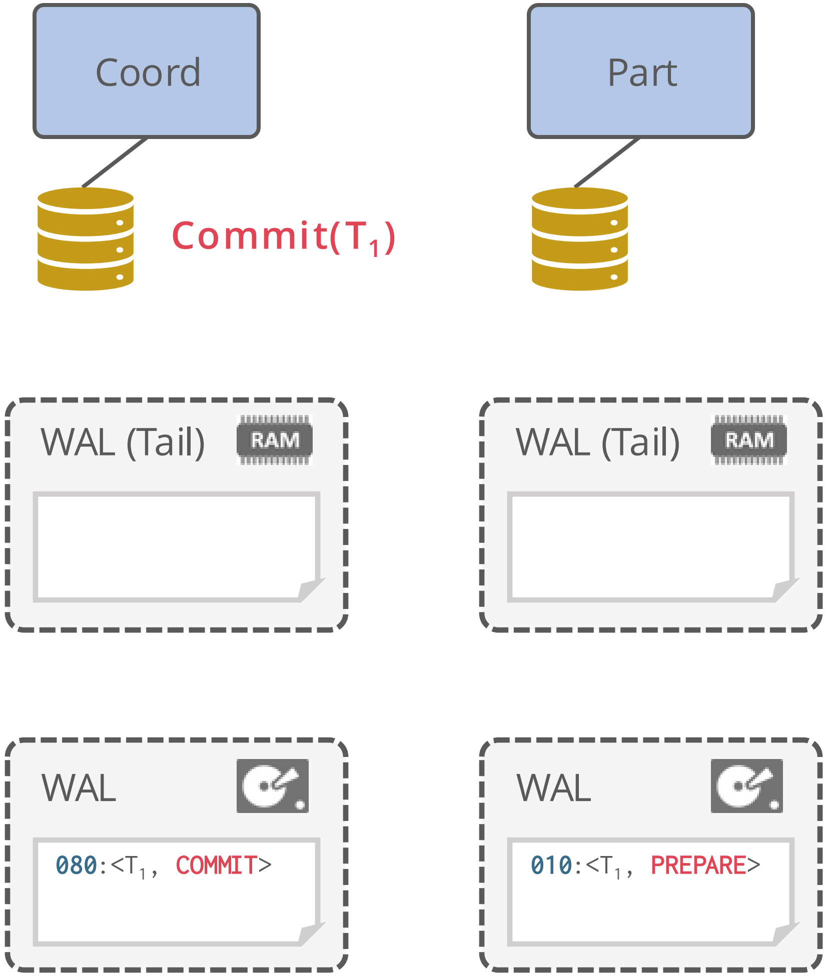
- Coordinator tells participants to "prepare"
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- **Coordinator flushes commit record**



One More Time, with Logging, Part 9

Phase 2:

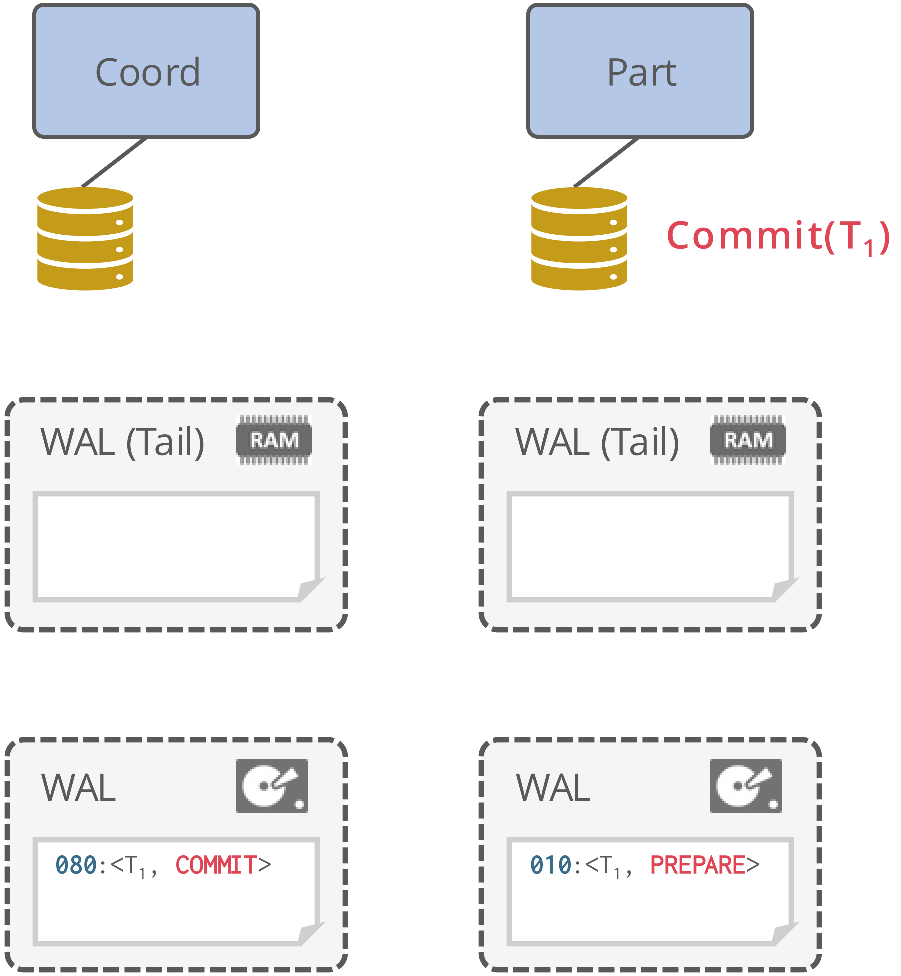
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 10

Phase 2:

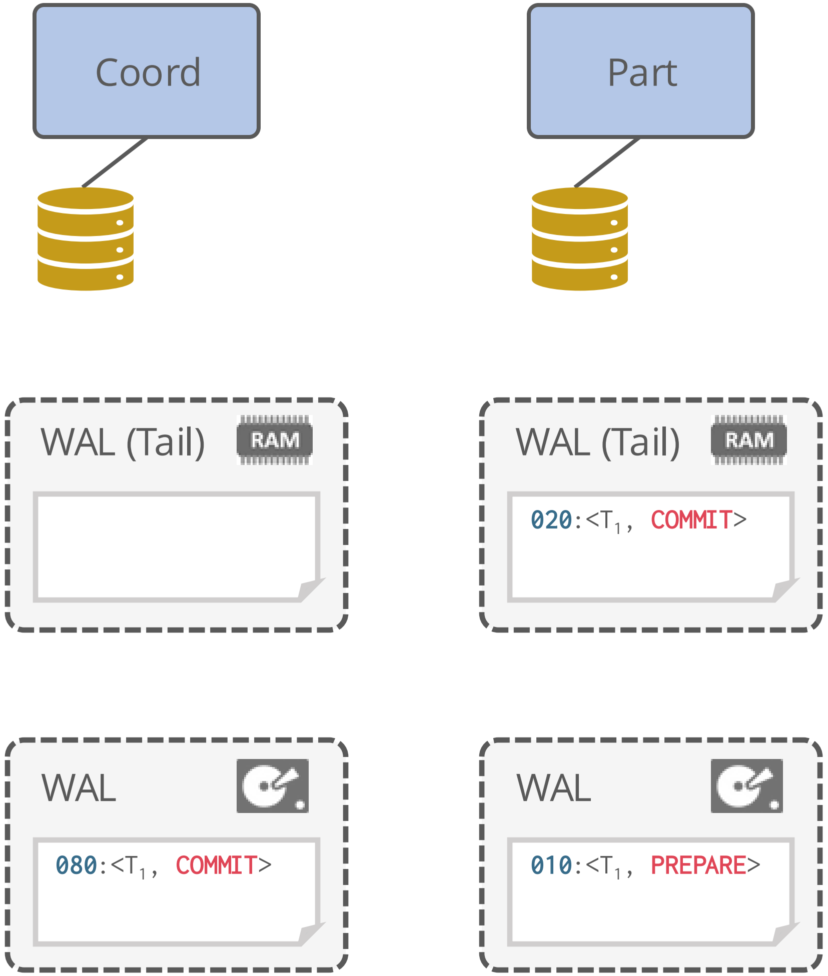
- **Coordinator broadcasts result of vote**
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 11

Phase 2:

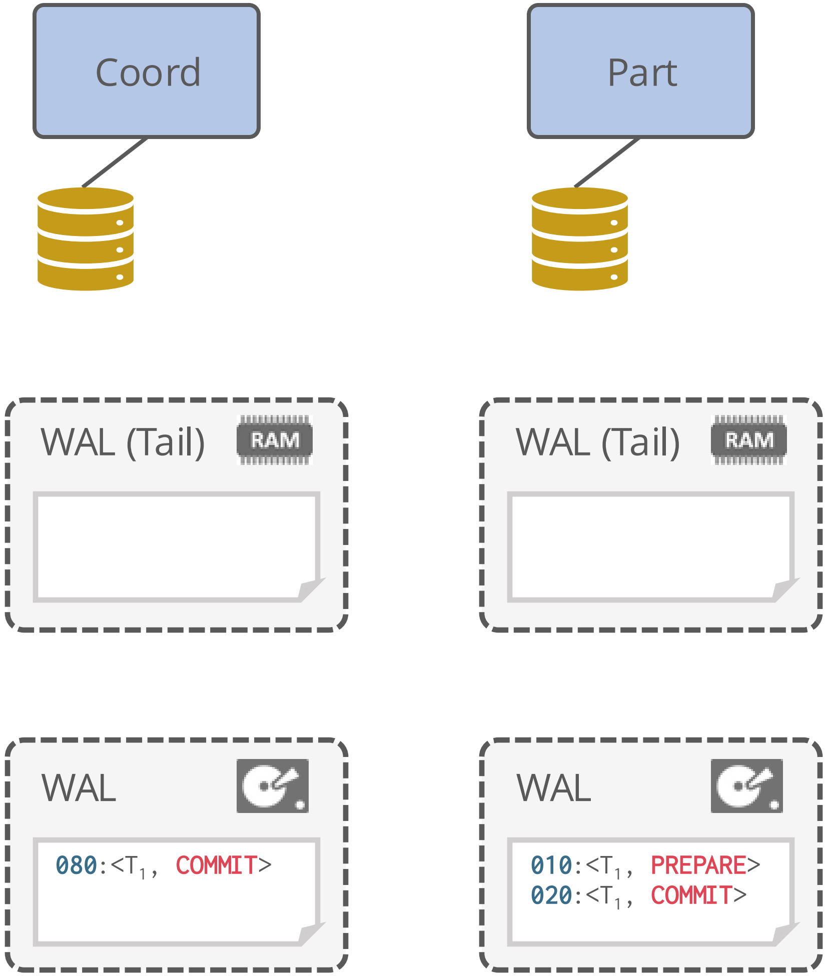
- Coordinator broadcasts result of vote
- **Participants make commit/abort record**
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 12

Phase 2:

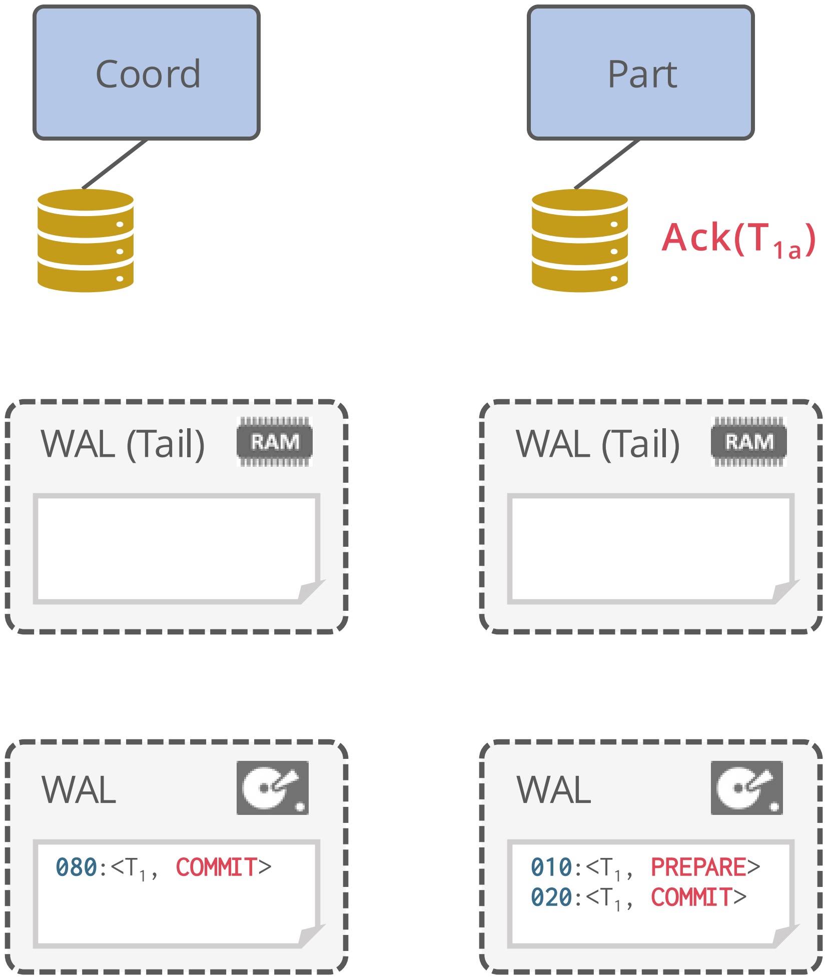
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- **Participants flush commit/abort record**
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 13

Phase 2:

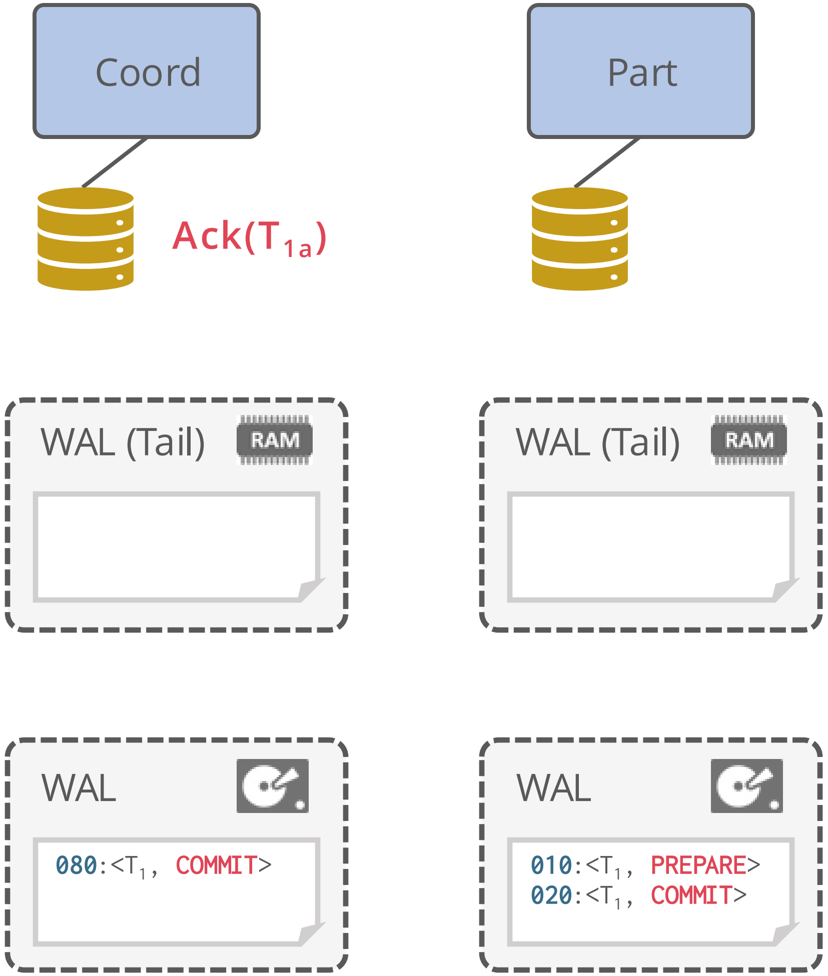
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- **Participants respond with Ack**
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 14

Phase 2:

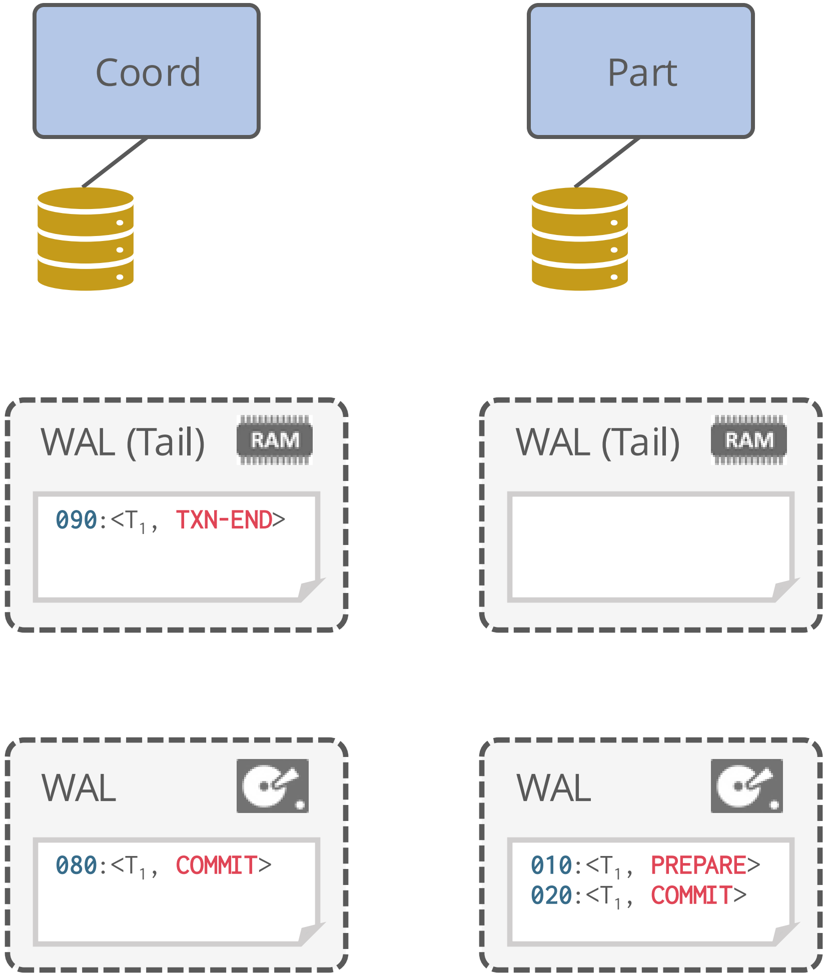
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- **Participants respond with Ack**
- Coordinator generates end record
- Coordinator flushes end record



One More Time, with Logging, Part 15

Phase 2:

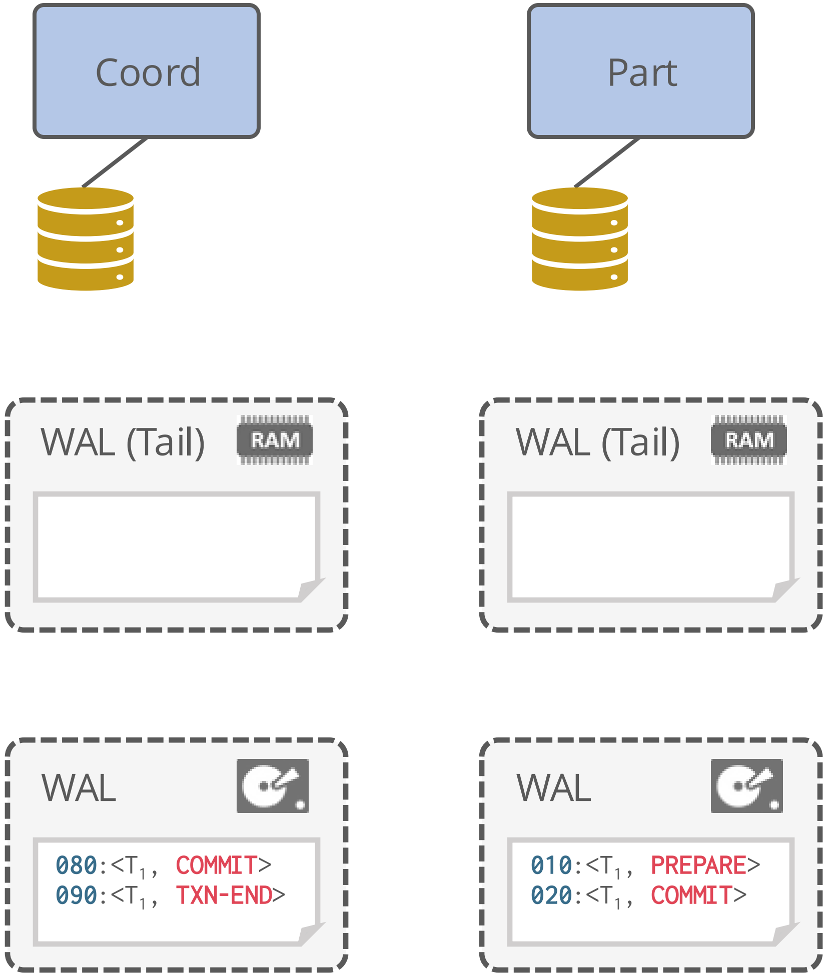
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- **Coordinator generates end record**
- Coordinator flushes end record



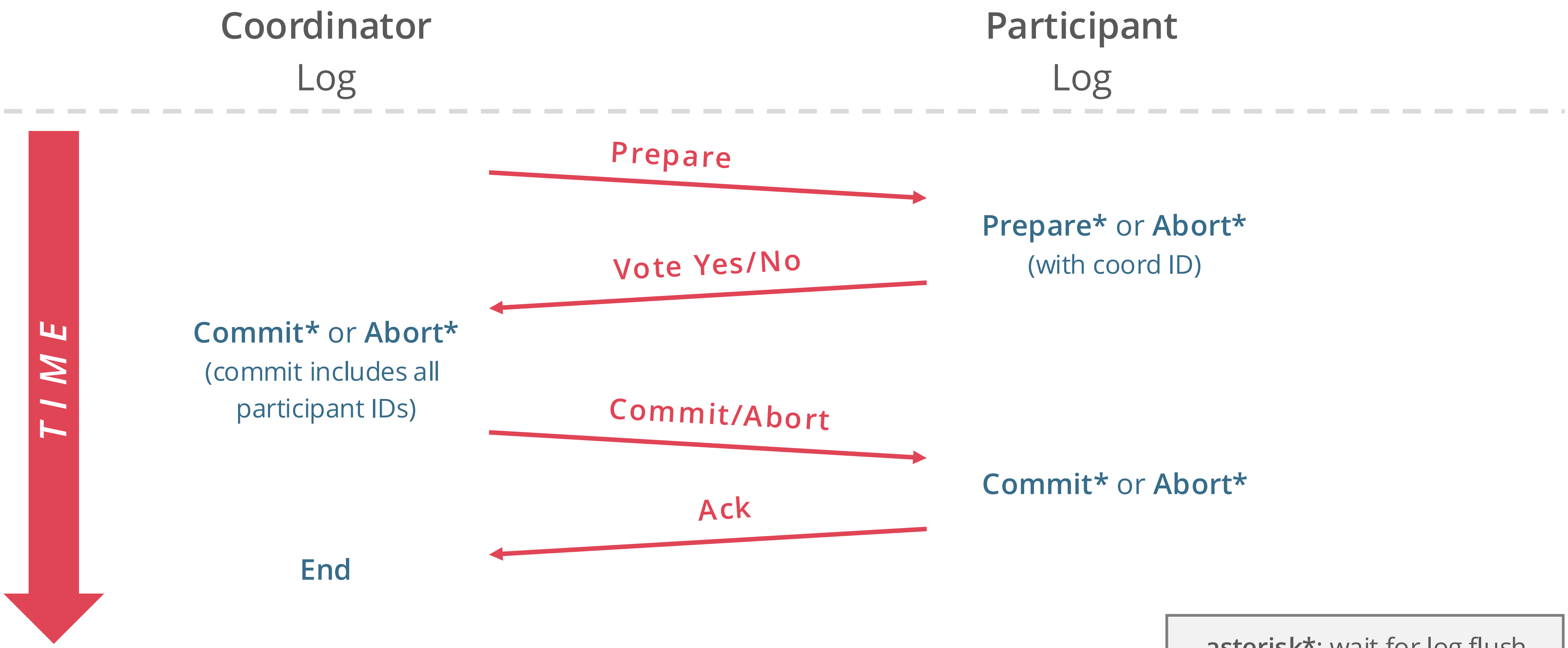
One More Time, with Logging, Part 16

Phase 2:

- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- **Coordinator flushes end record**



2PC In a Nutshell



asterisk*: wait for log flush before sending next msg

Outline

Distributed Locking

Distributed Deadlock Detection

Distributed Two-Phase Commit (2PC)

Recovery and 2PC

Failure Handling

Assume everybody recovers eventually

- Big assumption!
- Depends on WAL (and short downtimes)

Coordinator notices a Participant is down?

- If participant hasn't voted yet, coordinator aborts transaction
- If waiting for a commit Ack, hand to "recovery process"

Participant notices Coordinator is down?

- If it hasn't yet logged prepare, then abort unilaterally
- If it has logged prepare, hand to "recovery process"

Note

- Thinking a node is "down" may be incorrect!

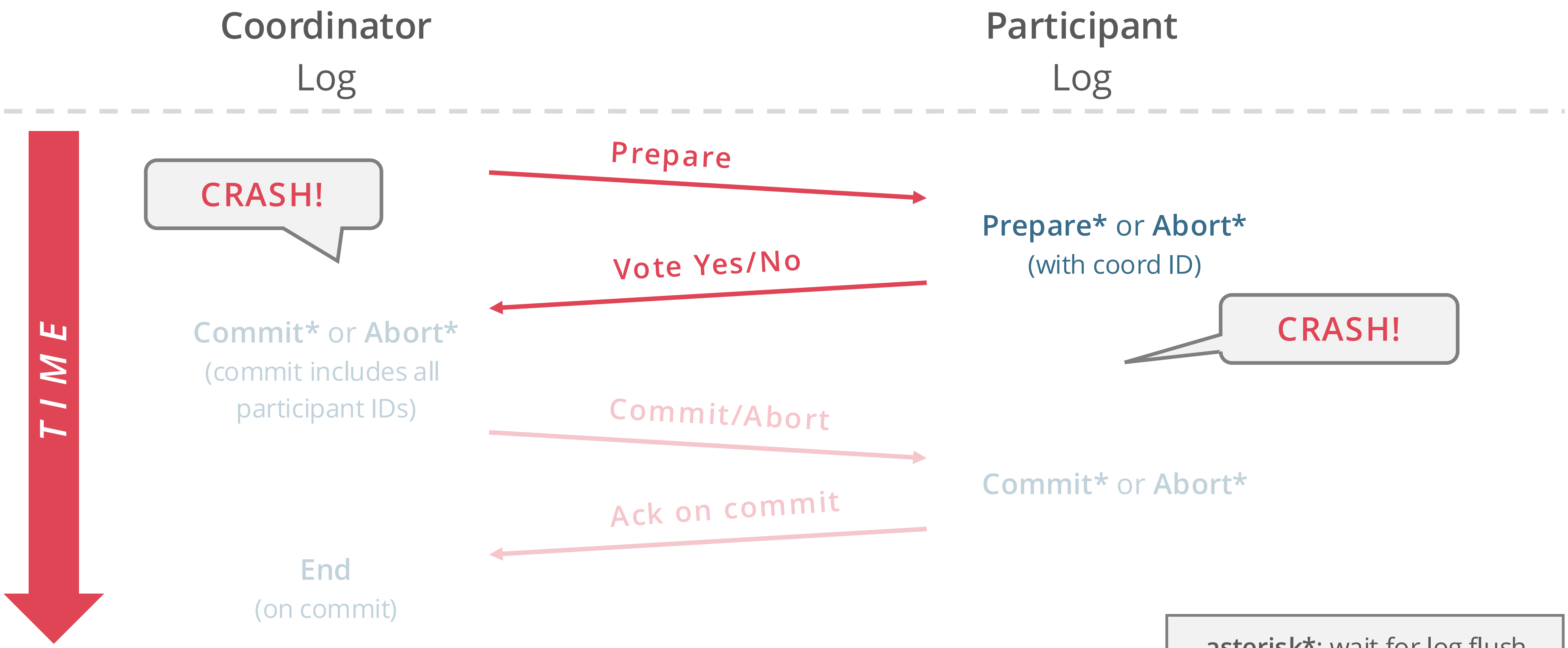
Integration with ARIES Recovery

- On recovery
  - Assume there's a "Recovery Process" at each node
  - It will be given tasks to do by the Analysis phase of ARIES
  - These tasks can run in the background (asynchronously)
- Note: multiple roles on a single node
  - Coordinator for some transactions, Participant for others

How Does Recovery Process Work?

- Coordinator recovery process gets inquiry from a “prepared” participant
  - If transaction table at coordinator says aborting/committing
    - Send appropriate response and continue protocol on both sides
  - If transaction table at coordinator says nothing: send ABORT
    - Only happens if coordinator had also crashed before writing commit/abort
    - Inquirer does the abort on its end

2PC In a Nutshell



asterisk*: wait for log flush before sending next msg

Recovery: Think it through

What happens when coordinator recovers?

- With “commit” and “end”?
- With just “commit”?
- With “abort”?

Commit iff coordinator logged a commit

What happens when participant recovers:

- With no prepare/commit/abort?
- With “prepare” and “commit”?
- With just “prepare”?
- With “abort”?

Recovery: Think it through

What happens when coordinator recovers?

- With "commit" and "end"? **Nothing**
- With just "commit"? **Rerun Phase 2!**
- With "abort"? **Nothing (presumed abort)**

Commit iff coordinator logged a commit

What happens when participant recovers:

- With no prepare/commit/abort? **Nothing (presumed abort)**
- With "prepare" and "commit"? **Send Ack to coordinator**
- With just "prepare"? **Send inquiry to coordinator**
- With "abort"? **Nothing (presumed abort)**

2PC + Strict 2PL

Ensure point-to-point messages are densely ordered

- 1,2,3,4,5…
- Dense per (sender/receiver/transaction ID)
- Receiver can detect anything missing or out-of-order
- Receiver buffers message k+1 until [1..k] received
- Effect: receiver considers messages in order

Commit:

- When a participant processes Commit request, it has all the locks it needs
- Flush log records and drop locks atomically

Abort:

- Its safe to abort autonomously, locally: no cascade
- Log appropriately to 2PC (presumed abort in our case)
- Perform local Undo, drop locks atomically

Availability Concerns

- What happens while a node is down?
  - Other nodes may be in limbo, holding locks
  - So certain data is unavailable
  - This may be bad...
- Dead Participants? Respawned by coordinator
  - Recover from log
  - And if the old participant comes back from the dead, just ignore it and tell it to recycle itself
- Dead Coordinator?
  - This is a problem!
  - 3-Phase Commit was an early attempt to solve it
  - Paxos Commit provides a more comprehensive solution
    - Gray + Lamport paper. Out of scope for this course

Summary

Data partitioning provides scale-up

Can also partition lock tables and logs

But need to do some global coordination:

- Deadlock detection: easy
- Commit: trickier

Two-phase commit is a classic distributed consensus protocol

- Logging/recovery aspects unique:
  - Many distributed protocols gloss over
- But 2PC is unavailable on any single failure
  - This is bad news for scale-up, because odds of failure go up with #machines
  - Paxos Commit addresses that problem