

# Advanced Database Systems

Spring 2025

Lecture #21:

## Distributed Transactions

R&G: Chapter 22

1

## PARALLEL / DISTRIBUTED DBMSs

2

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**

2

## PARALLEL VS. DISTRIBUTED DBMSs

3

### 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

3

## OBSERVATION

5

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?

5

## OUTLINE

6

Distributed Locking

Distributed Deadlock Detection

Distributed Two-Phase Commit (2PC)

Recovery and 2PC

6

## DISTRIBUTED CONCURRENCY CONTROL

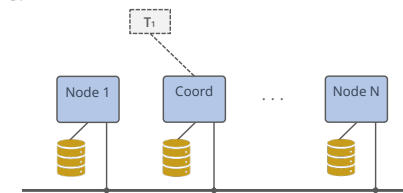
7

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



7

## WHERE IS THE LOCK TABLE?

8

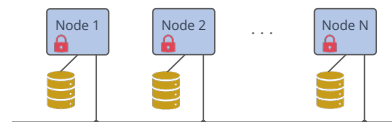
Typical design: Locks partitioned with the data

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



8

## WHERE IS THE LOCK TABLE?, PART 2

9

Typical design: Locks partitioned with the data

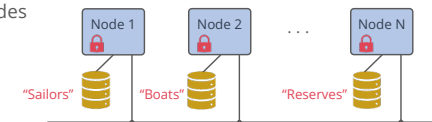
Independent: each node manages "its own" lock table

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



9

## WHERE IS THE LOCK TABLE?, PART 3

10

Typical design: Locks partitioned with the data

Independent: each node manages "its own" lock table

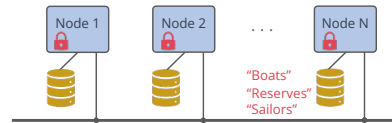
Works for objects that fit on one node (pages, tuples)

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



10

## IGNORE GLOBAL LOCKS FOR A MOMENT...

11

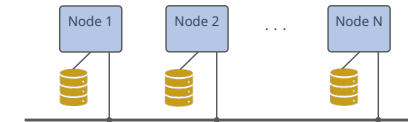
Every node does its own locking

Clean and efficient

"Global" issues remain:

Deadlock

Commit/Abort



11

## OUTLINE

12

Distributed Locking

Distributed Deadlock Detection

Distributed Two-Phase Commit (2PC)

Recovery and 2PC

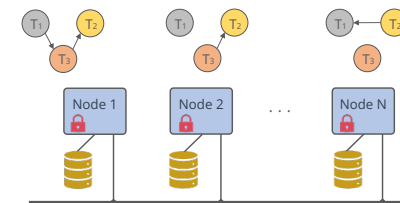
12

## WHAT COULD GO WRONG? #1

13

Deadlock detection

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



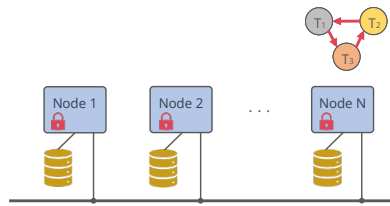
13

## WHAT COULD GO WRONG? #1, PART 2

14

### Deadlock detection

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



14

## OUTLINE

15

Distributed Locking

Distributed Deadlock Detection

Distributed Two-Phase Commit (2PC)

Recovery and 2PC

15

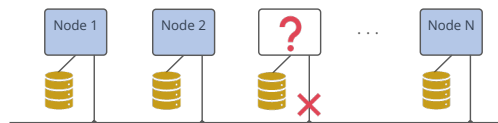
## WHAT COULD GO WRONG? #2

16

### Failures/Delays: Nodes

Commit? Abort?

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



16

## WHAT COULD GO WRONG? #2, PART 2

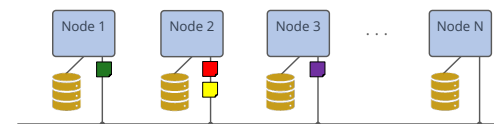
17

### Failures/Delays: Nodes

### Failures/Delays: Messages

Non-deterministic reordering per channel, interleaving across channels

"Lost" (very delayed) messages



17

## WHAT COULD GO WRONG? #2, PART 3

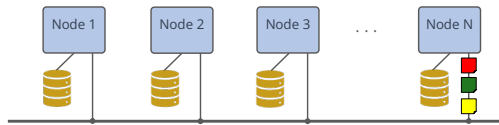
18

Failures/Delays: Nodes

Failures/Delays: Messages

Non-deterministic reordering per channel, interleaving across channels

"Lost" (very delayed) messages



18

## WHAT COULD GO WRONG? #2, PART 4

19

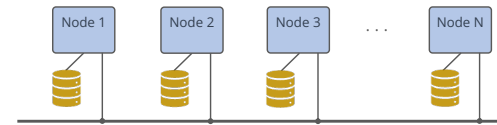
Failures/Delays: Nodes

Failures/Delays: Messages

Non-deterministic reordering per channel, interleaving across channels

"Lost" (very delayed) messages

How do all nodes agree on Commit vs. Abort?



19

## BASIC IDEA: DISTRIBUTED VOTING

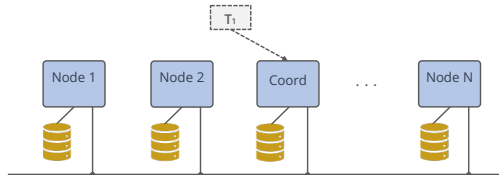
20

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



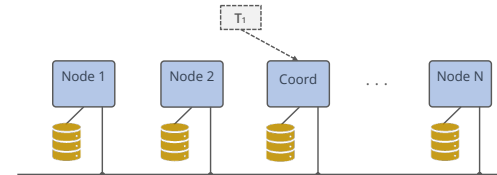
20

## DISTRIBUTED VOTING? HOW?

21

How do we implement distributed voting?!

In the face of message/node failure/delay?



21

## 2-PHASE COMMIT

22

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....

22

## 2-PHASE COMMIT, PART 1

23

Phase 1:

Coordinator tells participants to "prepare"

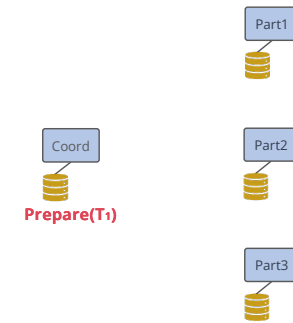
Participants respond with yes/no votes

Unanimity required for commit!

Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



23

## 2-PHASE COMMIT, PART 2

24

Phase 1:

Coordinator tells participants to "prepare"

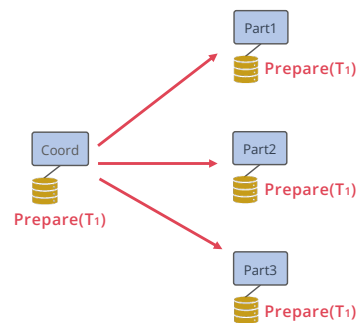
Participants respond with yes/no votes

Unanimity required for commit!

Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



24

## 2-PHASE COMMIT, PART 3

25

Phase 1:

Coordinator tells participants to "prepare"

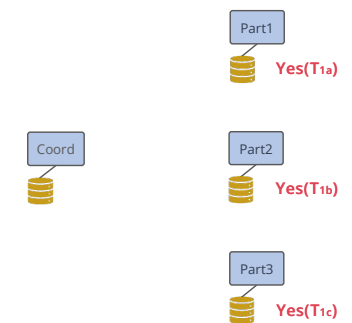
Participants respond with yes/no votes

Unanimity required for commit!

Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



25

## 2-PHASE COMMIT, PART 4

26

### Phase 1:

Coordinator tells participants to "prepare"

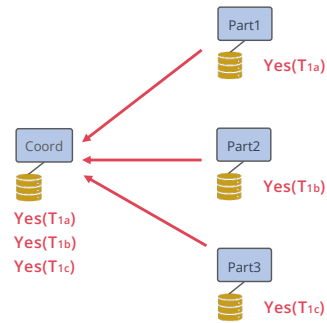
Participants respond with yes/no votes

Unanimity required for commit!

### Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



26

## 2-PHASE COMMIT, PART 5

27

### Phase 1:

Coordinator tells participants to "prepare"

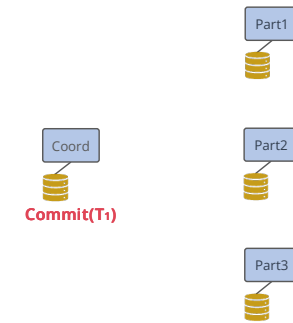
Participants respond with yes/no votes

Unanimity required for commit!

### Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



27

## 2-PHASE COMMIT, PART 6

28

### Phase 1:

Coordinator tells participants to "prepare"

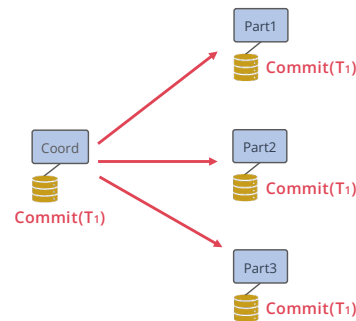
Participants respond with yes/no votes

Unanimity required for commit!

### Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



28

## 2-PHASE COMMIT, PART 7

29

### Phase 1:

Coordinator tells participants to "prepare"

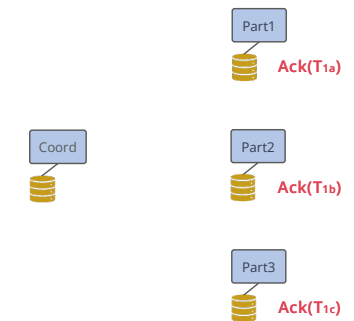
Participants respond with yes/no votes

Unanimity required for commit!

### Phase 2:

Coordinator disseminates result of the vote

Participants respond with Ack



29

## 2-PHASE COMMIT, PART 8

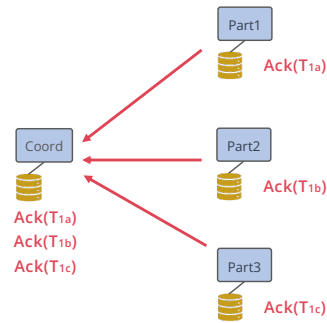
30

### 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**



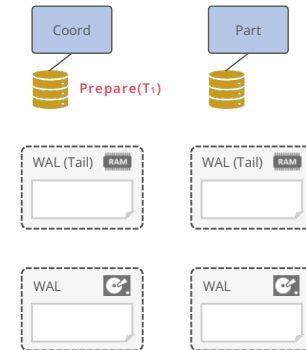
30

## ONE MORE TIME, WITH LOGGING

31

### 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



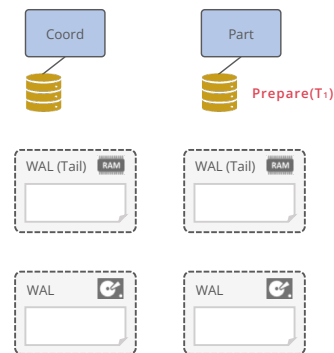
31

## ONE MORE TIME, WITH LOGGING, PART 2

32

### 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



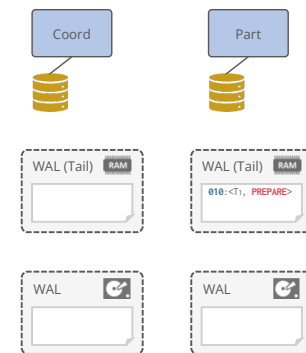
32

## ONE MORE TIME, WITH LOGGING, PART 3

33

### 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



33

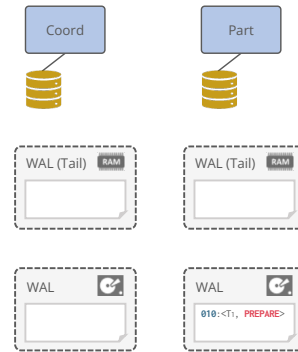


## ONE MORE TIME, WITH LOGGING, PART 4

34

### 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



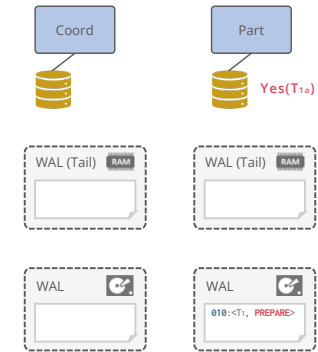
34

## ONE MORE TIME, WITH LOGGING, PART 5

35

### 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



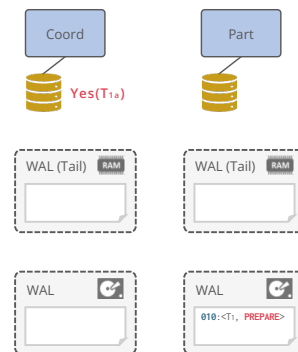
35

## ONE MORE TIME, WITH LOGGING, PART 6

36

### 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



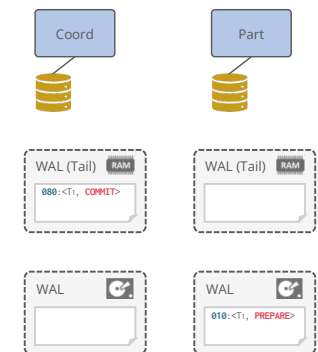
36

## ONE MORE TIME, WITH LOGGING, PART 7

37

### 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



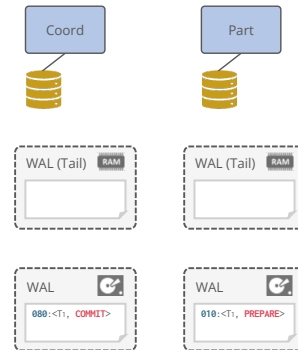
37

## ONE MORE TIME, WITH LOGGING, PART 8

38

### 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**



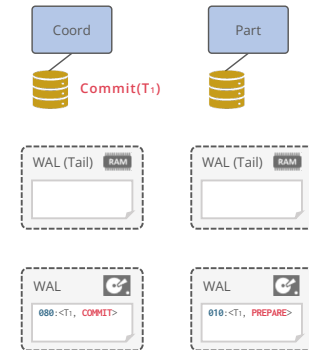
38

## ONE MORE TIME, WITH LOGGING, PART 9

39

### 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



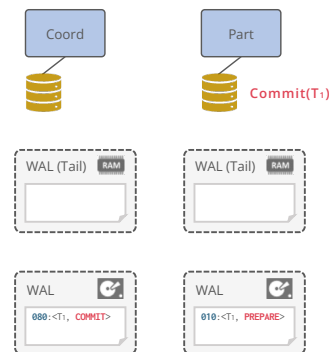
39

## ONE MORE TIME, WITH LOGGING, PART 10

40

### 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



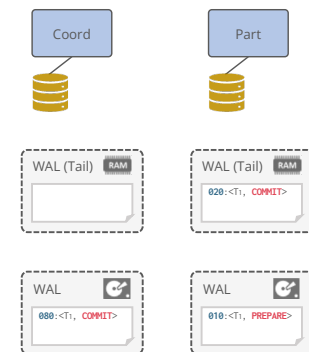
40

## ONE MORE TIME, WITH LOGGING, PART 11

41

### 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

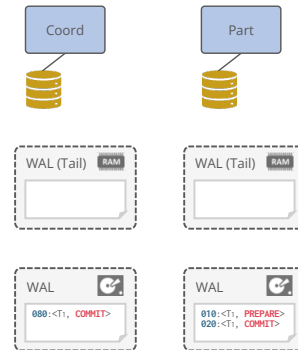


41

## ONE MORE TIME, WITH LOGGING, PART 12 <sup>42</sup>

### 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

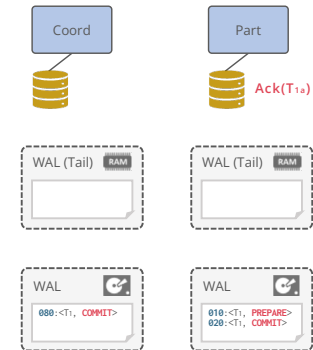


42

## ONE MORE TIME, WITH LOGGING, PART 13 <sup>43</sup>

### 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

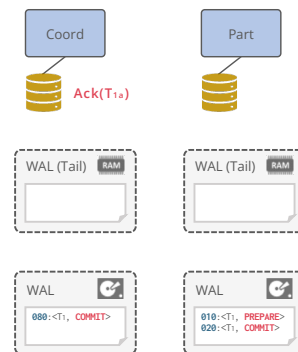


43

## ONE MORE TIME, WITH LOGGING, PART 14 <sup>44</sup>

### 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

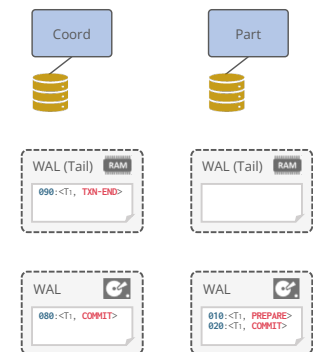


44

## ONE MORE TIME, WITH LOGGING, PART 15 <sup>45</sup>

### 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



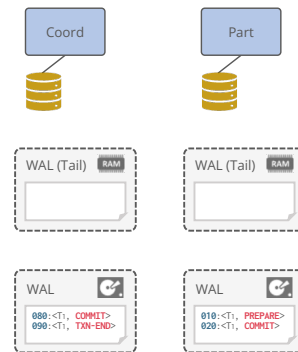
45

## ONE MORE TIME, WITH LOGGING, PART 16

46

### Phase 2:

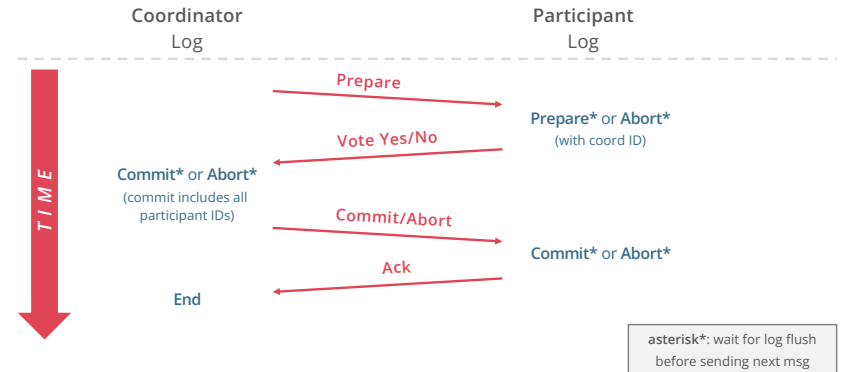
- Coordinator broadcasts result of vote
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record**



46

## 2PC IN A NUTSHELL

47



47

## OUTLINE

48

- Distributed Locking
- Distributed Deadlock Detection
- Distributed Two-Phase Commit (2PC)
- Recovery and 2PC

48

## FAILURE HANDLING

49

- 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!

49

## INTEGRATION WITH ARIES RECOVERY

50

### 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

50

## HOW DOES RECOVERY PROCESS WORK?

53

### 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**

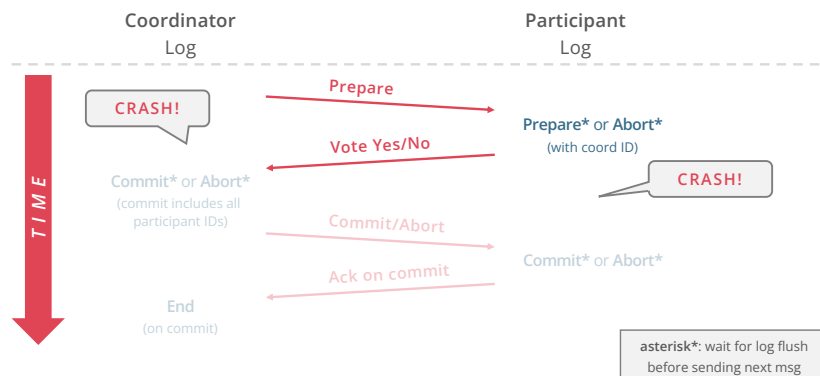
Only happens if coordinator had also crashed before writing commit/abort

Inquirer does the abort on its end

53

## 2PC IN A NUTSHELL

54



54

## RECOVERY: THINK IT THROUGH

55

### 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"?

55

## RECOVERY: THINK IT THROUGH

56

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)**

56

## 2PC + STRICT 2PL

57

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

57

## AVAILABILITY CONCERNS

58

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

58

## SUMMARY

59

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

59