Transaction management in DDBMS

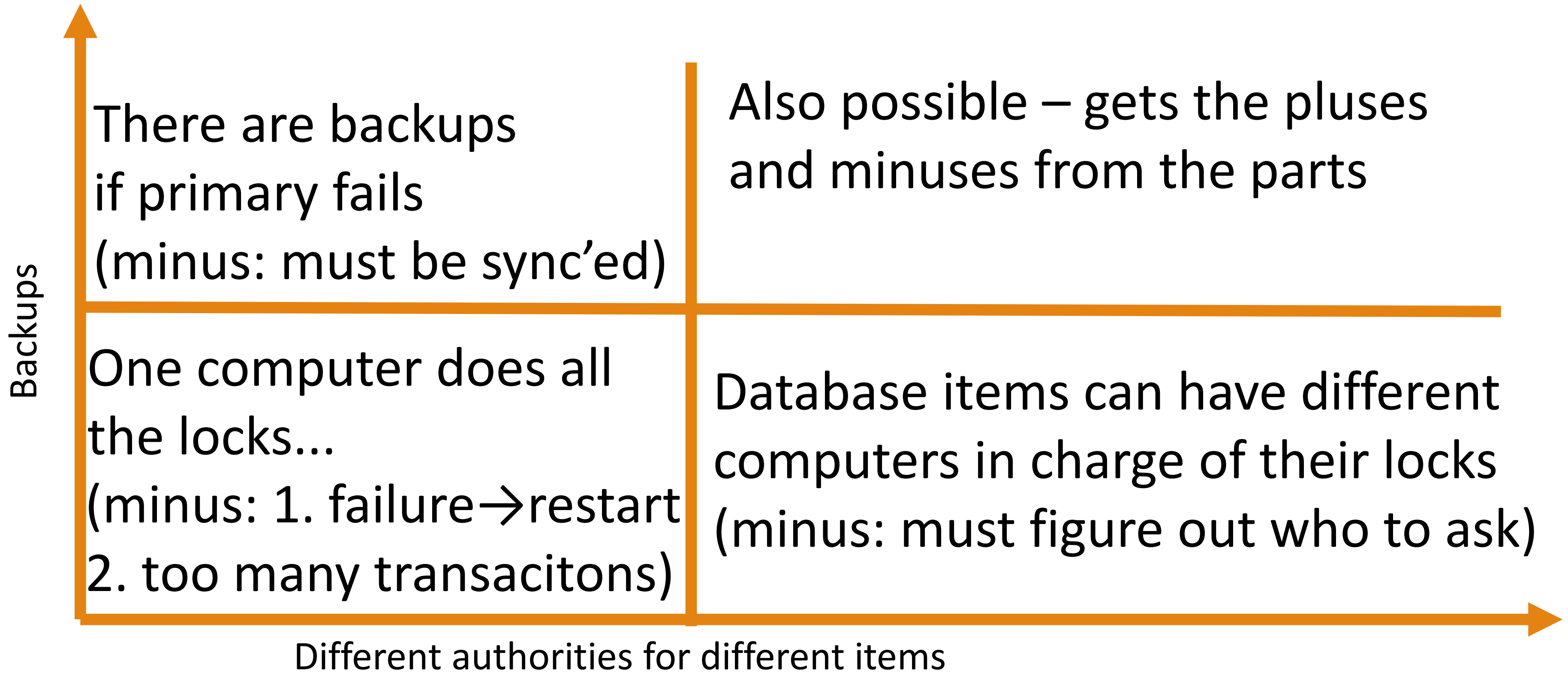
Overview over this video

Distributed databases creates new issues for transaction management

In this video we see what some of them are and ways to deal with these

Concurrency control in DDBMS

For full isolation/consistency, often based on locks:



There are backups if primary fails (minus: must be sync’ed)

Also possible – gets the pluses and minuses from the parts

One computer does all the locks... (minus: 1. failure→restart 2. too many transacitons)

Database items can have different computers in charge of their locks (minus: must figure out who to ask)

Concurrency control based on voting

Another approach to locks (instead of having a single designated computer granting them):

Voting!

Idea:

- Each site with a copy of an item has a local lock that it can grant transactions for that item
- If a transaction gets over half the local locks for an item, it has a global lock on the item
  - If so, it must tell the sites with a copy that it has the lock
  - If it takes too long, it must stop trying to get the lock
- Plus: Much more distributed than the non-voting approach
- Minus: Requires more communication

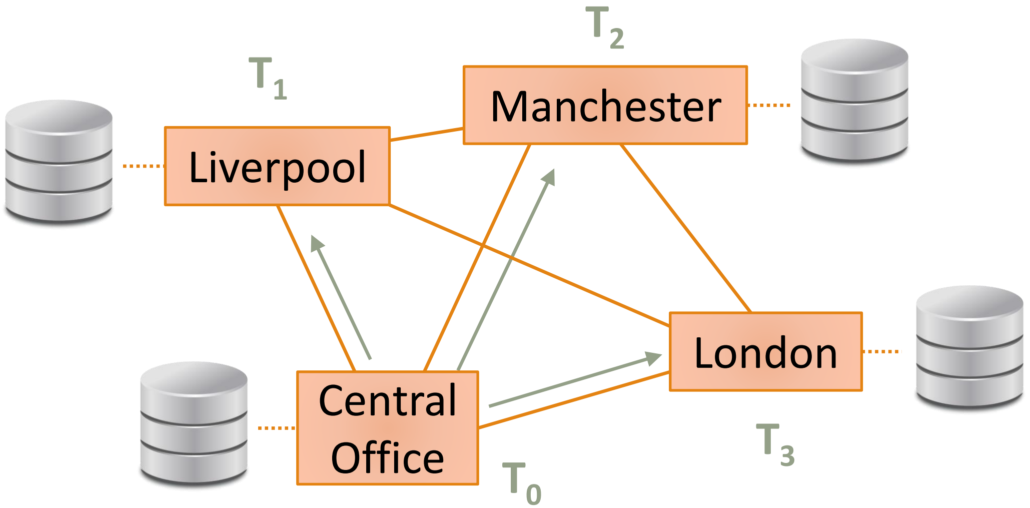
Recovery in DDBMS

Transactions in Distributed Databases

Let’s revisit our CS_Store chain…

At central office:

- Determine inventory for product X at each site
- Move product X between stores to balance inventory

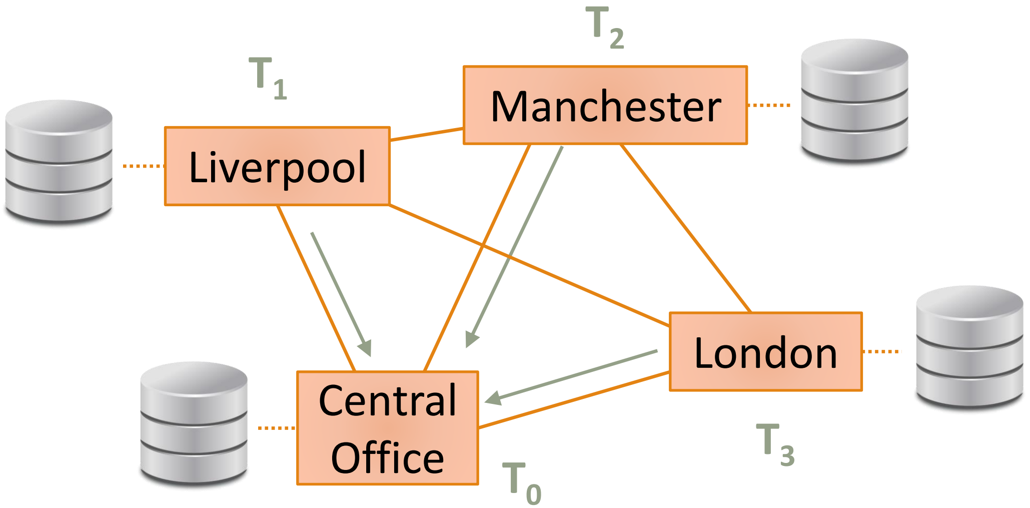


Global transaction T

- Starts **local transaction T_0** at central office
- T_0 instructs other sites to start **local transactions T_1, T_2, T_3**

Transactions in Distributed Databases

Let’s revisit our CS_Store chain…



At central office:

- Determine inventory for product X at each site
- Move product X between stores to balance inventory

Global transaction T

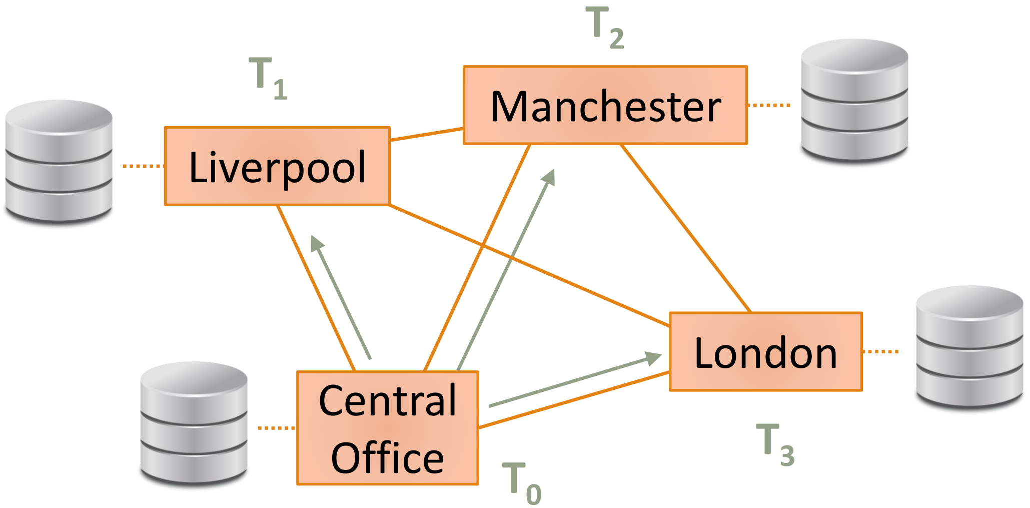
- Starts **local transaction T_0** at central office
- T_0 instructs other sites to start **local transactions T_1, T_2, T_3**
- T_1, T_2, T_3 find out inventory for product X at sites & send it back to T_0

Transactions in Distributed Databases

Let's revisit our CS_Store chain…

At central office:

- Determine inventory for product X at each site
- Move product X between stores to balance inventory



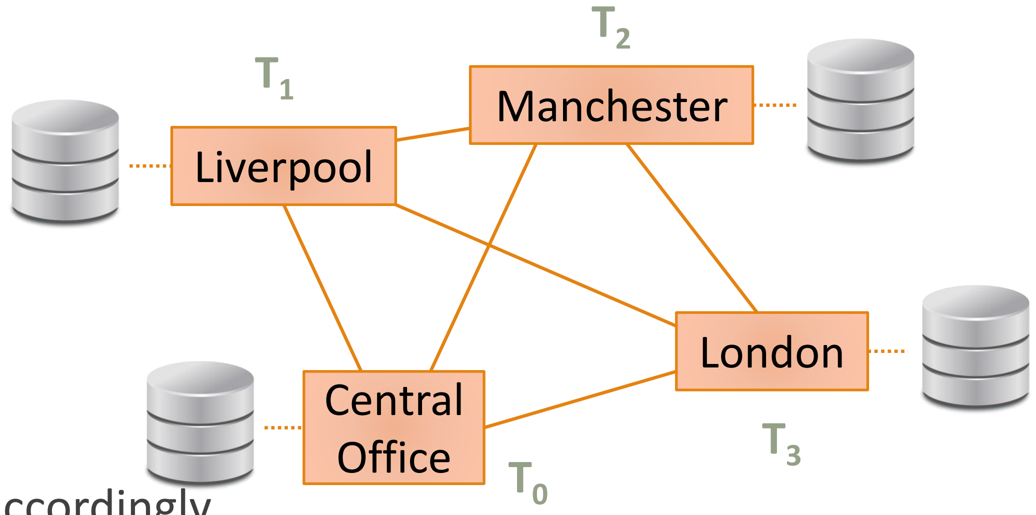
Global transaction T

- Starts **local transaction T_0** at central office
- T_0 instructs other sites to start **local transactions T_1, T_2, T_3**
- T_1, T_2, T_3 find out inventory for product X at sites & send it back to T_0
- T_0 determines how to move product X between sites
- T_0 instructs T_1, T_2, T_3 to move product X accordingly

Violation of Atomicity

Global transaction T

- Start T_0 at central office
- T_0 instructs other sites to start T_1, T_2, T_3
- T_1, T_2, T_3 report inventory for product X
- T_0 determines how to move product X
- T_0 instructs T_1, T_2, T_3 to move product X accordingly



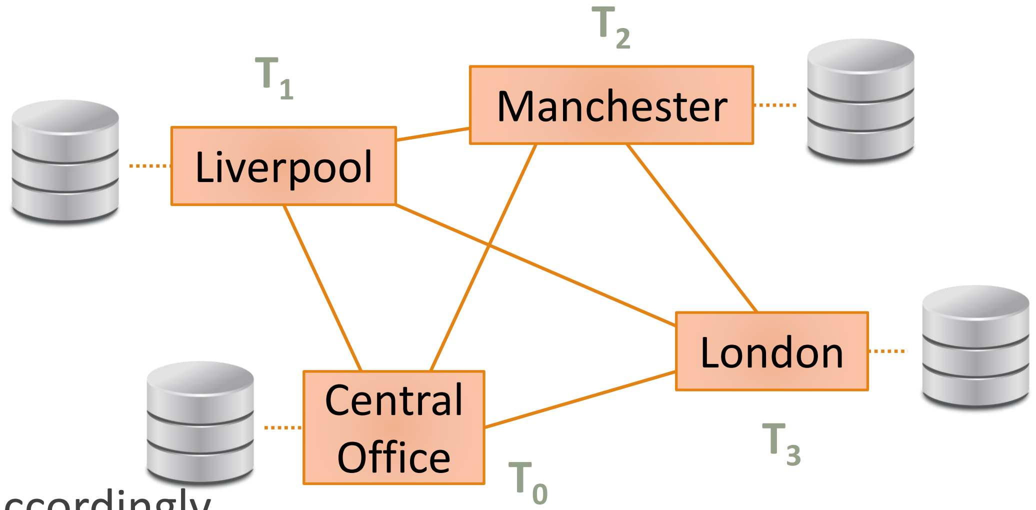
Atomicity:

- Can assume to be enforced at each node *locally*
- Could be violated *globally*

Problems With Failing Nodes

Global transaction T

- Start T_0 at central office
- T_0 instructs other sites to start T_1, T_2, T_3
- T_1, T_2, T_3 report inventory for product X
- T_0 determines how to move product X
- T_0 instructs T_1, T_2, T_3 to move product X accordingly



Nodes can fail during execution of T

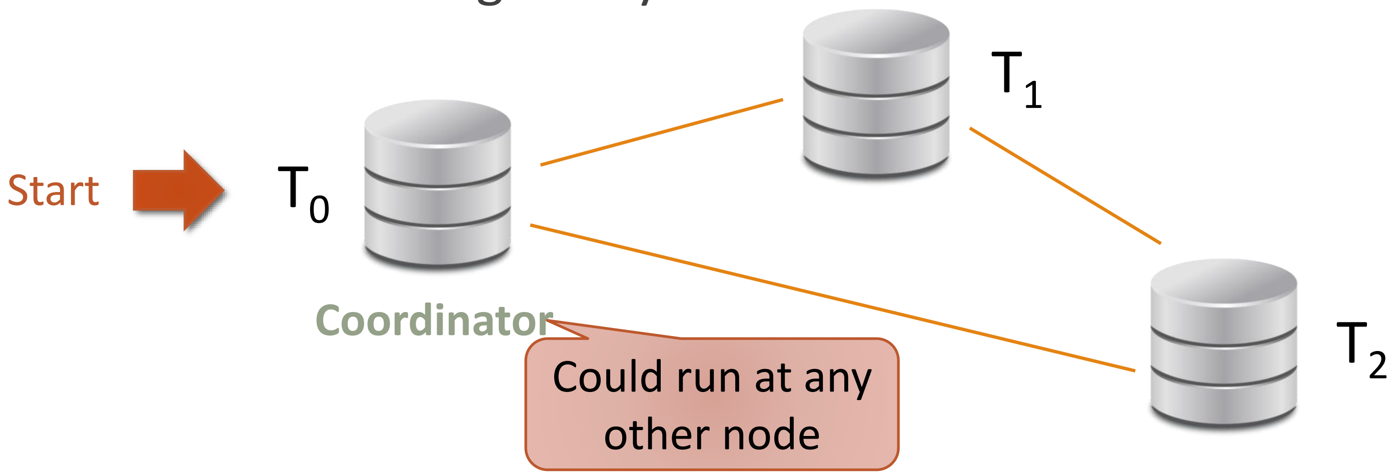
- Should we abort or wait?
- What about the failing node after recovery?

Distributed Commit

Not related to 2PL!

Two-Phase Commit Protocol

Coordinates commit actions globally



Could run at any other node

Coordinator: executed at some node & decides if and when local transactions can commit

Logging: at each node locally

- Messages sent to & received from other nodes are logged, too!

The Two Phases

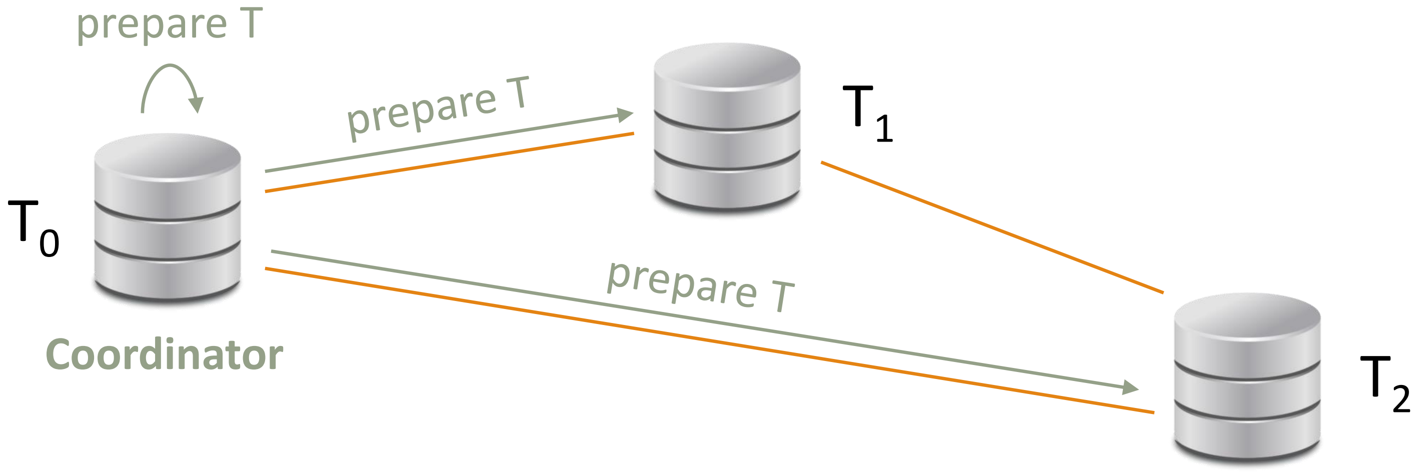
Phase 1: Decide when to commit or abort

Phase 2: Commit or abort

Phase 1: When To Commit?

Coordinator: ask nodes if they want to commit

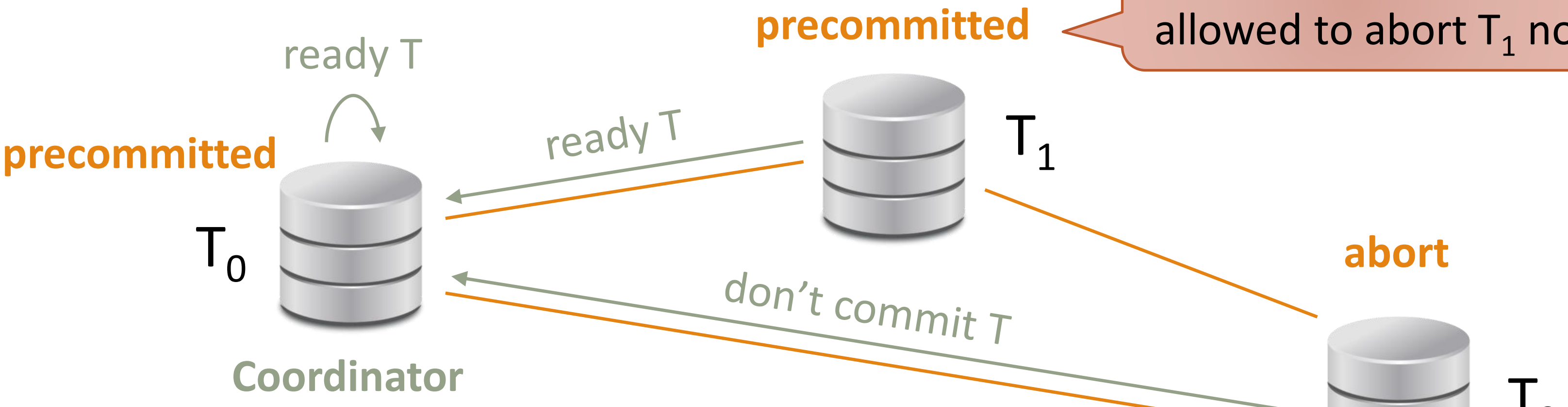
- Send message "prepare T"



Phase 1: Ready to Commit?

Coordinator: ask nodes if they want to commit

- Send message “prepare T”



Only the coordinator is allowed to abort T_1 now

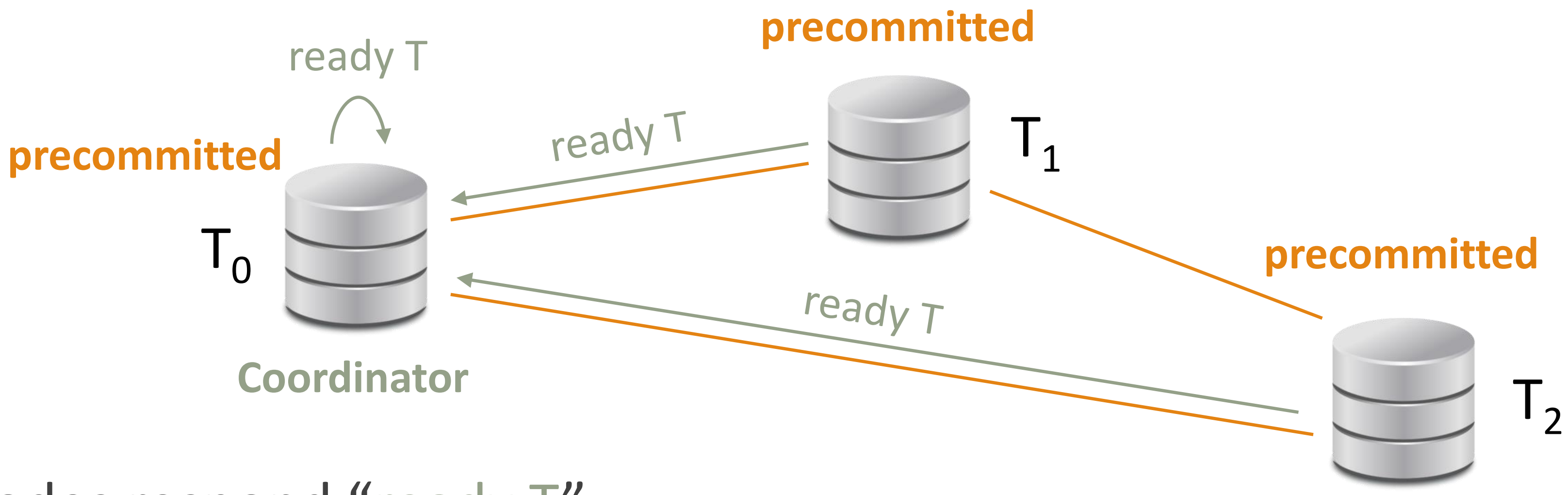
At each node: decide whether to commit or abort

- If commit → go into **precommitted** state & send back “ready T”
- If abort → send back “don’t commit T” and abort local transaction
- Can delay, but must decide eventually

Phase 2: Let’s Do It

Coordinator: waits for responses of nodes

- Assume nodes who don’t reply before a given timeout wish to abort

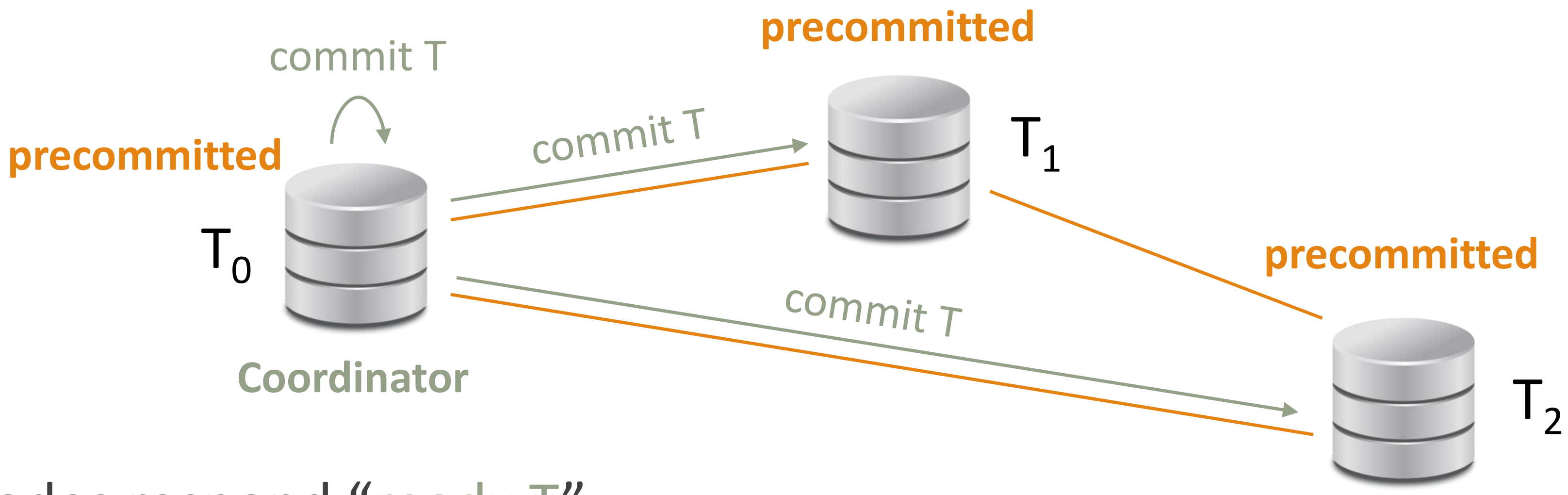


If all nodes respond “ready T”

Phase 2: Let’s Do It

Coordinator: waits for responses of nodes

- Assume nodes who don’t reply before a given timeout wish to abort



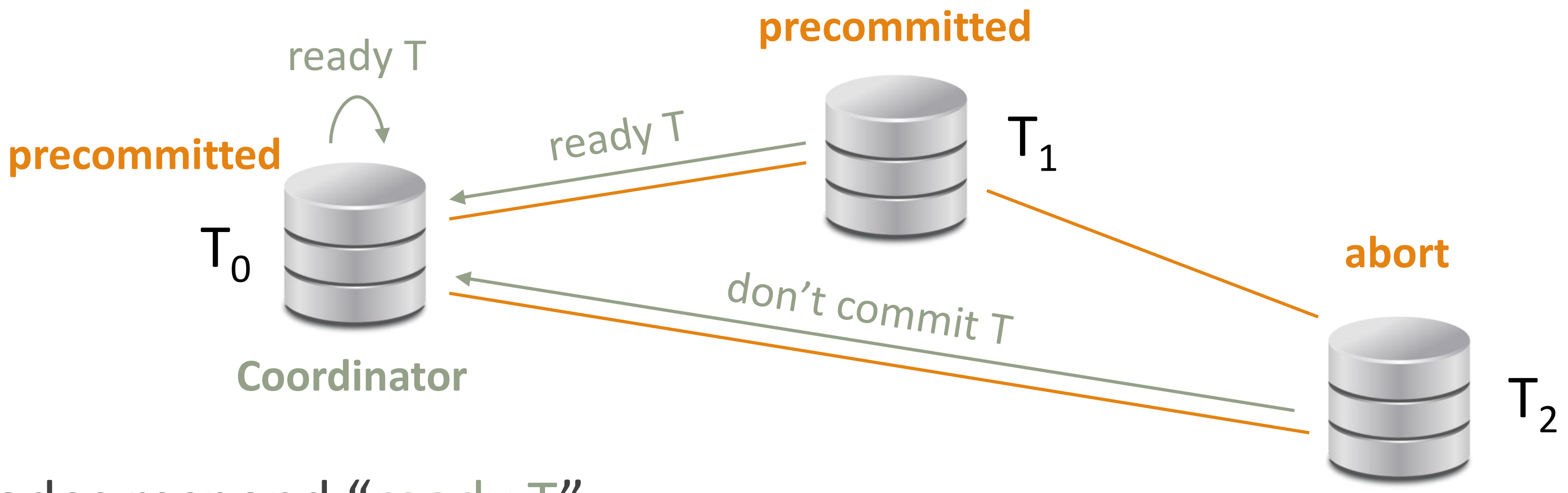
If all nodes respond “ready T”

- Send ”commit T” to all nodes → nodes commit

Phase 2: Let’s Do It

Coordinator: waits for responses of nodes

- Assume nodes who don’t reply before a given timeout wish to abort



If all nodes respond “ready T”

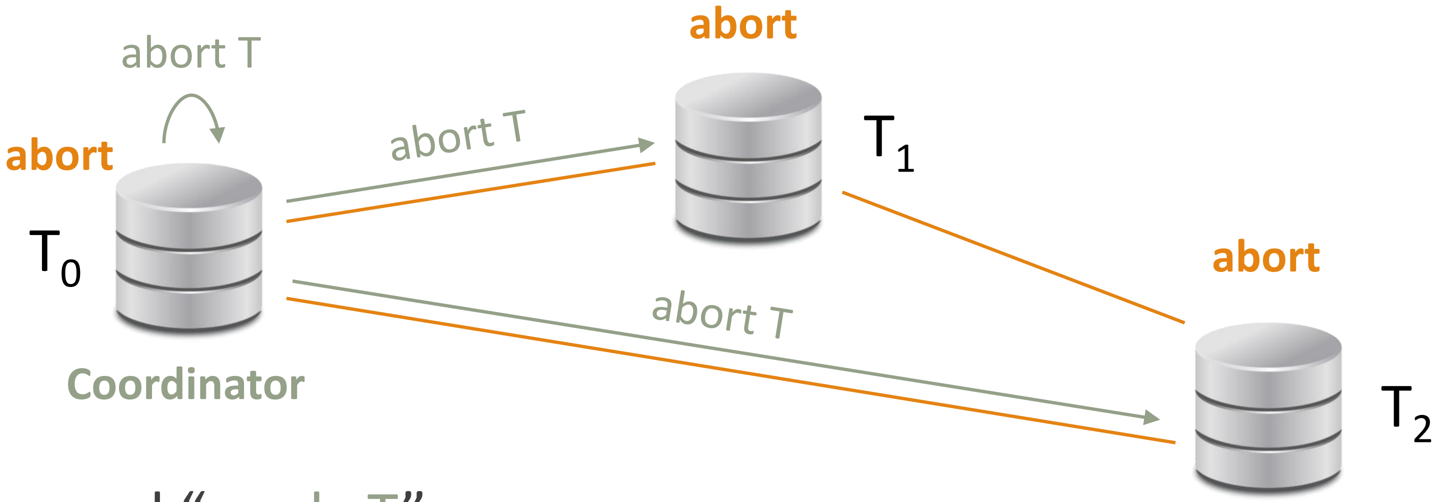
- Send ”commit T” to all nodes → nodes commit

If some node responds “don’t commit T”

Phase 2: Let’s Do It

Coordinator: waits for responses of nodes

- Assume nodes who don’t reply before a given timeout wish to abort



If all nodes respond “ready T”

- Send ”commit T” to all nodes → nodes commit

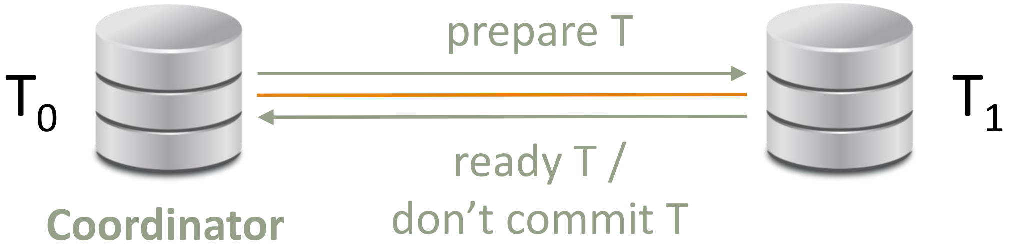
If some node responds “don’t commit T”

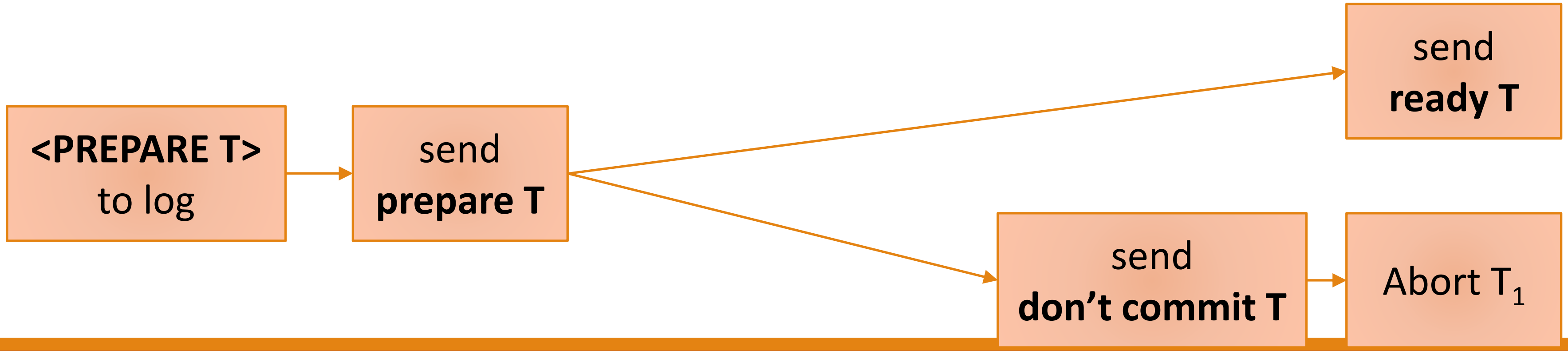
- Send “abort T” to all nodes → nodes abort

Logging: Phase 1

Again, we have to be careful in which order to write to disk and to the log

Phase 1:

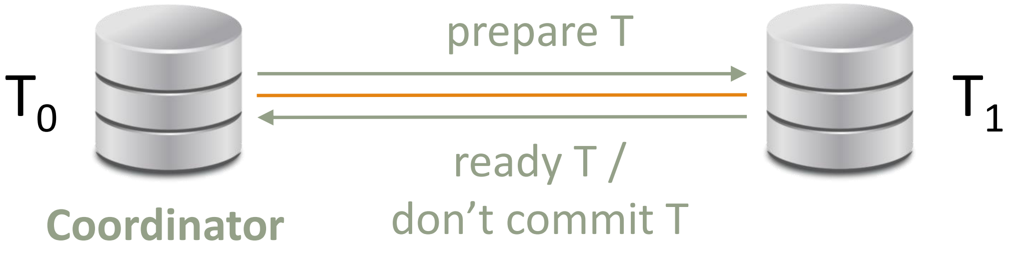


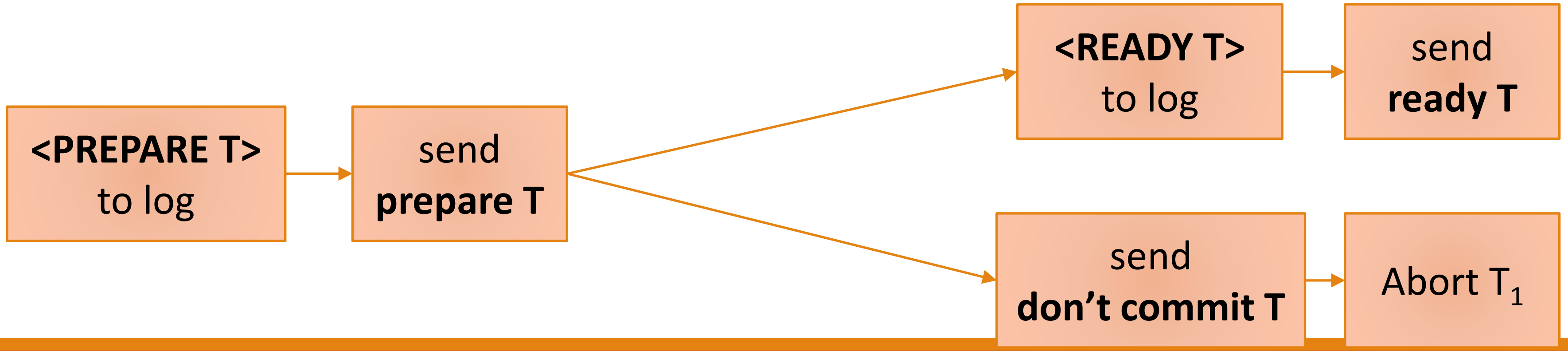


Logging: Phase 1

Again, we have to be careful in which order to write to disk and to the log

Phase 1:





<PREPARE T> to log

send **prepare T**

<READY T> to log

send **ready T**

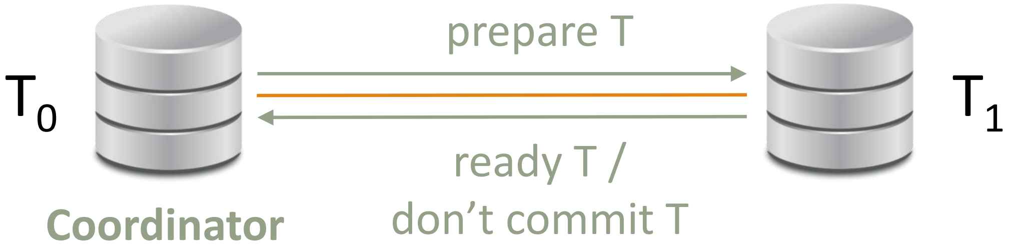
send **don’t commit T**

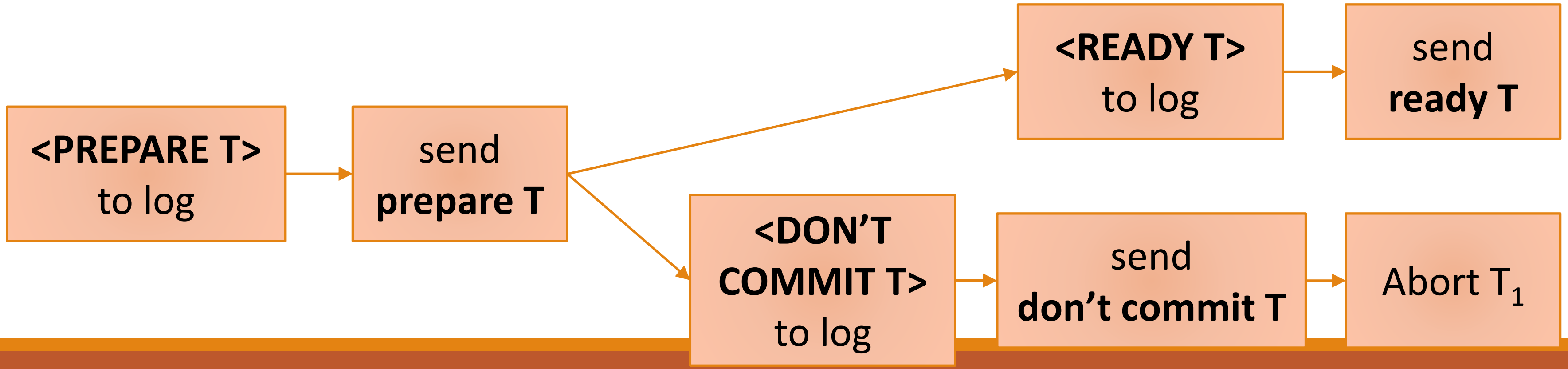
Abort T_1

Logging: Phase 1

Again, we have to be careful in which order to write to disk and to the log

Phase 1:





<PREPARE T> to log

send **prepare T**

<READY T> to log

send **ready T**

<DON'T COMMIT T> to log

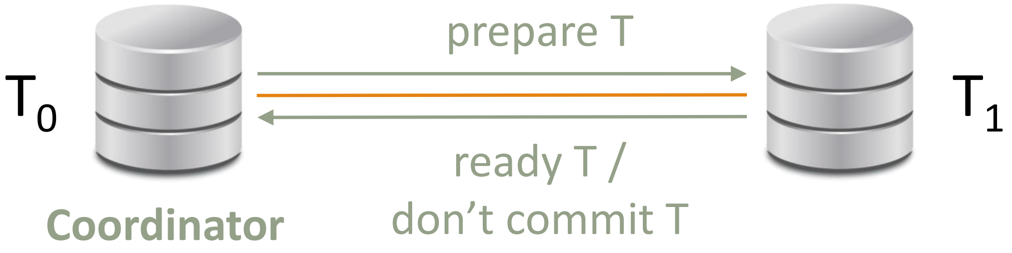
send **don't commit T**

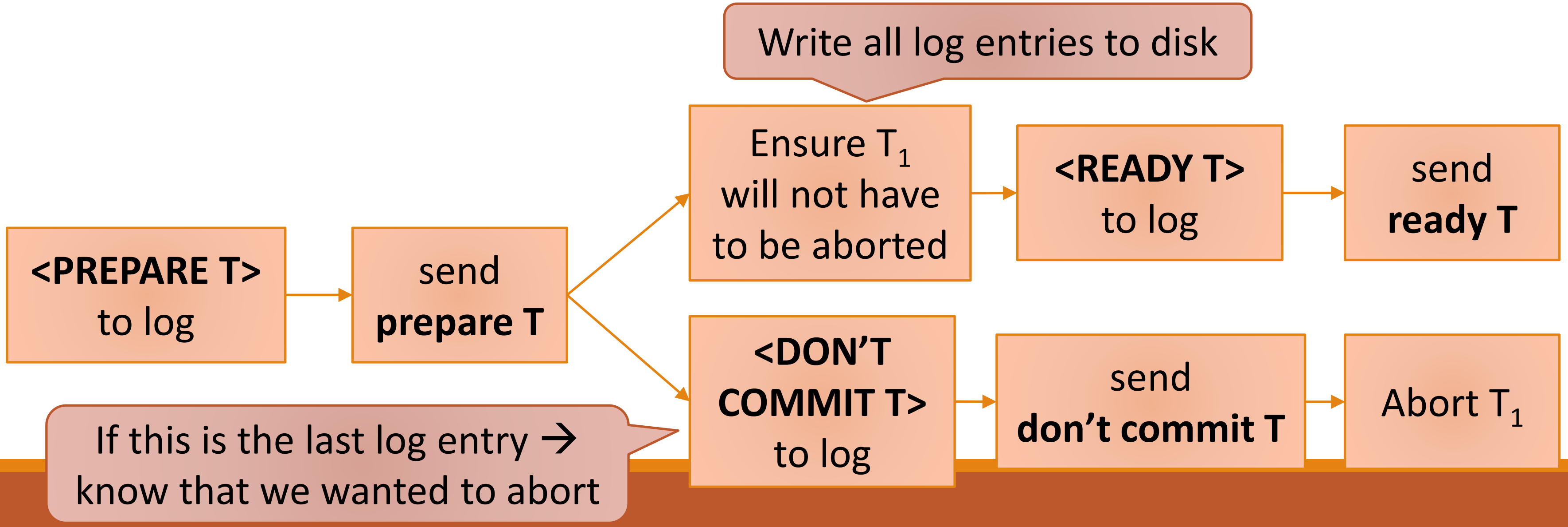
Abort T_1

Logging: Phase 1

Again, we have to be careful in which order to write to disk and to the log

Phase 1:





Write all log entries to disk

<PREPARE T> to log

send **prepare T**

Ensure T_1 will not have to be aborted

<READY T> to log

send **ready T**

<DON'T COMMIT T> to log

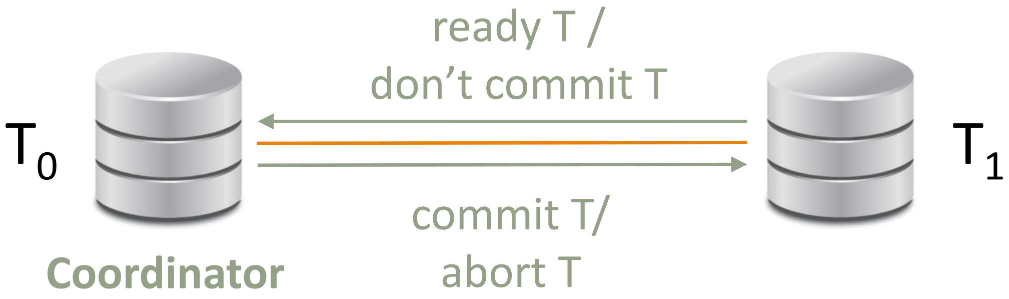
send **don't commit T**

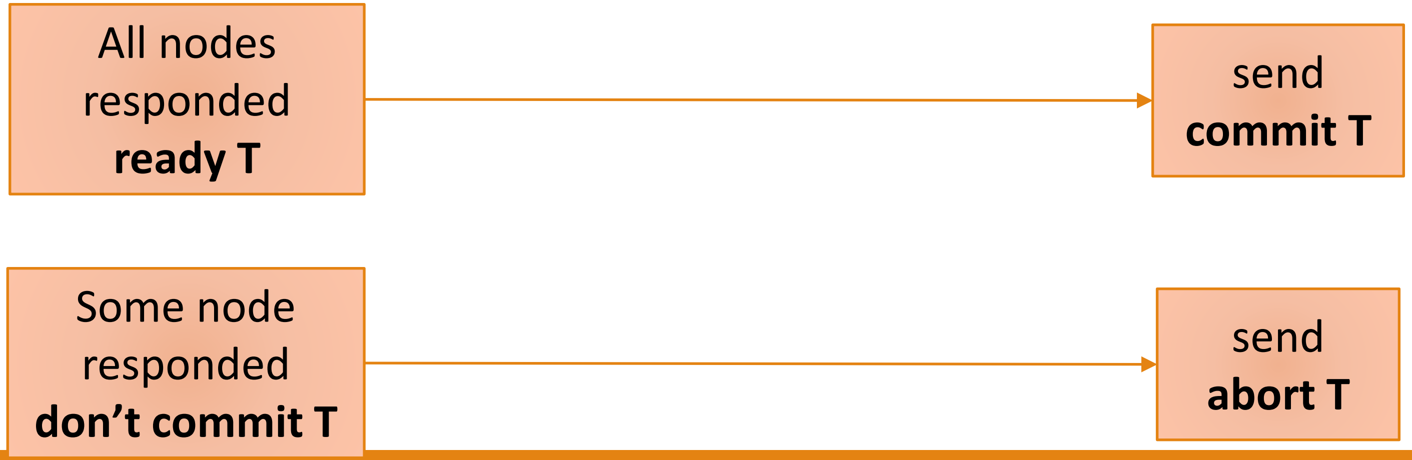
Abort T_1

If this is the last log entry → know that we wanted to abort

Logging: Phase 2

Phase 2:



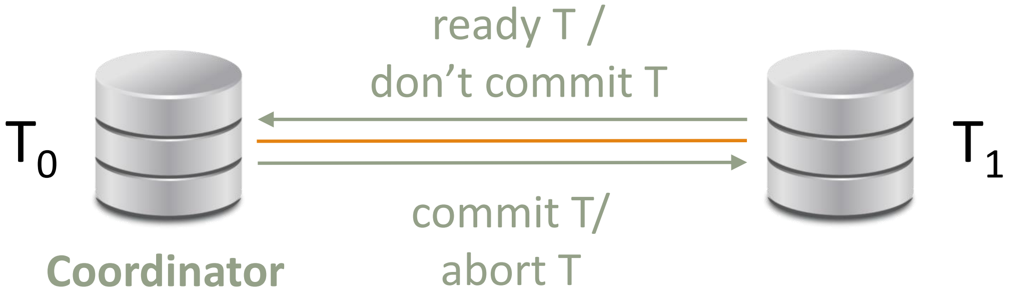


All nodes responded **ready T** → send **commit T**

Some node responded **don’t commit T** → send **abort T**

Logging: Phase 2

Phase 2:



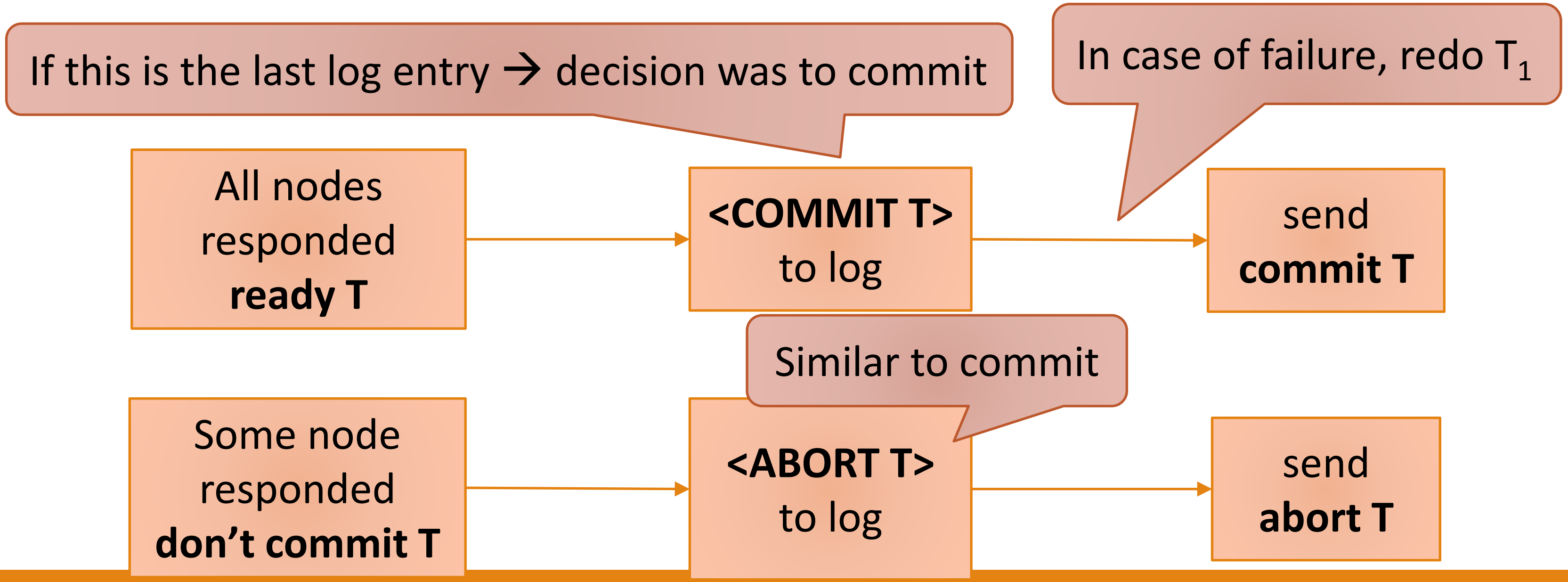
T_0

Coordinator

ready T /
don't commit T

commit T/
abort T

T_1



If this is the last log entry → decision was to commit

In case of failure, redo T_1

All nodes responded **ready T**

<COMMIT T> to log

send **commit T**

Similar to commit

Some node responded **don't commit T**

<ABORT T> to log

send **abort T**

Three-Phase Commit Protocol

Improvement of Two-Phase Commit

Can deal with the situation that in phase 2, the coordinator and some transaction crash, while everybody else are in precommited state

Idea: divide phase 2 into two parts

- Phase 2(a): “Prepare to Commit”
  - Send the decision (commit/abort) to all nodes
  - Nodes go into prepare-to-commit state
- Phase 2(b): “Commit”
  - The old Phase 2

Advantage: if the coordinator fails, all nodes know if they should commit/abort a transaction

Summary

Saw a few ways to deal with concurrency control in DDMBs

- Two dimensions:
    - Different computers could be the primary for different database items or not
    - Primary have backups or not
- Voting as alternative

Saw 2PC and 3PC (two- and three-phase commit – no relation to 2PL, besides the words “two” and “phase”) for coordinating commits to deal with recovery

Did not deal with deadlocks…