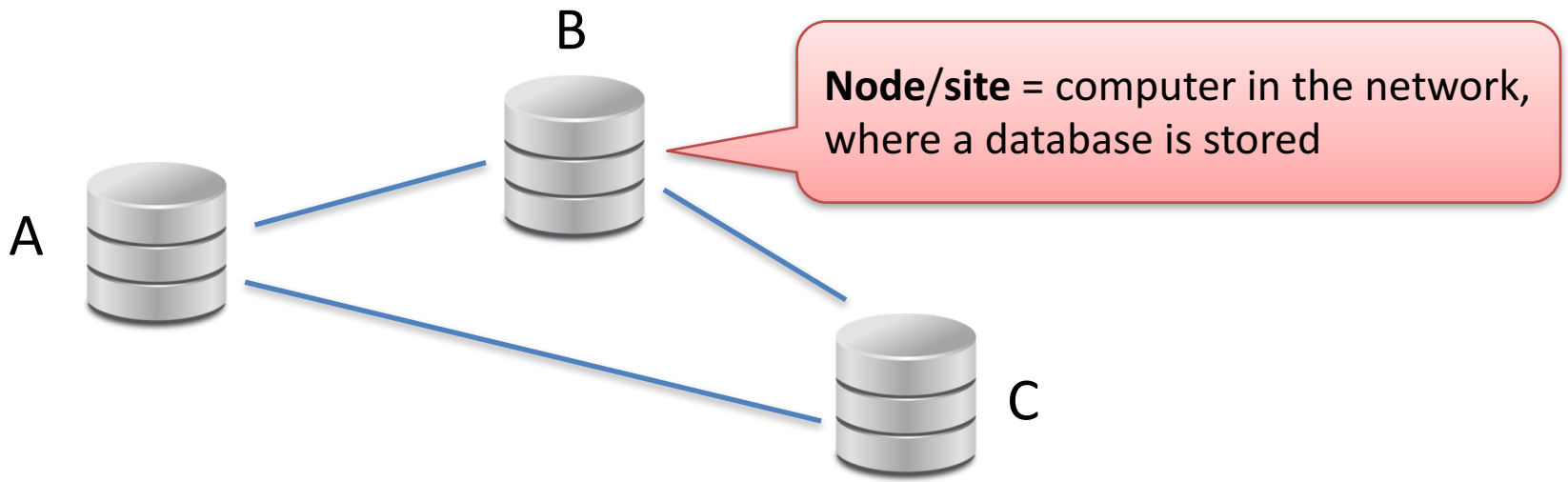
COMP207
Database Development

Lecture 18

Distributed Databases:
Transaction Management

Distributed Databases (Reminder)

- **Distributed Database**:
  - Collection of multiple **logically interrelated** databases
  - **Distributed** over a computer network



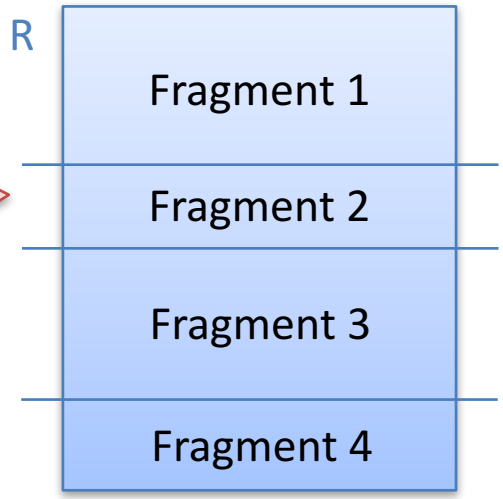
- **Distributed DBMS**: manages a distributed database

Fragmentation (Reminder)

- Split database into different parts that can then be stored at different nodes

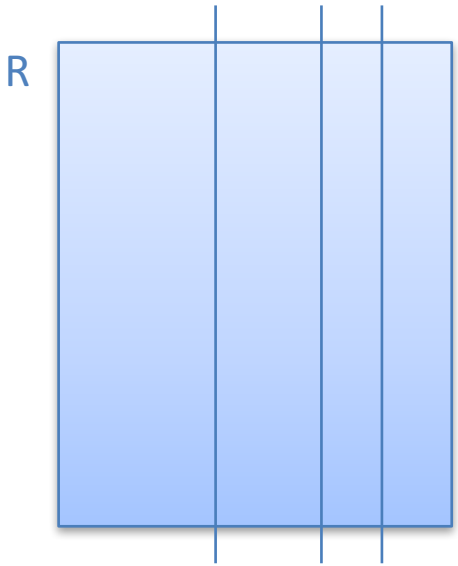
"Sharding"

Horizontal fragmentation



Fragments may overlap

Vertical fragmentation

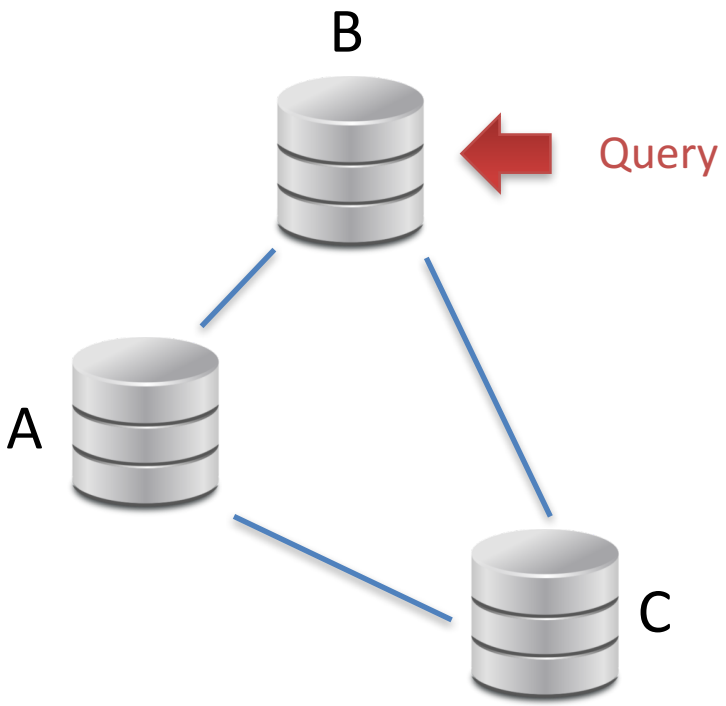


- Users don't see fragments, just the full relations

"Fragmentation transparency"

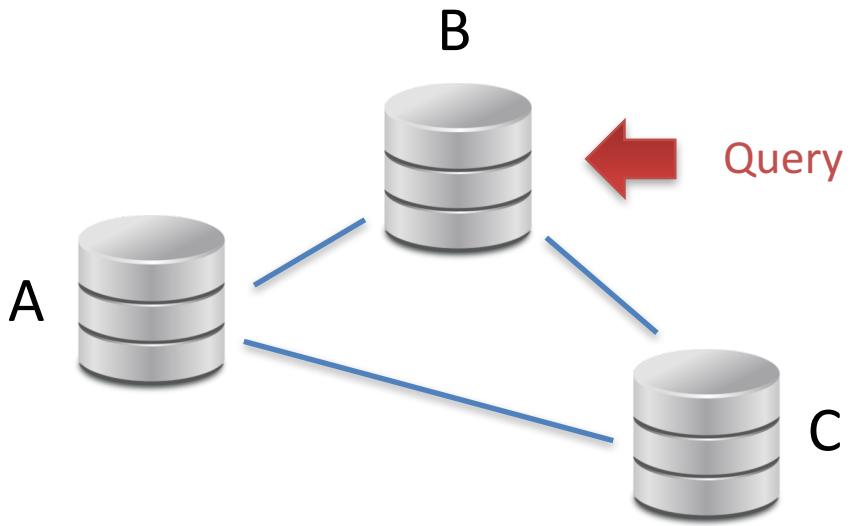
Challenges

- **Query Processing**
  - Data not available at local node has to be loaded from other nodes
  - Goal: avoid communication as much as possible
- **Transaction management**



Query Processing

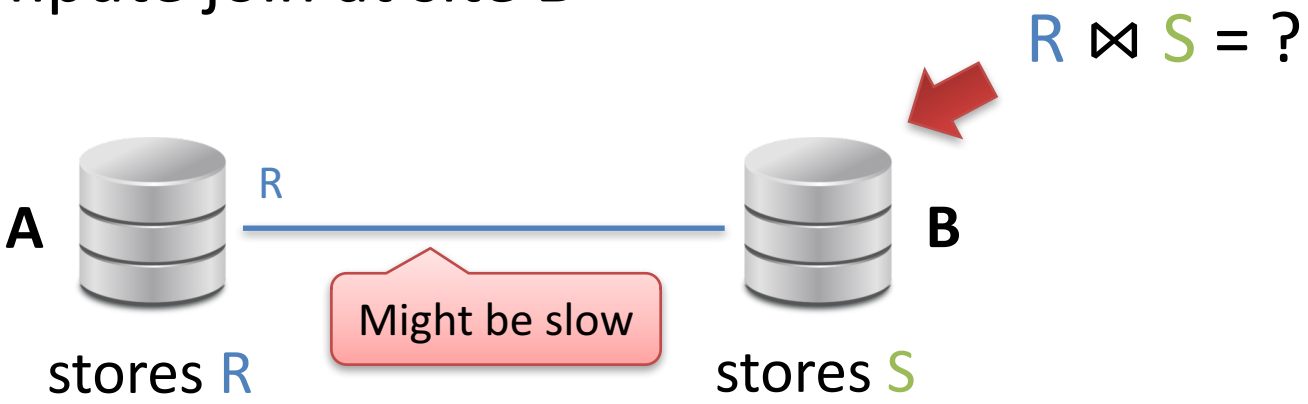
Query Processing in Distributed DBMS



- Try to answer query at site where query is raised
- If not possible: request information from other sites
  - Slow → design database to reduce this as much as possible
  - Most expensive: joins

Joins

- Goal: compute join at site **B**



- Obvious approach:
  - Site **B** asks site **A** to send R
  - Site **B** computes R ⋈ S

R might be very large – do we have to send all tuples?

- Better: only send data that is actually required

Semijoins ($\ltimes$)

- $R \ltimes S := R \bowtie \pi_{\text{common attributes of R and S}}(S)$

Recall: no duplicates

Modules

module	year
COMP105	1
COMP201	2
COMP207	2

Lecturers

name	module
J. Fearnley	COMP105
S. Coope	COMP201

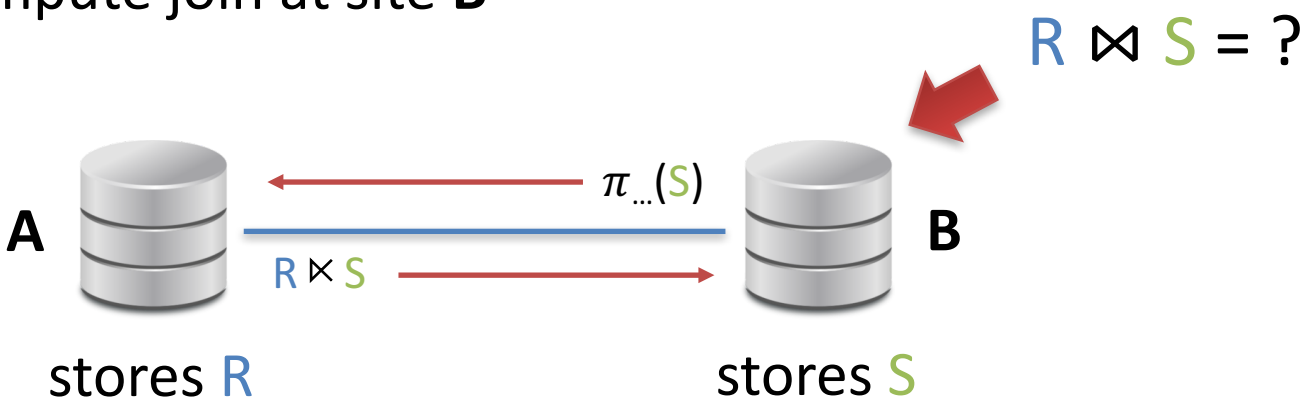
Modules $\ltimes$ **Lecturers** =

module	year
COMP105	1
COMP201	2

- Intuition: $R \ltimes S$ = set of all tuples in R that join with at least one tuple in S

Semijoin Reduction

- Goal: compute join at site **B**



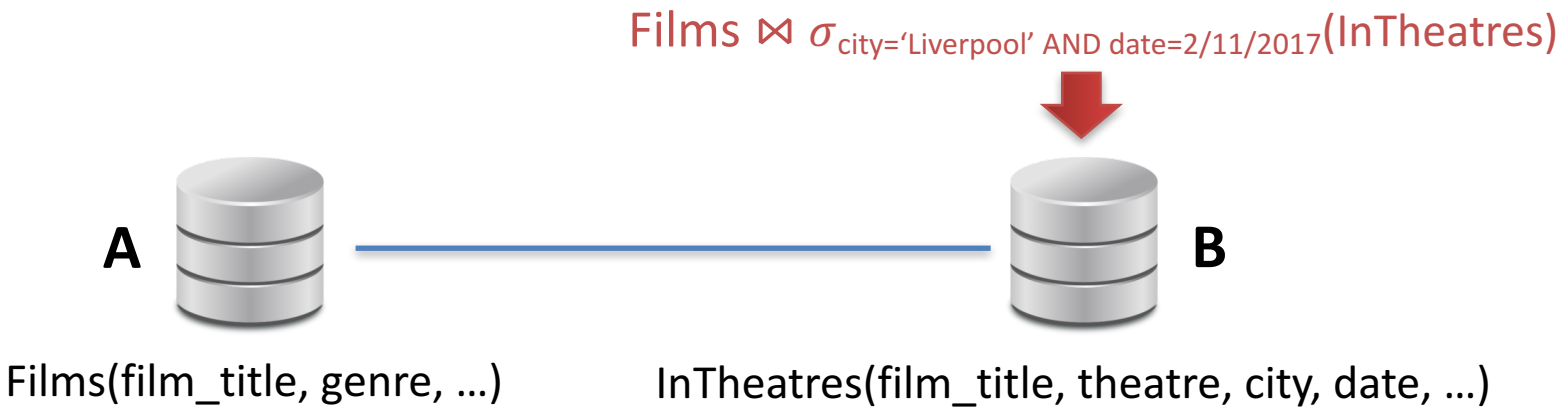
- With semijoins:
  - Site **B** sends S' := $\pi_{\text{common attributes of R and S}}$(S) to site **A**
  - Site **A** sends R' := R ⋉ S (= R ⋉ S') to site **B**
  - Site **B** outputs R' ⋈ S
- Communication costs ≈

  |S'| × (size of tuple in S') + |R'| × (size tuple in R')

Efficiency

- Is this more efficient than computing the join in the obvious approach way (exchanging relations)?
- Depends:
  - Is the projection much smaller than the full relation?
    - Many duplicates to be eliminated? I.e., do many tuples of S share values of the common attributes? Not the case if one of the join attributes is a key…
    - Do the columns that are projected out require much space?
  - Is the size of the semijoin much smaller?
  - In general: $|\pi_{\text{common attributes}}(S)| + |R \ltimes S|$ should be much smaller than S

Example

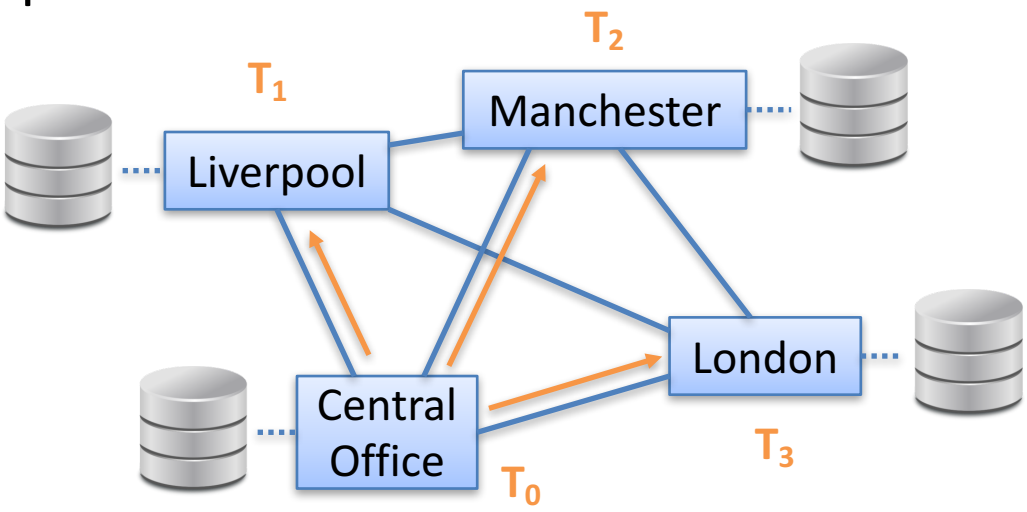


- Procedure:
  - At **B**, send S' := $\pi_{\text{film\_title}}(\sigma_{\text{city='Liverpool' AND date=2/11/2017}}(\text{InTheatres}))$ to **A**
  - At **A**, send R' := Films ⋉ S' to **B**
  - At **B**, output R' ⋈ $\sigma_{\text{city='Liverpool' AND date=2/11/2017}}(\text{InTheatres})$
- Communication costs:
  - Assume |Films| = 10,000, |S'| = |R'| = 20, and 1000 bytes per tuple
  - Communication cost = (20+20) × 1000 = 40,000 bytes

10,000,000 bytes for obvious approach

Transactions in Distributed Databases

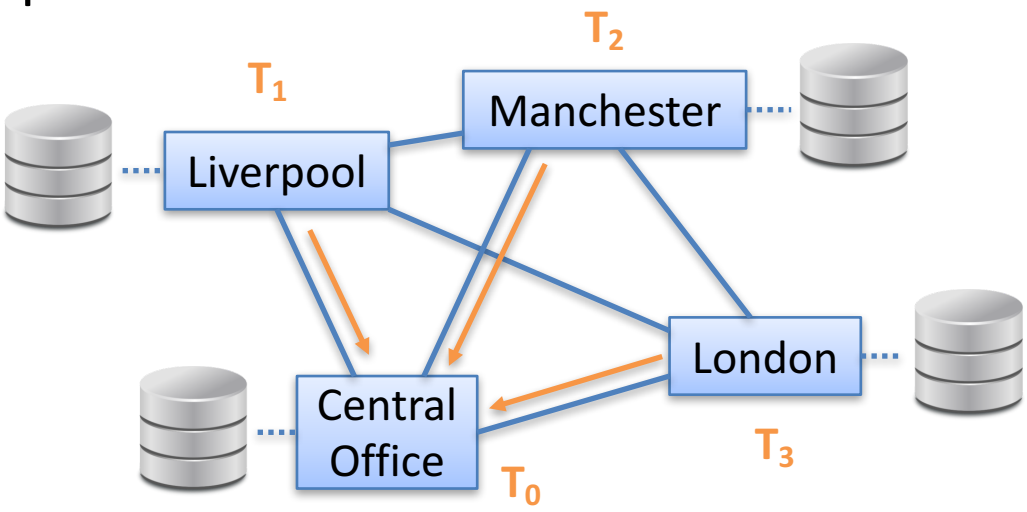
- Let's revisit our chain of department stores…
- At central office:
  - Determine inventory for product X at each site
  - Move product X between stores to balance inventory



- **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

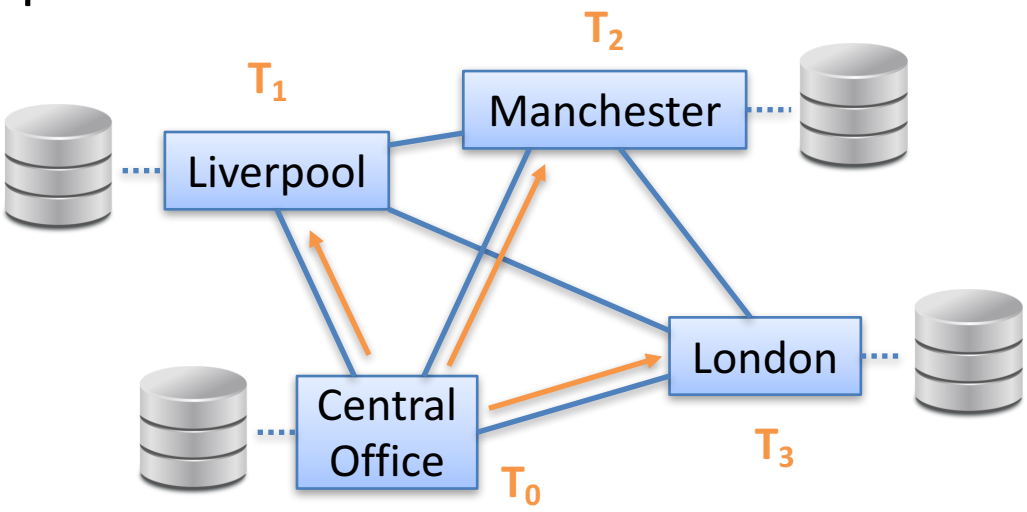
- Let's revisit our chain of department stores…
- At central office:
  - Determine inventory for product X at each site
  - Move product X between stores to balance inventory



- **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

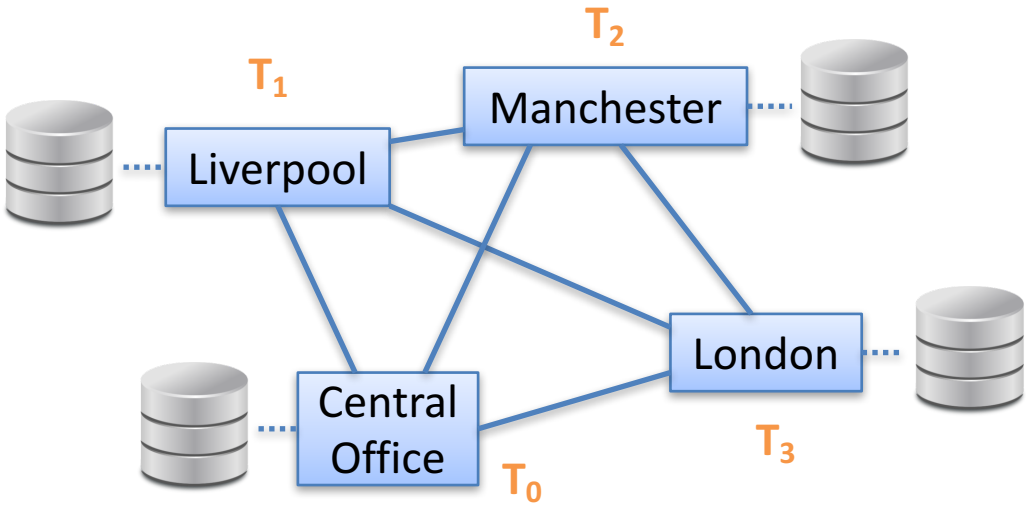
- Let's revisit our chain of department stores…
- At central office:
  - Determine inventory for product X at each site
  - Move product X between stores to balance inventory



- **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

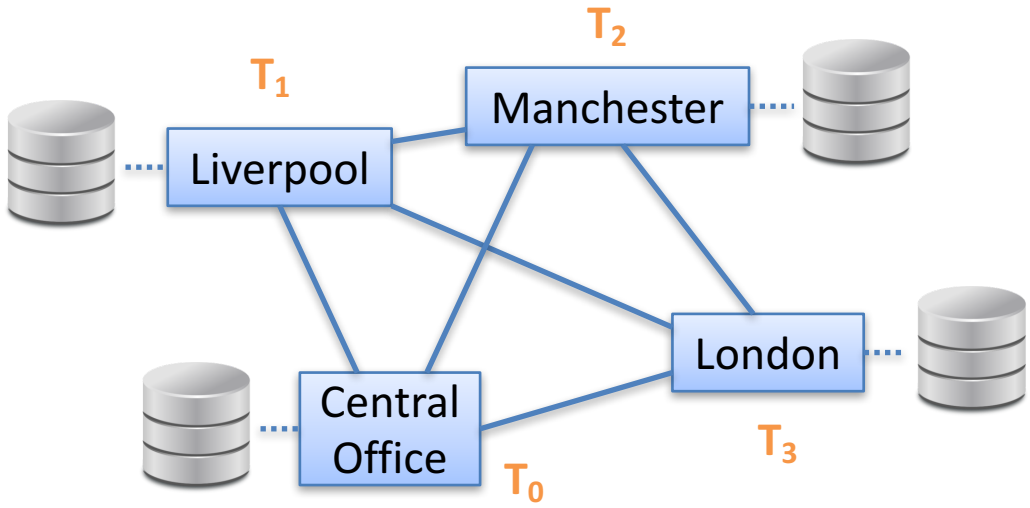


- **Atomicity:**
  - Can assume to be enforced at each node *locally*
  - Could be violated *globally*

What can go wrong?

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



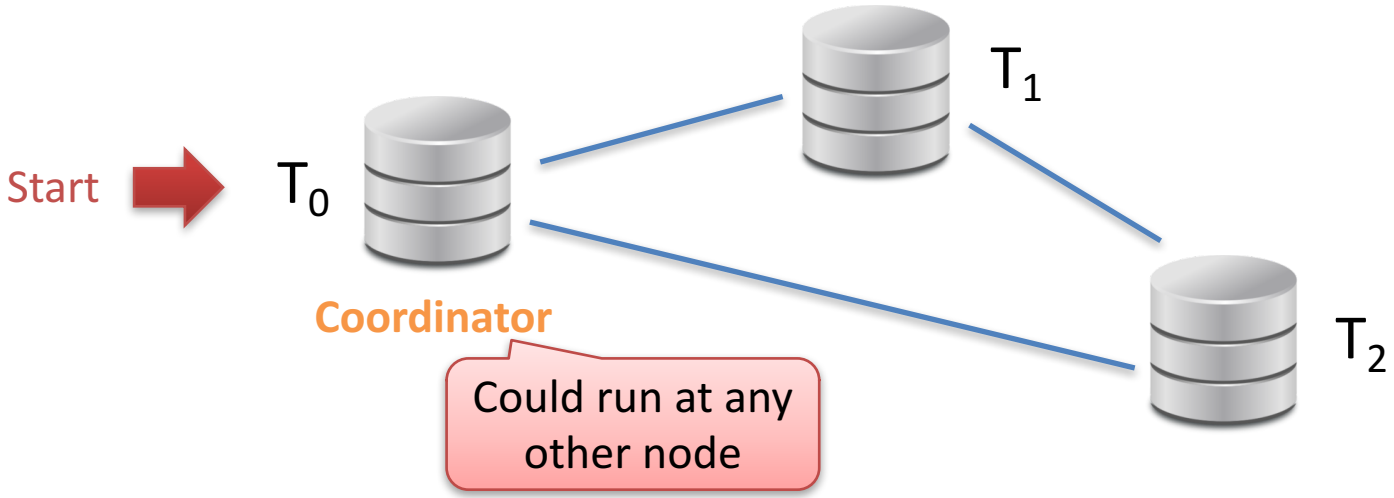
- **Nodes can fail during execution of T**
  - Should we abort or wait?
  - What about the failing node after recovery?

Distributed Commit

Two-Phase Commit Protocol

Not related to 2PL!

- Coordinates commit actions globally



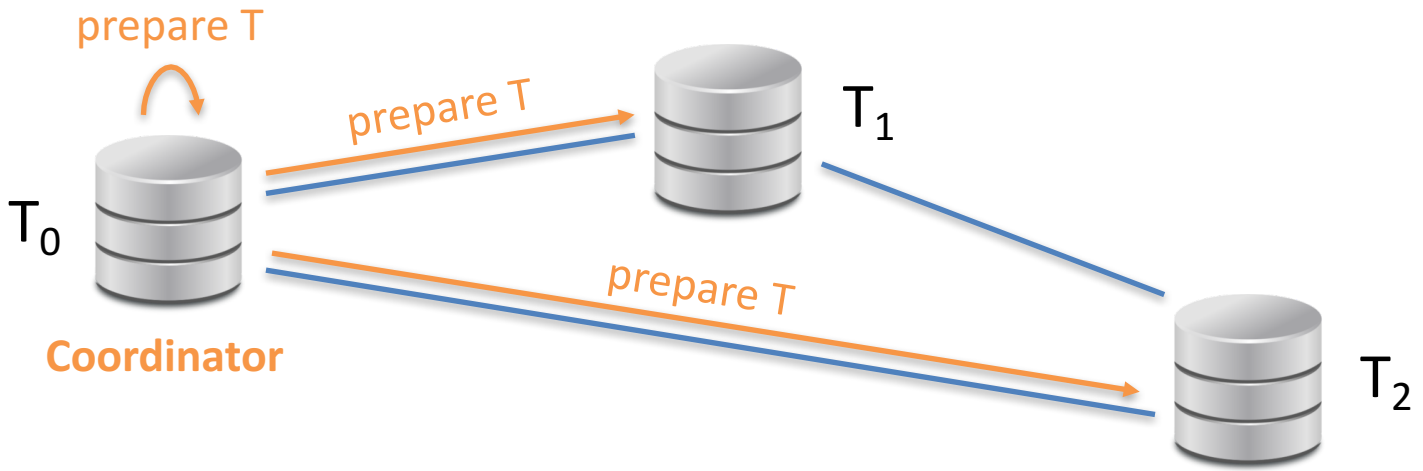
- **Coordinator:** executed at some node & decides if and when local transactions can commit
- **Logging:** at each node locally
  - Messages sent to & received from other nodes are logged, too!

The Two Phases

- Phase 1: Decide when to commit or abort
- Phase 2: Commit or abort

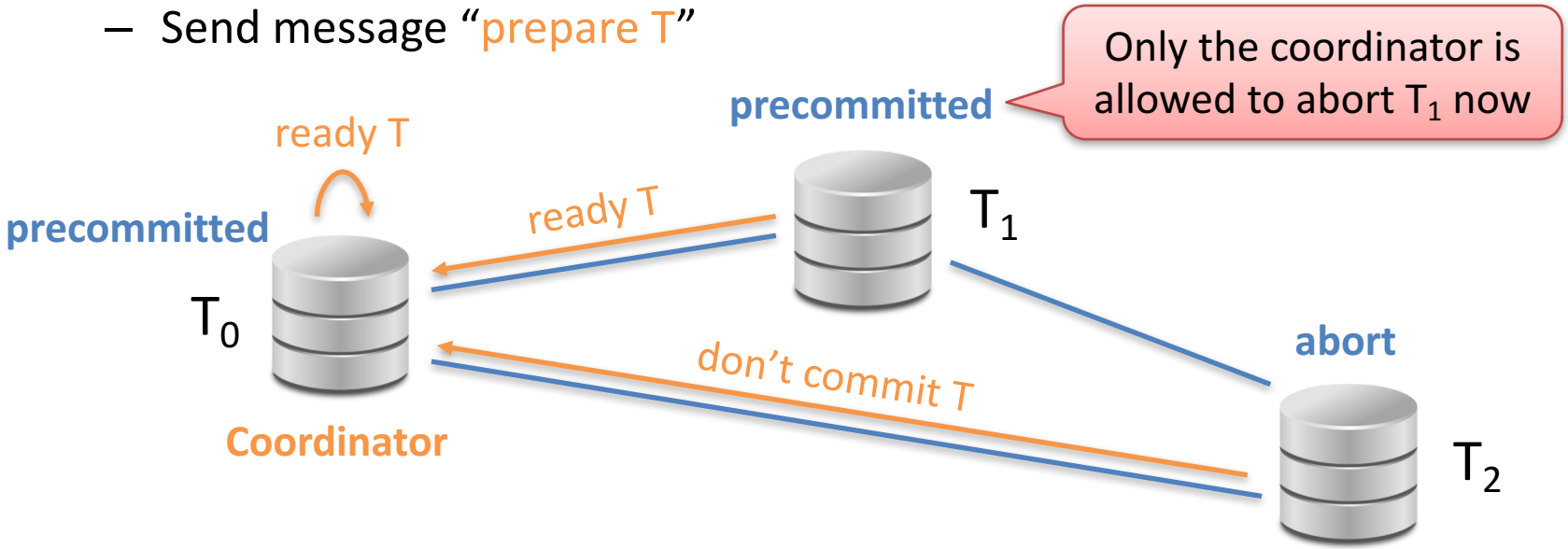
Phase 1: When To Commit?

- Coordinator: ask nodes if they want to commit
  - Send message “prepare T”



Phase 1: Ready to Commit?

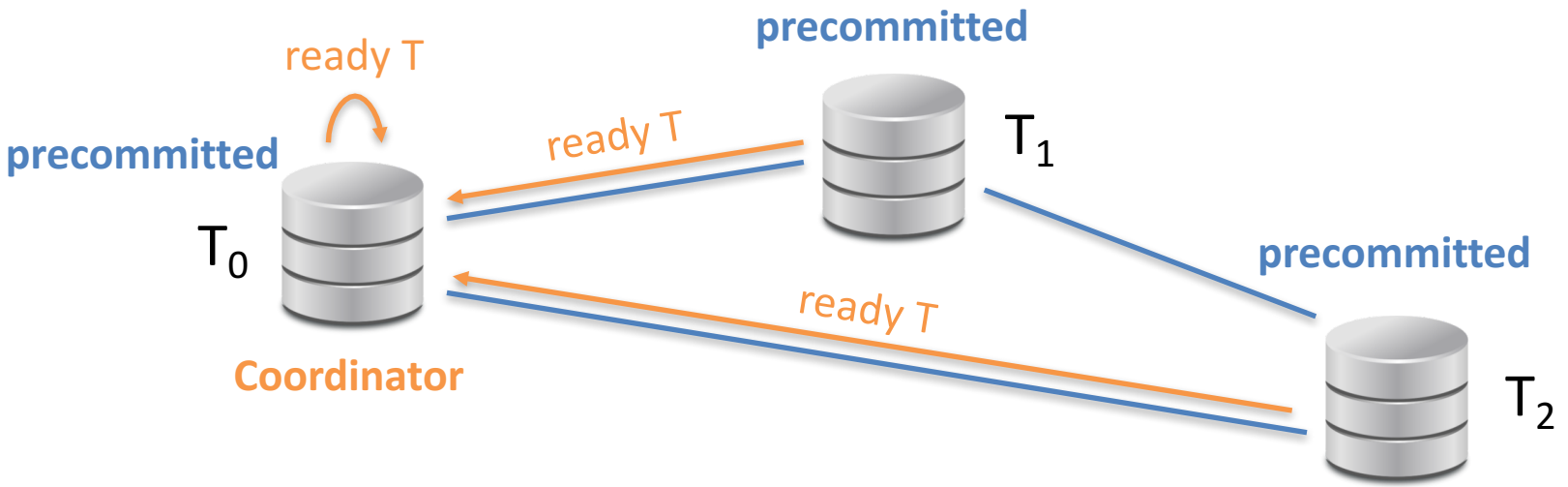
- Coordinator: ask nodes if they want to commit
  - Send message "prepare T"



- 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

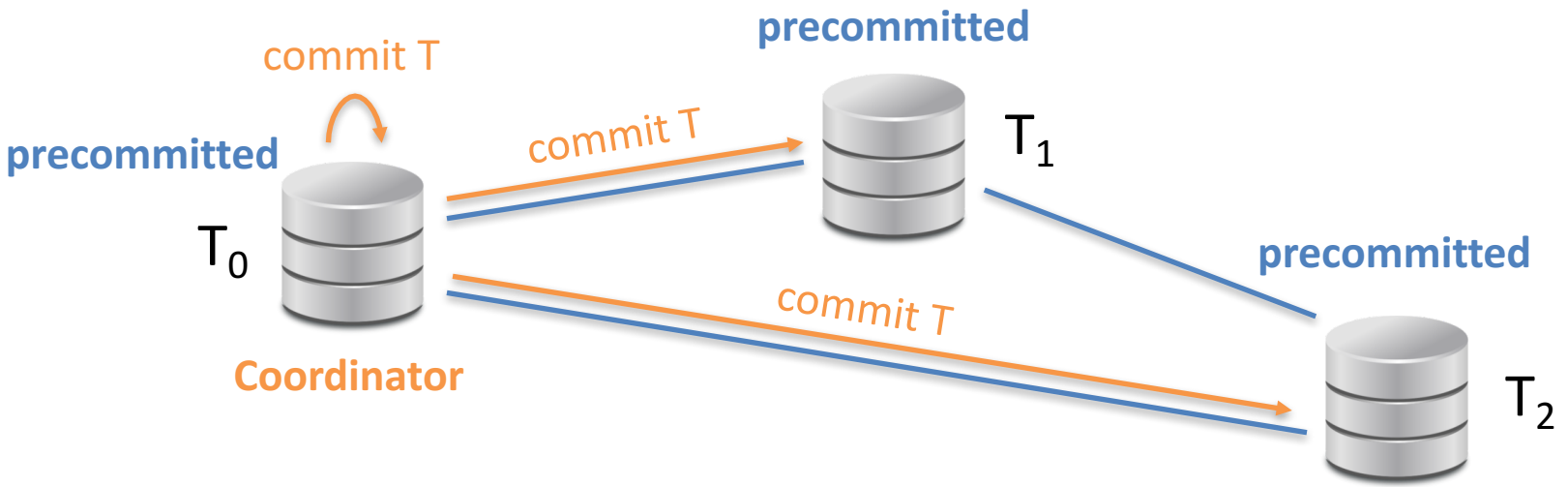
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort



- If all nodes respond "ready T"

Phase 2: Let's Do It

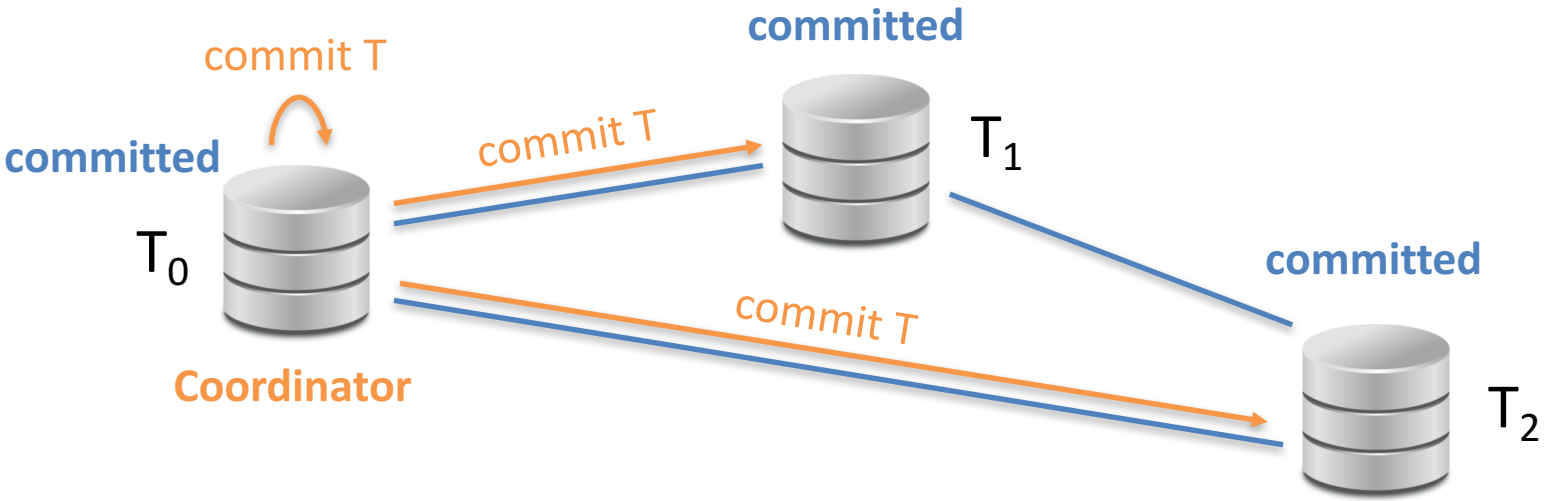
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort



- If all nodes respond "ready T"
  - Send "commit T" to all nodes → nodes commit

Phase 2: Let's Do It

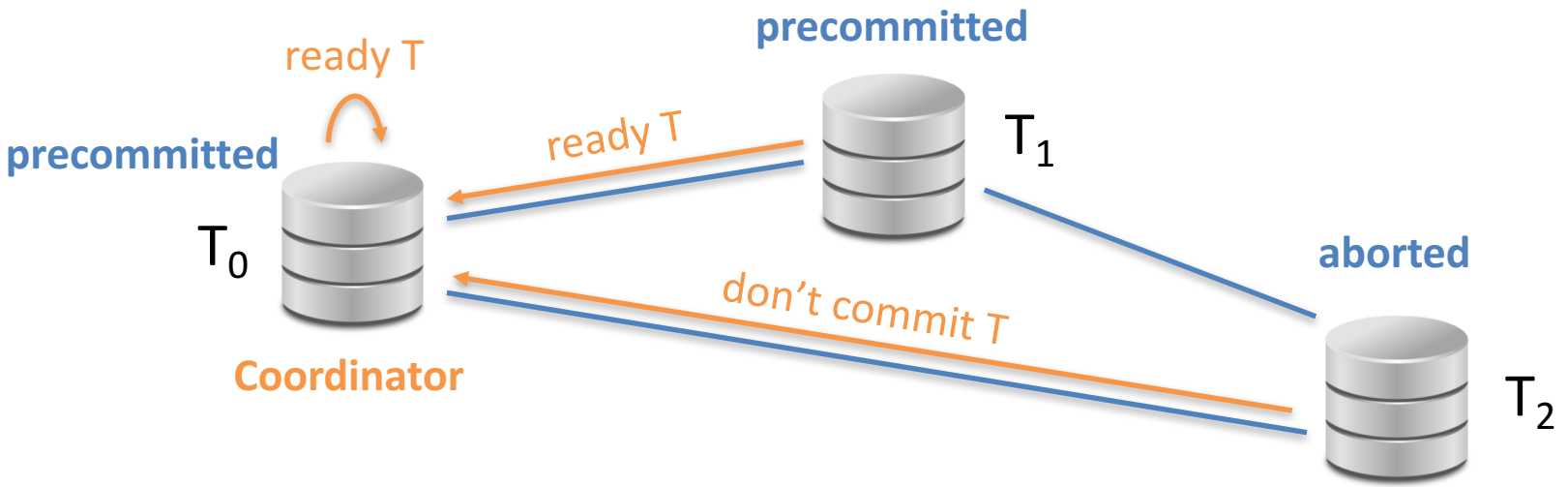
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort



- If all nodes respond "ready T"
  - Send "commit T" to all nodes → nodes commit

Phase 2: Let's Do It

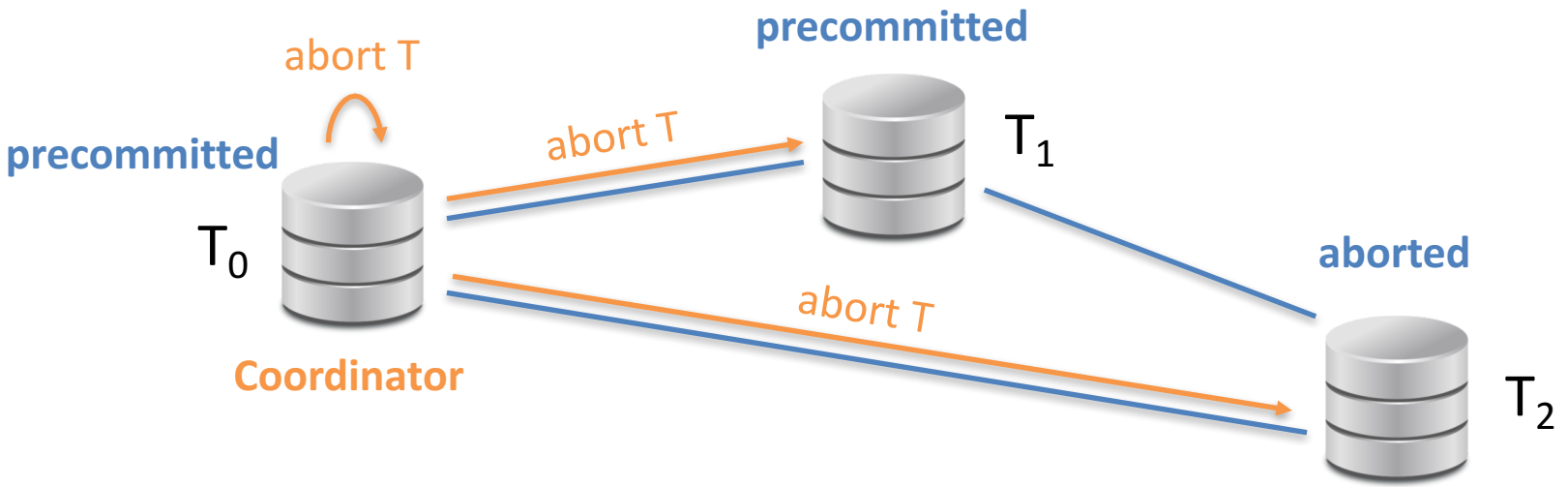
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort



- 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

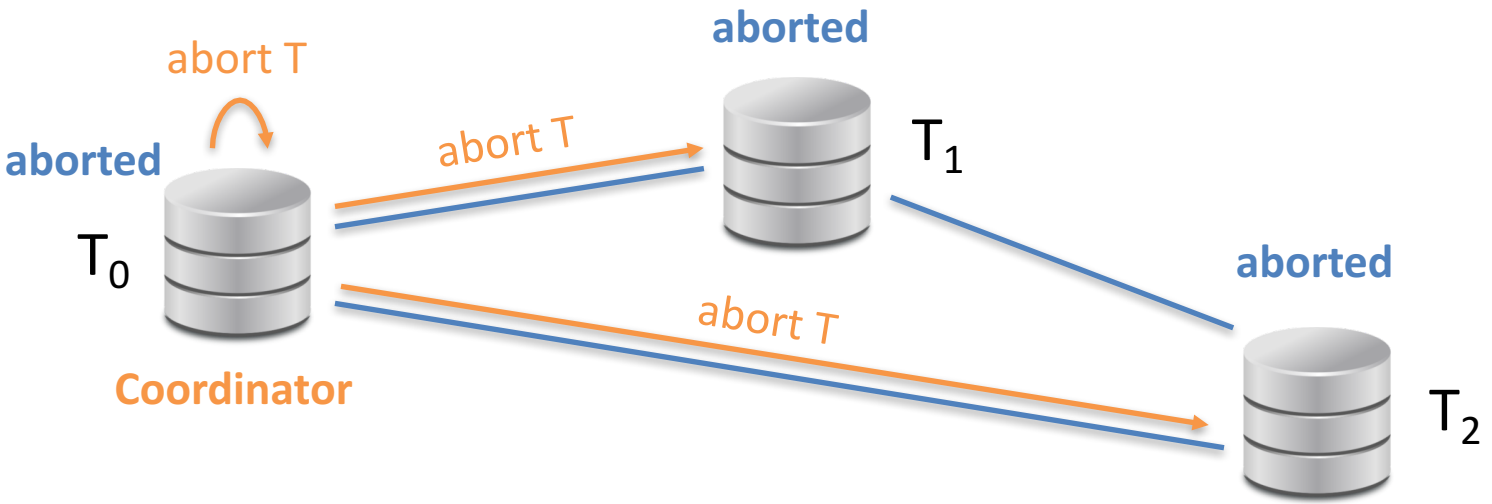
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort



- 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

Phase 2: Let's Do It

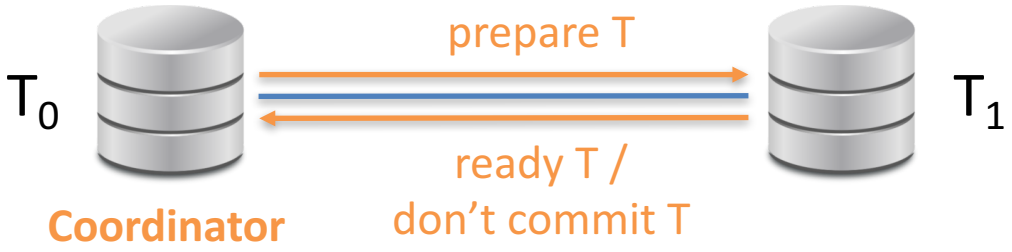
- Coordinator: waits for responses of nodes
  - Assume nodes who don't reply before a given timeout wish to abort

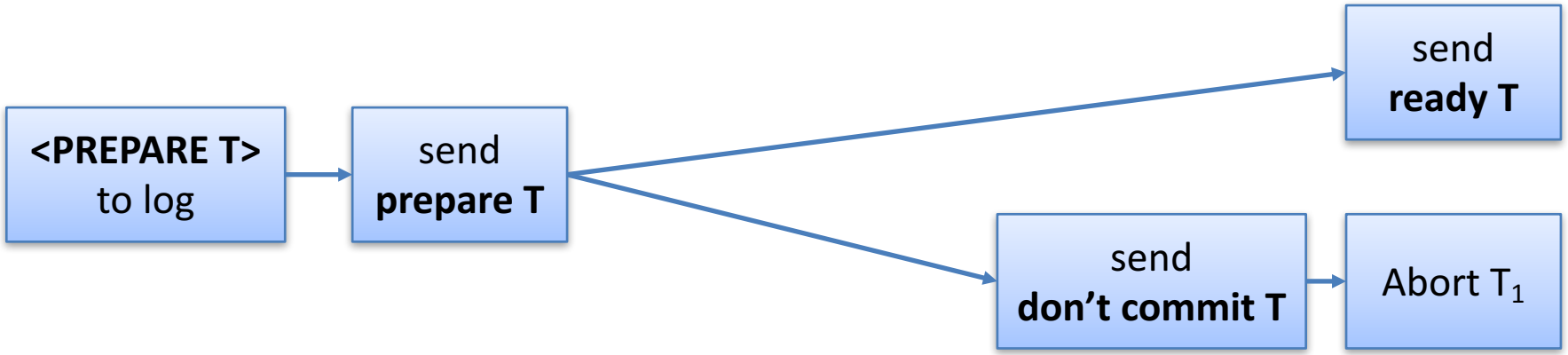


- If all nodes respond "ready T"
  - Send "commit T" to all nodes → nodes commit
- If some node responds "don't commit T"
  - Send "abort T" to all nodes → nodes abort

Logging: Phase 1

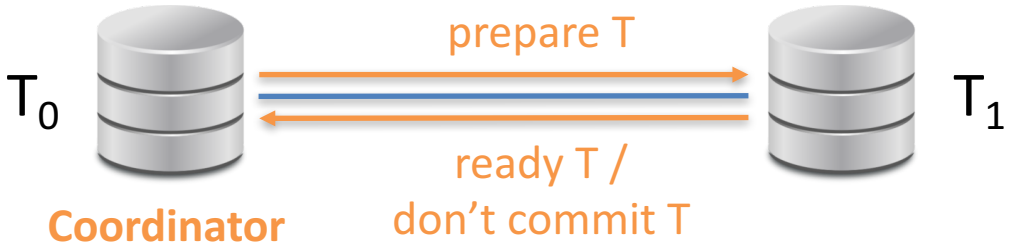
- Again, we have to be careful in which order to write to disk and to the log
- Phase 1:

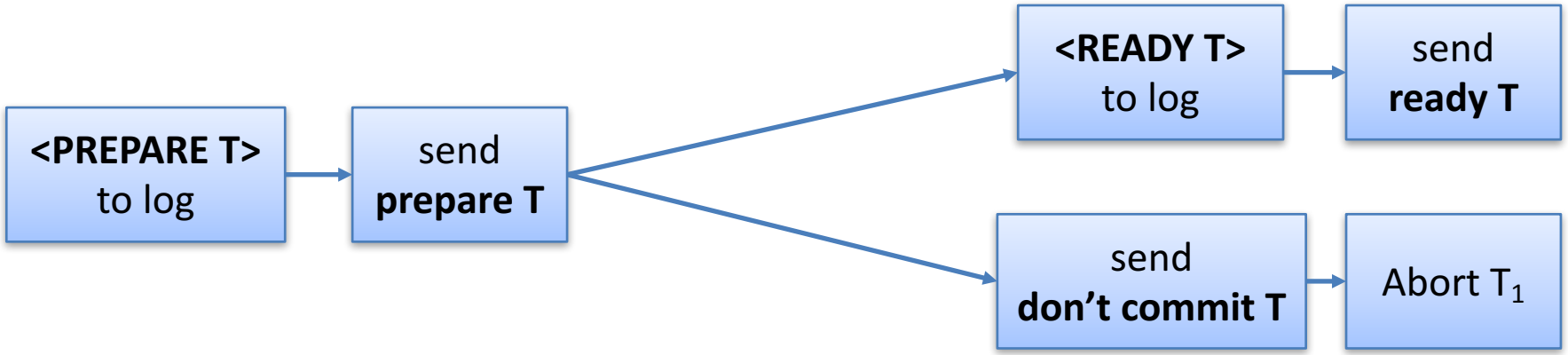




Logging: Phase 1

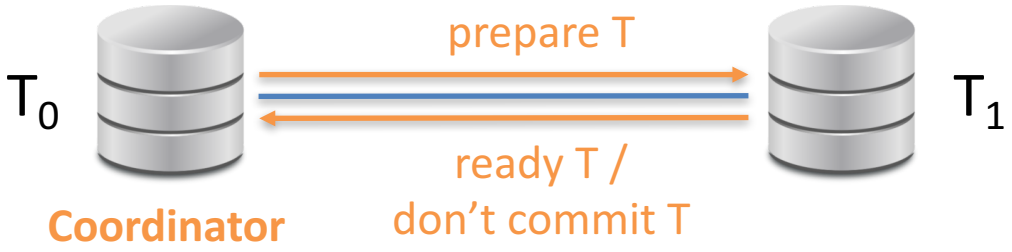
- Again, we have to be careful in which order to write to disk and to the log
- Phase 1:

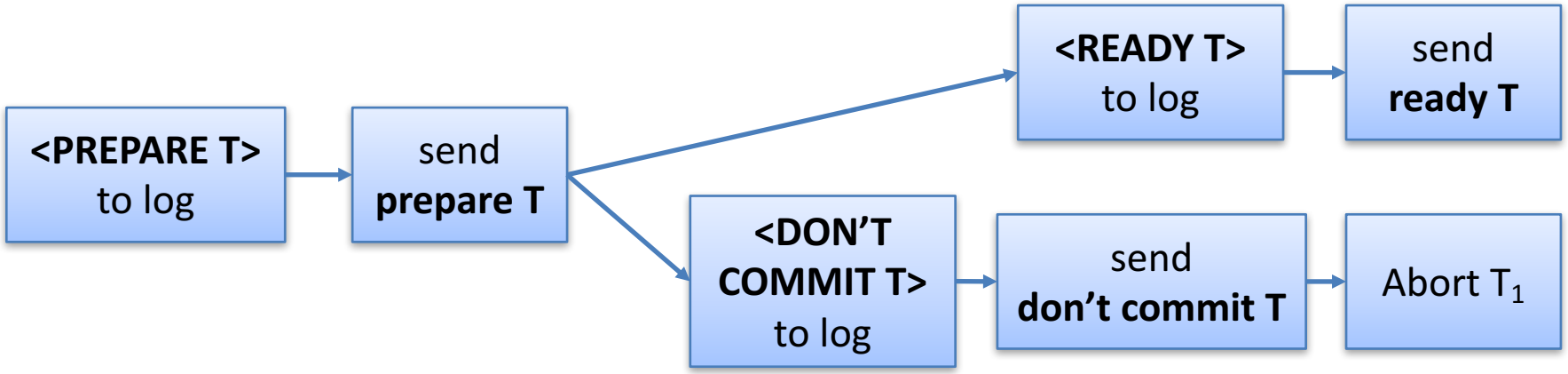




Logging: Phase 1

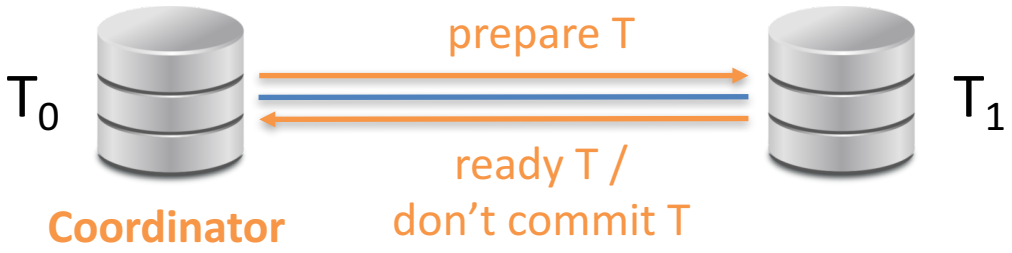
- Again, we have to be careful in which order to write to disk and to the log
- Phase 1:





Logging: Phase 1

- Again, we have to be careful in which order to write to disk and to the log
- Phase 1:



Write all log entries to disk

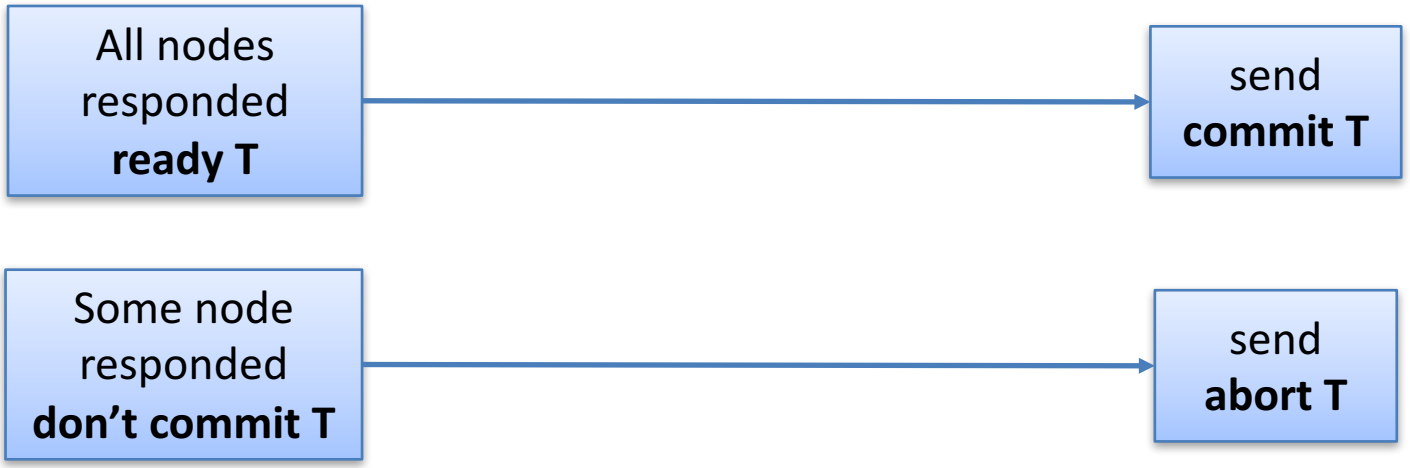
<PREPARE T> to log → send **prepare T** → Ensure T_1 will not have to be aborted → **<READY T>** to log → send **ready T**

send **prepare 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:



Logging: Phase 2

- Phase 2:

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**

<DON'T COMMIT T> to log

send **abort T**

Three-Phase Commit Protocol

- Improvement of Two-Phase Commit
- Can deal with the situation that the coordinator fails
- 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

- Distributed DBMSs are challenging
  - Query answering: avoid too much communication
  - Transactions: need global coordination
- **Two-phase commit**:
  - Coordinates commit actions of transactions
  - Problematic if coordinator can fail
  - Improvement: **three-phase commit** → announce action to everyone before executing them in the next round
- Challenges not covered here:
  - How to ensure **consistency** in the presence of replication?
  - What to do in case of **deadlocks** involving different nodes?