COURSE 4

Distributed Locking

Database Recovery

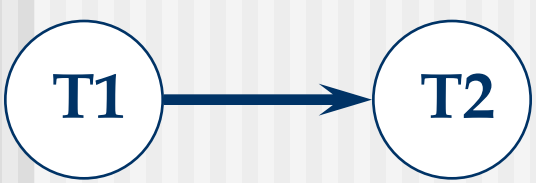
Distributed Locking

How do we manage locks for objects across many sites?

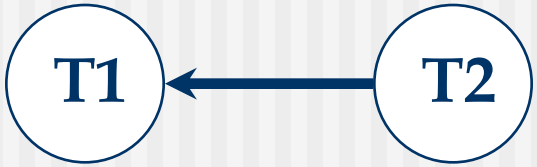
- Centralized: One site does all locking.
  - Vulnerable to single site failure.
- Primary Copy: All locking for an object done at the primary copy site for this object.
  - Reading requires access to locking site as well as site where the object is stored.
- Fully Distributed: Locking for a copy done at site where the copy is stored.
  - Locks at all sites while writing an object.

Distributed Deadlock Detection

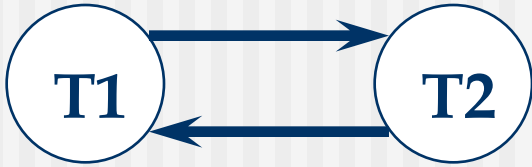
- Each site maintains a local waits-for graph.
- A global deadlock might exist even if the local graphs contain no cycles:



SITE A



SITE B



GLOBAL

Three solutions:

Centralized (send all local graphs to one site);

Hierarchical (organize sites into a hierarchy and send local graphs to parent in the hierarchy);

Timeout (abort transaction if it waits too long).

Recovery and the ACID Properties

Atomicity: All actions in the transaction happen, or none happen.

Consistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions.

Durability: If a transaction commits, its effects persist.

The **Recovery Manager** is responsible for ensuring two important properties of transactions: Atomicity and Durability.

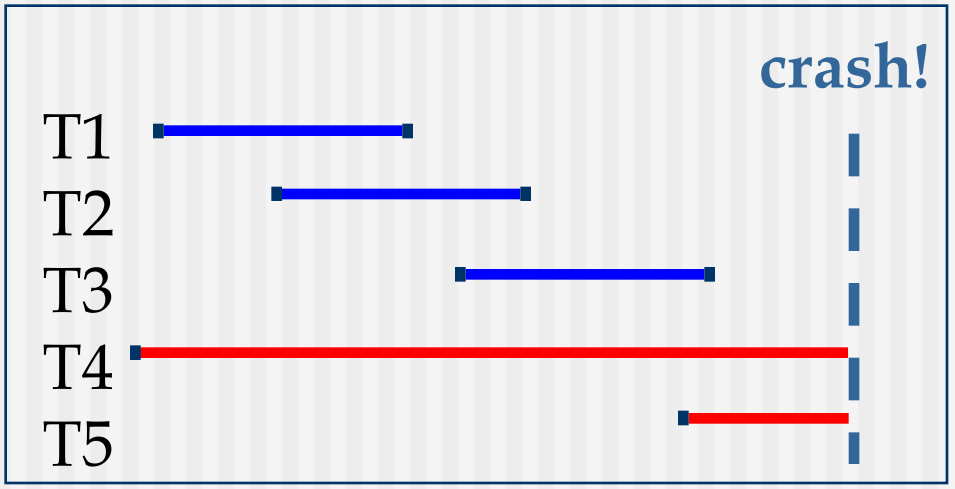
- Atomicity is guaranteed by undoing the actions of the transactions that did not commit.
- Durability is guaranteed by making sure that all actions of committed transactions survive crashes and failures.

Motivation

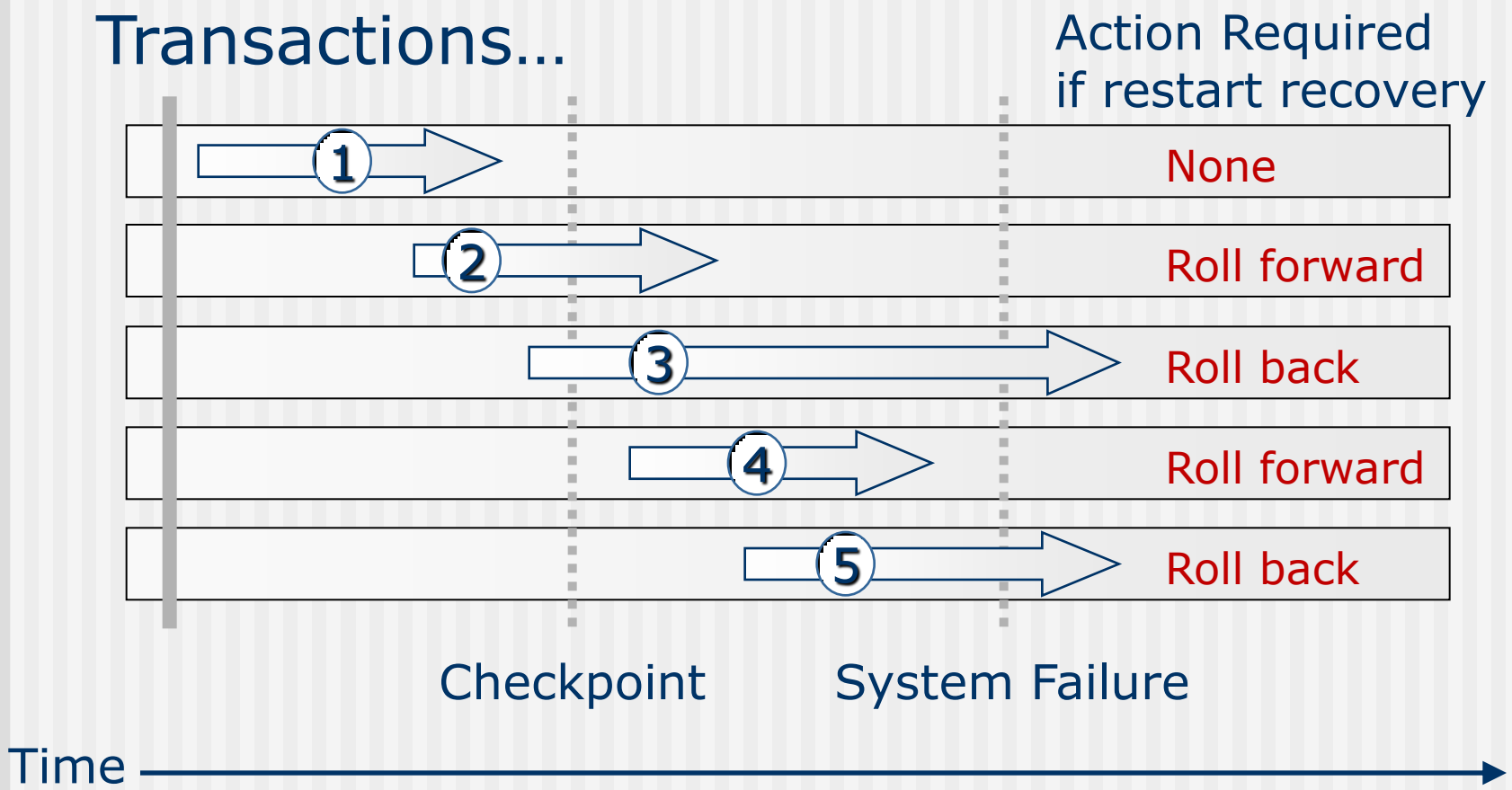
- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

Desired Behavior after system restarts:

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



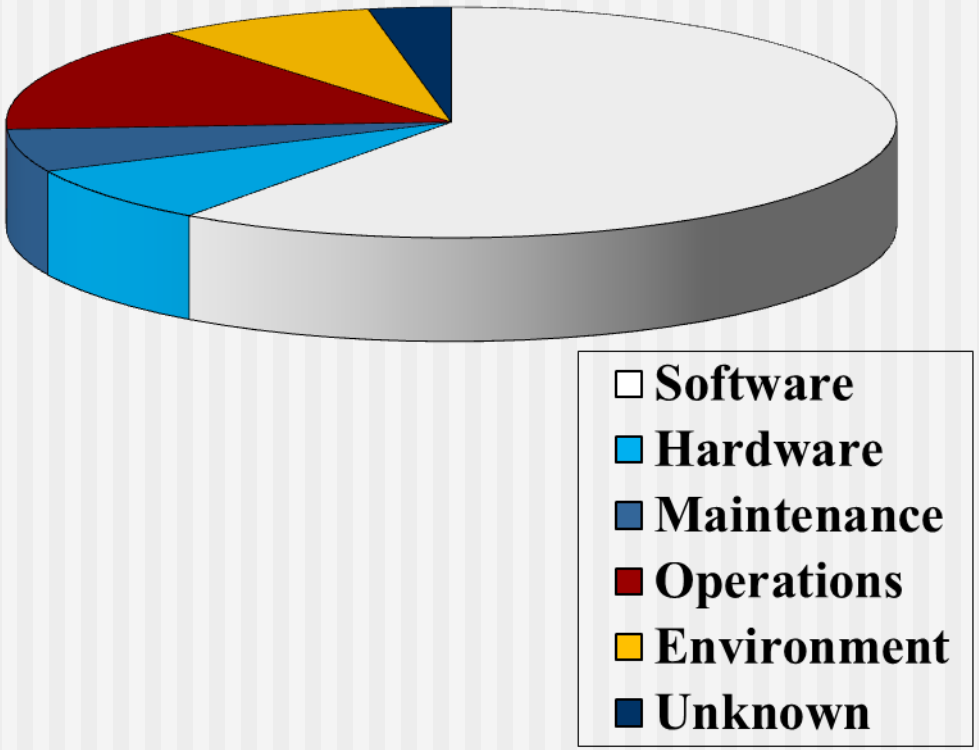
Another Motivating Example



Types of Failures

- **System crashes**: due to hardware or software errors, resulting in loss of main memory
- **Media failures**: due to problems with disk head or unreadable media, resulting in loss of parts of secondary storage.
- **Application errors**: due to logical errors in the program which may cause transactions to fail.
- **Natural disasters**: physical loss of media (fire, flood, earthquakes, terrorism, etc.)
- **Sabotage**: intentional corruption or destruction of data
- **Carelessness**: unintentional destruction of data by user or operator.

Failures Impact



General Failure Categories

(1) Transaction failures

- Transaction aborts (unilaterally or due to deadlock)
- 3% of transaction abort abnormally on average (bad input, data not found, overflow, res. limit exceeded)

(2) System failures

- Failure of processor, main memory, power supply…
- Main memory contents are lost but not secondary storage

(3) Media failures

- Failure of secondary storage
- Head crash or controller failure

Recovery

- For non-catastrophic failures (category 1 or 2):
  - Use transaction log to bring database into a consistent state (the way it was before failure)
  - Reverse any changes by **undoing** some operations (e.g., transaction did not commit yet)
  - **Redo** some operations to restore data (e.g., transaction committed but not all of the data was written out to disk yet)
- Catastrophic failure (category 3):
  - Use archival backup to **restore** past copy of database
  - Reconstruct most recent consistent state by **redoing** operations of committed operations from the log (backed up)

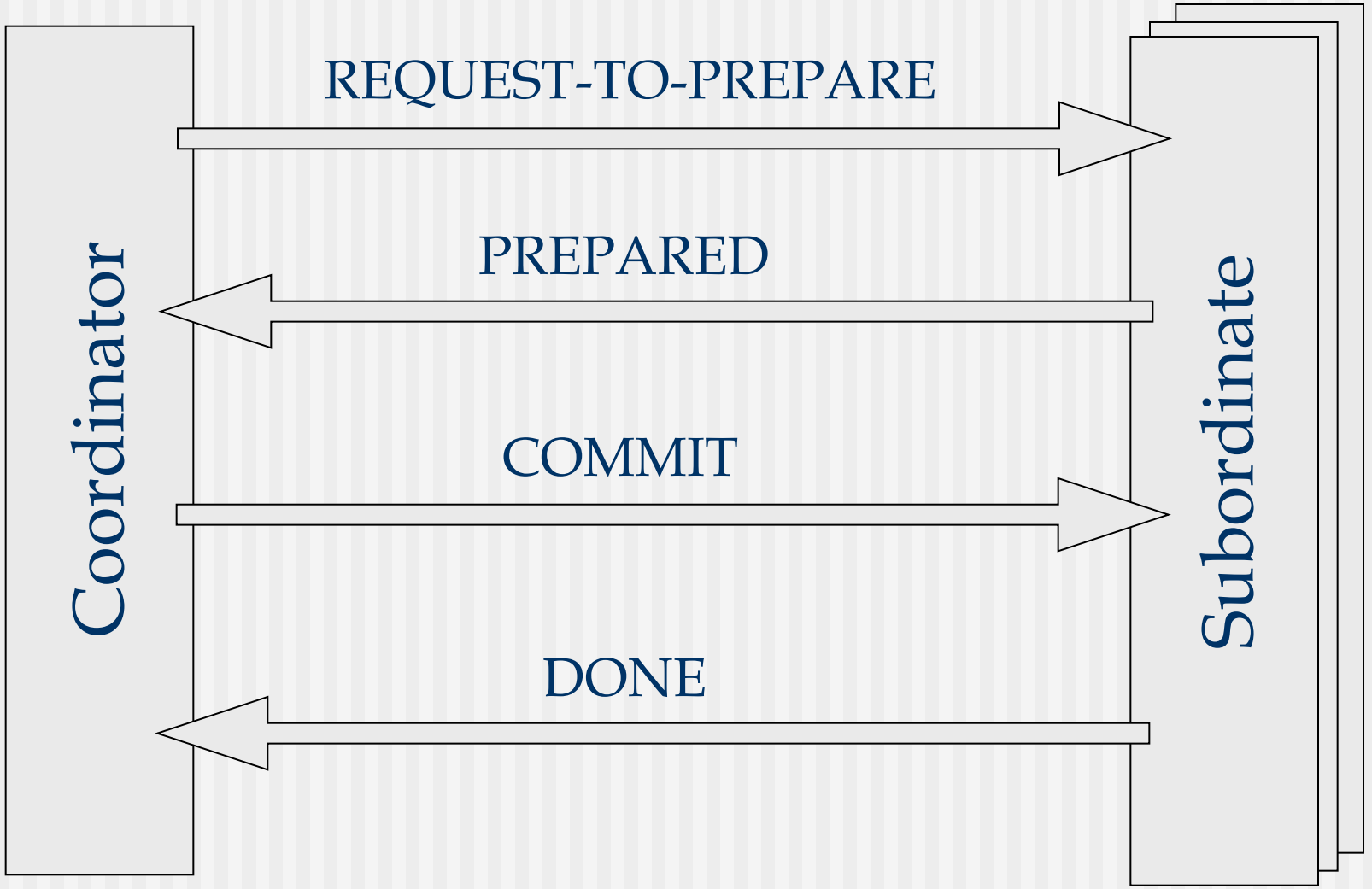
Distributed Recovery

- Two new issues:
  - New kinds of failure, e.g., links and remote sites.
  - If "sub-transactions" of a transaction execute at different sites, all or none must commit. Need a commit protocol to achieve this.
- A log is maintained at each site, as in a centralized DBMS, and commit protocol actions are additionally logged.

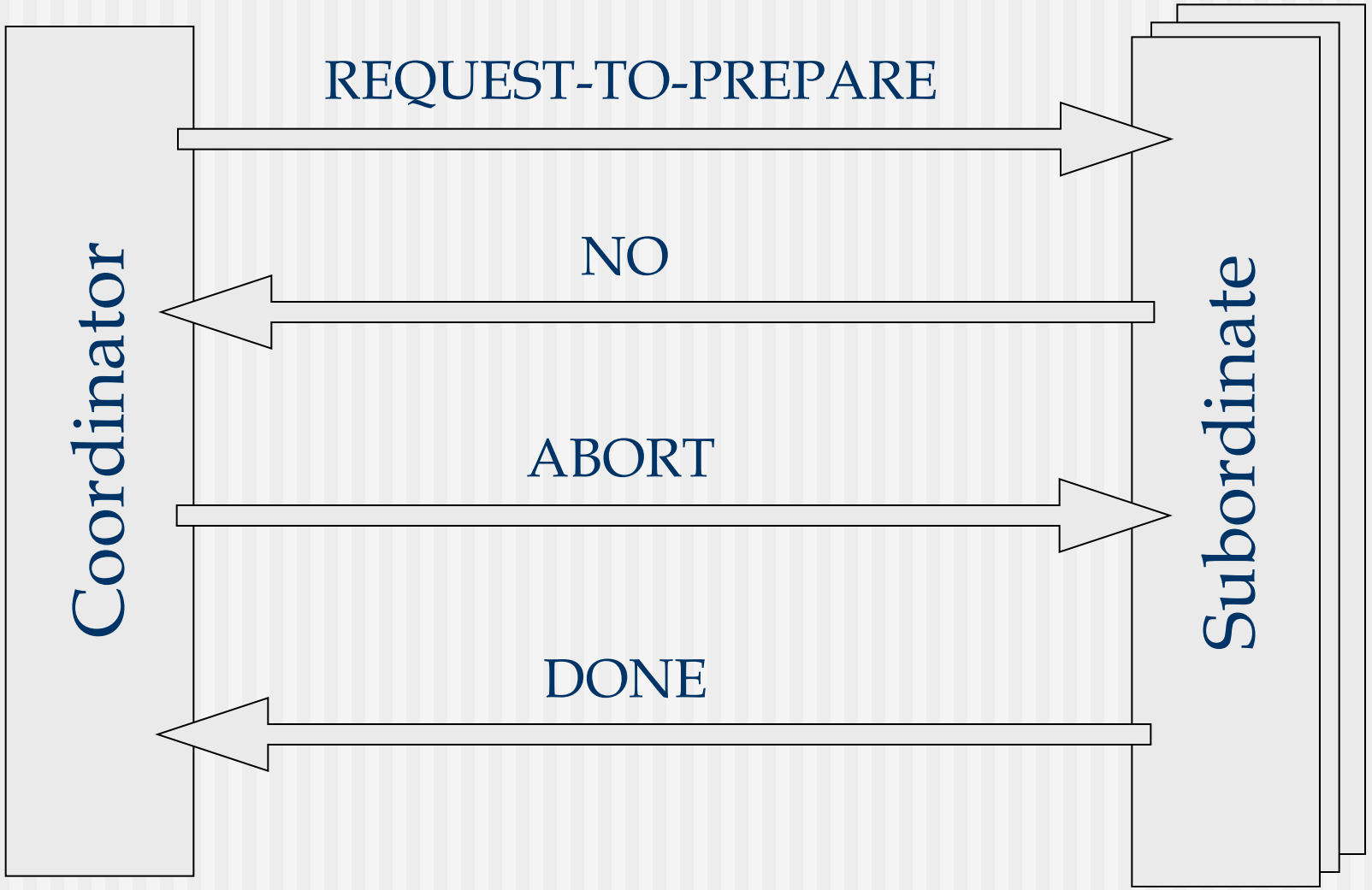
Two-Phase Commit (2PC)

- Site at which transaction originates is coordinator; other sites at which it executes are subordinates.
- When a transaction wants to commit:
  1. Coordinator sends **prepare** msg to each subordinate.
  2. Subordinate force-writes an **abort** or **prepare** log record and then sends a **no** or **yes** msg to coordinator.
  3. If coordinator gets unanimous yes votes, force-writes a **commit** log record and sends **commit** msg to all subs. Else, force-writes **abort** log rec, and sends **abort** msg.
  4. Subordinates force-write **abort/commit** log rec based on msg they get, then send **done** msg to coordinator.
  5. Coordinator writes **end** log rec after getting all *dones*.

Two-Phase Commit (2PC) (cont)



Two-Phase Commit (2PC) (cont)



Comments on 2PC

- Two rounds of communication: first, voting; then, termination. Both initiated by coordinator.
- Any site can decide to abort a transaction.
- Every msg reflects a decision by the sender; to ensure that this decision survives failures, it is first recorded in the local log.
- All commit protocol log recs for a transaction contain *TransactionID* and *CoordinatorID*. The coordinator's abort/commit record also includes ids of all subordinates.

Restart After a Failure at a Site

- If we have a commit or abort log rec for transaction T, but not an end rec, must redo/undo T.
  - If this site is the coordinator for T, keep sending **commit/abort** msgs to subs until **done** received.
- If we have a prepare log rec for transaction T, but not commit/abort, this site is a subordinate for T.
  - Repeatedly contact the coordinator to find status of T, then write **commit/abort** log rec; redo/undo T; and write **end** log rec.
- If we don't have even a prepare log rec for T, unilaterally abort and undo T.
  - This site may be coordinator! If so, subs may send msgs.

Blocking

- If coordinator for transaction T fails, subordinates who have voted yes cannot decide whether to *commit* or *abort* T until coordinator recovers.
  - T is blocked.
  - Even if all subordinates know each other (extra overhead in prepare msg) they are blocked unless one of them voted no.

Link and Remote Site Failures

- If a remote site does not respond during the commit protocol for transaction T, either because the site failed or the link failed:
  - If the current site is the coordinator for T, should abort T.
  - If the current site is a subordinate, and has not yet voted yes, it should abort T.
  - If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds.

Observations on 2PC

- Done msgs used to let coordinator know when it can "forget" a transaction; until it receives all dones, it must keep T in the Transaction Table.
- If coordinator fails after sending prepare msgs but before writing commit/abort log records, when it comes back up it aborts the transaction.
- If a subtransaction does no updates, its commit or abort status is irrelevant.

2PC with Presumed Abort

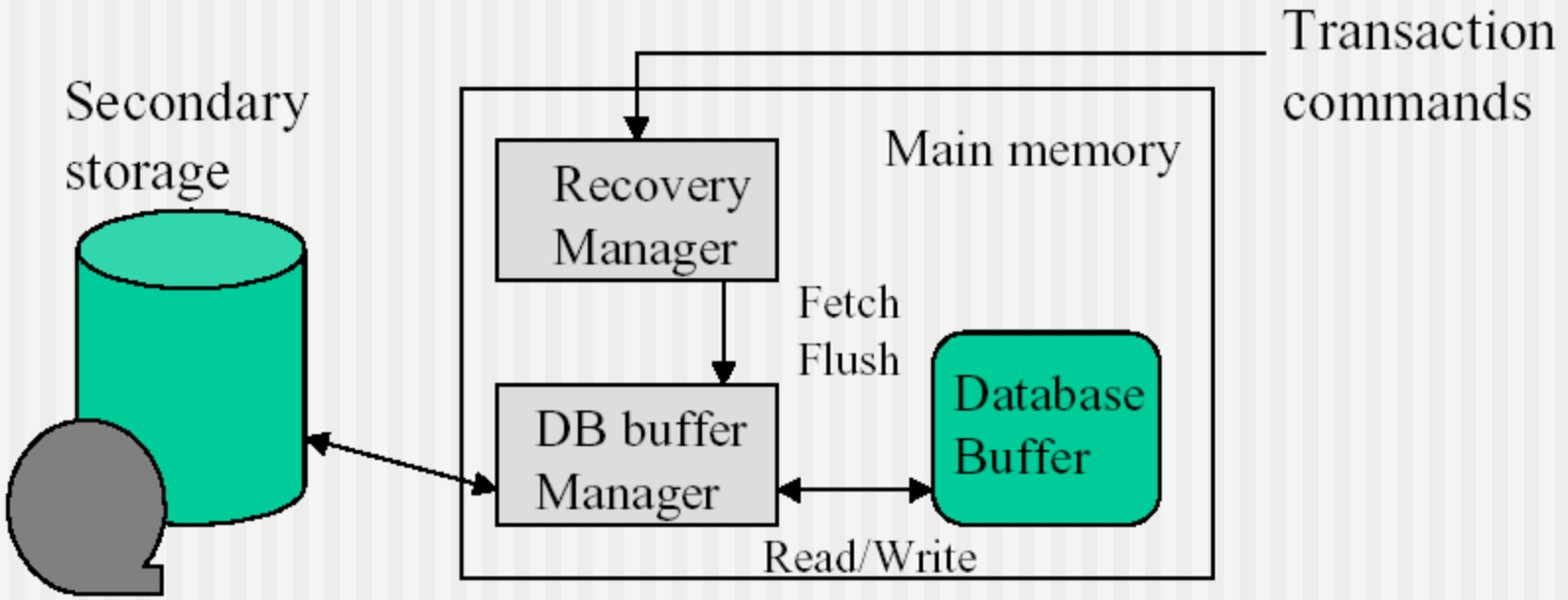
- When coordinator aborts T, it undoes T and removes it from the Transaction Table immediately.
  - Doesn't wait for **dones**; "presumes abort" if transaction not in Transaction Table. Names of subs not recorded in **abort** log rec.
- Subordinates do not send dones on abort.
- If sub-transaction does not do updates, it responds to prepare msg with reader instead of yes/no.
- Coordinator subsequently ignores readers.
- If all sub-transactions are readers, 2nd phase not needed.

Assumptions

- Concurrency control is in effect.
  - Strict 2PL, in particular.
- Updates are happening "in place".
  - i.e. data is overwritten on (deleted from) the disk.

- Is there a simple scheme to guarantee Atomicity & Durability?

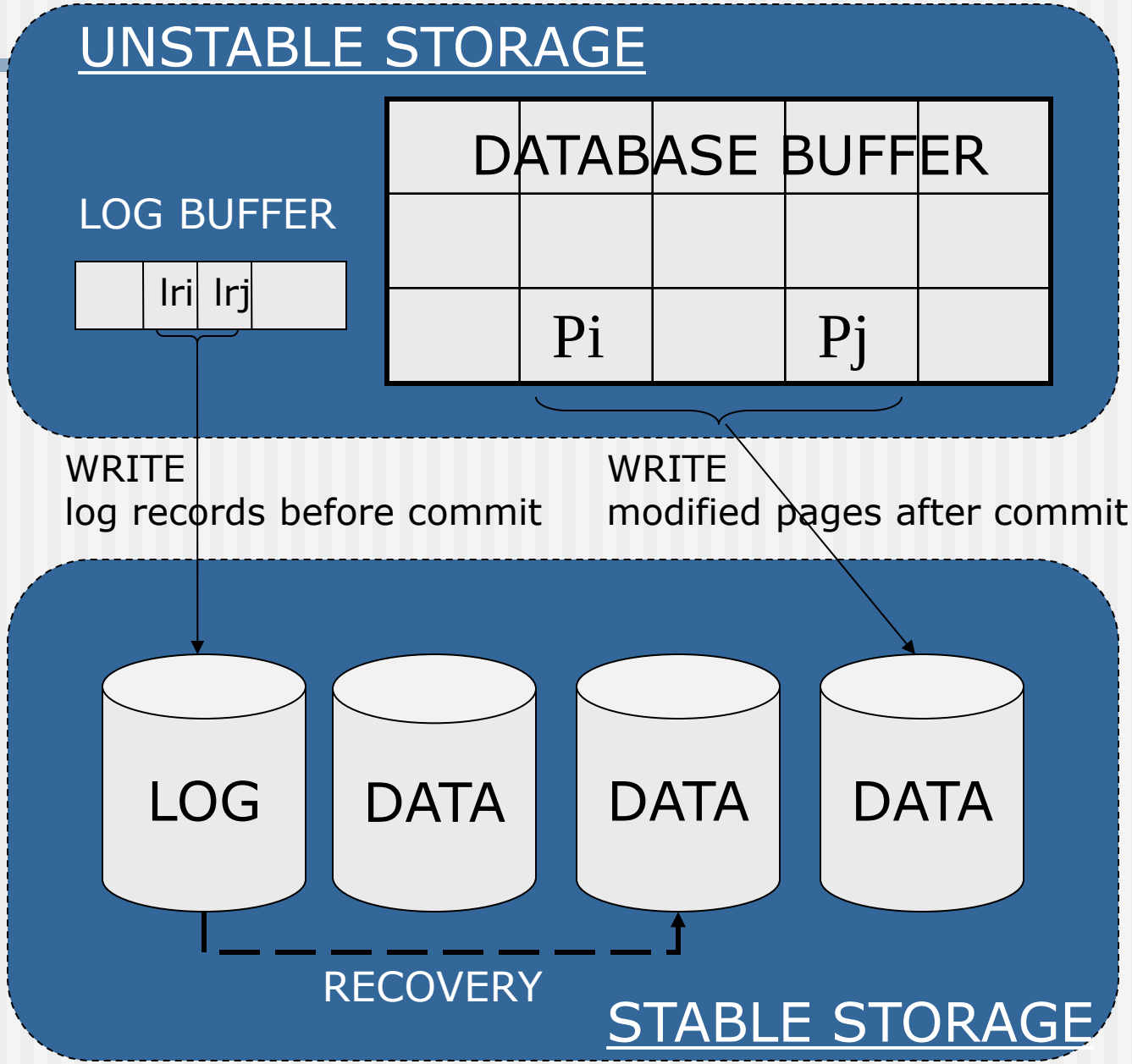
Recovery Manager



- Volatile storage: main memory (DB buffer)
- Stable storage: disks and other media. Resilient to failure and loses its content only in the presence of media failure or intentional attacks (Database and logs)

Action Logging

- Every update action of a transaction must also write a log entry to an append-only file.
- No record needs to be appended in the log if the action merely reads the database
- Why do we need a log? The log is consulted by the system to achieve both atomicity and durability.
- The log is generally stored on a different mass storage device than the database.


UNSTABLE STORAGE
DATABASE BUFFER
LOG BUFFER
lri
lrj
Pi
Pj
WRITE
log records before commit
WRITE
modified pages after commit
LOG
DATA
DATA
DATA
RECOVERY
STABLE STORAGE

Database Log

- The log contains records (or entries) that are appended.
- For recovery, the log is read backwards
- An entry in the log contains:
  - A transaction identifier
  - Type of operation
  - Objects accessed to perform action
  - Old value of object (before image)
  - New value of object (after image)
  - …
- Log entries (also called update records) are used to undo changes in case of aborts: using transaction ID replace object value in database with before image

Other Entries in Log

- Log may also contain action such as *begin-transaction, commit-transaction, abort-transaction*.
- If a transaction T is aborted, then rollback → scan log backwards and when T's update records are encountered, the before image is written to DB undoing the change.
  - However, how long do we have to scan backwards looking for T's entries in the log? → we need the ***begin-transaction*** entry indicating where to stop scanning for a transaction actions.
- In a rollback due to a crash, the system needs to distinguish between the transactions that completed and the still active transactions → **commit** and **abort** entries.

Problems With Log

- Does the commit request guarantee durability?
- If a crash occurs after the transaction has made changes and requested the commit, but before the commit record is written in log, the transaction will be aborted by the recovery procedure and the changes are not durable.

- What are the different scenarios?

Making Changes in Database

- Transaction T made changes to x. There is use of DB buffer. There is a crash while operations are performed.
- **Scenario1**: Neither operations made it to secondary storage → No problem. T is aborted.
- **Scenario2**: x is updated on disk but crash occurred before log is updated → No way to rollback since we don't have before image → inconsistent state.
- **Scenario3**: The update made it to the log but the changes to x on disk didn't make it → T aborted and before image is used to overwrite old value → no problem.
- What do we learn from these scenarios?

Write-Ahead Logging (WAL)

- The update record must always be appended to the log before the database is updated. The log is referred to as a ***write-ahead log***.
- The **Write-Ahead Logging** Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to disk.
  2. Must write all log records for a transaction *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.
- Exactly how is logging (and recovery!) done? We'll study the ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) algorithms.

Checkpointing

- In case of a crash, the recovery procedure needs to identify transaction that are still active in order to abort them. This is done by scanning the log backwards.
- How far do we have to scan the log before stopping and guaranteeing that there is not transaction still active?
- To avoid a complete backward scan of the log during recovery, we must include a mechanism to specify where to stop.
- New entry in the log called ***Checkpoint.***

Checkpoints

- The system periodically appends a checkpoint record to the log that lists the current active transactions.
- The recovery process must scan backward at least as far as the most recent checkpoint.
- If T is named in the checkpoint, then T was still active during crash → continue scan backward until *begin-transaction* T.

Checkpoints (cont.)

- Actions
  - Suspend execution of all transactions
  - Force-write all main memory buffers that have been modified to disk
  - Write checkpoint record to log, force-write log to disk
  - Resume execution of transactions
- Consequences?
  - If transaction has a commit before checkpoint, no need to redo write operations
- How often to do a checkpoint?
  - Every *m* minutes or *t* transactions