CSE 444: Database Internals

Section 9:

2-Phase Commit and Replication

Today

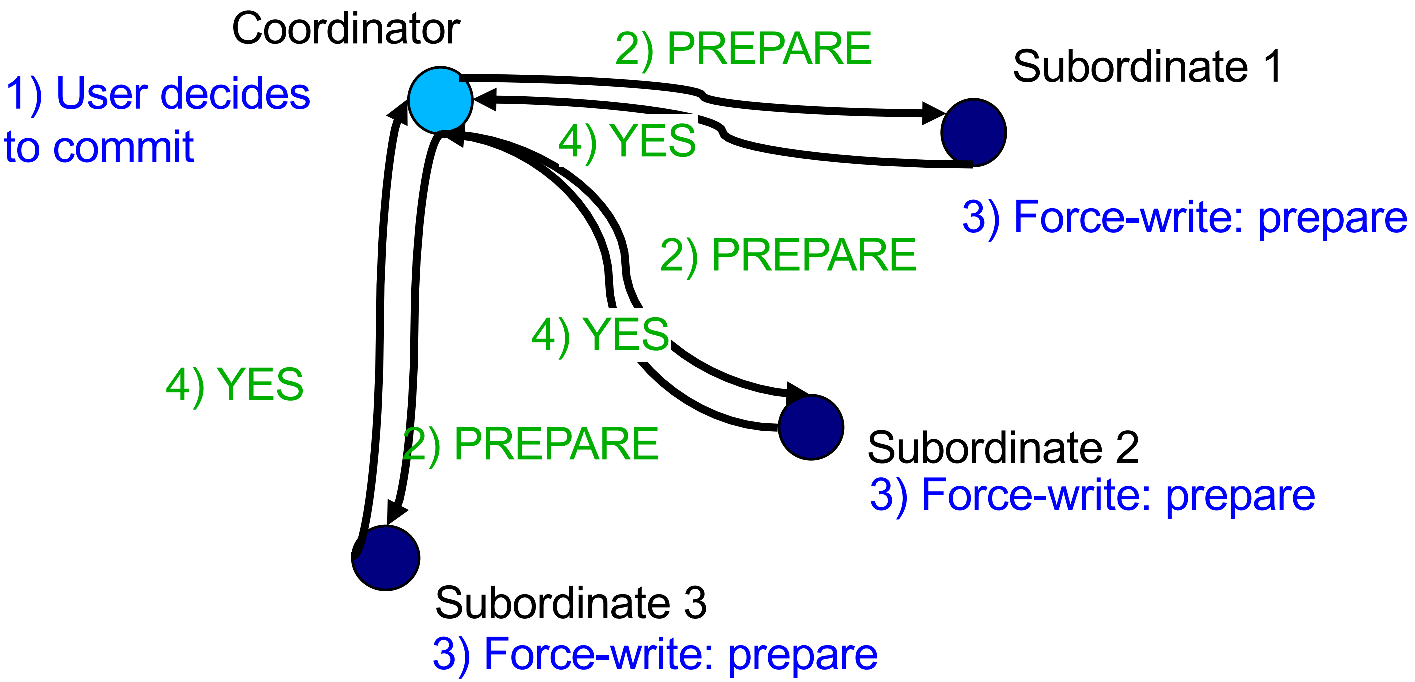
- **2-Phase Commit**
- **Replication**

Two-Phase Commit Protocol (2PC)

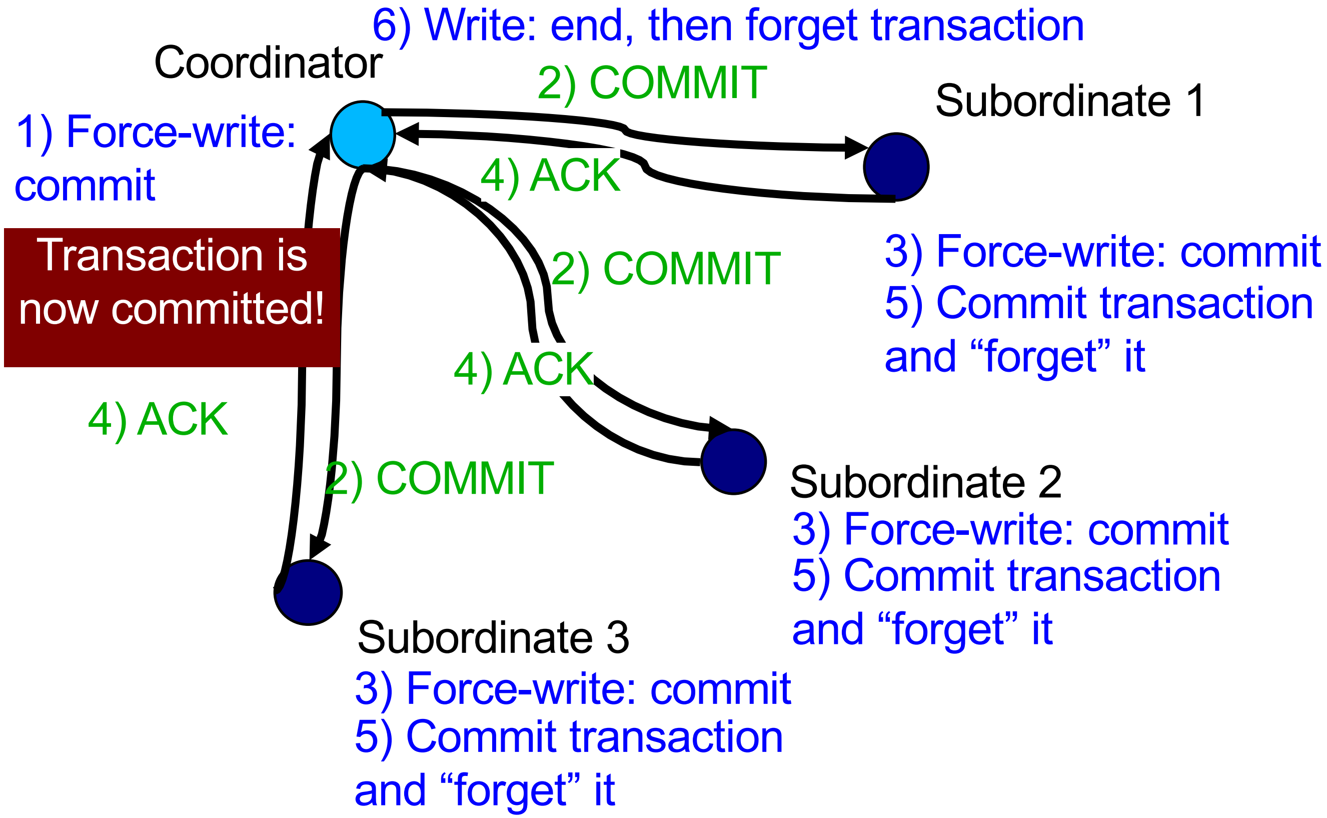
- One coordinator and many subordinates
  - **Phase 1**: Prepare
  - **Phase 2:** Commit or Abort

- **Principle**
  - When a process makes a decision: vote yes/no or commit/abort
  - When a subordinate wants to respond to a message: ack
  - **First force-write a log record** (to make sure it survives a failure) only then send message about decision

Phase 1: Prepare



Phase 2: Commit



6) Write: end, then forget transaction

Coordinator

1) Force-write: commit

Transaction is now committed!

2) COMMIT

4) ACK

2) COMMIT

4) ACK

4) ACK

2) COMMIT

Subordinate 1

3) Force-write: commit
5) Commit transaction and "forget" it

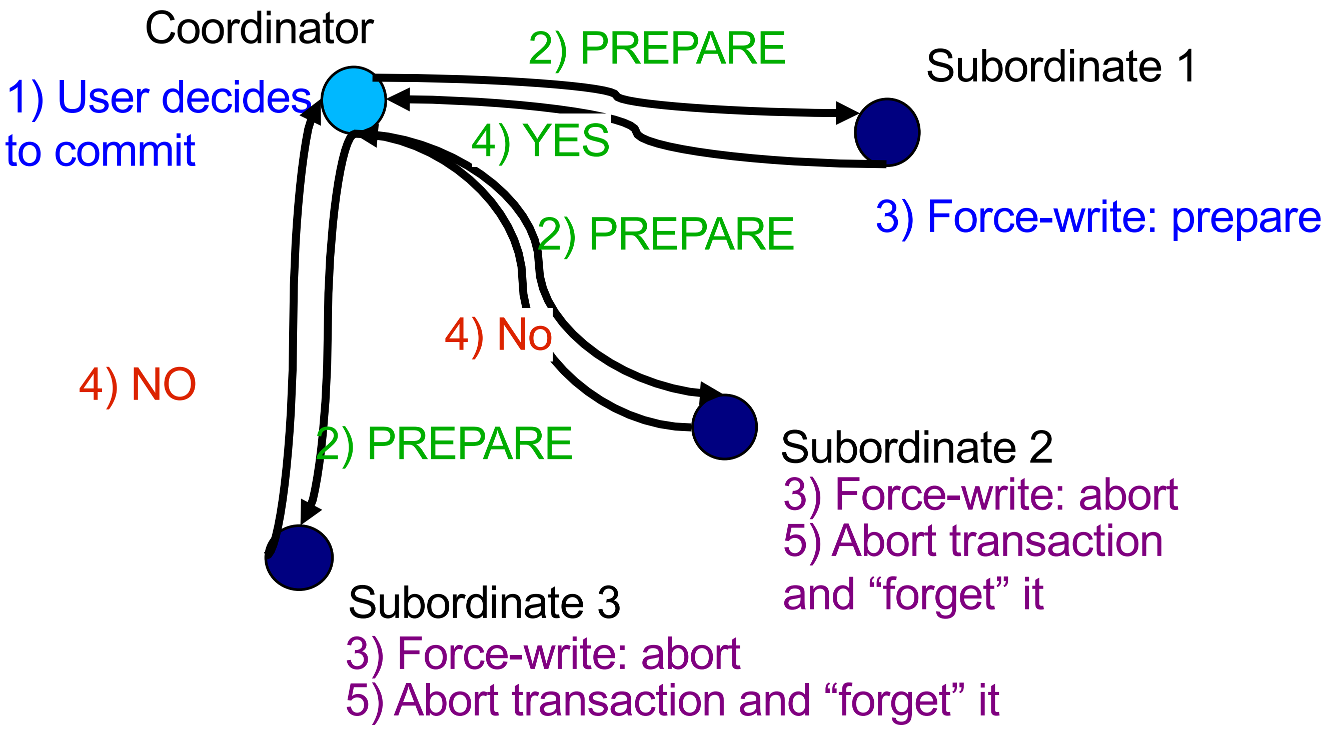
Subordinate 2

3) Force-write: commit
5) Commit transaction and "forget" it

Subordinate 3

3) Force-write: commit
5) Commit transaction and "forget" it

Review: 2PC with Abort



Coordinator

1) User decides to commit

2) PREPARE

Subordinate 1

4) YES

3) Force-write: prepare

2) PREPARE

4) No

4) NO

2) PREPARE

Subordinate 2

3) Force-write: abort

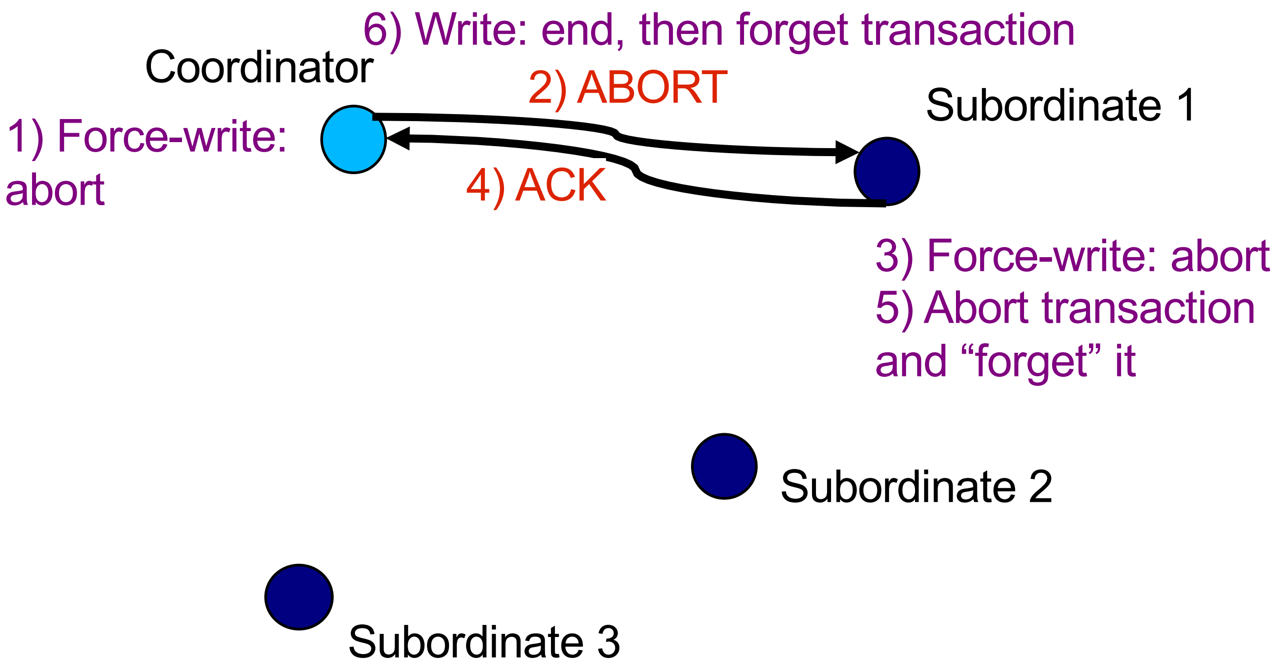
5) Abort transaction and “forget” it

Subordinate 3

3) Force-write: abort

5) Abort transaction and “forget” it

Review: 2PC with Abort



6) Write: end, then forget transaction

1) Force-write: abort

3) Force-write: abort
5) Abort transaction and "forget" it

2PC Crash/Recovery Scenarios

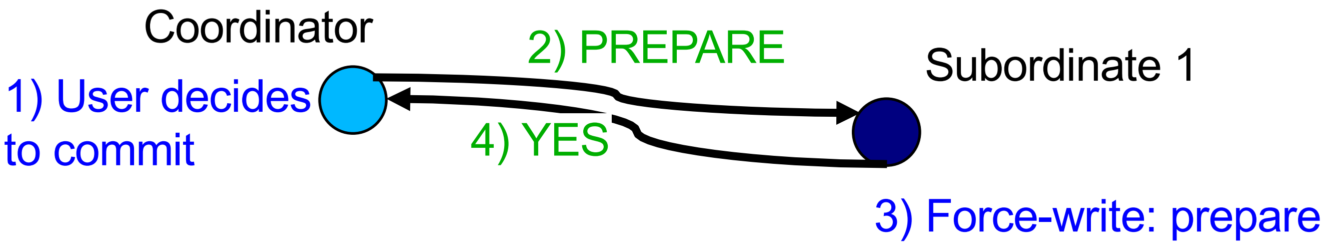
Recovery Process

- When a site comes back up after a crash, we invoke a **recovery** process that reads the log and processes all transactions at the time of the crash.
- The transactions at the site could have been the coordinator for some of these transactions or subordinates for others.

2PC – How a site recovers

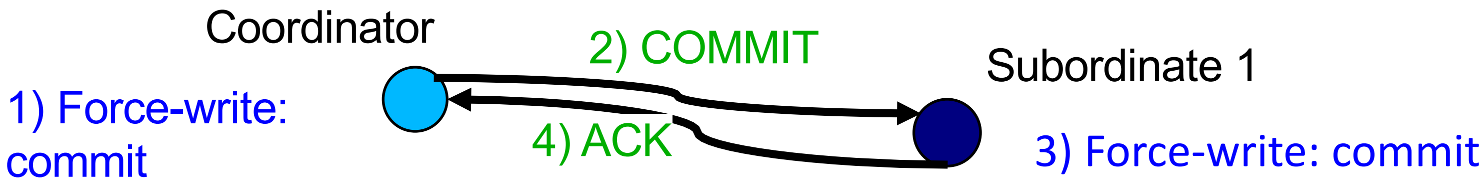
- **Scenario**: If we have a **PREPARE** log for T, but no **COMMIT/ABORT**.
  - *Subordinate*: Keep asking the coordinator for final decision on T. Once it responds, we can undo or redo the transaction.

Phase 1: Prepare



2PC – How a site recovers

- **Scenario**: If we find a **COMMIT** or **ABORT** log record for T
  - *Subordinate*: Will UNDO or REDO T.
  - *Coordinator*: Will periodically keep sending decision until it receives *acks* from all.



2PC – How a site recovers

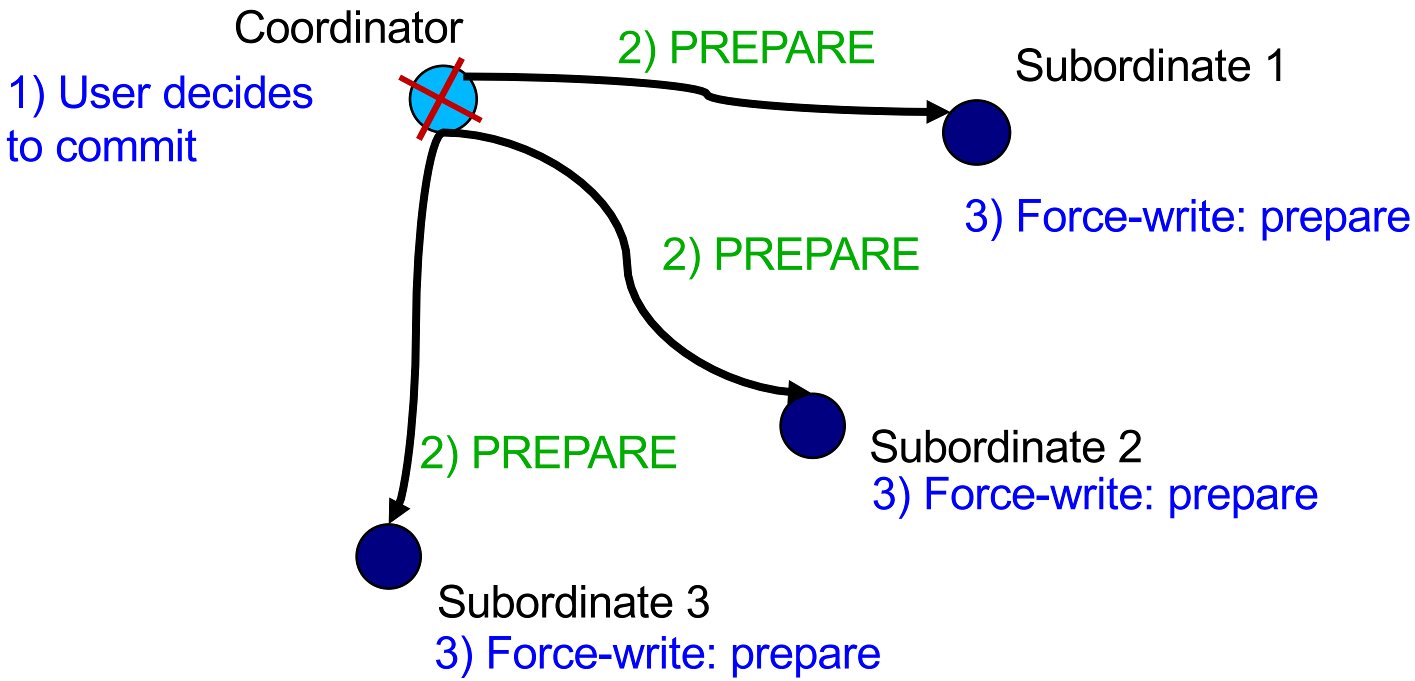
- **Scenario**: If we find no **PREPARE**, **COMMIT** or **ABORT**
  - *Coordinator*: Might have crashed after sending PREPARE. Might receive votes from subordinates for T, but coordinator does not have any information on T. Make decision to ABORT.
  - *Subordinate*: Could not have voted to commit before the crash, ABORT and UNDO T.

2PC – Communication Failures

- **Scenario**: Site we are communicating to has failed (assuming timeouts)
  - *Coordinator*: should simply **ABORT** the transaction if it waited too long to receive a vote from a subordinate.
  - *Subordinate*: If it has not yet responded with a decision to the prepare message (finds out coordinator is unavailable through a timeout), it should **ABORT**. If it voted "yes", it is simply blocked and needs to wait for the coordinator to recover.

Example #1

- In the two-phase commit protocol, what happens if the coordinator sends **PREPARE** messages and crashes before receiving any votes?



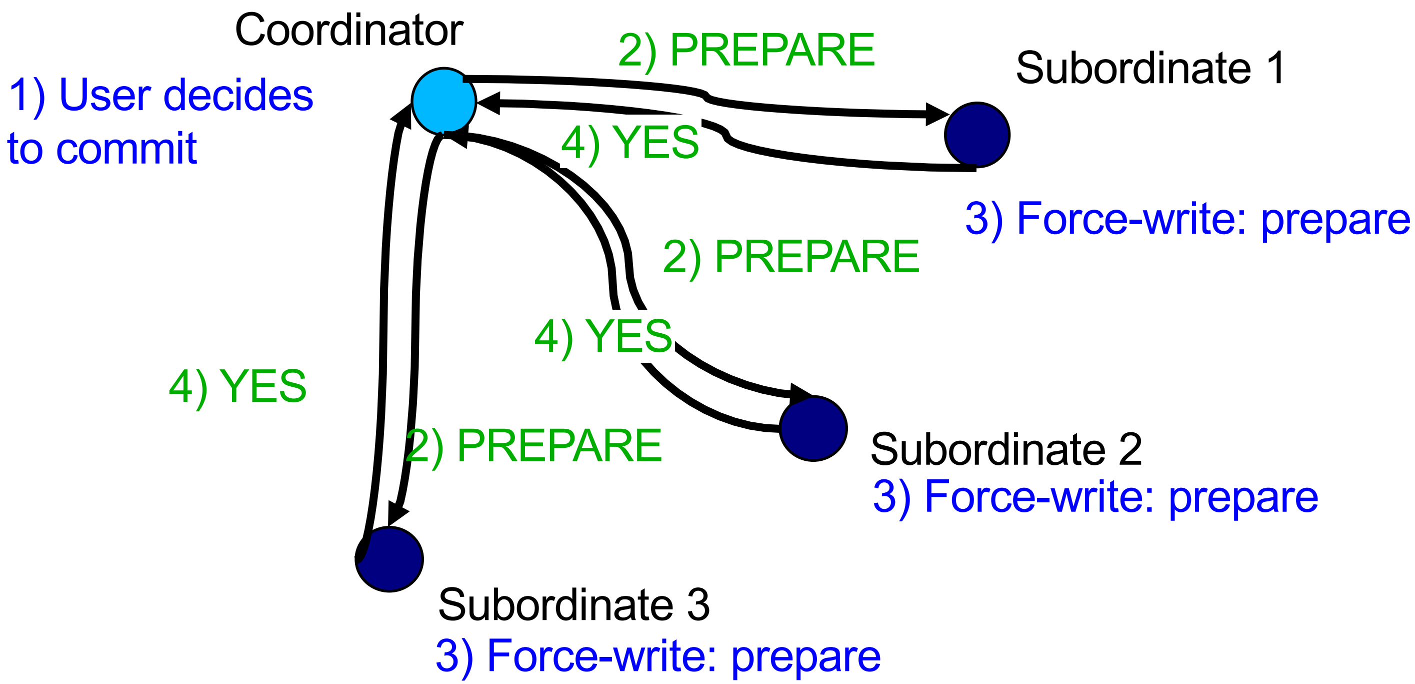
Example #1

- In the two-phase commit protocol, what happens if the coordinator sends **PREPARE** messages and crashes before receiving any votes?

- The coordinator will find that a transaction was executing, but no commit protocol record was written. Will **ABORT** T.

Example #2

- In 2PC, why do subordinates need to force-write a prepared log record before sending a YES VOTE?



Example #2

- In 2PC, why do subordinates need to force-write a **PREPARE** log record before sending a YES VOTE?

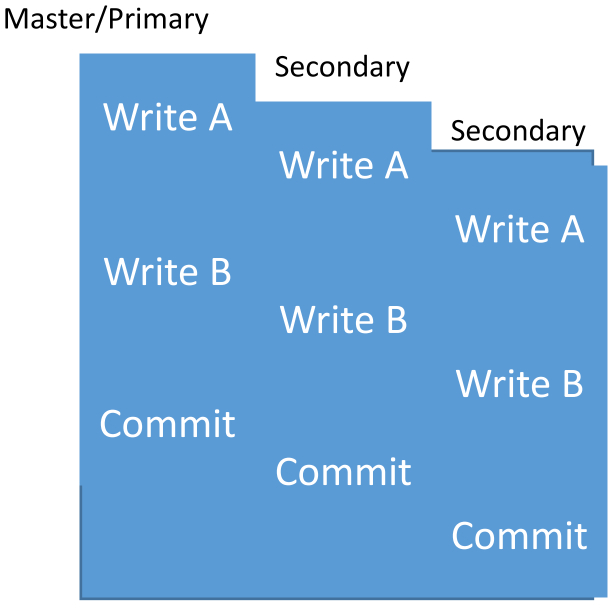
- Think about this case: Subordinate sends a YES vote without writing to the log. When recovering, it will find no commit record for the transaction. It will then ABORT, possibly resulting in an inconsistent state.

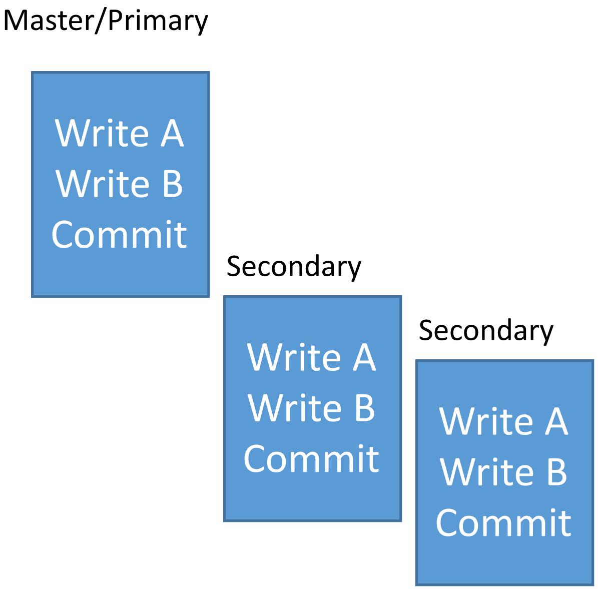
Replication

Synchronous vs. Asynchronous

- **Synchronous:** Updates are applied to all replicas of an object as part of the original transaction (needs global locks, 2PC).

- **Asynchronous:** One replica is updated by the originating transaction. Updates to other replicas propagate asynchronously, typically as a separate transaction for each node.





Master vs. Group

- **Master:**
  - Only the master can update.
  - All other replicas are read-only. If they want to update the object request the master do the update.

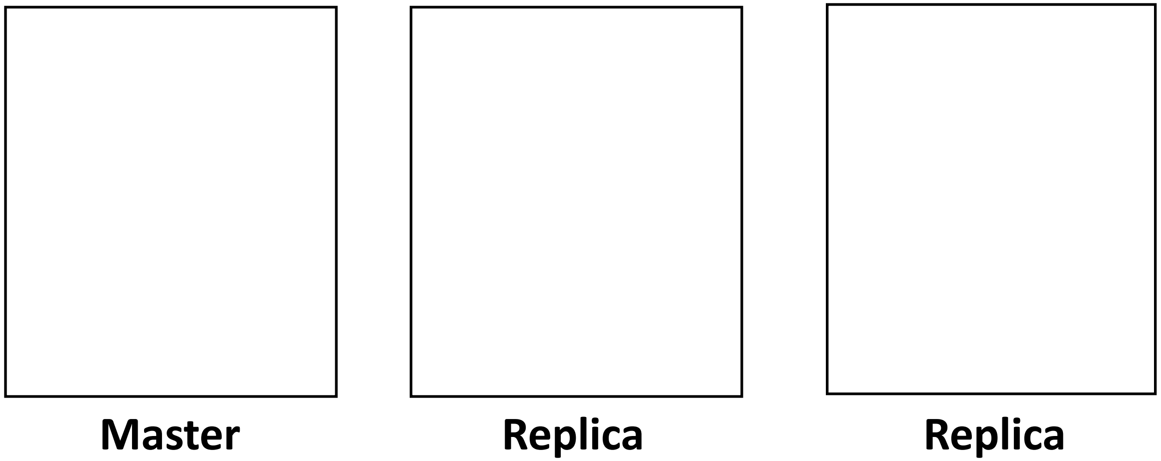
- **Group**:
  - Any node with a copy of a data item can update it (also called “update anywhere”)

Propagation vs. Ownership

	Synchronous	Asynchronous
Master	1 transaction 1 object owner	N transactions 1 object owner
Group	1 transactions N object owners	N transactions N object owners

Selecting a new primary

In ***synchronous master*** replication, when the master fails, why does a group of replicas need to have the majority of nodes in order to elect a primary?

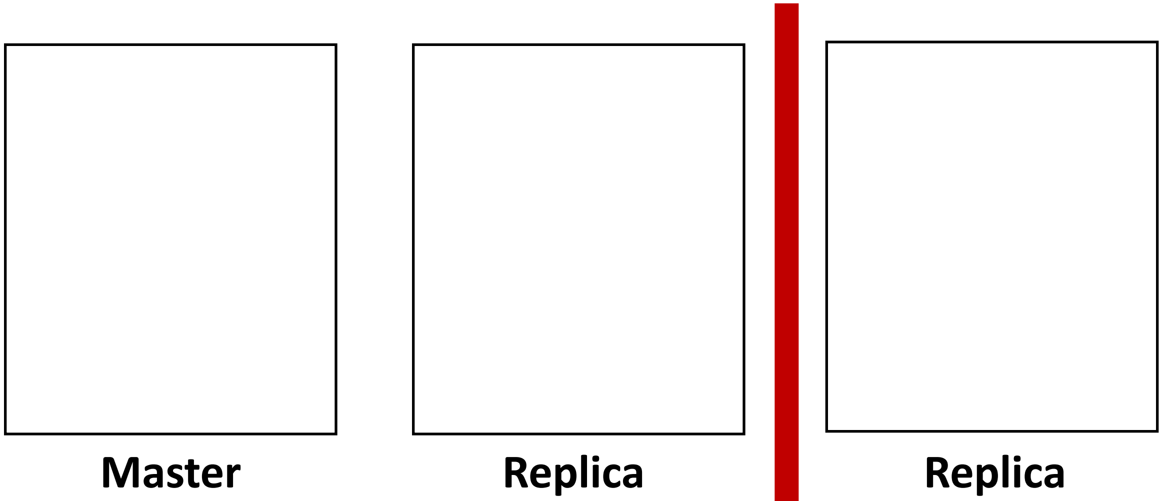


Master **Replica** **Replica**

Selecting a new primary

In ***synchronous master*** replication, when the master fails, why does a group of replicas need to have the majority of nodes in order to elect a primary?

The secondaries cannot differentiate between a crash failure of the master and a network partition.



Differences between synchronous and asynchronous

Synchronous

- **Option 1**: Use a master
- **Option 2**: Use a quorum (voting or read-many-copies)
- Favors consistency over availability
- High overhead

Differences between synchronous and asynchronous

Synchronous

- **Option 1**: Use a master
- **Option 2**: Use a quorum (voting or read-many-copies)
- Favors consistency over availability
- High overhead

Asynchronous:

- **Option 1**:   Use a master.
  - If it fails, might lose some of the most recent writes
- **Option 2**:   Allow updates to go everywhere. This is **multi-master.**
  - Can create conflicts that will need to be resolved
  - Generally in practice, this works best if the data is disjoint
- Favors availability over consistency