HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Distributed Data Management
Transactions

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut

An OLTP Topic

Motivation

- Most database interactions consist of multiple, coherent operations.
- Interactions can be affected by other interfering interactions and errors.
  - ➢ Database must ensure that interactions work correctly (→ transactions).

OLAP vs. OLTP

- OLAP systems …
  - prepare the data once.
  - send complex but individual, ungrouped read-queries.
  - resend failed queries and do not interfere.
- OLTP systems …
  - change the data frequently.
  - send coherent operations with mixed read/write load.
  - must ensure that interactions succeed consistently.

No real need for transactions

Transactions!

Definition

See lecture "Database Systems I" by Prof. Naumann

Transaction

- A sequence of database operations (read/write) that carry a database from one state into another (possibly changed) state.
- Transactions operate in different items (multi-object operations).
- Transactions succeed (commit) or fail (abort/rollback).
- The ACID safety guarantees must be satisfied:
  - Atomicity: A transaction is executed entirely or not at all.
  - Consistency: A transaction carries the database from a consistent state into a consistent state (consistent = logically and technically sound).
  - Isolation: A transaction does not contend with other transactions. Contentious access to data is moderated by the database so that transactions appear to run sequentially.
  - Durability: A transaction causes, if successful, a persistent change to the database.

Most distributed DBMSs do not support transactions and stick to the BASE consistency model

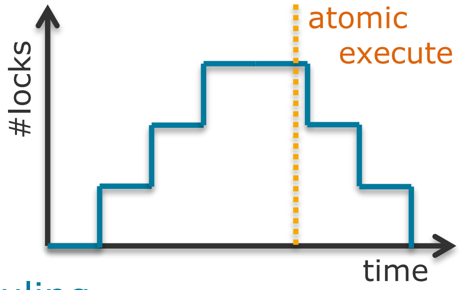
Transactions

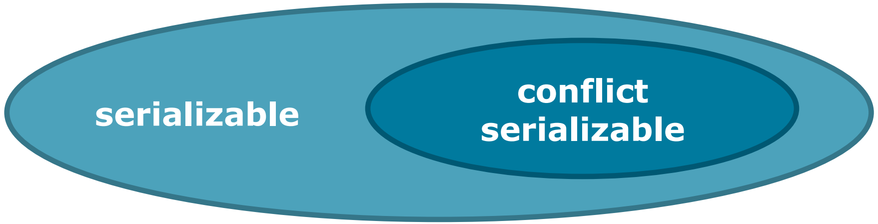
Achieving Isolation

See lecture “Database Systems I” by Prof. Naumann

Locking

- Block an item (row, document, …) for exclusive reads/writes of one transaction.
- Two-Phase Locking:
  - All locks in one transaction are set before the first lock is given up.
  - Technique to ensure conflict-serializable execution of transactions.





Scheduling

- Creating an execution order for transaction operations.
- See: serial schedule, serializable schedule, legal schedule

Locking is an issue if data is replicated!

Causal Ordering (recap)

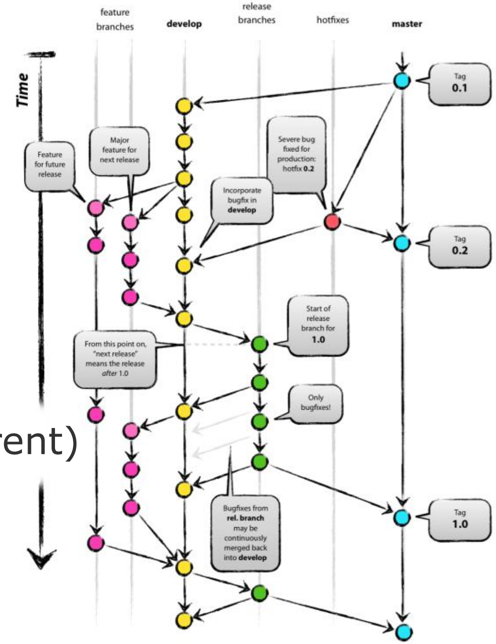
Thinking:
timelines that branch/merge;
events compare only along lines
➢ GIT

Linearizable (and Total Order Broadcast)

- Imposes a total order:
  - All events can be compared.
  - For one object, only the newest event is relevant.
- Implies causality:
  - A linear order is always also a causal order of the events.
- Is expensive (due to global order enforcement)

Causal ordering

- Imposes a partial order:
  - Some events are comparable (causal), others are not (concurrent)
  - For many events some partial order is just fine:
    - Order of writes, side-channel messages, **transactions** …
- Is cheaper (order enforcement only for related events)



Causal Consistency

Causal ordering:

- Example: reads and writes in transactional systems
  - Reads and writes are causally unrelated unless they …
    - target the same object or
    - connect through transactions.
- A system that guarantees causal ordering is causal consistent.

Inconsistencies

$(w/r)_{<transaction>}(<field>)$

Dirty Read: (write-read conflict)

- Reading a inconsistent value
- Example: $w_1(A)$ **$r_2(A)$** $w_1(A)$ — **A** was not finished and never supposed to be read.

Non-Repeatable Read: (read-write conflict)

- Reading an outdated value
- Example: **$r_1(A)$** $w_2(A)$ $r_1(A)$ — Re-reading **A** resulted in a different, inconsistent value.

Lost Update: (write-write conflict)

- Losing a written value
- Example: **$w_1(A)$** $w_2(A)$ $r_1(A)$ — Update of **A** is lost during the transaction.

Phantom Read: (read-write and write-read conflict)

- Reading/writing of inconsistent values
- Example: **$r_1(A)$** $w_2(B)$ **$r_1(B)$** $w_2(A)$ — Either **A**'s or **B**'s value is a phantom (should not be there).

Isolation

See lecture "Database Systems I" by Prof. Naumann

Isolation levels

- To ensure ACID, transactions must be serializable.
  - ➢ Very costly, but any weaker level breaks isolation.

Usually default

Isolations-Level	Lost Update	Dirty Reads	Non-Repeatable Reads	Phantom Rea
READ_UNCOMMITTED	prevented	possible	possible	possible
READ_COMMITTED	prevented	prevented	possible	possible
REPEATABLE_READ	prevented	prevented	prevented	possible
SERIALIZABLE	prevented	prevented	prevented	prevented

- READ_COMMITTED:
  - Read only committed values (remember local logical UNDO/REDO logs).
  - No dirty reads, because only consistent values are committed.
  - Still non-repeatable reads, because transactions interleave.

Isolation

Causally related operations are ordered (unrelated operations still occur concurrently)

Isolation levels

- Snapshot isolation: "readers don't block writers and vice versa"
  - Transactions see only data that was committed when they started.

Isolations-Level	Lost Update	Dirty Reads	Non-Repeatable Reads	Phantom Reads
READ_UNCOMMITTED	prevented	possible	possible	possible
READ_COMMITTED	prevented	prevented	possible	possible
REPEATABLE_READ	prevented	prevented	prevented	possible
SERIALIZABLE	prevented	prevented	prevented	prevented

  - Uncommitted transactions may read old values; hence, causal consistency but no linearizability!
  - Is expensive, because it not only orders the events for the same object but also for an entire transaction!
  - Implementations: shared/exclusive locks or multi-version concurrency control (MVCC)

Keep both old and new value until commit; let others read the old value

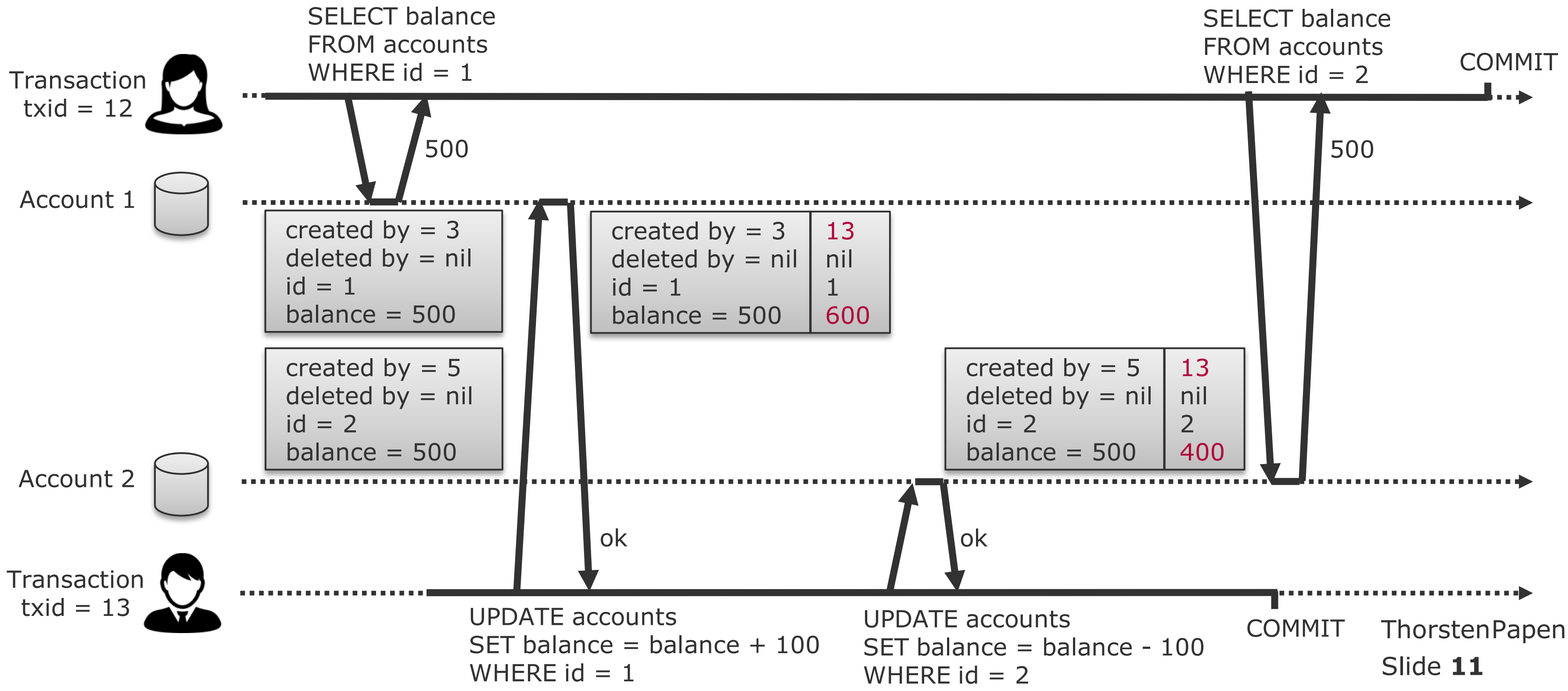
Snapshot Isolation via MVCC

Snapshot Isolation via MVCC

- For each entry (row, key-value pair, …) store `created by` and `deleted by` fields.
- Instead of changing entries directly, always append new versions.
- Transactions can now operate on consistent snapshots (= changes up to a fixed version).
- Algorithm:
  - At transaction start, make a list of all yet un-committed transactions.
  - During execution, ignore all changes made by …
    a) **un-committed transactions** from the start;
    b) **aborted transactions;**
    c) **newer transactions** (i.e. transactions with higher transaction id).
  - At transaction end, commit all changes; if write conflicts exist, rollback.

➢ MVCC is an optimistic approach that performs well if transactions do not collide frequently but causes many rollbacks otherwise.

Snapshot Isolation via MVCC



Transactions

Isolation

Isolation levels

- Snapshot isolation: “readers don’t block writers and vice versa”

Isolations-Level	Lost Update	Dirty Reads	Non-Repeatable Reads	Phantom Reads
READ_UNCOMMITTED	prevented	possible	possible	possi
READ_COMMITTED	prevented	prevented	possible	possi
REPEATABLE_READ	prevented	prevented	prevented	possi
SERIALIZABLE	prevented	prevented	prevented	preve

Read/Write locks avoid this problem, because all read applicants are locked for writing.

➢ SERIALIZABLE

- Although it avoids all standard anomalies, it is not truly SERIALIZABLE:
  - Write Skew: (related to phantom reads)
    - Same reads lead to different, non-conflicting but inconsistent writes.
    - Example: Two transactions scan a list of job applicants. Both see that no applicant was hired, yet, and mark one applicant as hired. If they hire different applicants, no conflict is created but the table is inconsistent (two hires for one job).

Consensus for Transaction Commits

Two-Phase Commit (2PC)

"Let's be ACID conform!"

- Goal:
  - Ensure that all nodes consistently commit or abort a transaction.
  - Consensus = "all agree"
- Requirements:
  - One node that acts as a coordinator for a transaction (e.g. leader).
  - Coordinator must be able to generate unique IDs for transactions.
- Steps: (coordinator view)
  - **Writing**: Send the data to all nodes.
  - **Phase 1**: Upon global success, send prepare requests to all nodes.
  - **Phase 2**: Upon global success, send commit request to all nodes.
    - ➢ 2PC transaction commits are blocking operations.

Consensus for Transaction Commits

Two-Phase Commit (2PC)

- Steps:

Obtain unique transaction ID

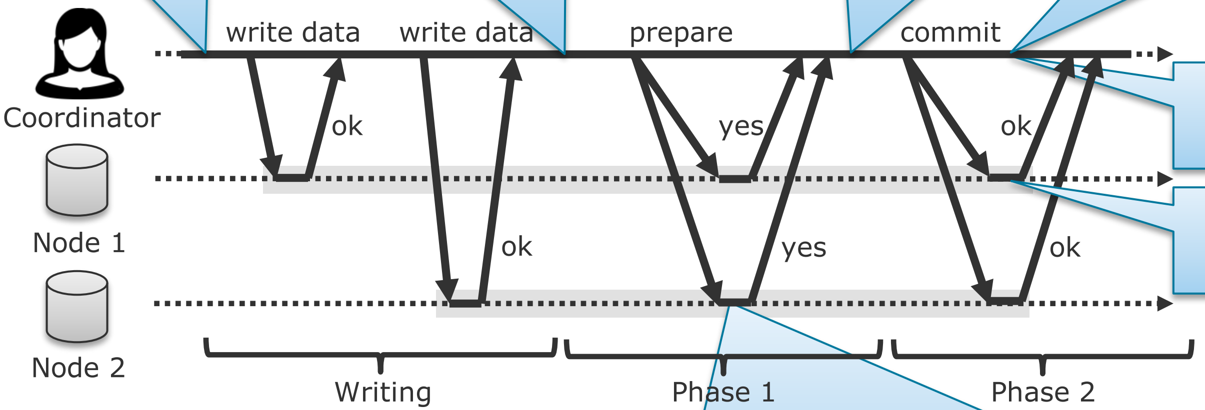
Whenever any response is missing/negative, abort transaction.

Make a decision and append it to log on disk.

➢ commit point

Keep sending commit messages until all nodes acknowledged.

What if coordinator cannot recover and a new coordinator must be elected?



If coordinator crashes: recover and continue sending commits/aborts.

If node crashes: recover (and query coordinator)!

= locks held by transaction

Get ready to commit (append all writes to log on disk).

➢ Crashes, power failures, exhausted memory, … are no excuses later on!

Consensus for Transaction Commits

Hasso Plattner Institut

Three-Phase Commit (3PC)

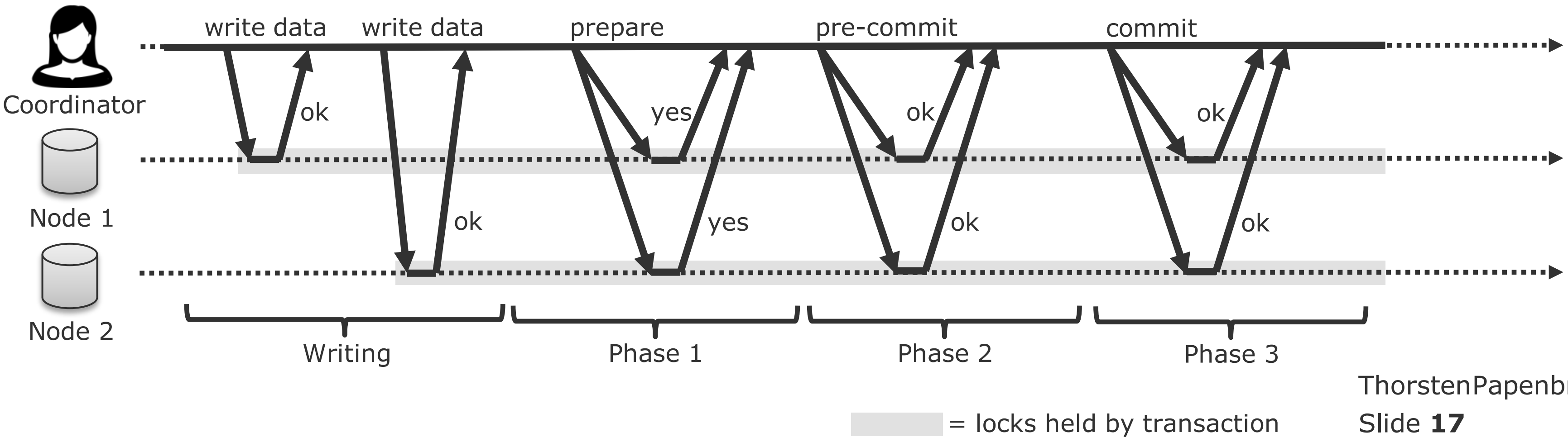
- Extension of the 2PC protocol that safely handles unrecoverable coordinators
- Idea: Spit the commit phase into two rounds.
- Steps: (coordinator view)
  - **Writing**: Send the data to all nodes.
  - **Phase 1**: Upon global success, send prepare requests to all nodes.
  - **Phase 2**: Upon global success, send pre-commit request to all nodes.
  - **Phase 3**: Upon global success, send commit request to all nodes.
- If the coordinator dies:
  - The new coordinator asks all nodes for their state.
  - If at least one node is in pre-commit phase, the new coordinator knows that the decision to commit was made and continues to push pre-commit (and then commit) messages.

Without the pre-commit phase, a new coordinator cannot know if a commit+close was already done by some node.

Consensus for Transaction Commits

Three-Phase Commit (3PC)

- Steps:

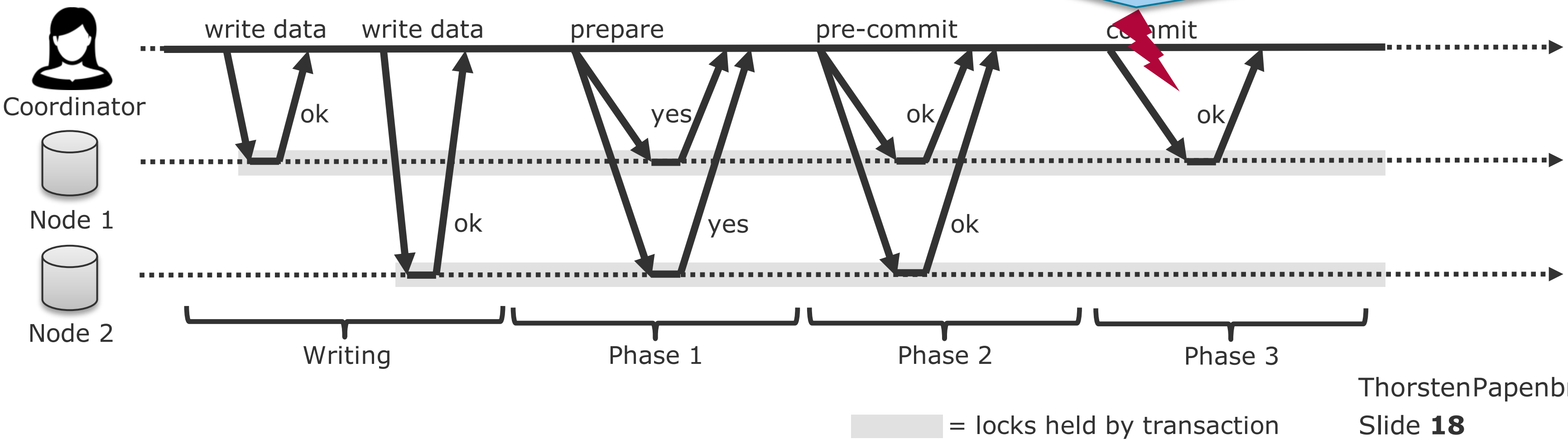


Consensus for Transaction Commits

Three-Phase Commit (3PC)

- Steps:

New coordinator sees node 2 in pre-commit state;
it re-sends pre-commit to node 1 and
continues pushing commit messages to node 1 and 2
(node 1 simply plays the protocol as it knows the transaction is already closed locally).

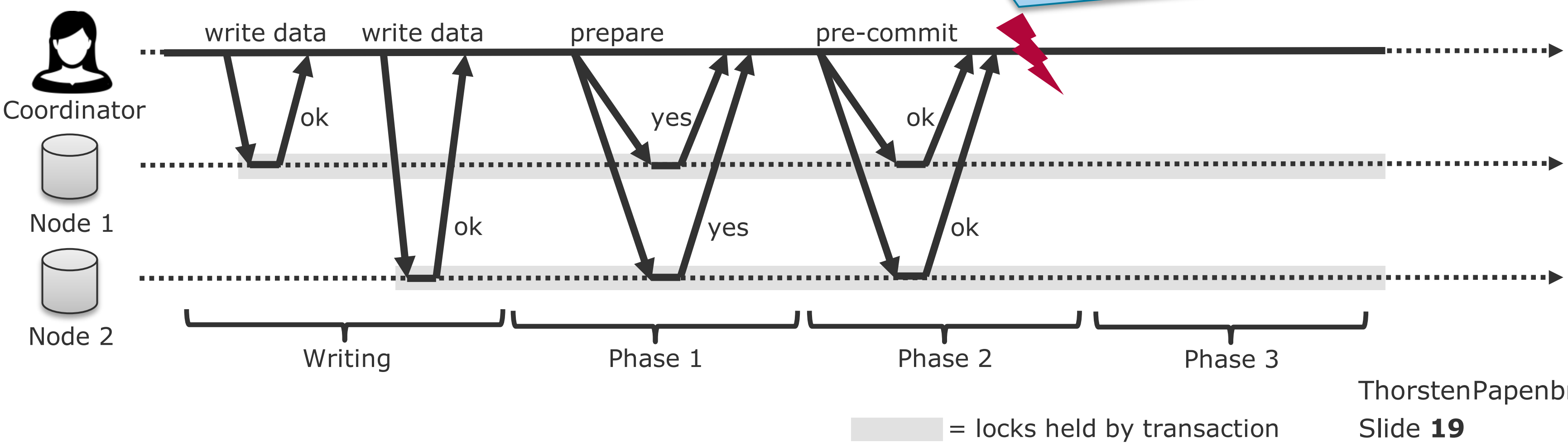


Consensus for Transaction Commits

Three-Phase Commit (3PC)

- Steps:

New coordinator sees node 1 and 2 in pre-commit state;
it continues pushing commit messages to node 1 and 2.

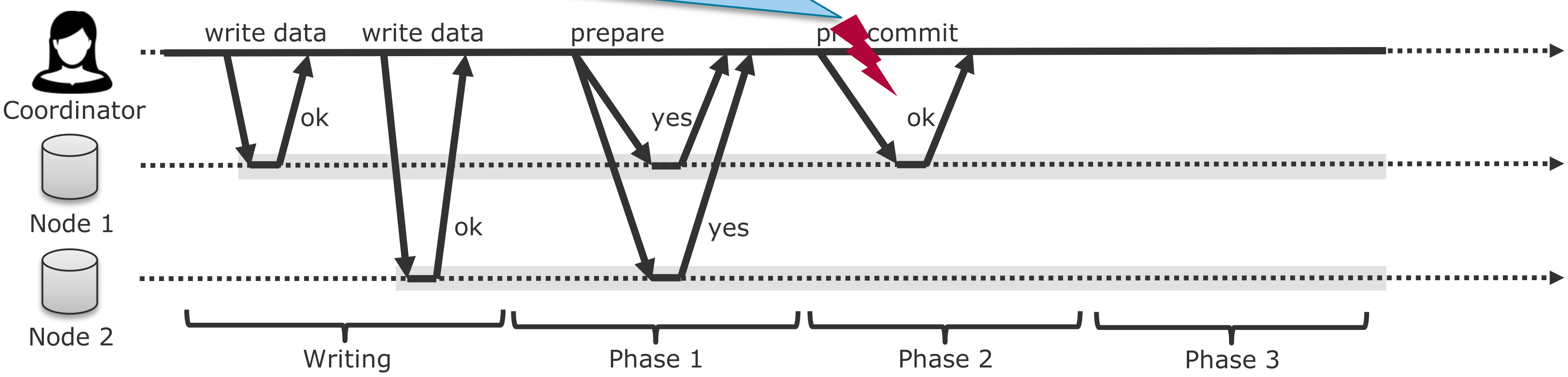


Consensus for Transaction Commits

Three-Phase Commit (3PC)

- Steps:

New coordinator sees node 1 in pre-commit state;
it sends pre-commit to node 2 and
then pushes commit messages to node 1 and 2.



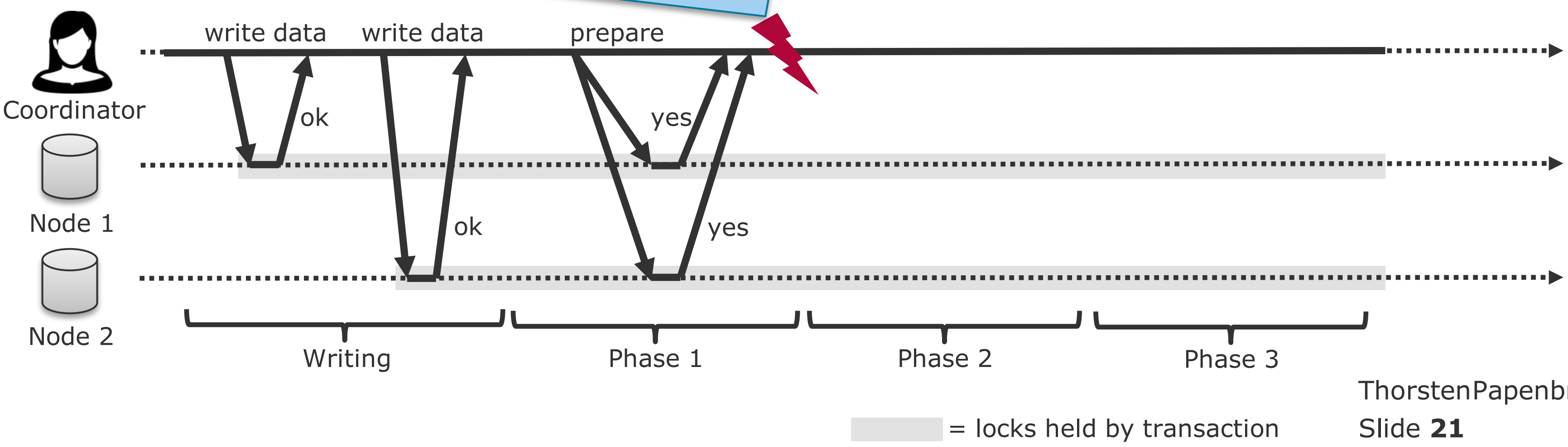
= locks held by transaction

Consensus for Transaction Commits

Three-Phase Commit (3PC)

- Steps:

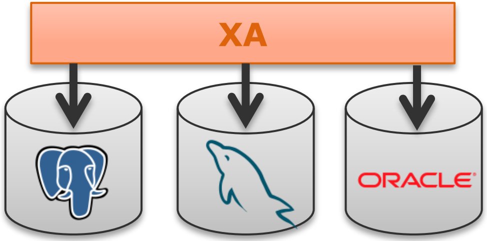
New coordinator sees all nodes in prepare (or earlier) state;
it sends abort messages to all nodes,
because the decision to commit was not made
(nothing was committed yet).



Consensus for Transaction Commits

Two/Three-Phase Commit (2PC / 3PC)

- What if the distributed database is a combination of different DBMS systems?



- eXtended Architecture (XA):
  - Standard for implementing 2PC across multiple DBMSs
  - Implemented as C API with bindings to e.g. Java:
    - Java Transaction API (JTA) supported by various drivers for …
      - databases, i.e., Java Database Connectivity (JDBC) and
      - message brokers, i.e., Java Message Service (JMS).
  - Used in:
    - Databases: PostgreSQL, MySQL, DB2, SQL Server, Oracle, …
    - Message Broker: ActiveMQ, HornetQ, MSMQ, IBM MQ, …

Consensus for Transaction Commits

Two/Three-Phase Commit (2PC / 3PC)

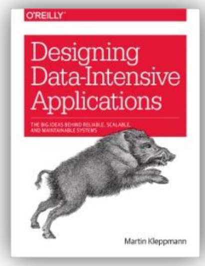
2PC is no good consensus protocol for **non-transactional votings.**

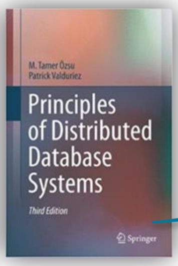
- Evaluation 2PC:
  - Expensive: about 10 times slower than single-node transactions in MySQL
  - Risky: locks are held indefinitely long if coordinator is lost
- Evaluation 3PC:
  - Expensive: even more expensive than 2PC
  - Blocking: locks are held for long times
  - Complex: automatically electing a new leader if the first failed is consensus voting inside a consensus protocol!

➢ Both are merely used in practical implementations.

Summary

- Transaction support costs memory resources:
  - Additional fields (`lock` or `changed`/`deleted`), versions, temporary lists …
- Transaction support costs computing resources:
  - Setting and checking locks, searching and cleaning versions …
- Transaction support scales badly in distributed systems:
  - Many actions require voting and/or change propagation.
- Transaction support is an open research area:
  - Achieving consistency for individual values in distributed systems is challenging; achieving the same for sequences of changes is even harder!





If you like to read more about distributed transaction handling, have a look at these two books!

Chapter 7. Transactions