Distributed Data Management
Transactions

Thorsten Papenbrock
Felix Naumann

F-2.03/F-2.04, Campus II

Hasso Plattner Institut

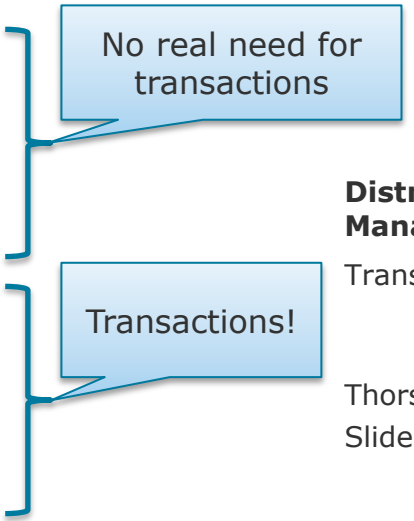
An OLTP Topic

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



Definition

See lecture "Database Systems I" by Prof. Naumann

Hasso Plattner Institut

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.

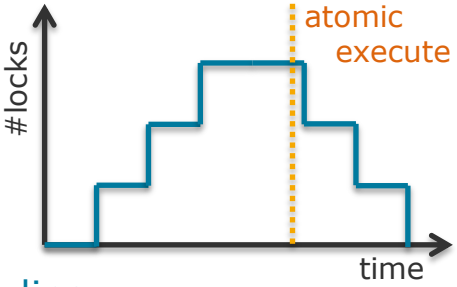
Most distributed DBMSs do not support transactions and stick to the BASE consistency model

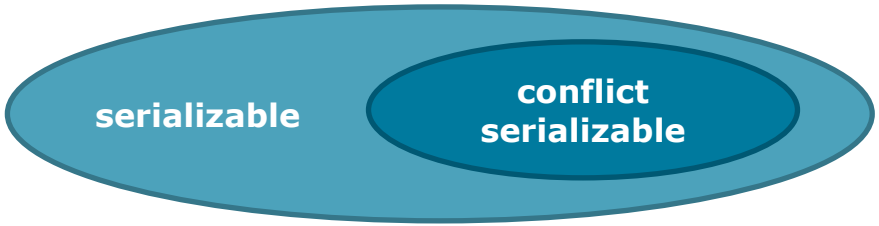
Achieving Isolation

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

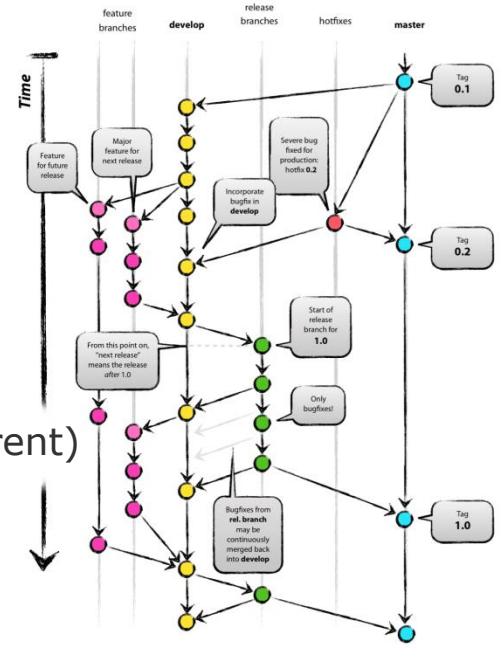
- Creating an execution order for transaction operations
- See: serial schedule, serializable schedule, legal schedule

Locking is an issue if data is replicated!

Causal Ordering

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

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

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

See lecture "Database Systems I" by Prof. Naumann

Dirty Read: (write-read conflict)

- Reading a wrong value
- Example: $w_1(A)$ $\mathbf{r_2(A)}$ $w_1(A)$

Non-Repeatable Read: (read-write conflict)

- Reading an outdated value
- Example: $\mathbf{r_1(A)}$ $w_2(A)$ $r_1(A)$

Lost Update: (write-write conflict)

- Losing a written value
- Example: $\mathbf{w_1(A)}$ $w_2(A)$ $r_1(A)$

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

- Reading/writing of inconsistent values
- Example: $\mathbf{r_1(A)}$ $w_2(B)$ $\mathbf{r_1(B)}$ $w_2(A)$

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

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

Usually default

Isolation

Isolation levels

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

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

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

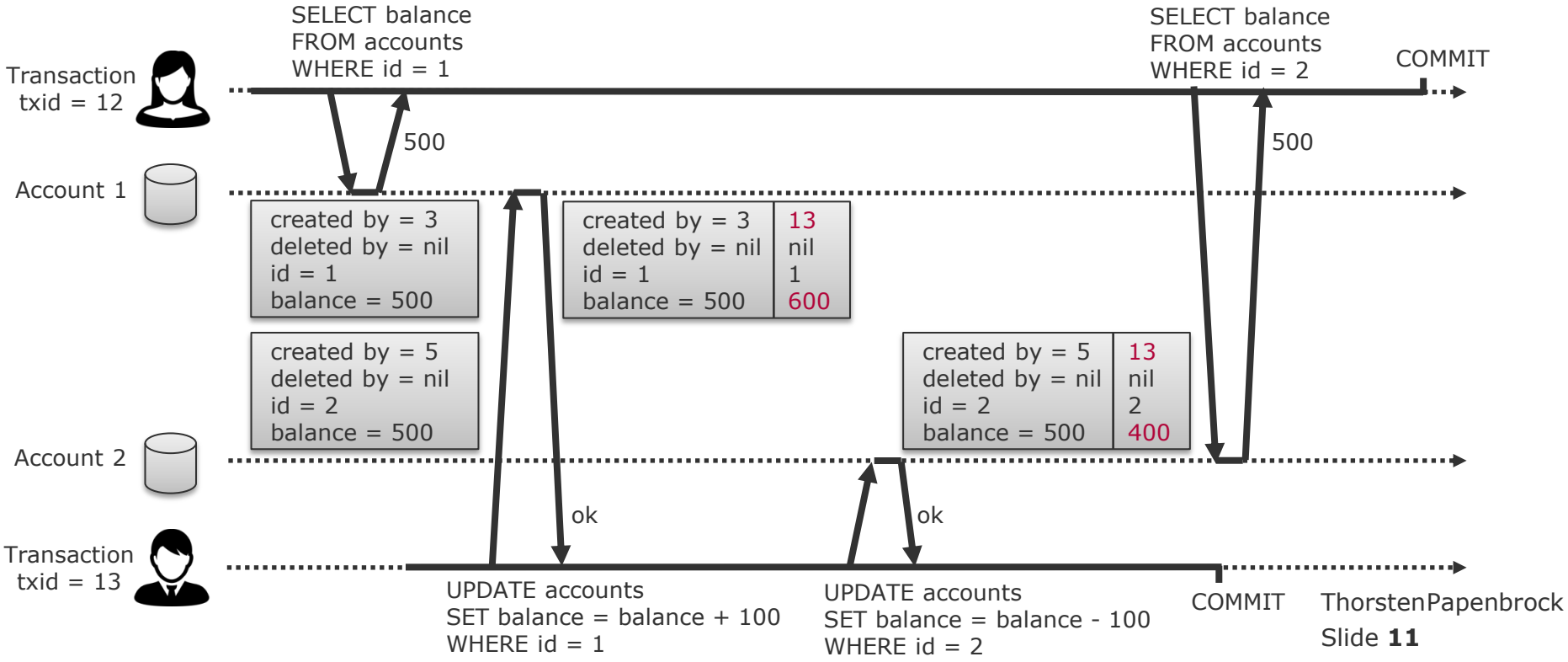
Keep both old and new value until commit; let others read the old value

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 …
    - **un-committed transactions** from the start
    - **aborted transactions**
    - **newer transactions** (i.e. transactions with higher transaction id)

Snapshot Isolation via MVCC



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

See lecture "Database Systems II" by Prof. Naumann for more details and 3PC

Two-Phase Commit (2PC)

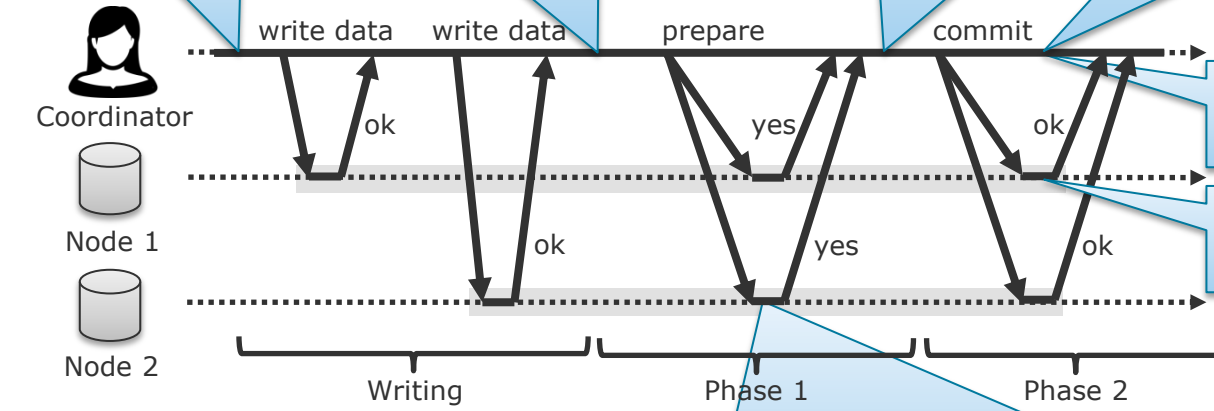
- Steps:

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



If coordinator crashes: recover and continue sending commits/aborts

If node crashes: recover (and query coordinator)

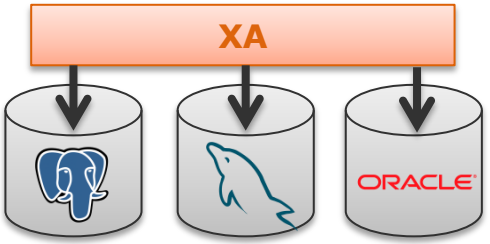
= locks held by transaction

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

Two-Phase Commit (2PC)

- 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-Phase Commit (2PC)

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

- Evaluation:
  - Expensive: e.g. 2PC is about 10 times slower than single-node transactions in MySQL
  - Blocking: locks are held for long times (indefinitely long if coordinator is lost)
- Extension:
  - Three-Phase-Commit (3PC):
    - Asynchronous, non-blocking transaction commits
    - Automatically choose another leader if the first one failed
      - ➢ Consensus voting inside a consensus protocol!
    - Complex and error prone (leader election = failover = risky)
      - ➢ 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!


PESSIMISTIC PENINSULA
No Phantoms
Actual serial execution
Two-phase locking
SERIALIZABLE
OPTIMISTIC PENINSULA
Serializable snapshot isolation
No Write Skew
SNAPSHOT ISOLATION
MVCC
No Read Skew
No Lost Updates
Row-level locking
No Dirty Writes
No Dirty Reads
READ COMMITTED
Temple of Multi-Object Atomicity
Atomic commit
To Distributed Transactions (Chapter 9)