Distributed Data Analytics
Transactions

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut

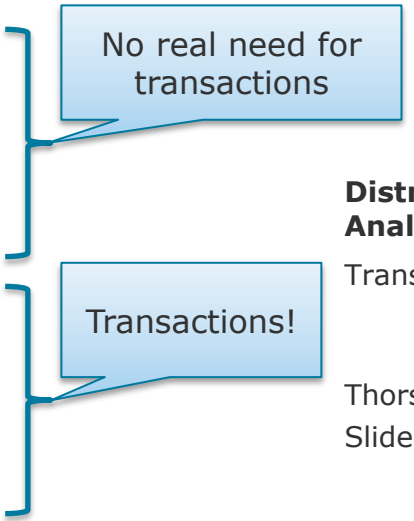
An OLTP Topic

Motivation

- Most database interactions consist of multiple, coherent operations
- Interactions can be disturbed 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 reads 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

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 can assume that it operates alone on the database; no other transaction can interfere with it.
  - 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

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- Reading an outdated value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

- Snapshot isolation: "reader don't block writers and vice versa"
  - Transactions see only data that was committed when they started
  - Implementations:
    shared/exclusive locks or multi-version concurrency control (MVCC)
  - Very important for long running backups and analytics

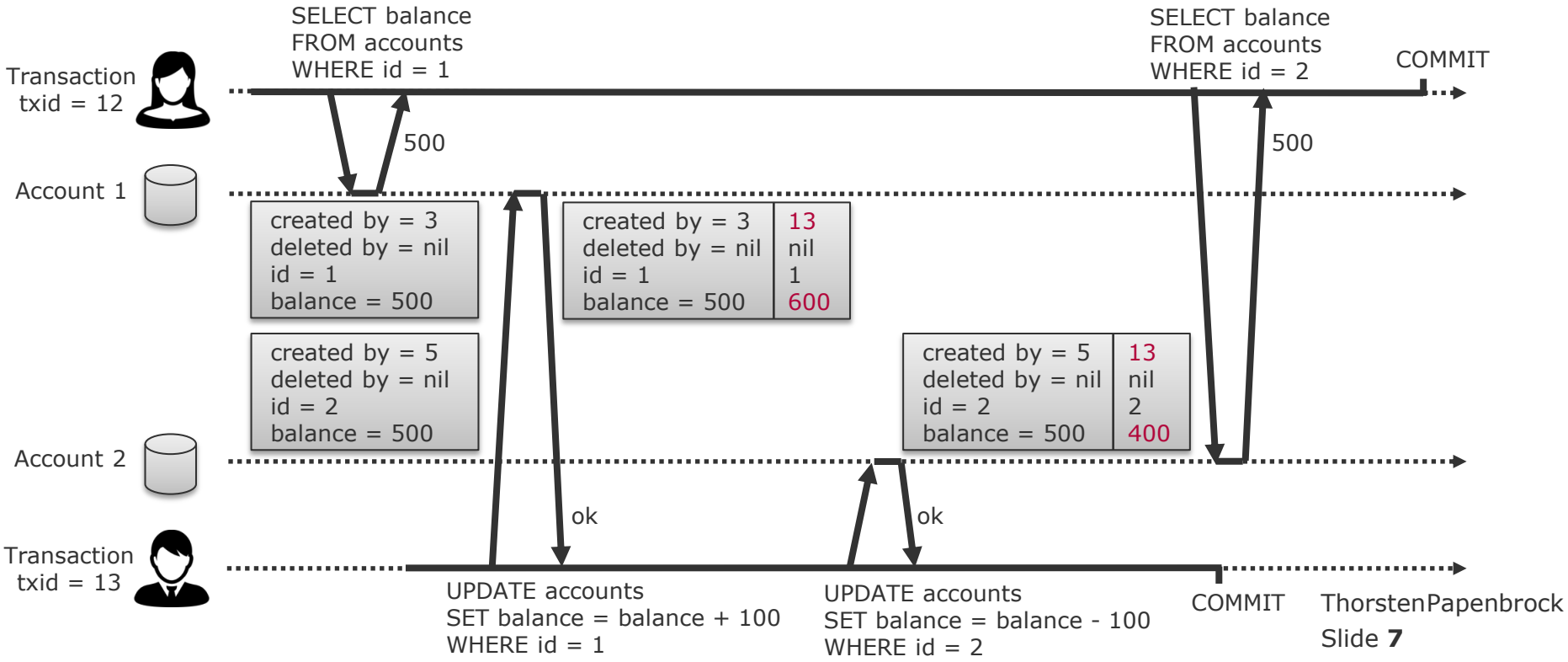
Keep both old and new value until commit; let others read the old value

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

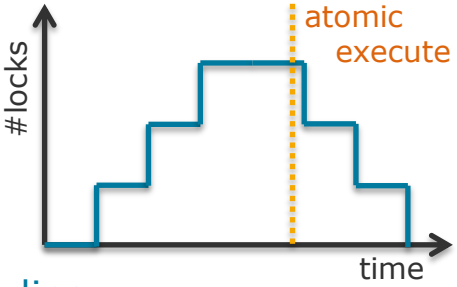


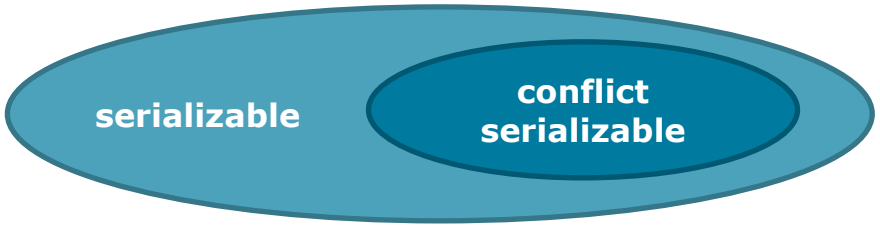
Achieving Isolation

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

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)