Datenbanksysteme II: Synchronization of Concurrent Transactions

Ulf Leser

Content of this Lecture

- Synchronization
- Serial and Serializable Schedules
- Locking and Deadlocks
- Timestamp Synchronization and SQL Isolation Levels

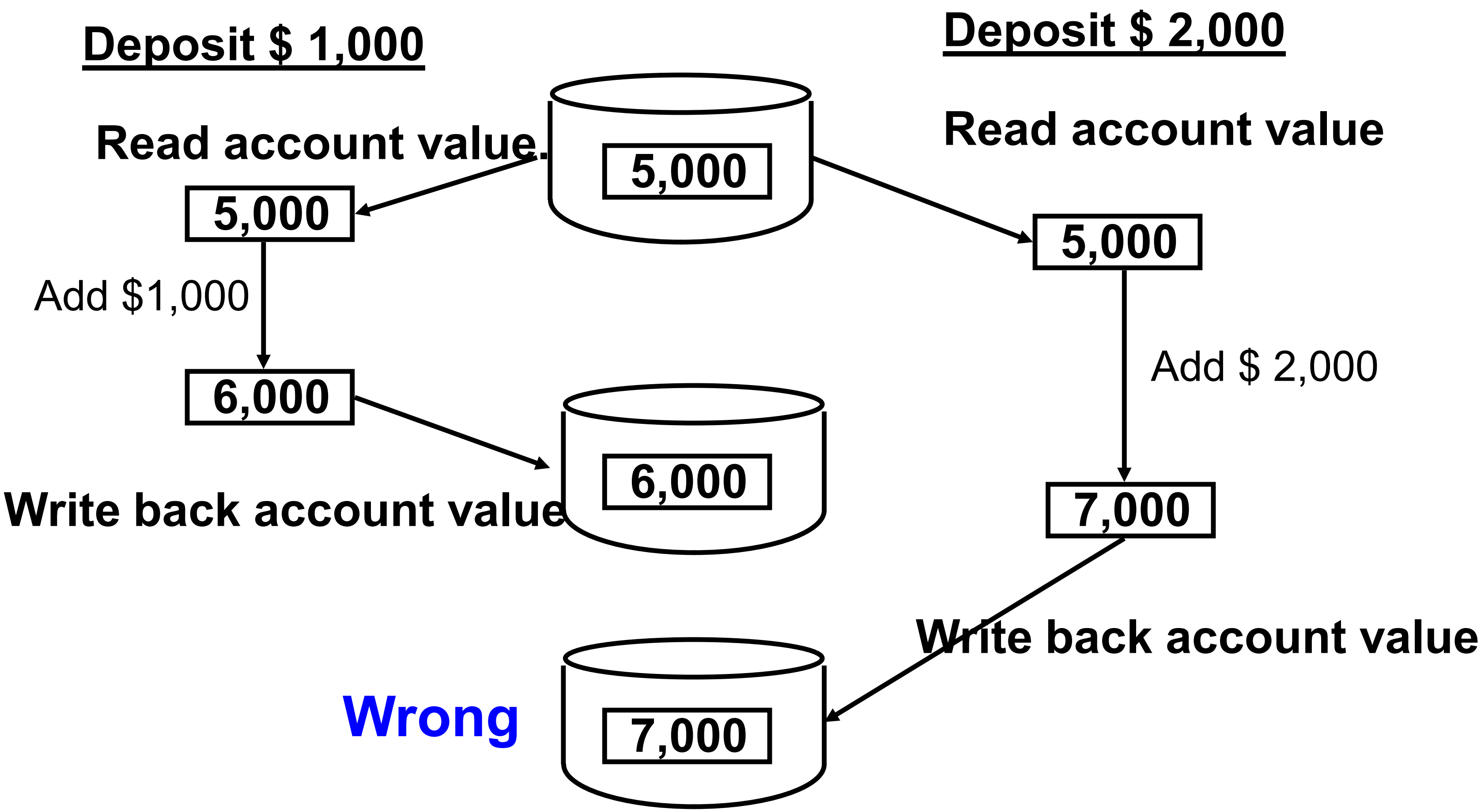
Synchronization

- Very important feature of RDBMS: Support for multiple users working concurrently on the same data
- "Work": Running transactions
- Synchronization = Preventing bad things from happening when transactions run concurrently
  - Inconsistent states
  - Lost or phantom changes
  - Starvation or deadlocks

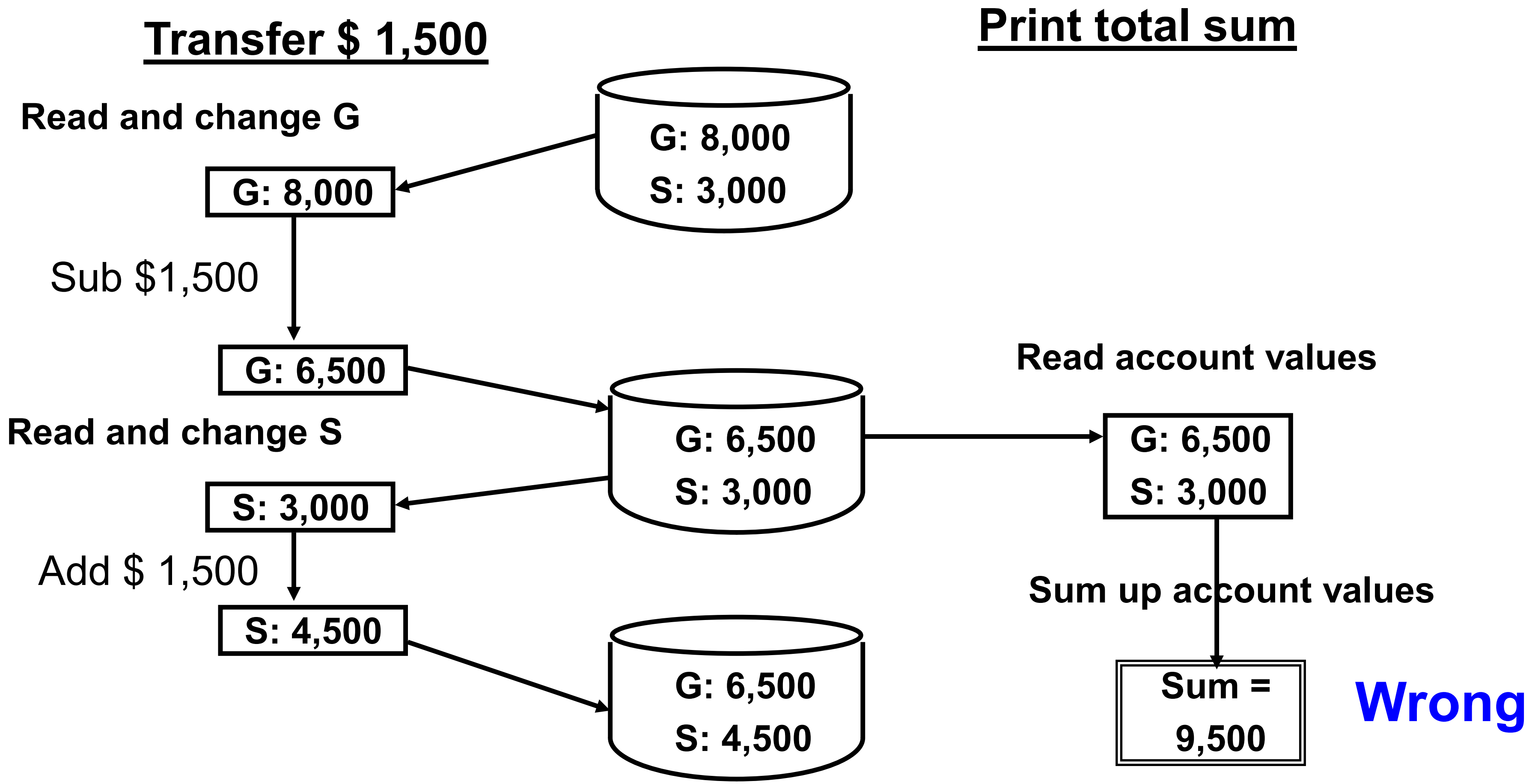
Trade-Off

- Trade-off between consistency and throughput
- High-performance OLTP systems often dominated by synchronization efforts
  - Much locking, TX wait and wait, frequent aborts through time-outs and deadlocks, frequent restarting leads to even more contention – breakdown
- Think carefully which degree of synchronization is necessary, respectively which types of errors are tolerable
  - Few applications really need full isolation
  - SQL defines different levels of isolation (later)

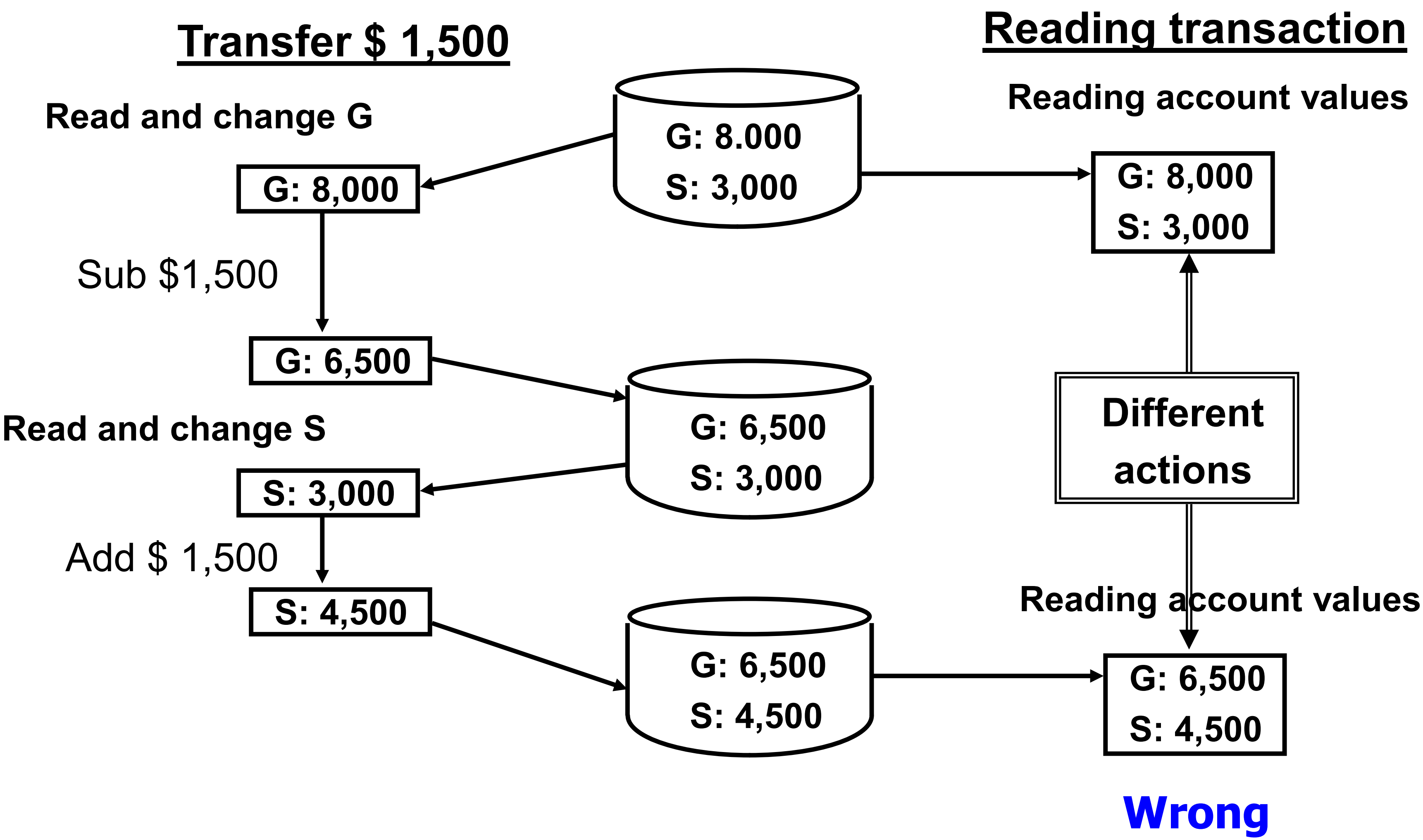
Lost Update Problem



Inconsistent Read Problem



Non-Repeatable Read



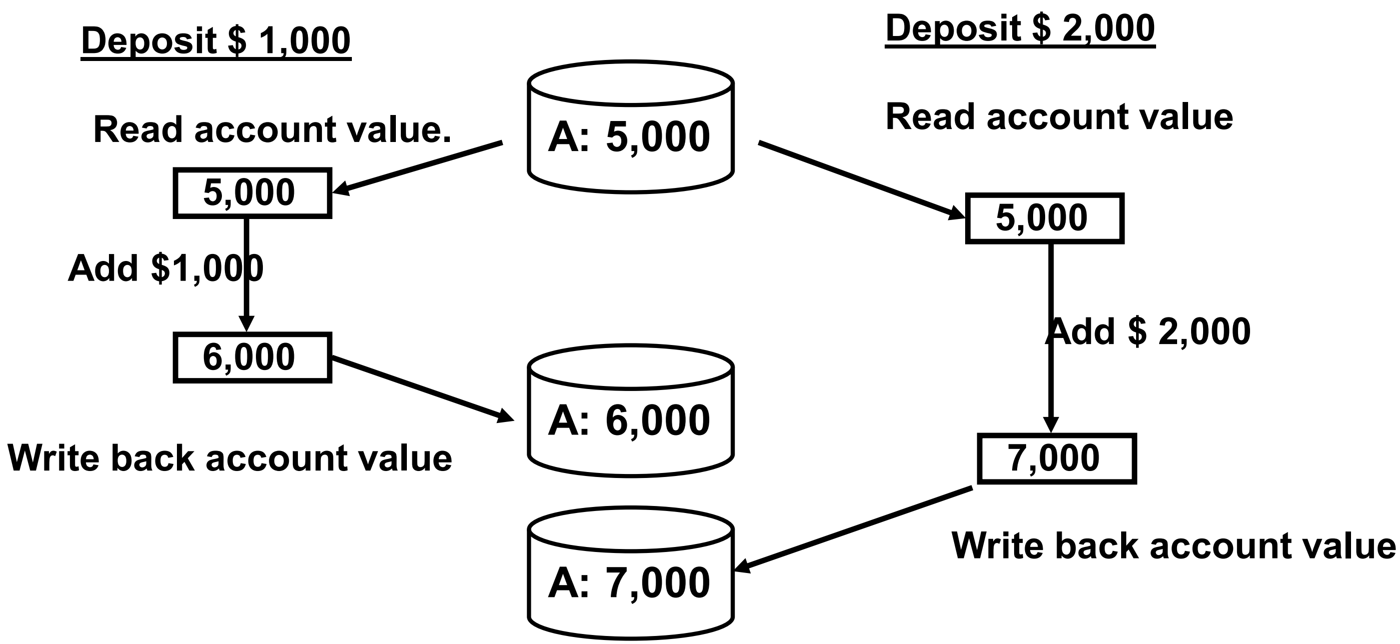
Other Problems

- Dirty Reads: T2 reads a value which was changed by T1, but T1 eventually aborts
  - T2 works on false premises
- Phantom reads: T2 computes an aggregate over table T, but T is changed by T1 before T2 uses its result
  - T2 results are already wrong when created
- Integrity constraint violations: T2 reads an intermediate state of a T1 which results in an IC violation (e.g.: T1 inserts primary key and deletes it again, T2 tries to insert the same key in-between)
  - T2 works on false premises
- Problems in clients: Dangling cursors (next tuple deleted)

Transaction Model

- Transactions work on objects (attributes, tuples, pages)
- Only two different operations
  - Read operation: R(X), R(Y), . . .
  - Write operation: W(X), W(Y), . . .
  - Other data (local variables) are assumed to have no sync problems
    - Local memory for each transaction
- A transaction T is a sequence of read and write operations
  - **T = <R_T(X),W_T(Y),R_T(Z),… >**
  - For sync, we do not care which values are read or written
    - But the recovery manager does!
  - We do not model when exactly reads/writes happens, but always assume the worst
    - Only the order within one TX is fixed
  - Sync should prevent all possible errors, not only real ones

Example



- Transaction T_1: $<R_{T1}(A), W_{T1}(A)>$
- Transaction T_2: $<R_{T2}(A), W_{T2}(A)>$
- Good order: $<R_{T1}(A), W_{T1}(A), R_{T2}(A), W_{T2}(A)>$
- Bad order: $<R_{T1}(A), R_{T2}(A), W_{T1}(A), W_{T2}(A)>$

Schedules

- We assume that each TX in itself has no problem
  - No intra-transaction parallelization, no speculative execution, …
  - Single operations are atomic, TX are not
- For now, we assume that all TX in T eventually commit
  - Hence, we don't include "commit" in our schedules
- Definition
  A schedule is a totally ordered sequence of all operations from a set T of transactions $\{T_1,\ldots, T_n\}$ such that all operations of any transaction are in correct order
- Examples (there exist 4! schedules)
  - $S_1 = <R_{T1}(A), R_{T2}(A), W_{T1}(A), W_{T2}(A)>$
  - $S_2 = <R_{T1}(A), W_{T1}(A), R_{T2}(A), W_{T2}(A)>$
  - $S_3 = <R_{T1}(A), R_{T2}(A), W_{T2}(A), W_{T1}(A)>$

Good Schedules

- Look at $\mathbf{s} = \langle R_{T1}(A), R_{T2}(A), W_{T1}(A), W_{T2}(A) \rangle$
  - This is exactly the “lost update” sequence
- Some other schedules do not have this problem
  - $S_2 = \langle R_{T1}(A),\ W_{T1}(A),\ R_{T2}(A),\ W_{T2}(A) \rangle$
  - $S_4 = \langle R_{T2}(A),\ W_{T2}(A),\ R_{T1}(A),\ W_{T1}(A) \rangle$
- Apparently, some schedules are fine, others not
- Synchronization – prevent “bad” schedules

Content of this Lecture

- Synchronization
- Serial and Serializable Schedules
- Locking and Deadlocks
- Timestamp Synchronization and SQL Isolation Levels

Preface

- In the following, we lay the theoretical foundations for TX synchronization
- We characterize when a given order of operations (schedule) is acceptable
- Real databases don't do such reasoning: They enforce acceptable orders of operations
  - See "Locking and Deadlocks"
- Acceptable: ACID properties are guaranteed

Serial Schedules

- Definition
  A schedule for a set T of transactions is called serial if all its transactions are totally ordered
  - Totally ordered: Each TX starts when no other TX is active and finishes before any other TX starts
- Clearly, serial schedules have no problem with interference, isolation is ensured
- There is a cost: No concurrent actions -> bad performance
  - TX cannot work on other data items in parallel
  - Most TX do never interfere with others – should not be halted
- We need a weaker criterion

Acceptable Schedules

- For a set T of transactions there are |T|! serial schedules
- These are not equivalent, i.e., different serial schedules for the same set of TX may produce very different results
  - S_1, = <R_{T1}(A), A=A+10, W_{T1}(A), R_{T2}(A), A=A*2, W_{T2}(A)>
  - S_2, = <R_{T2}(A), A=A*2, W_{T2}(A), R_{T1}(A), A=A+10, W_{T1}(A)>
- Consistency only requires TX to be atomic and without interference, but does not dictate the order of transactions
  - In particular, there is no guaranteed or canonical order of TX
    - Such as time of start
    - "Time" is always difficult in concurrent processes
- Hence, every serial schedule is acceptable (ACID)

Serializable Schedules

- Definition
  A schedule for a set T of transactions is serializable, if its result is equal to the result of at least one serial schedule of T
- Result means
  - The final state of the DB after executing all TX from T
  - The outputs of all involved TXs (intermediate results)
- Informally: Some intertwining of operations is OK, as long as the same result could have been achieved with a serial schedule

Conflicts

- To define the "harmfulness" of intertwining, we need a notion of conflict
- Observation: It does not matter if two TX read the same object, in whatever order
- All other cases matter because they may generate different results depending on execution order
  - Assume the worst!
- Definition
  Two operations $op_1 \in T_1$ and $op_2 \in T_2$ conflict iff both operate on the same object X and at least one is a write

Serializability of Schedules

- Definition

  Two schedules S und S' over T are conflict-equivalent, if
  - *For all $op \in T_1$ and $op' \in T_2$: If op and op' are in conflict, then they are executed in the same order in S and in S'*

  A schedule is called conflict-serializable if it is conflict-equivalent to at least one serial schedule
- Explanation
  - All critical operations (R/W, W/W) must be executed in the same order in the serial schedule and the schedule under study
  - None-critical operations (R/R) do not matter – all conflict-serializable schedules are acceptable
  - Order of ops is constrained, but less then in serial schedules

Example

```
Start T1;
Read( x, t);
Write( x, t+5);
Read( y, t);
Write( y, t+5);
```

```
Start T2;
Read( x, s);
Write( x, s*3);
Read( y, s);
Write( y, s*3);
```

S=<R1(X),W1(X),R2(X),W2(X),R2(Y),W2(Y),R1(Y),W1(Y)>

- Imagine initially x=y=10
- Result of schedule S is x=45 and y=35
- Serial1: <T1;T2>, leading to x=45 and y=45
- Serial2: <T2;T1>, leading to x=35 and y=35
- S is not serializable
- But is it conflict-serializable?

Conflicting Orders

```
Start T1;
Read( x, t);
Write( x, t+5);
Read( y, t);
Write( y, t+5);
```

```
Start T2;
Read( x, s);
Write( x, s*3);
Read( y, s);
Write( y, s*3);
```

```
S=<R1(X),W1(X),R2(X),W2(X),R2(Y),W2(Y),R1(Y),W1(Y)>
```

- Conflicts
  - R1(X)<W2(X), W1(X)<R2(X), W1(X)<W2(X)
  - R2(Y)<W1(Y), W2(Y)<R1(Y), W2(Y)<W1(Y)

Serial schedules

R1(X)	R2(X)
W1(X)	W2(X)
R1(Y)	R2(Y)
W1(Y)	W2(Y)
R2(X)	R1(X)
W2(X)	W1(X)
R2(Y)	R1(Y)
W2(Y)	W1(Y)

Efficiently Testing Conflict-Serializability

- We should not try to check conflict-serializability by looking at all possible serial orders of its transactions and check for conflict-equivalence by considering all conflicting pairs of operations
- Instead, we lift the problem from pairs of operations to pairs of transactions – we order transactions, not operations
- Precedence constraints between TX can be encoded in a graph

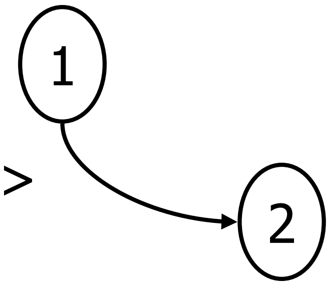
Serializability Graphs

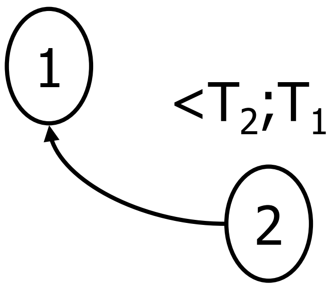
- Definition
  The serializability graph SG(S) of a schedule S is the graph formed by
  - *Each transaction forms a vertex*
  - *There is an edge from vertices T_i to T_k, iff in S there are conflicting operations $op_i \in T_i$ and $op_k \in T_k$ and op_i is executed before op_k*

```
Start T1;
Read( x, t);
Write( x, t+5);
Read( y, t);
Write( y, t+5);
```

```
Start T2;
Read( x, s);
Write( x, s*3);
Read( y, s);
Write( y, s*3);
```

$<T_1;T_2>$





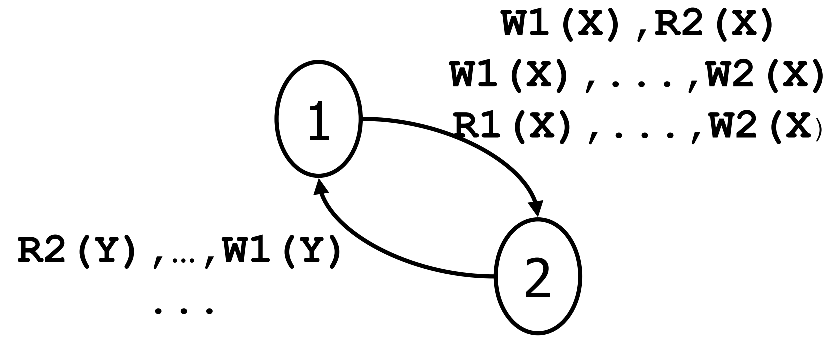
$<T_2;T_1>$

Testing Serializability

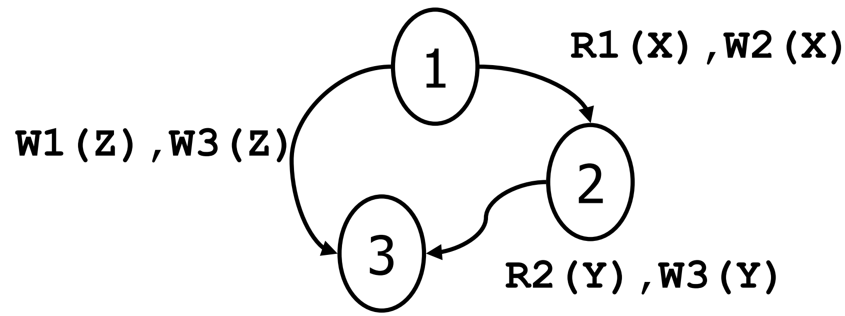
- Theorem
  A schedule S is conflict-serializable iff SG (S) is cycle-free
- Formal proof: Omitted (see literature)
- Intuition (one direction)
  - If two operations are in conflict, we need to preserve their order in any potential conflict-equivalent serial schedule
  - Thus, each conflict puts a constraint on the possible orders
  - If SG(S) contains a cycle, not all of these constraints can be fulfilled by any serial schedule
- That's good: Testing for cycles is linear in |SG|

Examples

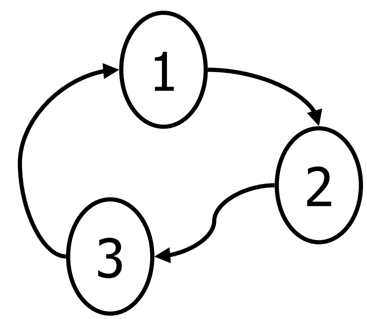
- **<R1(X),W1(X),R2(X),W2(X), R2(Y),W2(Y),R1(Y),W1(Y)>**
  - Not serializable



- **<R1(X),R2(Y),W1(Z),W3(Z), W2(X),W3(Y)>**
  - Serializable: **<T1;T2;T3>**



- **<R1(X),R2(Y),W3(Z),W1(Z), W2(X),W3(Y)>**
  - Not serializable



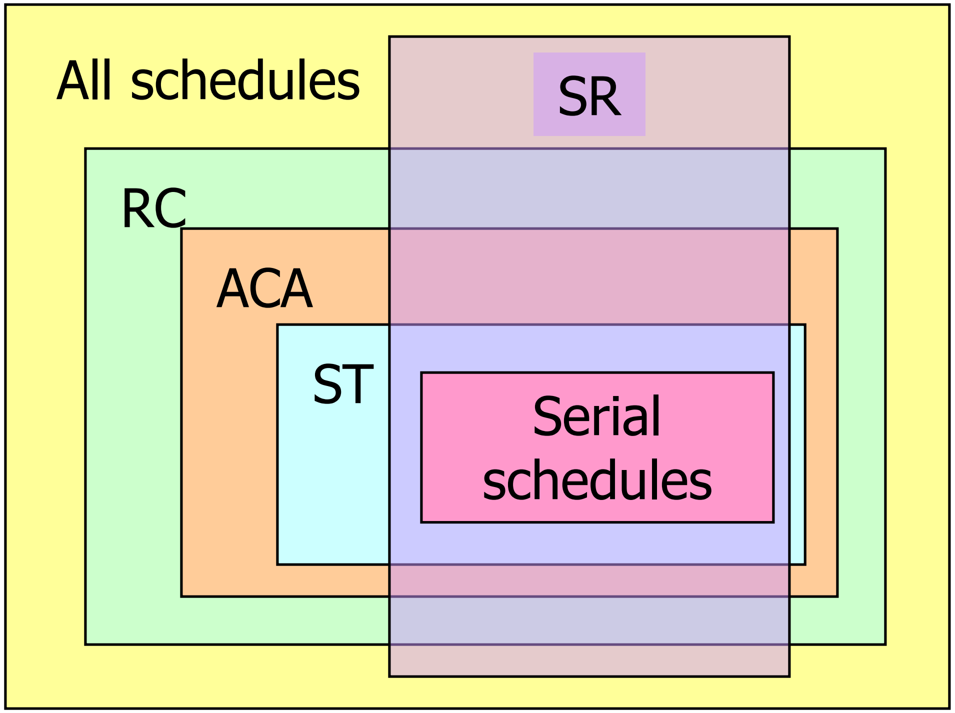
Transactions Do more Than Read and Write

- In particular, they commit or abort
- This has implications – which data is valid when?
- Imagine $S=<W_1(X), R_2(X), W_2(X), commit_2, abort_1>$
  - By our definitions, S is serializable
  - But T2 has read what it should not have read; when T1 aborts, T2 should also be aborted; but T2 cannot be aborted any more
  - S is not recoverable
- Imagine $<W_1(X), R_2(X), W_2(X), abort_1>$
  - Scheduler must and may abort T2 (because of dirty read), although schedule <T2;T1> would have been fine
  - Problem of cascading aborts

Definitions

- Definition
  - *A schedule S is called recoverable, if, whenever a committed T2 reads or writes an object X whose value was before written by a unfinished T1, then S contains a commit for T1 before the commit of T2*
    - Avoids un-abortable transactions
  - *A schedule S is called strict, if, whenever a T1 writes an object X that is later read or written by a T2, then S contains a $commit_1$ or $abort_1$ before the respective operation of T2*
    - Avoids cascading aborts (and problems in recovery – see literature)
- Lemmata
  - Every strict schedule is recoverable
  - A conflict-serializable schedule can be recoverable (or strict) or not
  - Details: Literature

Relationships



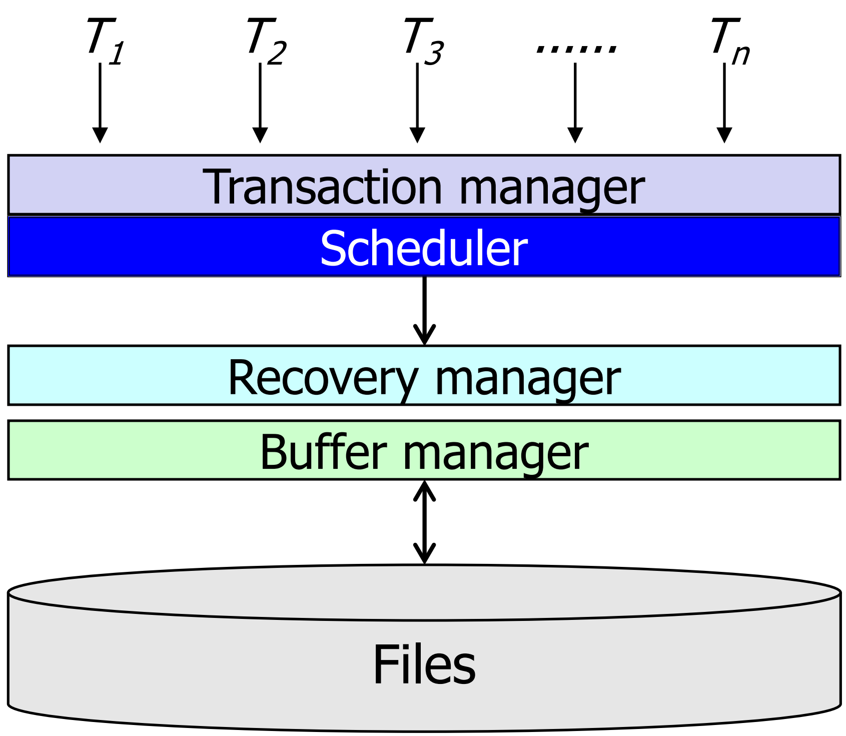
- RC: Recoverable schedules
- ACA: Schedules avoiding any cascading aborts
- ST: Strict schedules
  - Usually, we want strict schedules in databases
- SR: Serializable schedules

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Locking and Deadlocks
- Timestamp Synchronization and SQL Isolation Levels

Locking

- Practice: DBs do not check schedules before/while they run
- Instead, a scheduler ensures properties of schedules at runtime



New Component: Scheduler

- Responsible for
  - Generating schedules as wanted (e.g. strict, serializable, …)
  - Handling deadlocks
- Operations of the schedulers
  - Pass on operations of transactions: R, W, Abort, Commit
    - And do bookkeeping (i.e. set locks, maintain waits-for graph, …)
  - Reject operations
    - In extreme case, scheduler aborts running TX
    - E.g. necessary to resolve deadlocks
  - Delay operations
    - Wait with the requested action
    - TX held in a waiting queue

Two Flavors of Schedulers

- Pessimistic scheduling (locking – discussed here)
  - Delay problematic actions and avoid aborts
  - Advantage: Few aborts
  - Disadvantage: Reduced parallelism
  - Use when many conflicts are expected
- Optimistic scheduling (sketched later)
  - Let TXs perform as if they were isolated
  - Check for synchronization problems while running or afterwards
  - If problem encountered, abort critical TX
  - Advantage: No delays, fast parallel execution of conflict-free TXs
  - Disadvantages: More aborts in case of conflicting TX
  - Use when few conflicts are expected

Pessimistic Scheduling

- Main idea: Check each incoming operation
- If problems may occur (e.g. non-serializable order), either delay operation or abort TX
- Usual implementation: Manage locks on objects
  - No global locks, but one "controller" per data object
    - Less of a bottleneck
  - TX may only perform operations if proper locks have been acquired
  - Other TX may block such acquisitions
- Many issues: Which types of locks, how manages the locks, when may TX release/acquire locks, …

Locks and Lock Manager

- Lock: A temporary access privilege to an object
- Lock manager (LM) administers requests and locks
  - Bottleneck! But: hardware support
- Types of locks
  - Read lock (sharable lock): S
  - Write lock (exclusive lock): X
  - Read and write locks are not compatible, i.e. there cannot exist a X/S-lock and a S-lock from different TX on the same object
- If an incompatible lock is requested, LM refuses request and scheduler delays requesting TX
- Locks must be released
  - Either explicitly by the transaction
  - Or automatically at commit or abort time

Lock Protocols

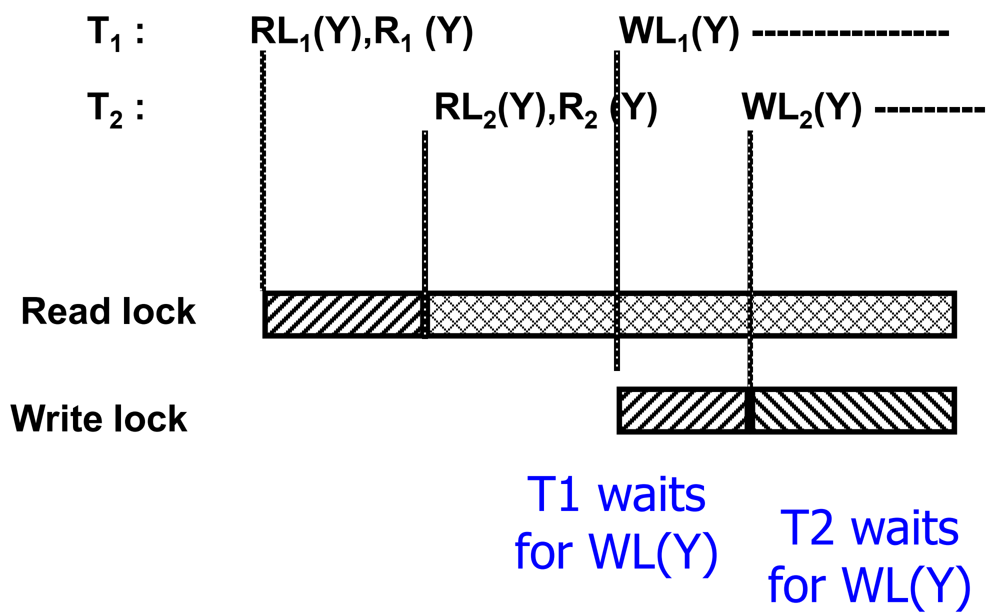
- Lock protocol: At what points in time TXs may acquire and release locks
- Example – A simple read/write lock protocol
  - A read lock (S or RL) must be acquired before a read
  - A write lock (X or WL) must be acquired before a write
  - Compatibility matrix for read and write locks
    - "+": compatible
    - "–": incompatible
- Will create many deadlocks

	S	**X**
S	+	-
X	-	-

Deadlocks

T1: $<RL_1(Y),R1(Y),WL_1(Y),W1(Y),U_1(Y)>$

T2: $<RL_2(Y),R2(Y),WL_2(Y),W2(Y),U_2(Y)>$



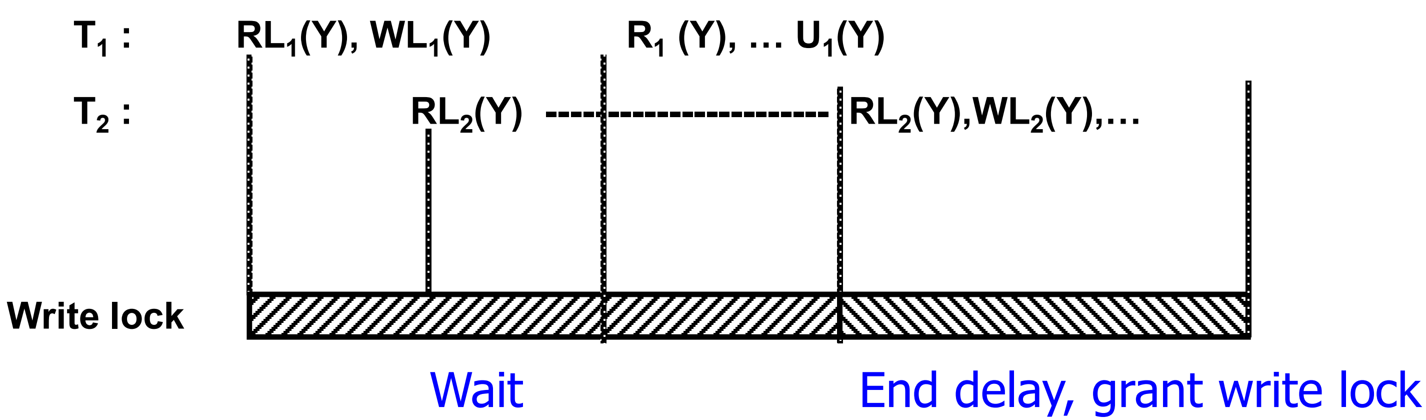
- Both RL are granted
- Both WL-requests are delayed
- Both TX wait for each other
- Locks are never released, because neither TX can proceed
- Deadlock

Option 1: Deadlock Prevention

- "Preclaiming"
  - All locks must be requested before first data access
  - Requires that a TX knows all its lock needs at the start of the TX
  - Requesting all locks must be atomic
    - We lock the operation "locking objects"

T1: $\langle RL_1(Y), WL_1(Y), R1(Y), W1(Y), U_1(Y) \rangle$

T2: $\langle RL_2(Y), WL_2(Y), R2(Y), W2(Y), U_2(Y) \rangle$



Option 1: Deadlock Prevention

- "Preclaiming"
  - All locks must be requested before first data access
  - Requires that a TX knows all its lock needs at the start of the TX
  - Requesting all locks is atomic
- Consequences
  - TX are delayed only at start-up time
  - Delayed TX cannot acquire any locks
  - Delayed TX cannot block other TX – no deadlocks
- Disadvantages
  - If uncertain, typically more locks then needed are requested
  - Locks are kept longer than necessary
  - Low throughput: Only entirely conflict-free TXs run concurrently

Option 2: Deadlock Detection

- Build waits-for graph on TX representing requests
- Scheduler must regularly (or prior to edge insertion) check for cycles
  - If cycle is detected – chose a transaction and abort it
  - Often: Also abort TX after a fixed time (timeout)
- Which TX to abort?
  - TX that can be aborted with minimal overhead (locks, REDO logs)
  - TX that has executed the least operations (redo log) so far
  - TX that has requested the most locks
  - TX that participates in more than one cycle
  - …

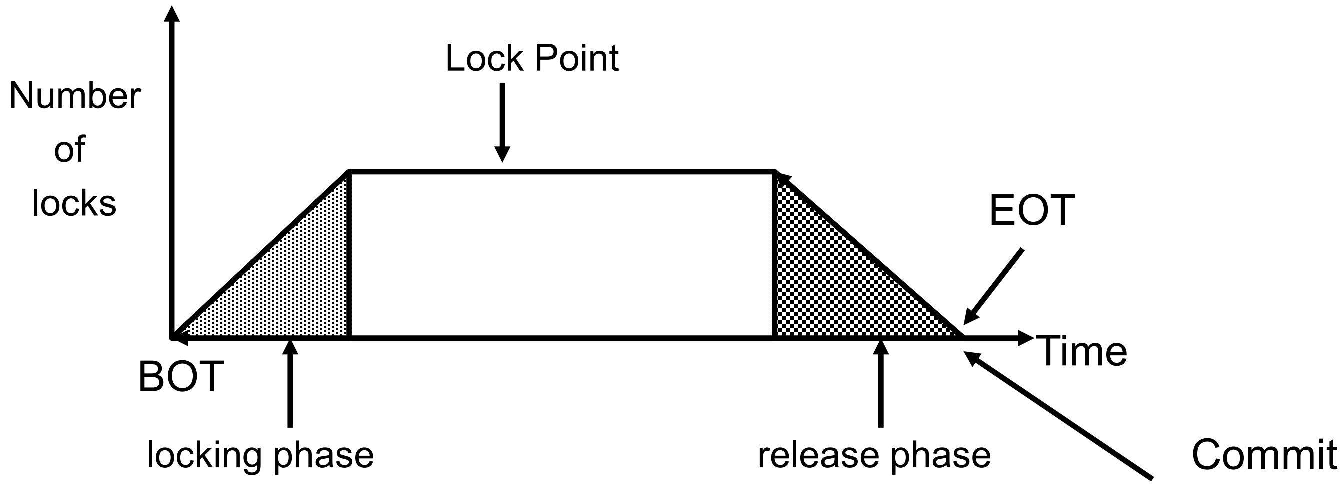
Preclaiming or Deadlock Detection?

- Preclaiming: No deadlocks, only serial schedules, reduced concurrency
- Deadlock detection: No deadlocks, no serializability guarantees, high concurrency

- We what: Only save schedules, higher concurrency, no (rare) deadlocks

2-Phase Lock Protocol (2PL)

- Less conservative than preclaiming: 2-Phase Locking
  - Before TX can read object X, it must own a read or write lock on X
  - Before a TX can write object X, it must own a write lock on X
  - Once a TX starts to release locks, it cannot be granted new locks
- Very prominent



2PL Schedules are Serializable

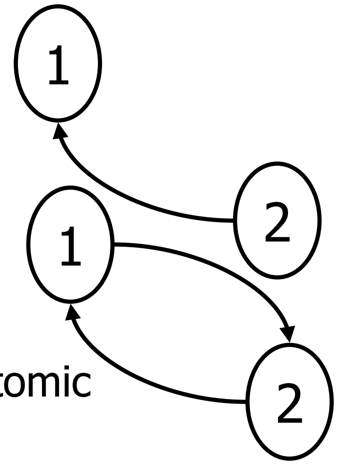
- 2PL does not prevent deadlocks (example next slide)
- Theorem
  All 2PL schedules are conflict-serializable
- Proof
  - We prove that the (runtime) serializability graph SG of any 2PL schedule S cannot contain a cycle
  - Step 1: If there exists an edge between T_i and T_j, then T_i's lock point happens before T_j's lock point
    - Since there exists an edge from T_i to T_j, there exists an object X on which both TXs want to execute operations that are in conflict
    - Assume T_i owns a lock on X (following 2PL). T_j can get this lock only after T_i has performed an unlock operation (because T_i and T_j are in conflict). Therefore T_i has left its lock point before T_j can reach its lock point

2PL Schedules are Serializable

- 2PL does not prevent deadlocks, but …
- Theorem
  All 2PL schedules are conflict-serializable
- Proof (cont)
  - Step 2: Now assume that SG(S) contains a cycle
    - Then there exist edges
      $$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \; . \; . \; . \; . \; \rightarrow T_n \rightarrow T_1$$
    - According to step 1, this cycle implies that the lock point of T_2 occurs before the lock point of T_1 (by transitivity)
    - Contradiction
  - Q.e.d.

Examples

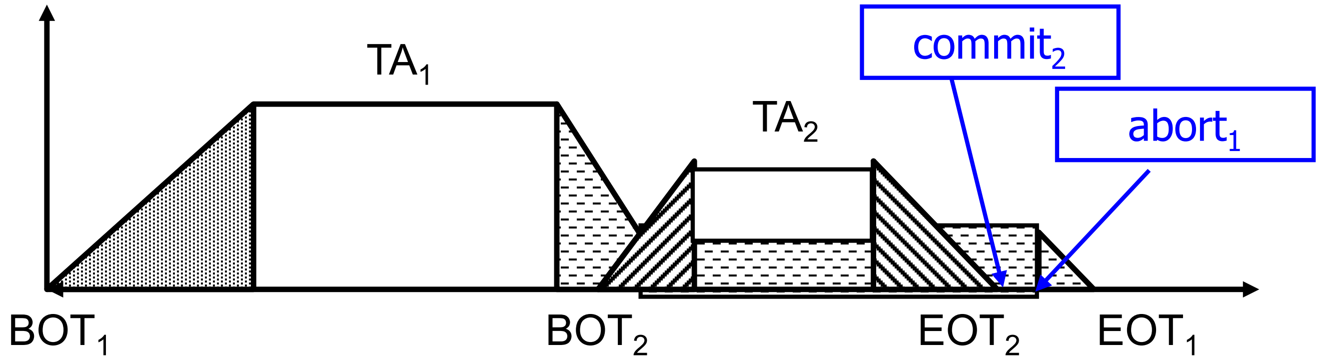
<R1(X),W1(X),R2(X),W2(X),R2(Y),W2(Y),R1(Y),W1(Y)>



- With 2PL, the following may happen
  - WL_1(X),WL_1(Y),R_1(X),W_1(X),<T2 must wait>,R_1(Y), W_1(Y),U_1(X,Y),<T1 finished>,WL_2(X),<T1 commits>,…
    - Fine
  - RL_1(X),R_1(X),RL_2(X),<T1 must wait>,<T2 must wait>
    - 2PL does not prevent deadlocks because lock phase is not atomic
  - WL_2(X),R2(X),W2(X),<T1 must wait>,WL_2(Y), …
    - Fine
  - …
- U_i(X,Y,…) means: TX_i unlocks objects X, Y, …

Observation

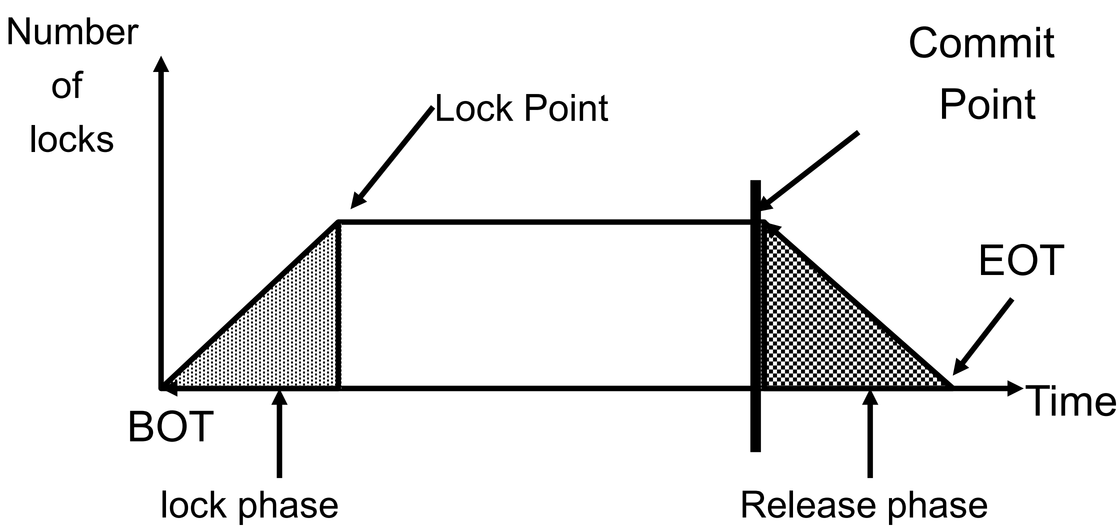
- 2PL does not guarantee recoverable schedules
  - Recall: A schedule S is called recoverable, if, whenever a committed T2 reads or writes an object X whose value was before written by a unfinished T1, then S contains a commit for T1 before the commit of T2



  - When T2 starts, it may lock and write objects locked and written by T1 before
  - If T1 aborts late (looong release phase), T2 might have committed already

Strong and Strict 2PL Protocol (SS2PL)

- SS2PL ensures recoverable schedules
- Locks are released only after passing "Commit Point"
  - Only after commit/abort has been acknowledged by scheduler
  - Less parallelization, less throughput, but recoverable
  - Deadlocks may still happen (solve by atomic lock/unlock phase)



Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Locking and Deadlocks
- Timestamp Synchronization and SQL Isolation Levels

Optimistic Locking by Timestamps (sketched)

- Create a “timestamp” (sequential ID) for new TX
- Manage timestamps for each object: Last reading TX, last writing TX, last committed TX
- When T accesses an object X, compare TS(X) and TS(T)
  - In case of potential conflicts, abort transactions
    - No delays, no locks, no deadlocks
  - Example: “Read too late”: <R2(X),R1(Y),W1(Y),R2(Y)>
    - R2 tries to read Y whose value has changed after T2 started
    - Unsure situation, not serializable – abort T2
  - Requires rule set over different situations
  - Not covered here

Opt Locking by Multi-Version Synchronization

- Idea: When changing data (here T1), only change a copy
  - TX always read the last committed value (no dirty reads)
  - In example: T2 would read old value of Y (before T1)
  - Requires keeping multiple versions of each object
  - Writes must still be synchronized, but reads are "freed"
- Optimistic: Don't sync, but validate changes at end of TX
  - Upon abort, do nothing (discard local changes)
  - Upon commit, check
    - Whether read objects have changed in the meantime
    - Whether written objects have been read or written in the meantime
  - If yes: abort transaction
  - Otherwise, copy local values to database
- Used in many systems: Oracle, PostGreSQL, …

Discussion

- Advantage
  - No lock manager, no delays
  - “Reads never wait”
  - Very fast if conflicts are rare
- Disadvantage
  - Even if conflicts would appear early, TX first has to finish
    - Waste of CPU cycles
  - Management of timestamps (space, CPU)
    - Need to stamp all accesses to any object across and within transactions
    - Use higher granularity: Timestamps of blocks, tuples, etc.
  - Main memory management: Many versions, garbage collection, …

SQL Degrees of Isolation

- Goal
  - Let the user/program decide what as specific TX needs
  - Trade-off: Performance versus level-of-isolation
- SQL isolation levels
  - Lost update is never accepted
  - Oracle only supports "read committed" (default) and "serializable" (and "read-only")
  - #

Isolationsebene	Dirty Read	Unrepeatable Read	Phantom Read
Read Uncommitted	+	+	+
Read Committed	–	+	+
Repeatable Read	–	–	+
Serializable	–	–	–

Details

- „Read uncommitted“
  - Can only be used for read-only transactions
  - Do not generate locks, will never wait
- “Read committed”
  - Will only read committed data, but repeatable reads not guaranteed
  - In MV-S, reads won’t wait and writes are not delayed
- “Repeatable reads”
  - Reads read from local copy (in MV-S), TX only checked at commit/abort time
- “Serializable”
  - Full locking protocol, e.g. 2PL

Issues not Discussed

- Optimistic, time-stamped and multi-version scheduling
- Inserts: Locking a non-existing object?
- Managing locks (and locking the lock table …)
- Lock propagation (from value to tuple to table …)
- Locking data with (hierarchical) indexes
- Advanced TX models: Nested, compensating operations, distributed, …
- …