Datenbanksysteme II: Synchronization of Concurrent Transactions

Ulf Leser

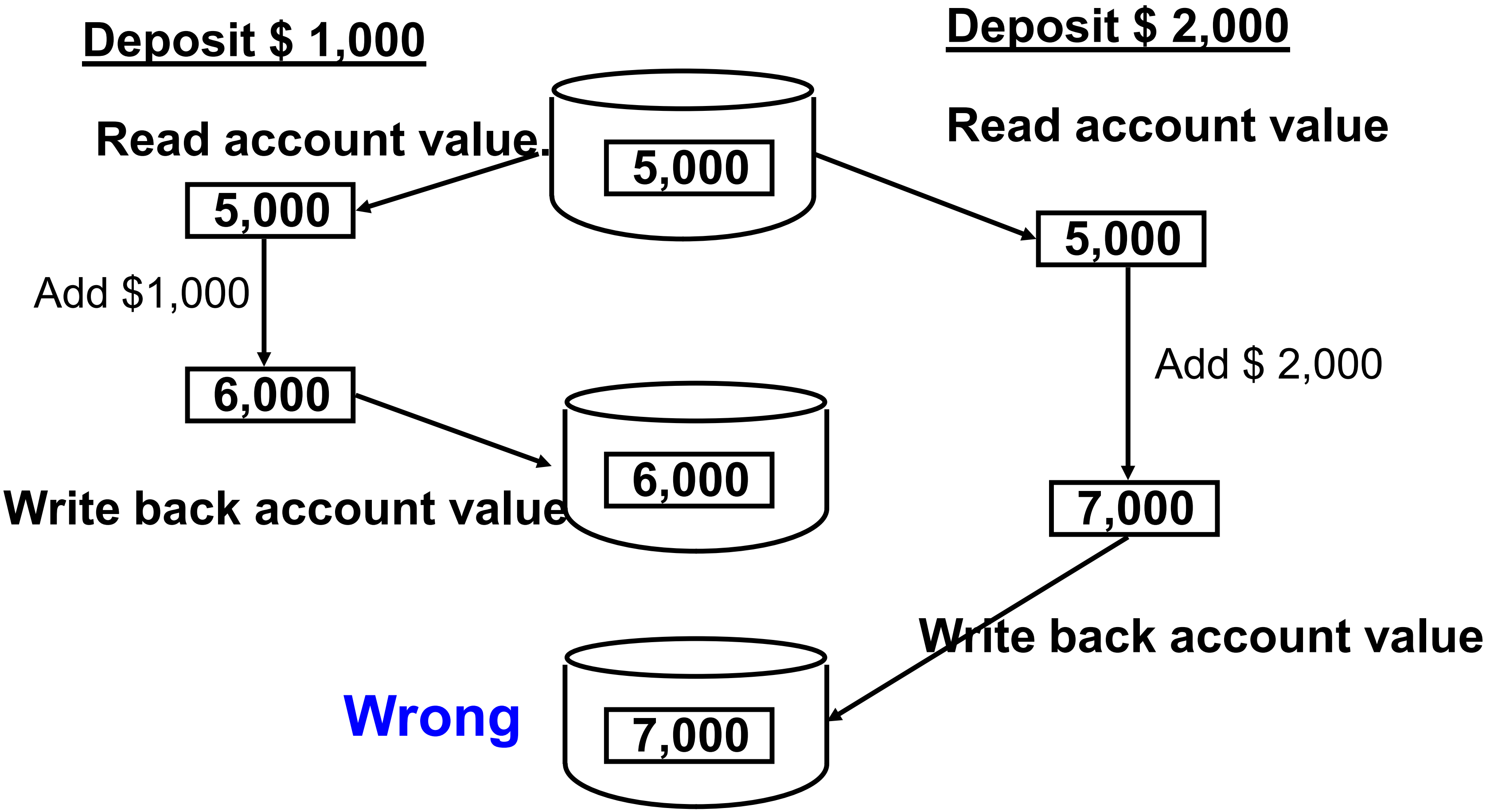
Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

Synchronization

- Very important feature of RDBMS: Synchronizing the concurrent work of multiple users on the same data
- "Work" = Running transactions
- Synchronization = Preventing bad things from happening
  - Unintended (=inconsistent) states
  - Lost or phantom changes
  - Starvation or deadlocks

Lost Update Problem



Deposit $ 1,000

Read account value

5,000

Add $1,000

6,000

Write back account value

Deposit $ 2,000

Read account value

5,000

Add $ 2,000

7,000

Write back account value

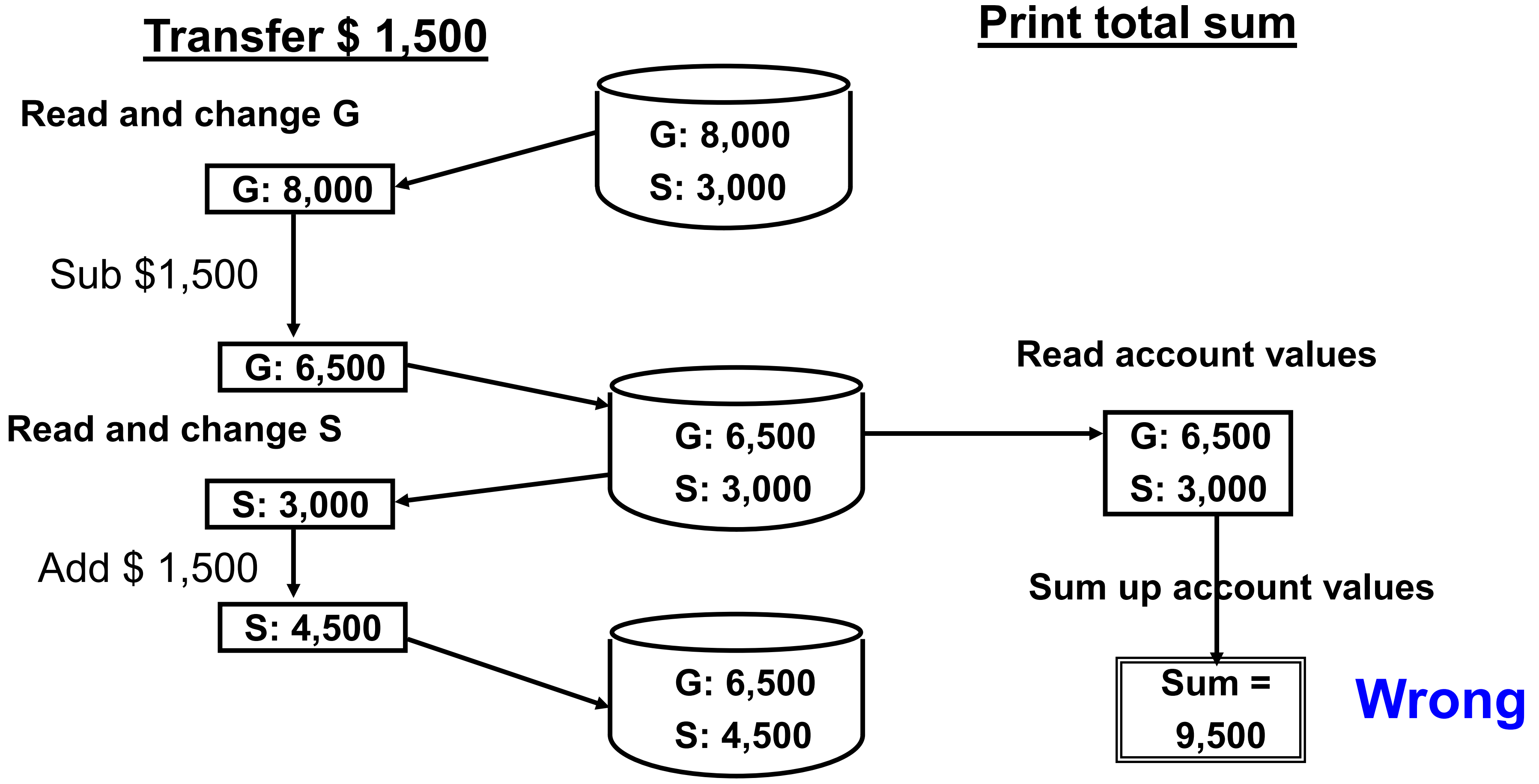
5,000

6,000

Wrong

7,000

Inconsistent Read Problem



Transfer $ 1,500

Print total sum

Read and change G

G: 8,000
S: 3,000

G: 8,000

Sub $1,500

G: 6,500

Read account values

Read and change S

G: 6,500
S: 3,000

G: 6,500
S: 3,000

S: 3,000

Add $ 1,500

Sum up account values

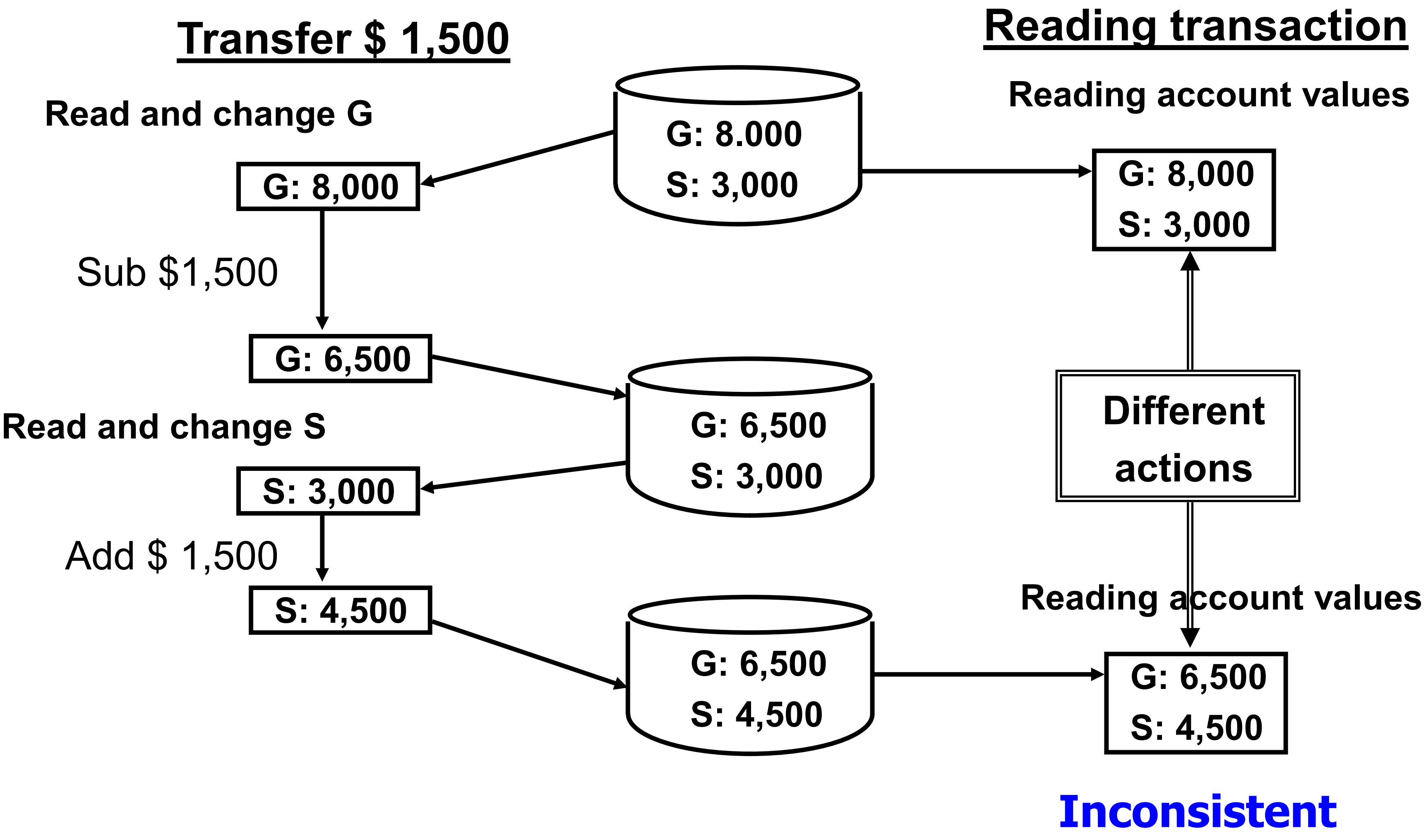
S: 4,500

G: 6,500
S: 4,500

Sum = 9,500

Wrong

Non-Repeatable Read



Transfer $ 1,500

Read and change G

G: 8,000

Sub $1,500

G: 6,500

Read and change S

S: 3,000

Add $ 1,500

S: 4,500

G: 8.000
S: 3,000

G: 6,500
S: 3,000

G: 6,500
S: 4,500

Reading transaction

Reading account values

G: 8,000
S: 3,000

Different actions

Reading account values

G: 6,500
S: 4,500

Inconsistent

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 during execution of T2
  - T2 results are outdated when used
- 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)

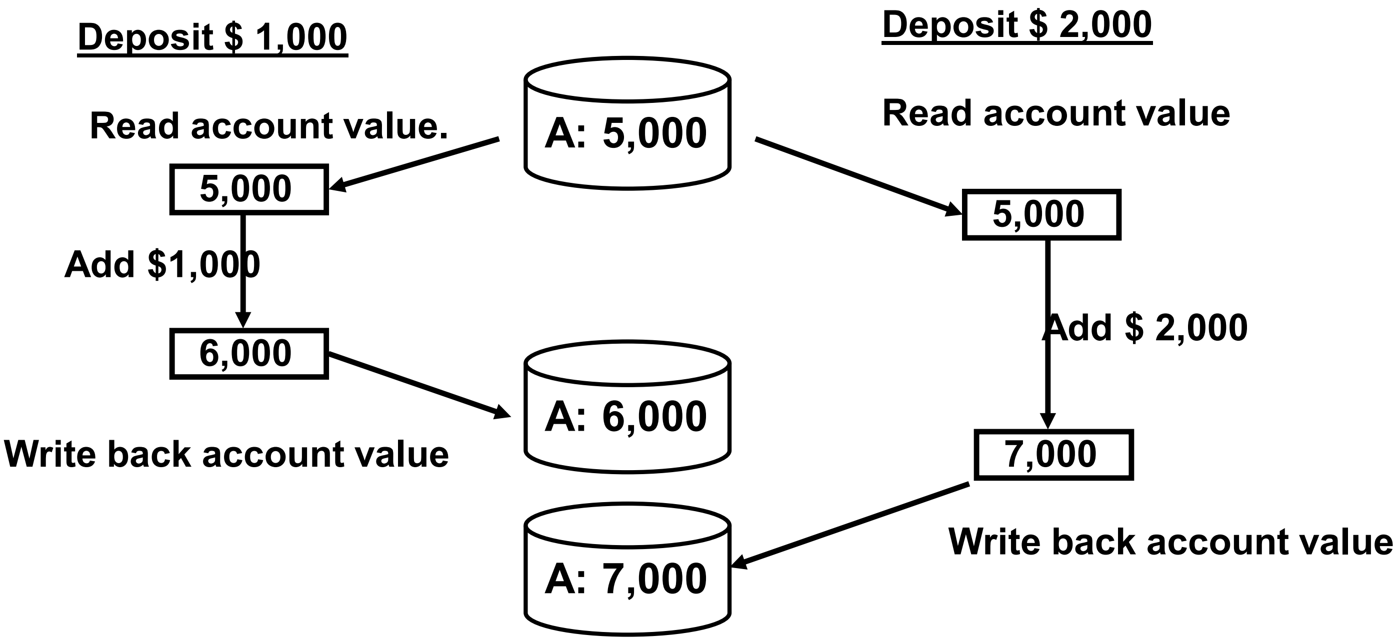
Trade-Off

- 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 restarts lead 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)

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!
- Sync: Prevent all possible errors, assuming worst case
  - Within a TX, the order of events is fixed (programmed)
  - But order of events from different TX is not – assume the worst

Example



Deposit $ 1,000

Read account value.

5,000

Add $1,000

6,000

Write back account value

A: 5,000

A: 6,000

A: 7,000

Deposit $ 2,000

Read account value

5,000

Add $ 2,000

7,000

Write back account value

- 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

- For now, we assume that all TX in T eventually commit
  - Hence, we don't include "commit" in our schedules
  - Aborts – see later

- Definition
  A schedule is a totally ordered sequence of all operations from a set T of (concurrent) transactions $\{T_1,\ldots, T_n\}$ *such that all operations of any transaction are in correct order*
- Examples
  - $S_1 = \langle R_{T1}(A), R_{T2}(A), W_{T1}(A), W_{T2}(A)\rangle$
  - $S_2 = \langle R_{T1}(A), W_{T1}(A), R_{T2}(A), W_{T2}(A)\rangle$
  - $S_3 = \langle R_{T1}(A), R_{T2}(A), W_{T2}(A), W_{T1}(A)\rangle$

Good Schedules

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

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

Preface

- We first lay the theoretical foundations for synchronization
- We characterize when a given order of operations (schedule) is acceptable (= “fine”)
- Acceptable (for now): ACID properties are guaranteed
  - Synchronization only deals with the I
- Real databases do not do such reasoning: They enforce acceptable orders of operations
  - See later: locking

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 concurrency, isolation is ensured
- Very simple to implement – one global TX semaphor
- There is a cost: No concurrent actions -> bad performance
  - TX cannot work on different 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
- 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: The final state of the DB after executing all TX in T
- Informally: Some intertwining of operations is OK, as long as the same result could have been achieved with a serial schedule

- But how should we check this?
- We need a criterion we can check efficiently

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 pairs of operations (R/W, W/W) must be executed in the same order in the serial schedule as in 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

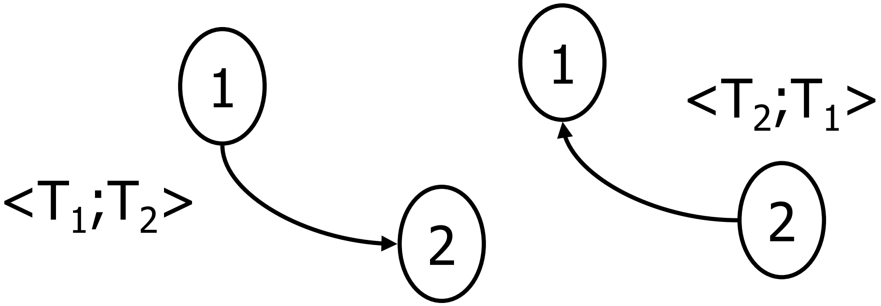
- Definition

  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);
```

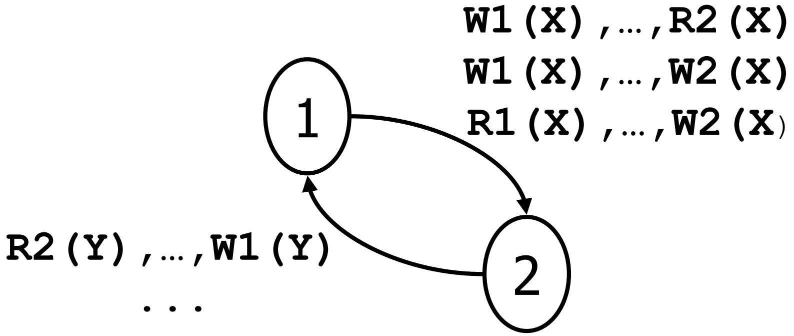


Testing Serializability

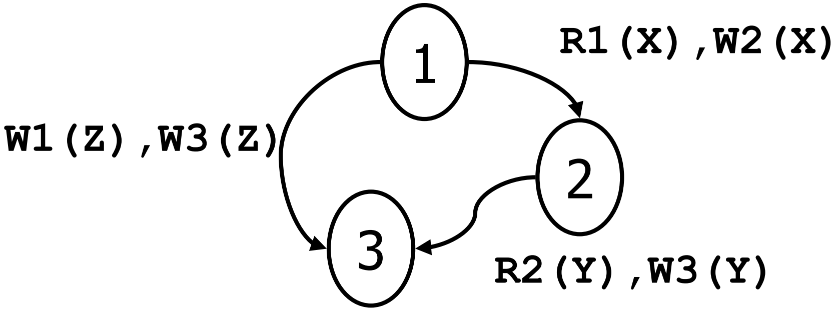
- Theorem
  A schedule S is conflict-serializable iff SG (S) is cycle-free
- Formal proof: Omitted (see literature)
- Intuition
  - If two operations are in conflict, we need to preserve their order in any potential conflict-equivalent serial schedule
  - Thus, each conflict puts a constraint on the possible orders
  - If SG(S) contains a cycle, no serial conflict can exist that fulfills all of these constraints
- Very good! Can be tested efficiently
  - Building SG(S) is $O(|T|^2*n^2)$
    - There are $O(|T|^2)$ pairs of transactions, and each has O(n2) paris
    - n: length of longest transaction in S
  - Testing for cycles is in $O(|SG(S)|) = O(|T|)$

Examples

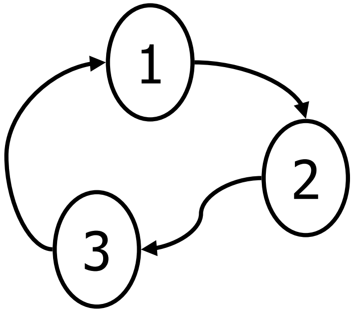
- **<R1(X),W1(X),R2(X),W2(X), R2(Y),W2(Y),R1(Y),W1(Y)>**
  - Not serializable



- **<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



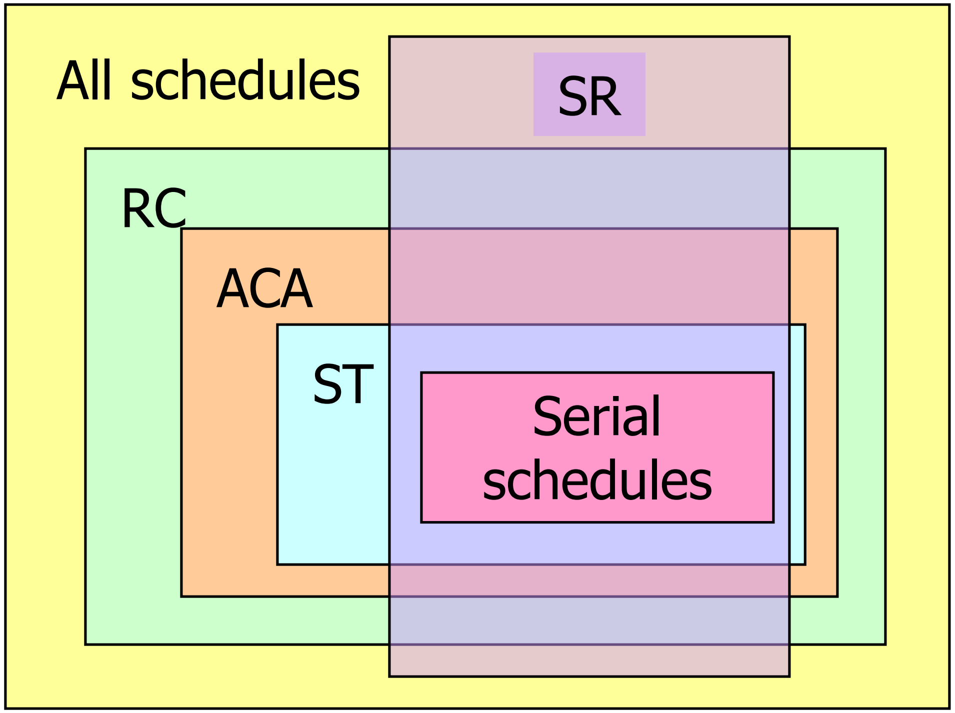
Transactions Do more Than Read and Write

- In particular, they commit or abort
- This has implications – which data is valid when?
- Imagine <$W_1(X)$, $R_2(X)$, $W_2(X)$, abort_1>
  - Scheduler must and may abort T2 (because of dirty read), although schedule <T2;T1> would have been fine
  - Problem of cascading aborts
- Worse: S=<$W_1(X)$, $R_2(X)$, $W_2(X)$, commit_2, abort_1>
  - By our definitions, S is serializable (we assumed all TX commit)
  - 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

Definitions

- Definition
  - *A schedule S is called recoverable, if, whenever a committed T2 has read or written an object X that before was written by a unfinished T1, then S contains a commit for T1 before the end 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
- Lemmata
  - Every strict schedule is recoverable
  - A conflict-serializable schedule can be recoverable (or strict) or not
  - Details: Literature

Relationships



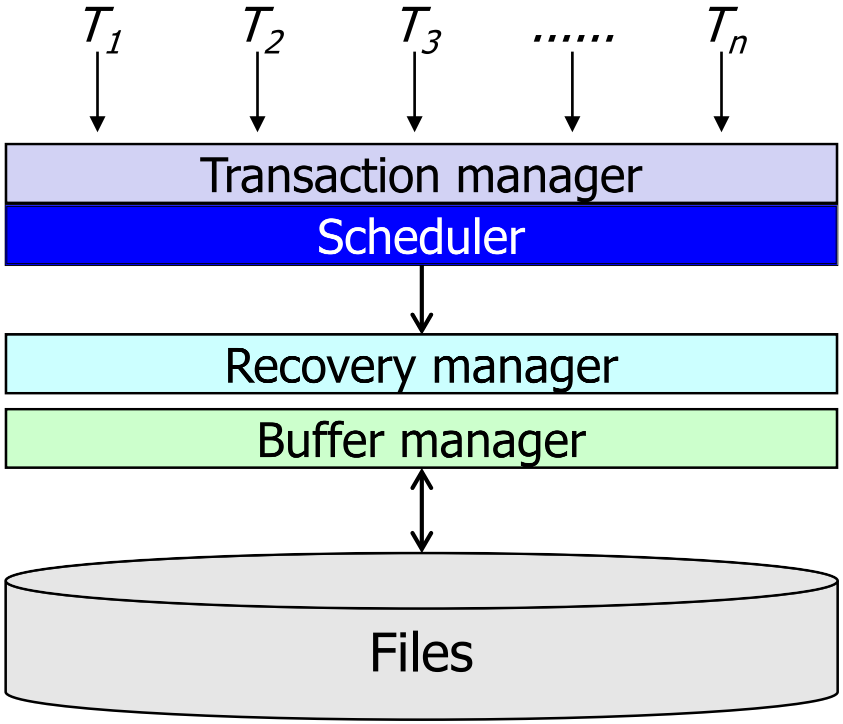
- RC: Recoverable schedules
- ACA: Schedules avoiding any cascading aborts
- ST: Strict schedules
  - Usually, we want strict schedules in databases
- SR: Serializable schedules

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

Locking

- Real DBs do not check schedules after they finished
- Instead, a scheduler ensures the desired properties of schedules by controlling their actions at runtime



New Component: Scheduler

- Operations of the schedulers
  - Pass on operations of transactions: R, W, Abort, Commit
    - And do bookkeeping (i.e. set locks, maintain waits-for graph, …)
  - Delay operations
    - Wait with the requested action
    - Scheduler must manage a waiting queue of TXs
  - Revitalize operations when locks have been released
  - Control for deadlocks
    - Might incur aborting (many) transactions

Two Flavors of Schedulers

- Pessimistic scheduling (locking – discussed here)
  - Delay problematic actions and avoid aborts
  - Advantage: Few aborts
  - Disadvantage: Reduced parallelism – overly cautious
  - 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
  - One “controller” per data object (value, tuple, page)
    - Granularity defines the type of “X” in something like “R_1(X)”
    - The more coarse-grain, the less effort, but also the less parallelism
  - 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 X-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 managed and released
  - Either explicitly by the transaction
  - Or automatically at commit or abort time

Lock Protocols

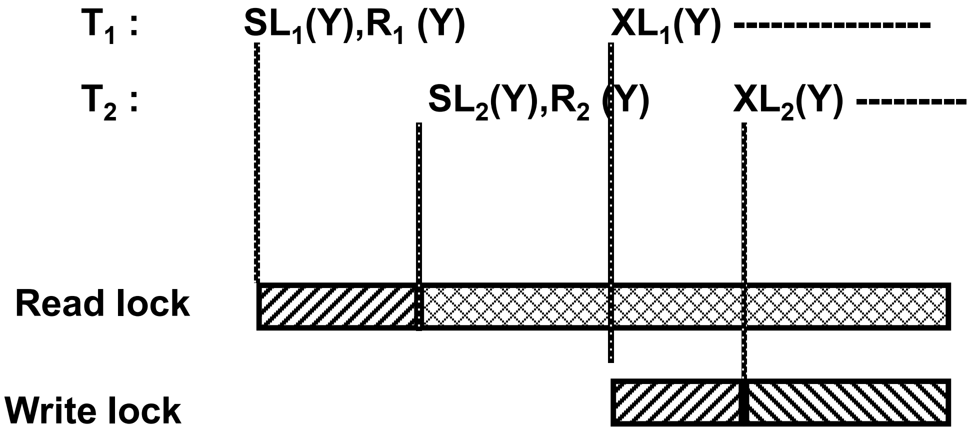
- Lock protocol: At what points in time TXs may acquire and release locks
- Example – A simple read/write lock protocol
  - A read lock (SL) must be acquired before a read
  - A write lock (XL) must be acquired before a write
  - Compatibility matrix for read and write locks
    - "+": compatible
    - "–": incompatible
- Danger: Deadlocks

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

Deadlocks

T1: <SL$_1$(Y),R1(Y),XL$_1$(Y),W1(Y),U$_1$(Y)>

T2: <SL$_2$(Y),R2(Y),XL$_2$(Y),W2(Y),U$_2$(Y)>



T1 waits for XL(Y)

T2 waits for XL(Y)

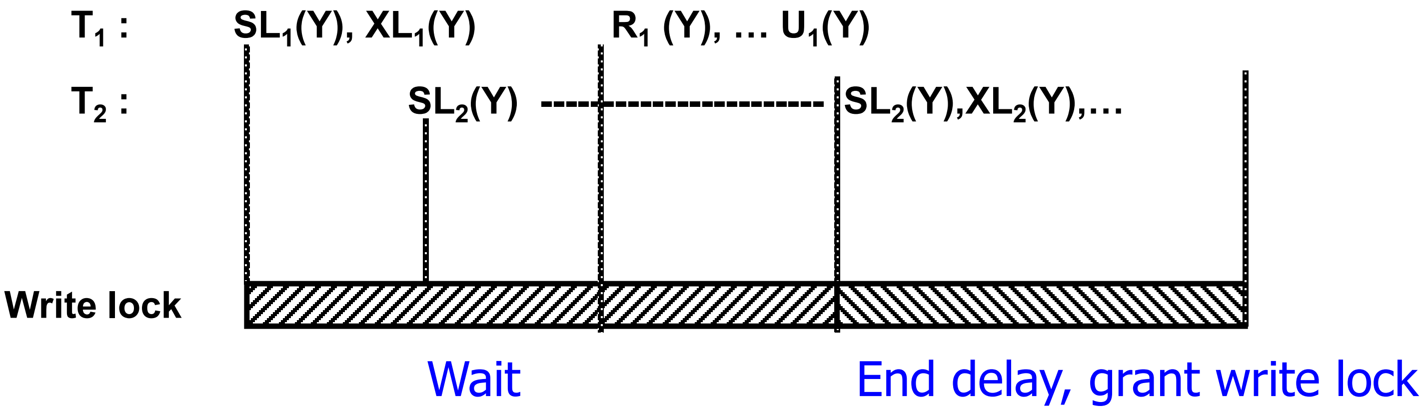
- Both SL are granted
- Both XL-requests are delayed
- Both TX wait for each other
- Locks are never released, because neither TX can proceed
- Deadlock

Option 1: Deadlock Prevention

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

T1: <SL_1(Y),XL_1(Y),R1(Y),W1(Y),U_1(Y)>

T2: <SL_2(Y),XL_2(Y),R2(Y),W2(Y),U_2(Y)>



T_1 : SL_1(Y), XL_1(Y) R_1 (Y), … U_1(Y)

T_2 : SL_2(Y) ------------------------ SL_2(Y),XL_2(Y),…

Write lock

Wait

End delay, grant write lock

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 directly at start-up time, but not anymore later
  - Delayed TX cannot acquire any locks
  - Delayed TX cannot block other TX – no deadlocks
- Disadvantages
  - If uncertain, typically more locks than needed are requested
  - Locks are kept longer than necessary
  - Lock on lock phase is a global lock, independent of conflicts

Option 2: Deadlock Detection

- LM builds a waits-for graph over all active TXs
- Graph is regularly (or prior to edge insertion) checked for cycles
  - If cycle is detected – choose a transaction and abort it
  - Often: Also abort TX after a fixed waiting 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
  - …

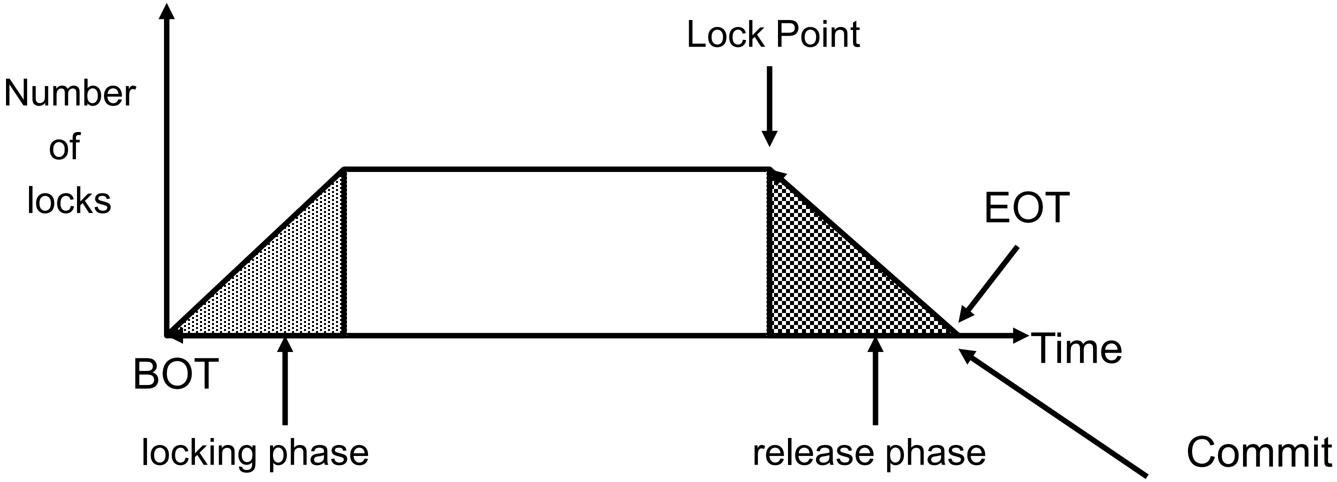
Preclaiming or Deadlock Detection

- Preclaiming: No deadlocks, no enforced aborts, conflict-serializable schedules, reduced concurrency
- Deadlock detection: Enforced aborts, no serializability guarantees, higher concurrency

- We what: Only conflict-serializable schedules, higher concurrency, few deadlocks

2-Phase Lock Protocol (2PL)

- Very prominent alternative: 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 request new locks



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 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 reached its lock point before T_j can reach its lock point

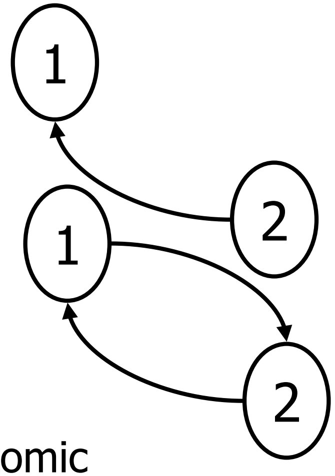
2PL Schedules are Serializable

- 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), R1(Y),W1(Y)>

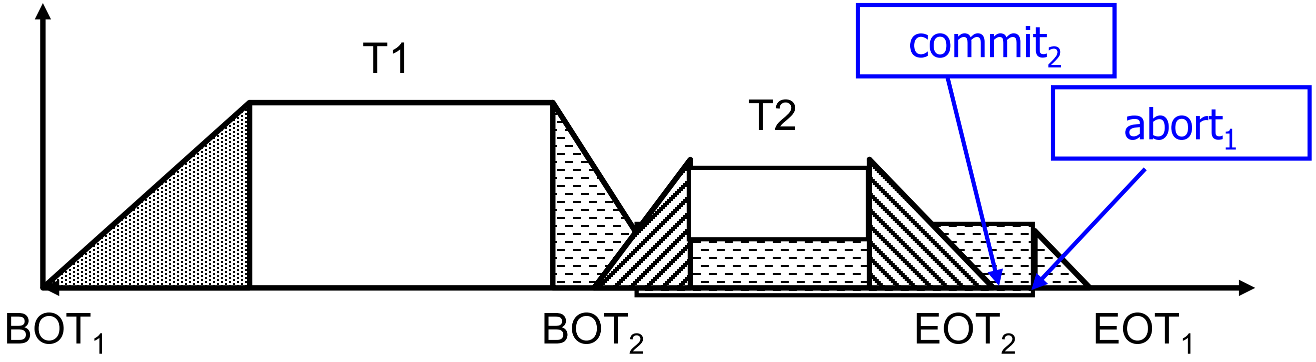
<R2(X),W2(X), R2(Y),W2(Y)>



- With 2PL, the following may happen
  - **XL_1(X),XL_1(Y),R_1(X),W_1(X),<T2 must wait>,R_1(Y), W_1(Y),U_1(X,Y),<T1 finished>,XL_2(X),<T1 commits>,…**
    - Fine
  - **SL_1(X),R_1(X),SL_2(X),<T1 must wait>,<T2 must wait>**
    - 2PL does not prevent deadlocks because lock phase is not atomic
  - **SL_2(X),XL_2(X), <T1 must wait>,R2(X),W2(X), XL_2(Y), …**
    - Fine
  - …
- U_i(X,Y,…) means: TX_i unlocks objects X, Y, …

Observation

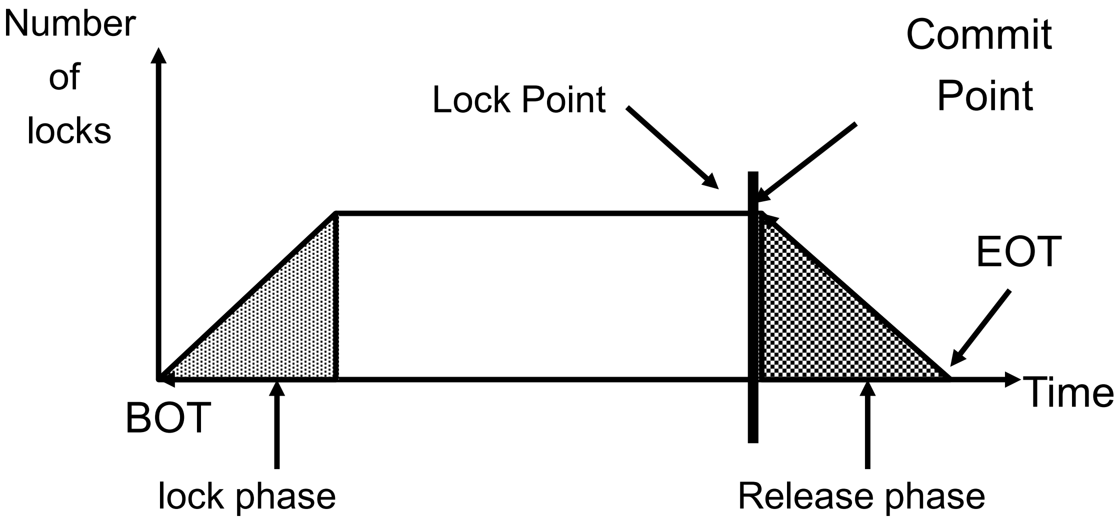
- 2PL does not guarantee recoverable schedules
  - Recall: A schedule S is called recoverable, if, whenever a committed T2 reads or writes an object X whose value was before written by a unfinished T1, then S contains a commit for T1 before the commit of T2



  - 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 also 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



Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- 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
    - Not serializable – abort T2
  - Requires rule set over all different situations
  - Not covered here

Multi-Version Synchronization

- Idea: When changing data (here T1), only change a copy
  - TX always read the last committed value (no dirty reads)
  - In previous 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 versions / of timestamps
    - Increased main memory requirements
    - Additional CPU effort

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Pessimistic synchronization: Lock protocols and deadlocks
- Optimistic synchronization: Timestamp and MVS
- SQL Isolation Levels

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 MVS, reads don never have to wait and writes are not delayed
- "Repeatable reads"
  - Reads read from local copy (in MVS), TX only checked at commit/abort time
- "Serializable"
  - Full locking protocol, e.g. 2PL

Issues not Discussed

- Optimistic scheduling, e.g. time-stamped and MVS
- Inserts: Locking a non-existing object?
- Data structures for managing locks
- Lock escalation to reduce management effort
  - From value to tuple to table …
- Locking data with (hierarchical) indexes
- Advanced TX models: Nested, compensating operations, distributed, …
- …