Transactions & Concurrency

CS460 Databases for Data Scientists

UNIVERSITY OF MANNHEIM
Data and Web Science Group

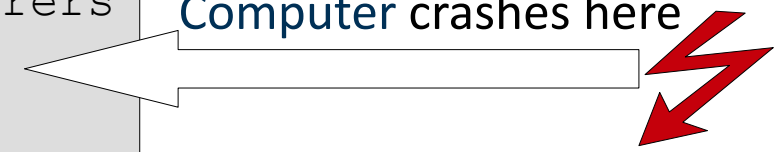
Flashback to First Lecture

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- We already stumbled upon transactions

```
Delete from file: active lecturers

Add to file: retired lecturers
```



File: **active** lecturers

Prof. Smith
Dr. Stevens
Prof. Miller

File: **retired** lecturers

Dr. Hawkins
Prof. Brown
Prof. Wilson

Flashback to First Lecture

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- ... and we already stumbled upon concurrency


```
Read num_participants from file

If num_participants < limit:
    add participant to file
```

User 1

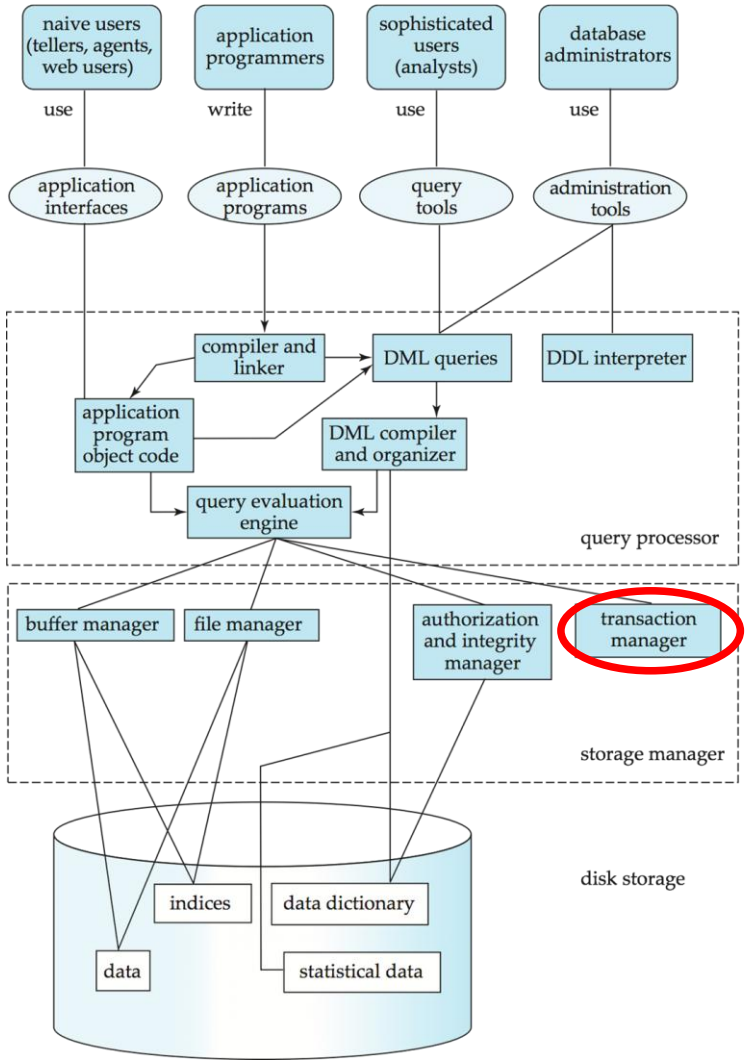
```
Read num_participants from file

If num_participants < limit:
    add participant to file
```

User 2

Flashback to First Lecture

- One of the tasks of a DBMS:
  - handle transactions
  - take care of concurrency



Today's Lecture

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Transactions
- Schedules
  - Serializability
  - Recoverability
- Protocols for Concurrent Execution
  - Lock-based protocols
    - Two-Phase Locking Protocol
    - Deadlocks
  - Timestamp-Based Protocols
  - Validation-Based Protocols

Concept of a Transaction

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- A **transaction** is a **unit** of program execution that accesses and possibly updates various data items
- E.g., transaction to transfer $50 from account A to account B:

T_1
read(A) A := A – 50 **write** (A) **read** (B) B := B + 50 **write** (B)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
    - Lecture next week
  - Concurrent (=parallel) execution of multiple transactions (today)

Requirements for Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Atomicity requirement**
  - If the transaction fails after writing to account A and before writing to account B, money will be “lost” leading to an inconsistent database state
  - Failure could be due to software or hardware
  - DBMS should ensure that updates of a partially executed transaction are not reflected in the database

Requirements for Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Consistency requirement**
  - The sum of A and B is unchanged by the execution of the transaction
  - In general, consistency requirements include
    - Explicitly specified integrity constraints, e.g., primary keys and foreign keys
    - Implicit integrity constraints
      - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction, when starting to execute, must see a consistent database
  - During transaction execution, the database may be temporarily inconsistent
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Requirements for Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Isolation requirement**
  - if between steps 3 and 9, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database

Step	T_1	T_2
1	**read**(A)	
2	A = A – 50	
3	**write**(A)	
4		**read**(A)
5		**read**(B)
6		print(A+B)
7	**read**(B)	
8	B = B + 50	
9	**write**(B)	

- Isolation can be ensured trivially by running transactions serially
  - i.e., one after the other
  - however, parallel execution is often desired due to performance benefits

Requirements for Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

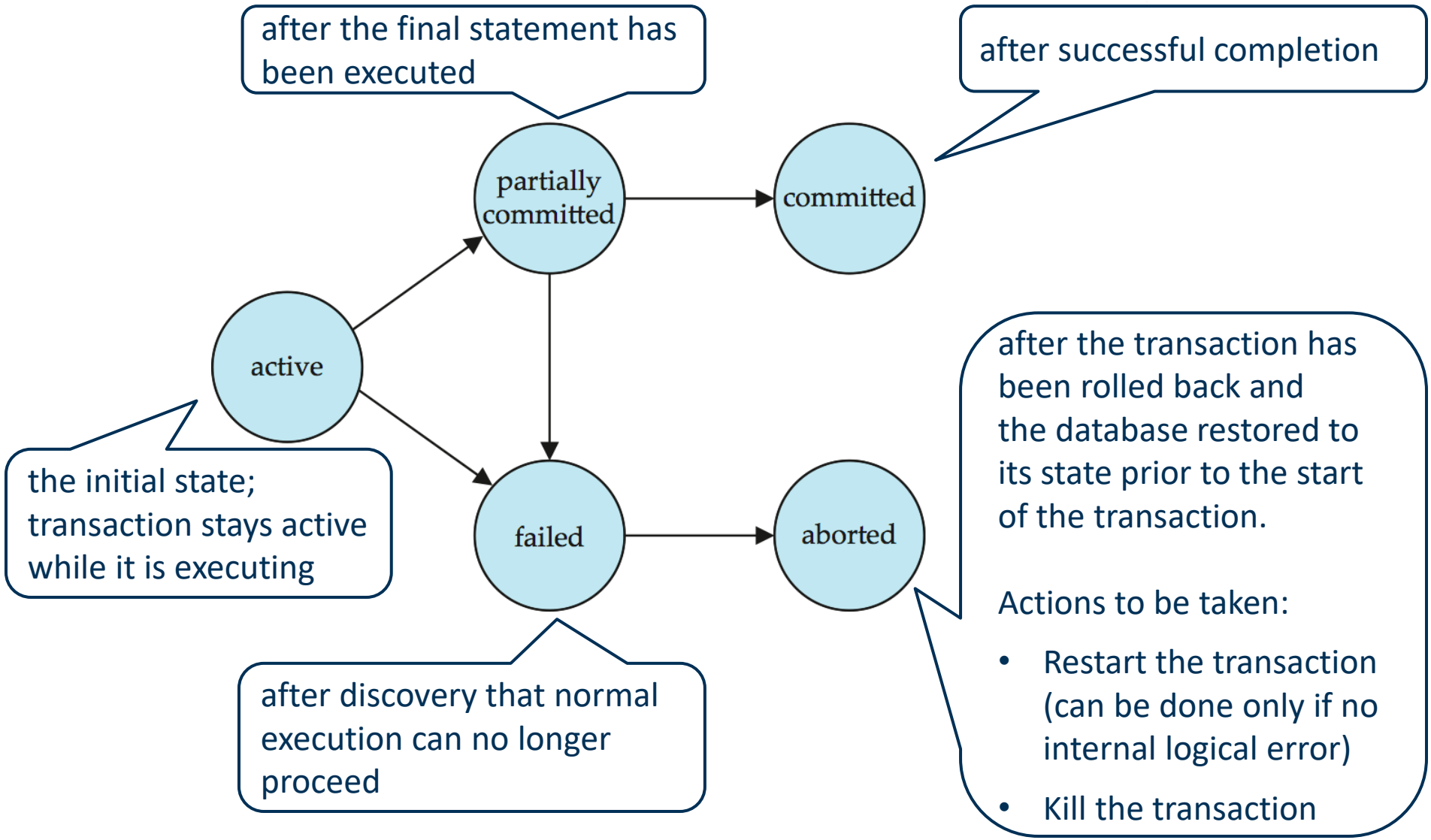
- **Durability requirement**
  - once the user has been notified that the transaction has completed,
    - i.e., the transfer of the $50 has taken place,
  - the updates to the database by the transaction must persist
    - even if there are software or hardware failures

ACID Properties

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Atomicity:** Either all operations of the transaction are properly reflected in the database, or none
- **Consistency:** Execution of a full transaction preserves the consistency of the database
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Transaction States



Concurrent Execution of Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Multiple transactions are allowed to run concurrently in the system
  - **Increased processor and disk utilization**, leading to better transaction throughput
    - e.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response** time for transactions
    - e.g., short transactions need not wait behind long ones
- **Concurrency control schemes**
  - mechanisms to achieve isolation
  - control the interaction among the concurrent transactions
  - prevent them from destroying the consistency of the database

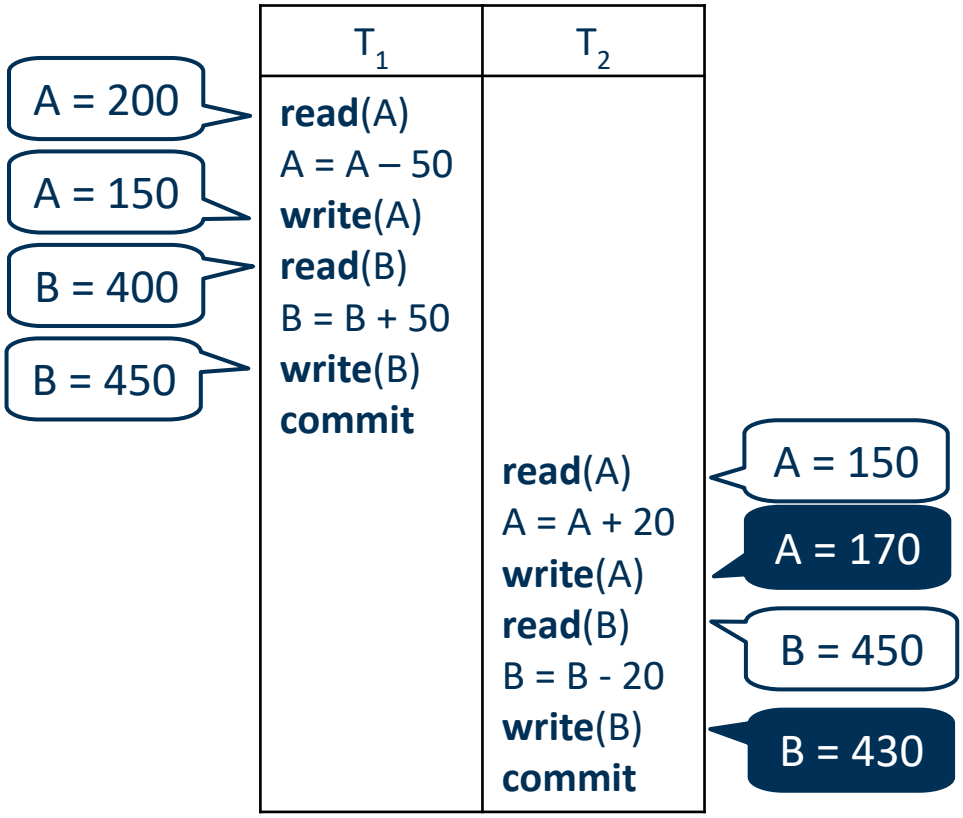
Schedules

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Schedule**
  - A sequence of instructions that specifies the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction
- A transaction that successfully completes its execution will have a **commit** instruction as the last statement
  - By default, a transaction is assumed to execute a COMMIT instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

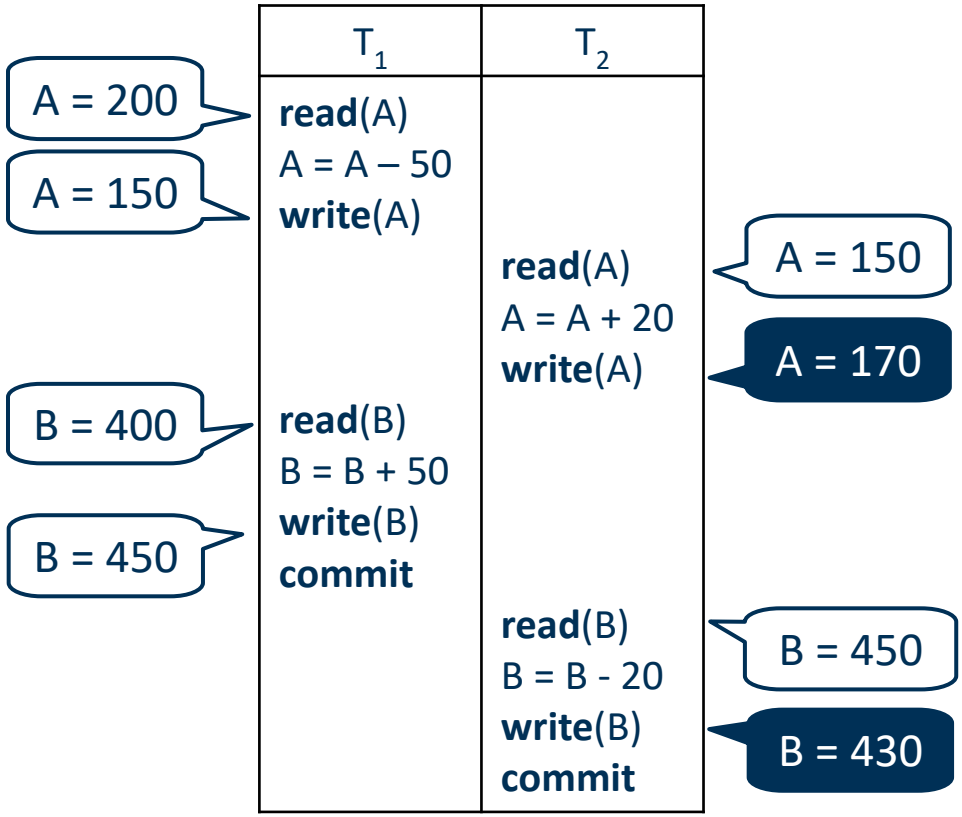
Schedule Example: Serial Schedule

- Let T_1 transfer \$ 50 from *A* to *B*, and T_2 transfer \$ 20 of the balance from *B* to *A*
- Start with e.g.
  - A: \$ 200
  - B: \$ 400
- **Serial** schedule: T_1 is executed as a whole, followed by T_2 :



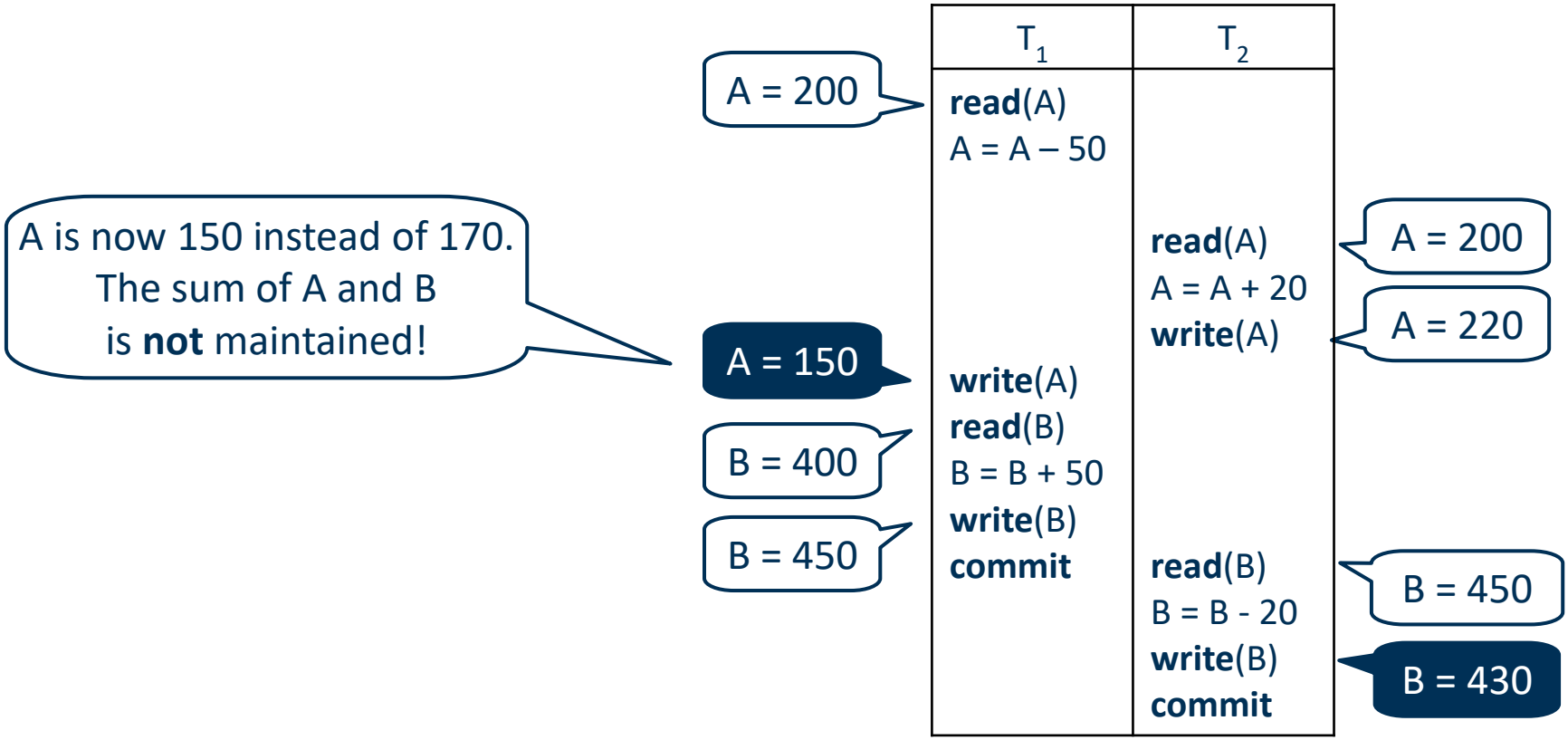
Schedule Example: Intertwined Schedule

- Let T_1 transfer \$ 50 from *A* to *B*, and T_2 transfer \$ 20 of the balance from *B* to *A*
- **Intertwined** schedule: parts of T_1 are executed, interrupted by parts of T_2
  - the sum A+B is maintained



Schedule Examples:
Wrong Schedule

- Let T_1 transfer \$ 50 from *A* to *B*,
  and T_2 transfer \$ 20 of the balance from *B* to *A*



Serializability

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Basic assumption: transactions preserve database consistency
  - i.e., serial execution of a set of transactions
    also preserves database consistency
- A (possibly concurrent) schedule is **serializable**
  if its outcome is equivalent to a serial schedule
  - We ignore operations other than read and write instructions
  - Transactions may perform arbitrary computations on data in between
  - Our simplified schedules consist of only read and write instructions

Conflicting Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- When can we swap instructions of transactions?

T_1	T_2
read(A) **write**(A)	 **read**(B) **write**(B)

T_1	T_2
read(A) **write**(A)	 **read**(B) **write**(B)

T_1	T_2
read(A) **write**(A)	 **read**(B) **write**(B)

  - We can swap all statements if transactions work on completely different sets of data items
    - In the example T_1 work on data item A and T_2 work on data item B
    - Result will always be the same
  - Conflicts can only occur if the same data item is accessed

Conflicting Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- When can we swap instructions of transactions? (four cases)

1.

T_1	T_2
read(A)	
	read(A)

The order of T_1 or T_2 doesn't matter
Same value of A is read

2.

T_1	T_2
read(A)	
	write(A)

If swapped, T_1 reads a different value.
Order of $T_1$and T_2 matters -> conflict

3.

T_1	T_2
write(A)	
	read(A)

If swapped, T_2 reads a different value.
Order of $T_1$and T_2 matters -> conflict

4.

T_1	T_2
write(A)	
	write(A)

Only value written by T_2 is preserved in the DB.
If swapped, only value of T_1 is preserved.
Order of $T_1$and T_2 matters -> conflict

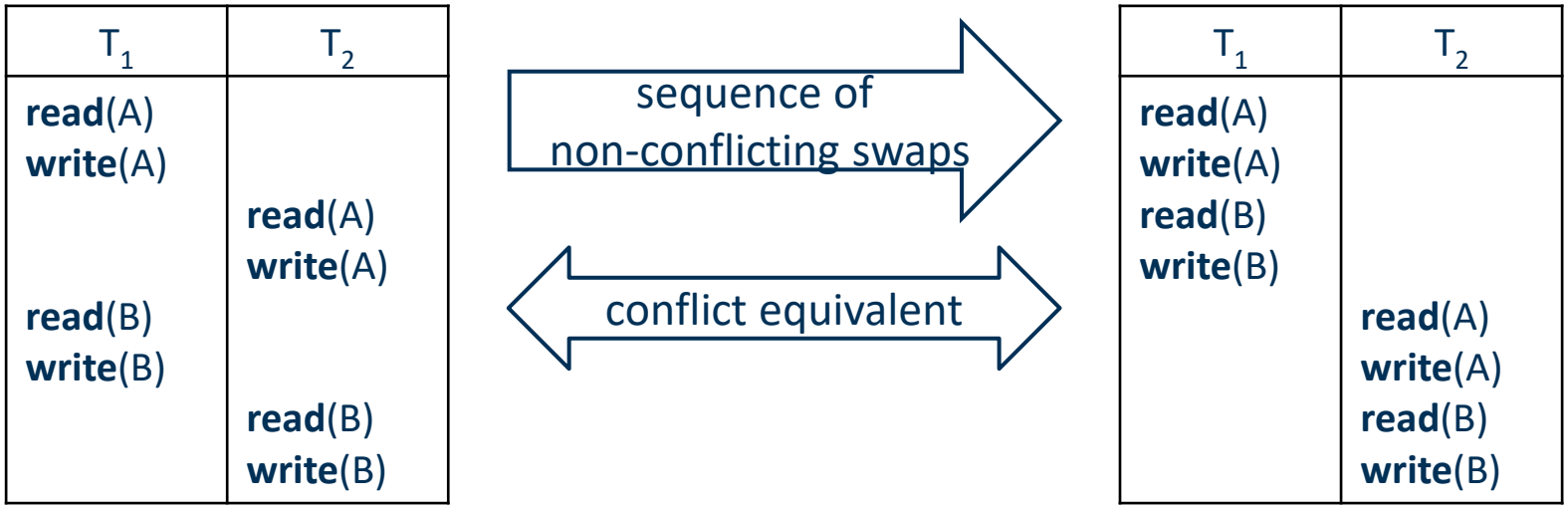
Conflicting Transactions

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- When can we swap instructions of transactions?
- Let I_i and I_j be two instructions of transactions T_i and T_j
  - Instructions I_i and I_j **conflict**
    - if and only if there exists some data item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q
- Implications on serializability:
  - Non-conflicting instructions can be executed in **any** order
  - A conflict between I_i and I_j forces a **temporal order** between them

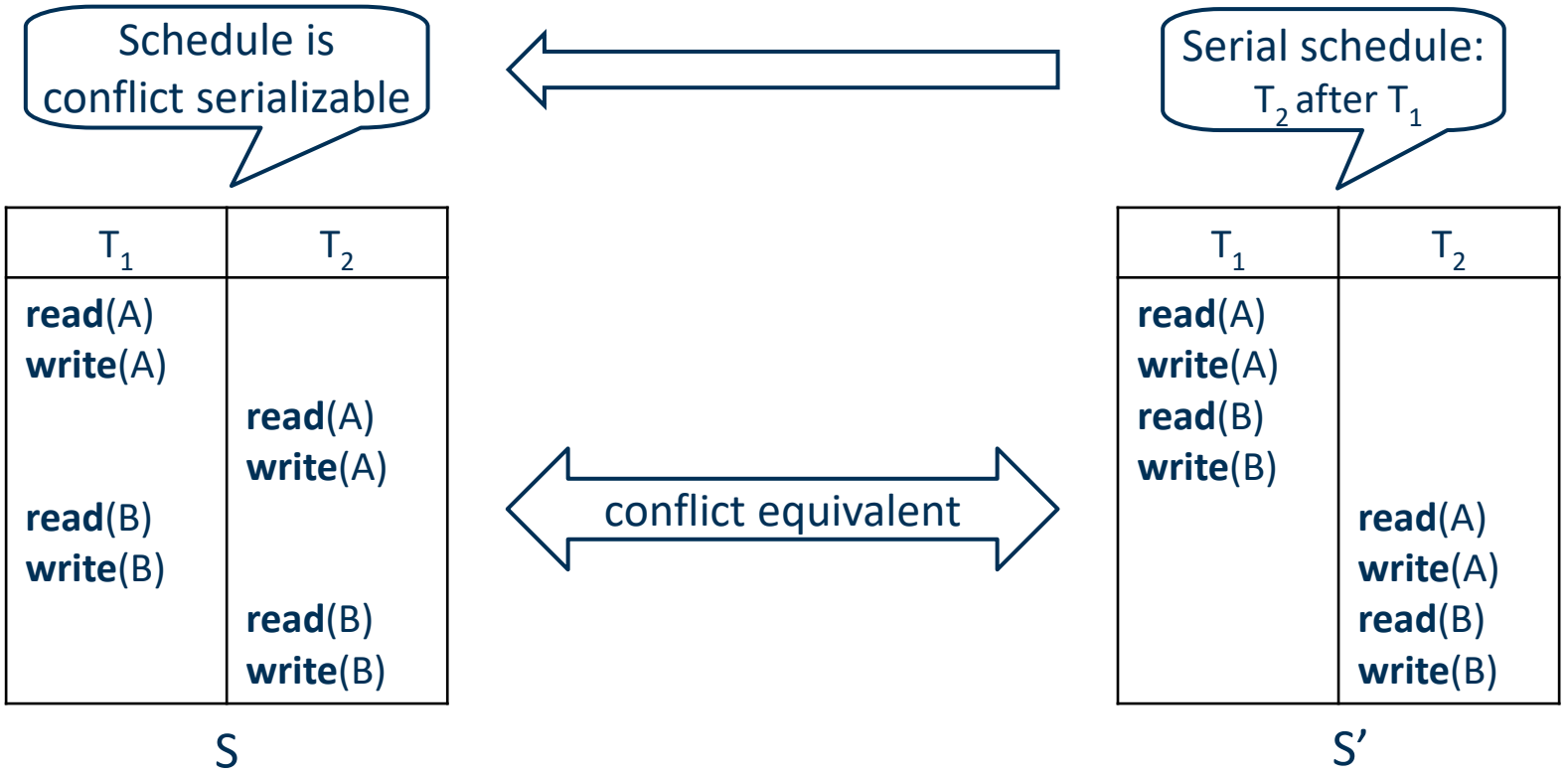
Conflict Equivalence

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.



Conflict Serializability

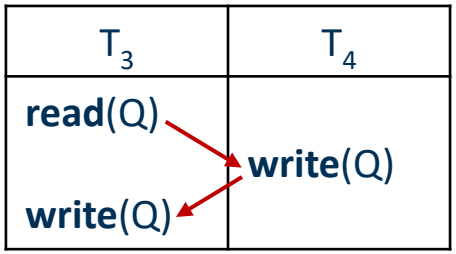
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule



Conflict Equivalence and Serializability

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Example of a schedule that is not conflict serializable:

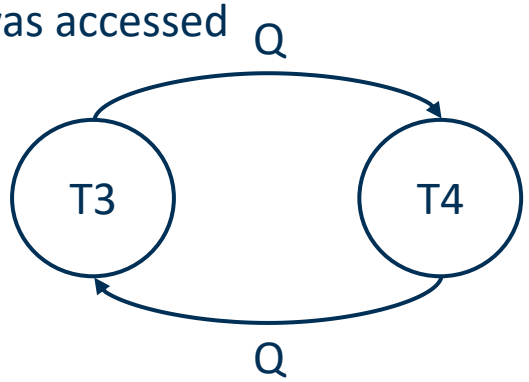


- **write**(Q) in T_3 conflicts with **write**(Q) in T_4 and **write**(Q) in T_4 conflicts with **read**(Q) in T_3
  - i.e., we are unable to swap instructions in the above schedule to obtain either the **serial** schedule < T_3, T_4 >, or the **serial** schedule < T_4, T_3 >

Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph:** a directed graph where
  - Transactions are the vertices
  - Conflicts between statements of these transactions are edges
    - draw an edge from T_i to T_j if one of the conditions hold
      - T_i executes **write**(Q) before T_j executes **read**(Q)
      - T_i executes **read**(Q) before T_j executes **write**(Q)
      - T_i executes **write**(Q) before T_j executes **write**(Q)
    - We may label the edge by the item that was accessed

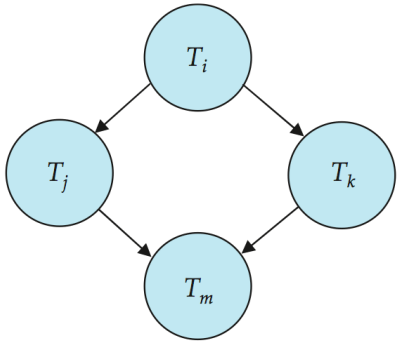
T_3	T_4
read(Q)	
	write(Q)
write(Q)	



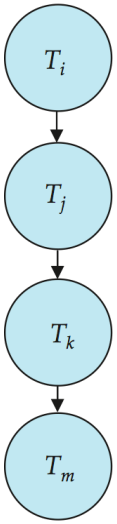
Testing for Conflict Serializability

UNIVERSITY OF MANNHEIM
Data and Web Science Group

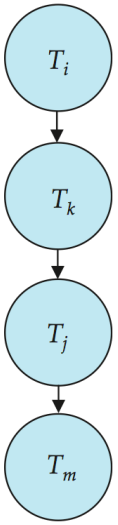
- A schedule is conflict serializable
  - if and only if its precedence graph is acyclic
  - serializability order can be obtained by a topological sorting of the graph
    - i.e., a linear order consistent with the partial order of the graph
    - Example: both (b) and (c) are possible partial orders of (a)
- Cycle-detection algorithms in $O(n^2)$ exist
  - where n is the number of vertices in the graph
  - better algorithms are in O(n+e) where e is the number of edges



(a)



(b)



(c)

Recoverable Schedules

- Consider the following schedule:

T_8	T_9
read(A)	
write(A)	
	read(A)
	commit
read(B)	

T_8 aborts here

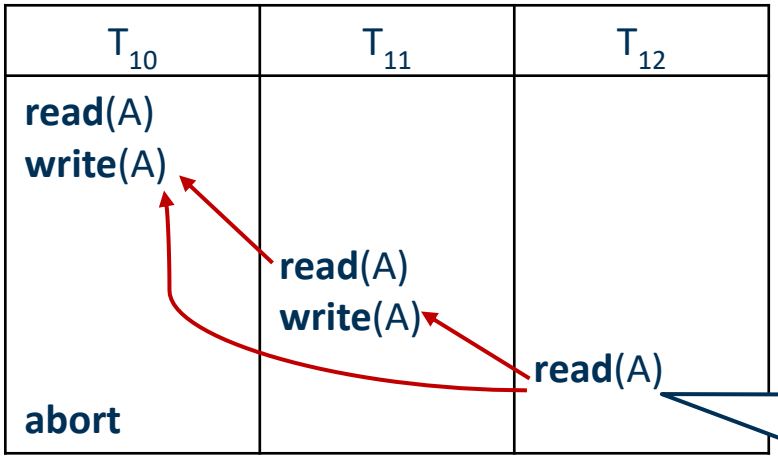
- What happens if T_8 aborts after T_9 commits?
  - T_9 would have read an inconsistent database state and possibly shown the value of A to the user (because T_9 is commited) -> this should be avoided
- A schedule is ***recoverable*** if the following holds:
  - if a transaction T_j (T_9 in the example) reads a data item previously written by a transaction T_i
  - then the **commit** operation of T_i **must** appear before the **commit** operation of T_j

T_{8a}	T_{9a}
read(A)	
write(A)	
	read(A)
read(B)	
commit	
	commit

Cascading Rollbacks

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Consider the following schedule:



Recoverable schedule means: If T_{11} or T_{12} wants to commit, they need to wait for T_{10} to commit

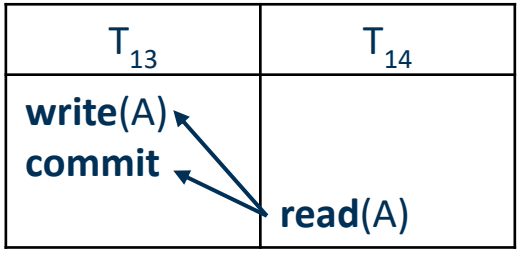
- On the abort of T_{10}
  - all three transactions need to abort and be rolled back
  - can mean undoing a significant amount of work

Still cascading rollbacks will happen

Cascadeless Schedules

UNIVERSITY OF MANNHEIM
Data and Web Science Group

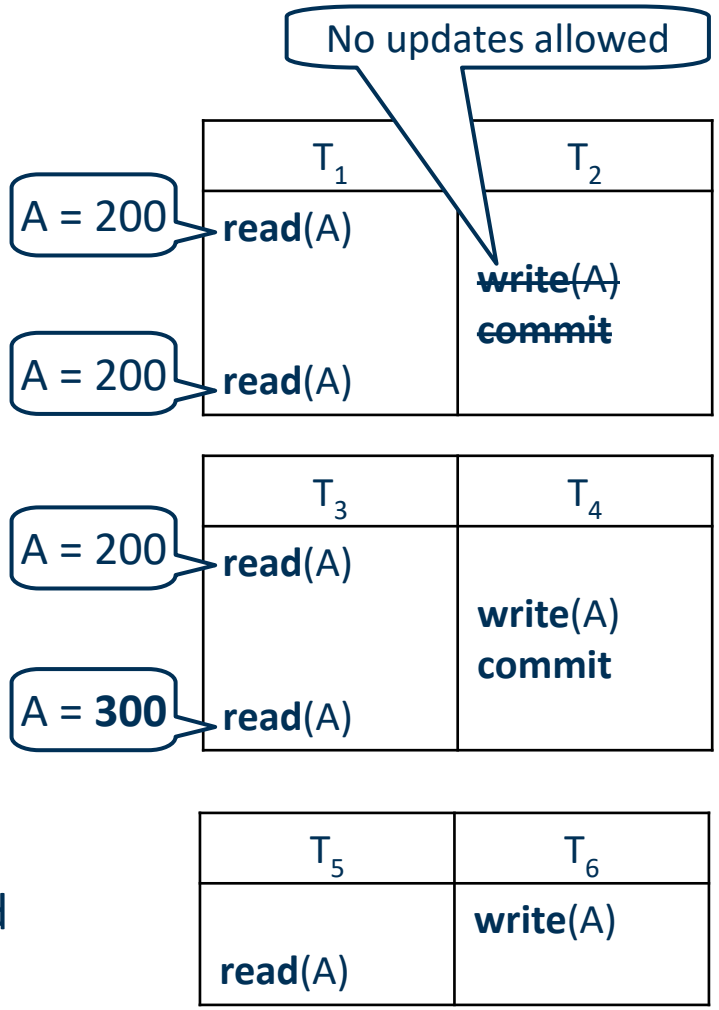
- To prevent cascading rollbacks, a stronger version exists
  - A schedule is **cascadeless** if and only if
    - for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i,
    - the **commit** operation of T_i appears before the **read** operation of T_j
  - It is desirable to restrict the schedules to those that are cascadeless
  - Every cascadeless schedule is also recoverable
    - the reverse need not hold



Cascadeless (commit before read)

Levels of Consistency

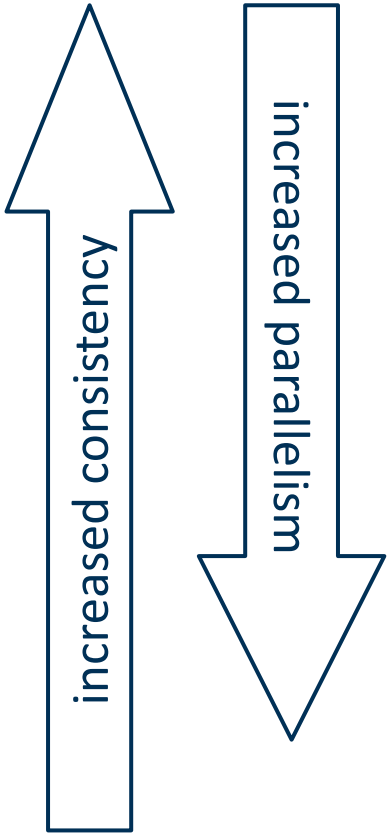
- **Serializable:** default
- **Repeatable read:**
  - only committed records to be read
  - successive reads of same record must return the same value
- **Read committed:**
  - only committed records can be read,
  - successive reads of record may return different (but committed) values
- **Read uncommitted:**
  - even uncommitted records may be read



Levels of Consistency

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Serializable:** default
- **Repeatable read:**
  - only committed records to be read
  - successive reads of same record must return the same value
- **Read committed:**
  - only committed records can be read,
  - successive reads of record may return different (but committed) values
- **Read uncommitted:**
  - even uncommitted records may be read



Transaction Definition in SQL

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL
  - A transaction begins implicitly
  - A transaction ends by:
    - **Commit work** commits current transaction and begins a new one
    - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - implicit commit can be turned off by a database directive
  - e.g., in JDBC, `connection.setAutoCommit(false);`

Concurrency Control in DBMS

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules
  - but provides a poor degree of parallelism
- Concurrency control protocols have to trade off
  - degree of parallelism they achieve
  - amount of overhead they incur

Lock-based Protocols

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written.
     X-lock is requested using **lock-X** instruction
  2. *shared (S) mode*. Data item can only be read.
     S-lock is requested using **lock-S** instruction
- Lock requests are made to the concurrency-control manager
  - by the application accessing the database
  - transaction can proceed only after request is granted

Requesting and Granting Locks

UNIVERSITY OF MANNHEIM
Data and Web Science Group

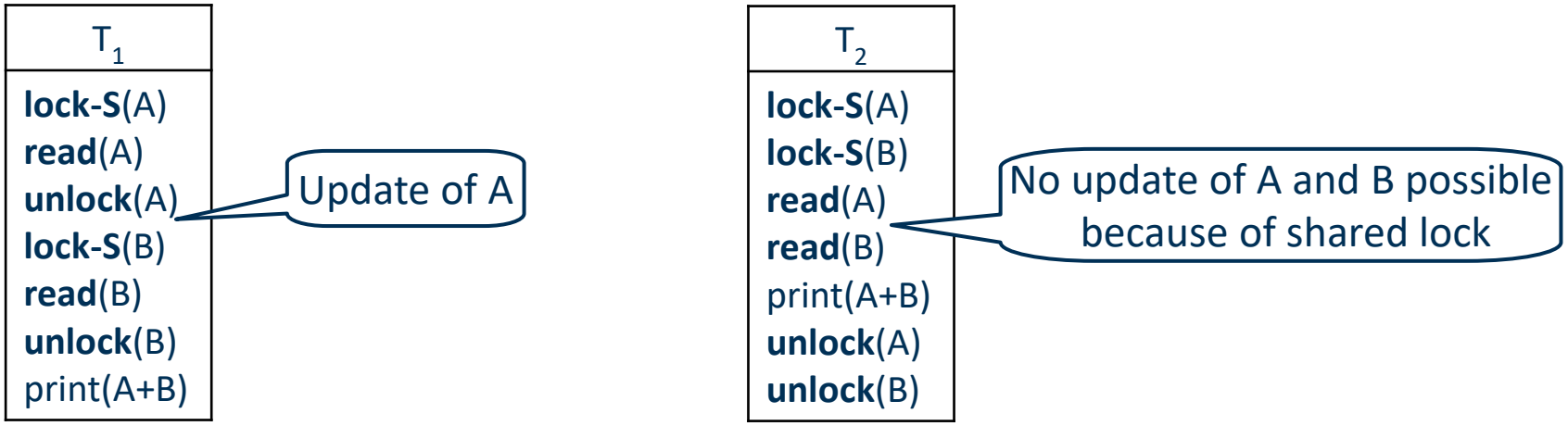
- Transactions request locks
  - can be granted if the requested lock is compatible
- Compatibility:
  - Any number of transactions can hold shared locks on an item
  - If any transaction holds an exclusive on the item, no other transaction may hold any lock on the item
- If a lock cannot be granted
  - the requesting transaction has to wait until all incompatible locks are released

requested \ already granted	S	X
S	true	false
X	false	false

Lock-based Protocols

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Example of two transactions performing locking:

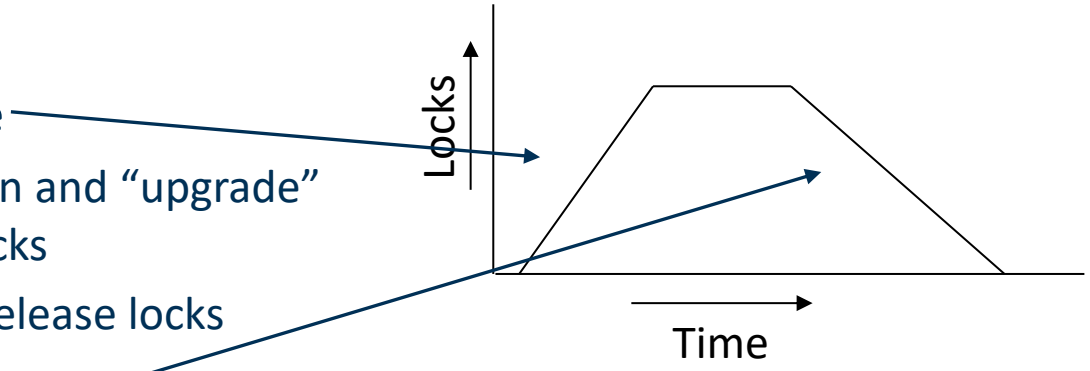


- Only T_2 is serializable
  - in T_1, if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be inconsistent
- A **locking protocol** is a set of rules followed by all transactions
  - Locking protocols restrict the set of possible schedules

The Two-Phase Locking Protocol

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Protocol that ensures conflict serializable schedules
- Runs in two phases
  - Phase 1: Growing Phase
    - Transaction may obtain and “upgrade” shared to exclusive locks
    - Transaction may not release locks
  - Phase 2: Shrinking Phase
    - Transaction may release and “downgrade” exclusive to shared locks
    - Transaction may not obtain locks



- The protocol assures serializability
  - It can be proved that the transactions can be serialized in the order of their **lock points**,
  - i.e., the point where a transaction acquired its final lock

Automatic Acquisition of Locks

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- A transaction T_i issues the standard read/write instruction, without explicit locking calls
- The operation **read**(D) is processed by the DBMS as:

```
if T_i has a lock on D
    read(D)
else
    if necessary wait until no other transaction has a lock-X on D
    grant T_i a  lock-S on D
    read(D)
```

Automatic Acquisition of Locks

UNIVERSITY OF MANNHEIM
Data and Web Science Group

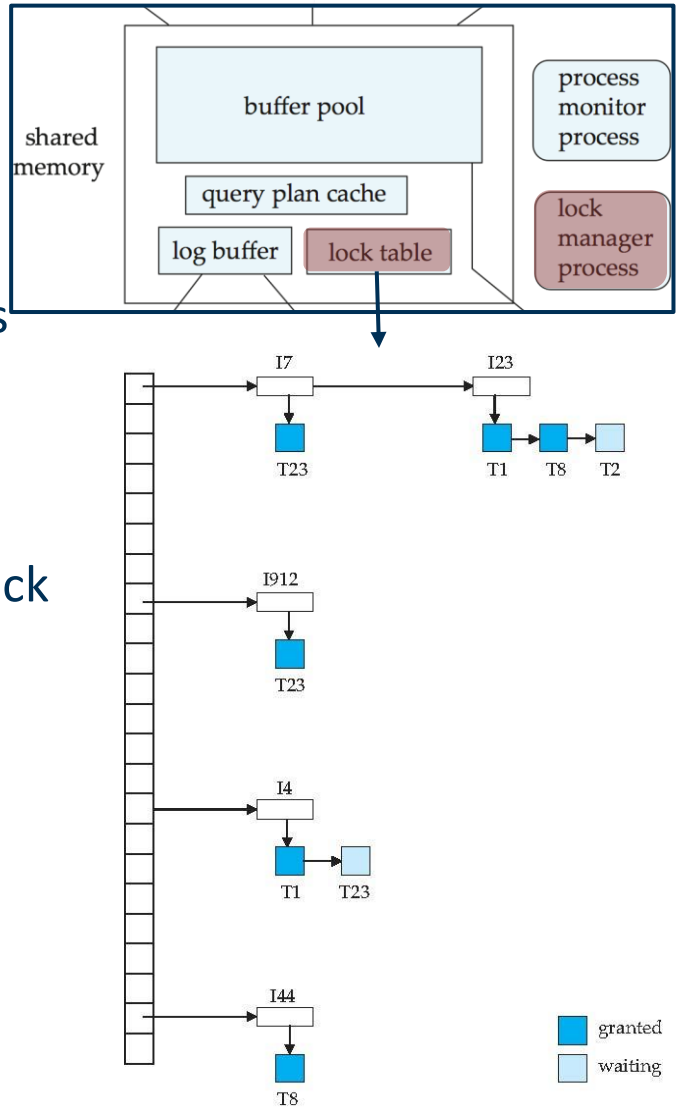
- A transaction T_i issues the standard read/write instruction, without explicit locking calls
- The operation **write*(D)*** is processed by the DBMS as:

```
if T_i has a lock-X on D
    write(D)
else
    if necessary wait until no other transaction has any lock on D
    if T_i has a lock-S on D
        upgrade lock on D to lock-X
    else
        grant T_i a lock-X on D
    write(D)
```

- All locks are released after commit or abort

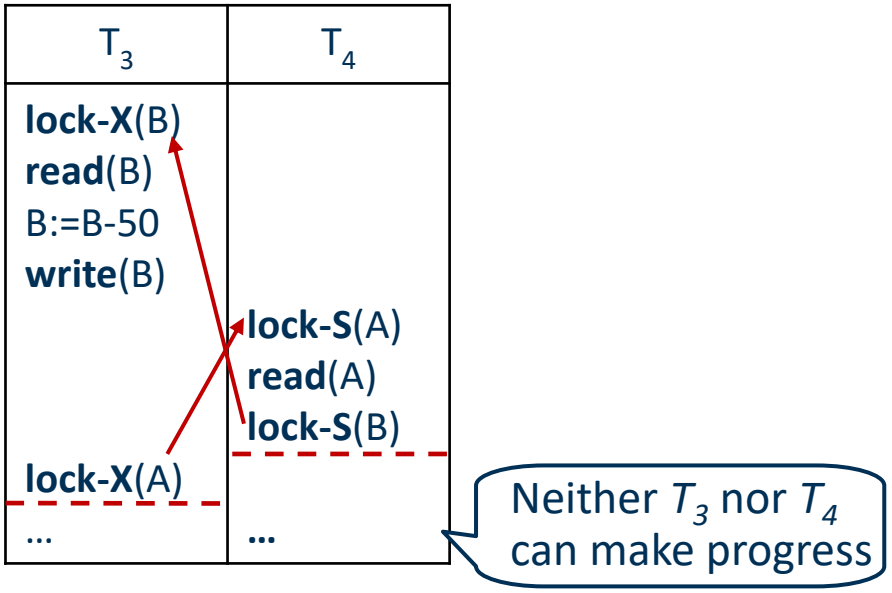
Implementation of Locking – Lock Table

- A **lock manager** can be implemented as a separate process
  - transactions send lock and unlock requests to the lock manager
  - lock manager replies to a lock request by sending a lock grant message or a message asking the transaction to roll back in case of a deadlock
- The lock manager maintains **lock table** to record granted locks and pending requests
  - implemented as an in-memory hash table indexed on the name of the data item



Deadlocks

- Consider the partial schedule



- Situation:
  - executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on B,
  - executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*
- Such a situation is called a **deadlock**
  - to handle the problem,
    one of T_3 or T_4 must be rolled back and its locks released

Deadlocks

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Two-phase locking protocol
  - guarantees **serializability** but does **not** ensure freedom from deadlocks
- The potential for deadlock exists in most locking protocols
  - but there are prevention mechanisms (see later)
- When a deadlock occurs
  - rollbacks are necessary
  - there is a possibility of cascading roll-backs
    (also under two-phase locking protocol)
    - but cascading rollbacks can be expensive
    - Modified protocol called **strict two-phase locking**
      - a transaction must hold all its exclusive locks until it commits/aborts
      - avoids cascading rollbacks

Starvation

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- In addition to deadlocks, there is a possibility of **starvation:**
  - A transaction may be waiting for an X-lock on an item
  - while a sequence of other transactions request and are granted an S-lock on the same item

- **Starvation** occurs if the concurrency control manager is badly designed
  - Unfair Prioritization etc
  - Concurrency control manager can be designed to prevent starvation

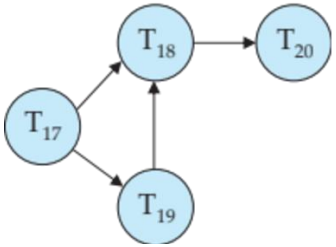
Might wait forever

T_1	T_2	T_3	T_4
lock-X(B)	**lock-S**(B) …. **unlock**(B)		
		lock-S(B) … **unlock**(B)	
			lock-S(B) … **unlock**(B)

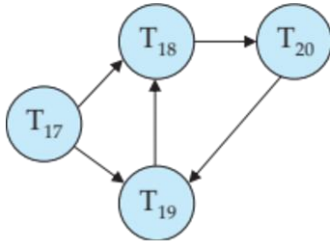
Deadlock Detection

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Deadlocks can be detected using a **wait-for graph**
  - Vertices are the transactions
  - Edges:
    - T_i requests a lock on a data item currently being locked by T_j, the edge $T_i \rightarrow T_j$ is inserted (T_i is waiting for T_j)
    - T_j releases lock on a data item needed by T_i (or T_i is rolled back), the edge $T_i \rightarrow T_j$ is removed from the wait-for graph
  - System is in a deadlock state $\leftrightarrow$ the wait-for graph has a cycle
    - invoke a deadlock-detection algorithm to look for cycles



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

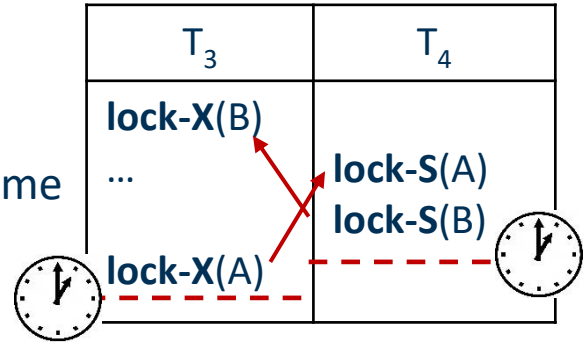
UNIVERSITY OF MANNHEIM
Data and Web Science Group

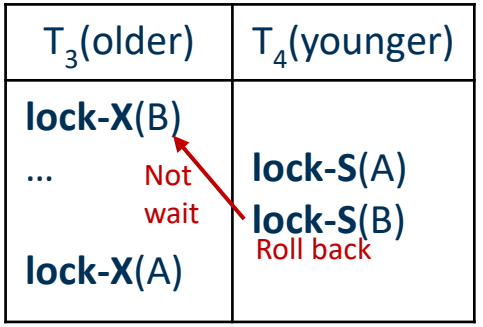
- When deadlock is detected :
  - some transaction will have to rolled back (made a victim)
  - select a transaction as a victim that will incur minimum cost
- Rollback – determine how far to roll back transaction
  - Total rollback: Abort the transaction and then restart it
  - More effective: roll back transaction only as far as necessary to break deadlock
- Starvation happens if same transaction is always chosen as victim
  - Solution: include the number of rollbacks in the cost factor to avoid starvation

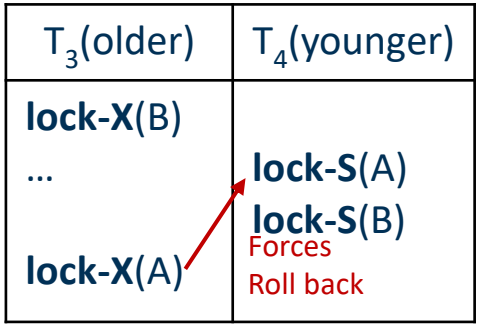
Deadlock Prevention Protocols

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Timeout-Based** Scheme
  - transactions wait for a lock for specified amount of time
  - if the lock has not been granted within that time → roll back
- **Wait-Die** Scheme
  - older transaction may wait for younger one to release data item
  - younger transactions never wait for older ones (they are rolled back instead)
  - a transaction may die several times
- **Wound-Wait** Scheme
  - older transaction *wounds* (forces rollback) of younger transaction (instead of waiting for it)
  - younger transactions may wait for older ones
  - may cause fewer rollbacks than *wait-die* scheme







Timestamp-based Protocols

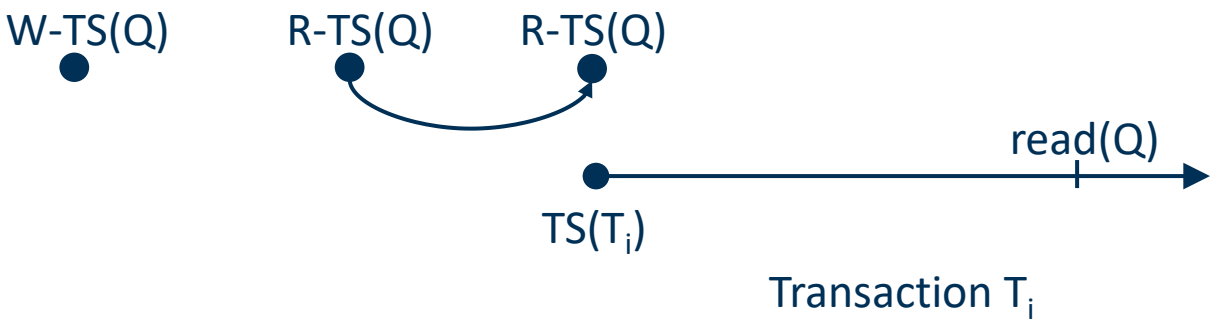
UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Locking protocol (before) determine order by first lock
- Timestamp-based protocol determine order in advance
  - Each transaction T_i is issued a timestamp **TS(T_i)** when it enters the system
    - timestamps must be free of duplicates
  - The protocol manages concurrent execution such that the time-stamps determine the serializability order
  - In order to assure such behavior, the protocol maintains two timestamp values for each data *Q*:
    - **W-TS**(*Q*) is the largest time-stamp **of any transaction** that executed **write**(*Q*) successfully
    - **R-TS**(*Q*) is the largest time-stamp **of any transaction** that executed **read**(*Q*) successfully

Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

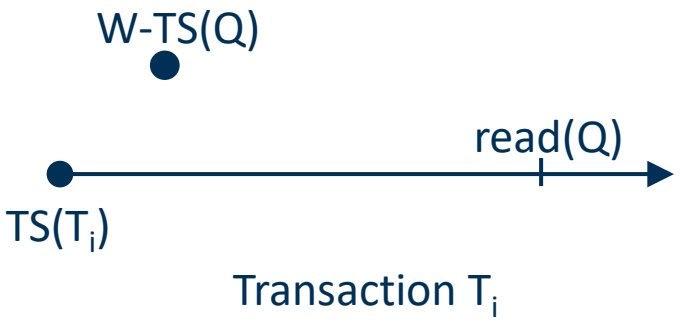
- Transaction T_i issues a **read**(Q)
  - if **TS**(T_i) > **W-TS**(Q)
    - execute **read** operation,
      set **R-TS**(Q) to **max**(**R-TS**(Q),**TS**(T_i))



Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

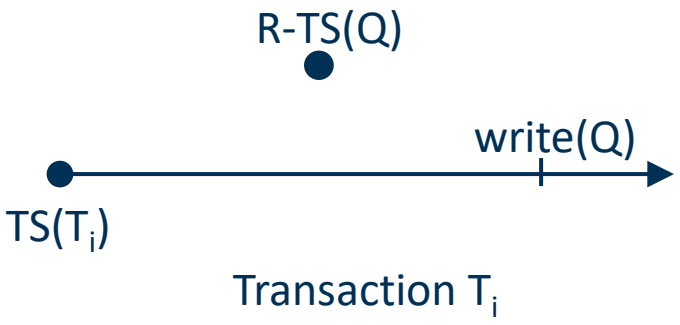
- Transaction T_i issues a **read**(Q)
  - if **TS**(T_i) $\leq$ **W-TS**(Q),
    then T_i needs to read a value of Q that was already overwritten
    → reject **read**, rollback T_i



Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

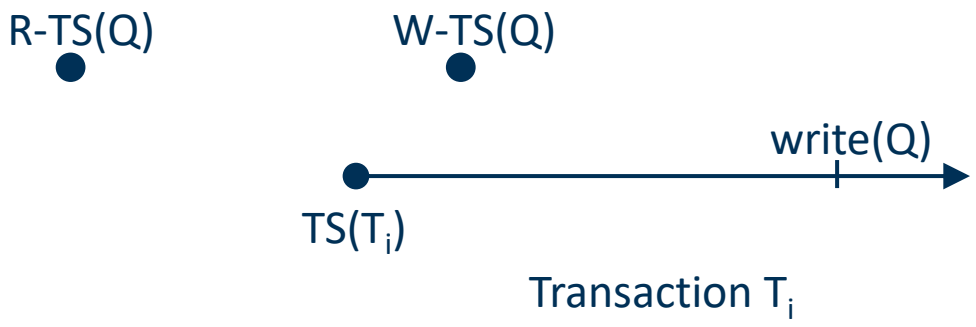
- Transaction T_i issues **write**(Q)
  - if **TS**(T_i) < **R-TS**(Q),
    then the value of Q that T_i is producing was read previously
    → reject **write**, rollback T_i



Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

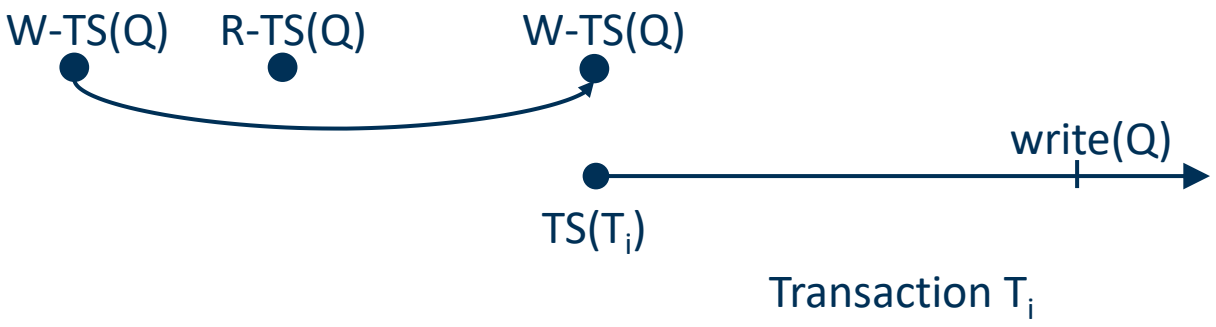
- Transaction T_i issues **write**(Q)
  - if **TS**(T_i) < **W-TS**(Q),
    then T_i is attempting to write an obsolete value of Q
    → reject **write**, rollback T_i



Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

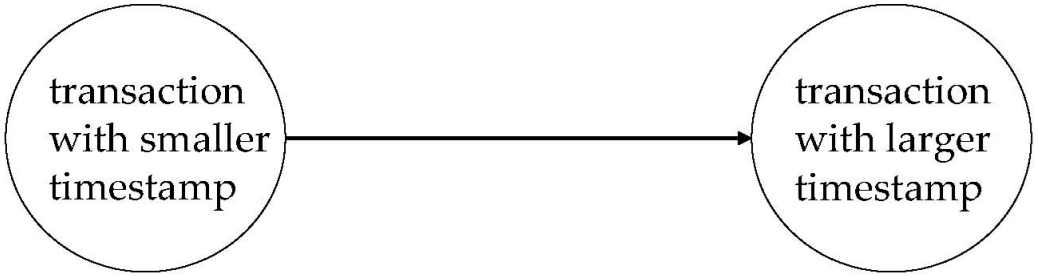
- Transaction T_i issues **write**(Q)
  - **TS**(T_i) ≥ **R-TS**(Q) and **TS**(T_i) ≥ **W-TS**(Q)
    - execute **write** and set **W-TS**(Q) to **TS**(T_i)



Timestamp-based Scheduling

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock
  - no transaction ever waits, there are only rollbacks
- But the schedule may not be cascade-free
  - and may not even be recoverable

Validation Based Protocol

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Execution of transaction T_i is done in three phases
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a ''validation test'' to determine if local variables can be written without violating serializability
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back
- The three phases of concurrently executing transactions can be interleaved
  - but each transaction must go through the three phases in that order
- Assume for simplicity that the validation and write phase occur together, atomically and serially
  - i.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

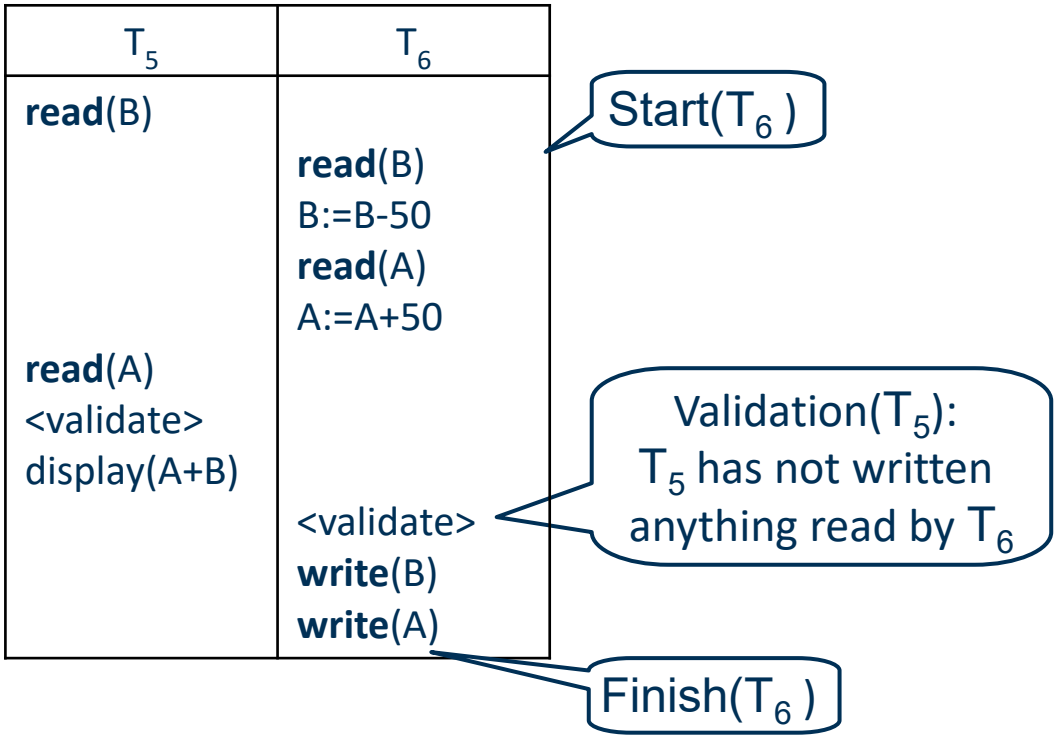
Validation Based Protocol

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time; this is done to increase concurrency.
  - Thus, TS(T_i) is given the value of Validation(T_i)
- This protocol is useful and gives greater degree of concurrency
  - if probability of conflicts is low
  - serializability order is not pre-decided
  - relatively few transactions will have to be rolled back

Validation Test for Transaction T_j

- Example schedule using validation



Validation Test for Transaction T_j

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j
- then validation succeeds and T_j can be committed
  - otherwise, validation fails and T_j is aborted
- *Explanation*: Either the first condition is satisfied, i.e., there is no overlapped execution, or the second condition is satisfied, i.e.,
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i

Summary

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Parallel access to databases brings challenges
  - easy solution: process one transaction after the other
  - higher performance solution: support parallelism
- Transactions & Serializability
  - Methods for generating serializations
- Locks & Deadlocks
- Protocols

Questions?

UNIVERSITY OF MANNHEIM
Data and Web Science Group