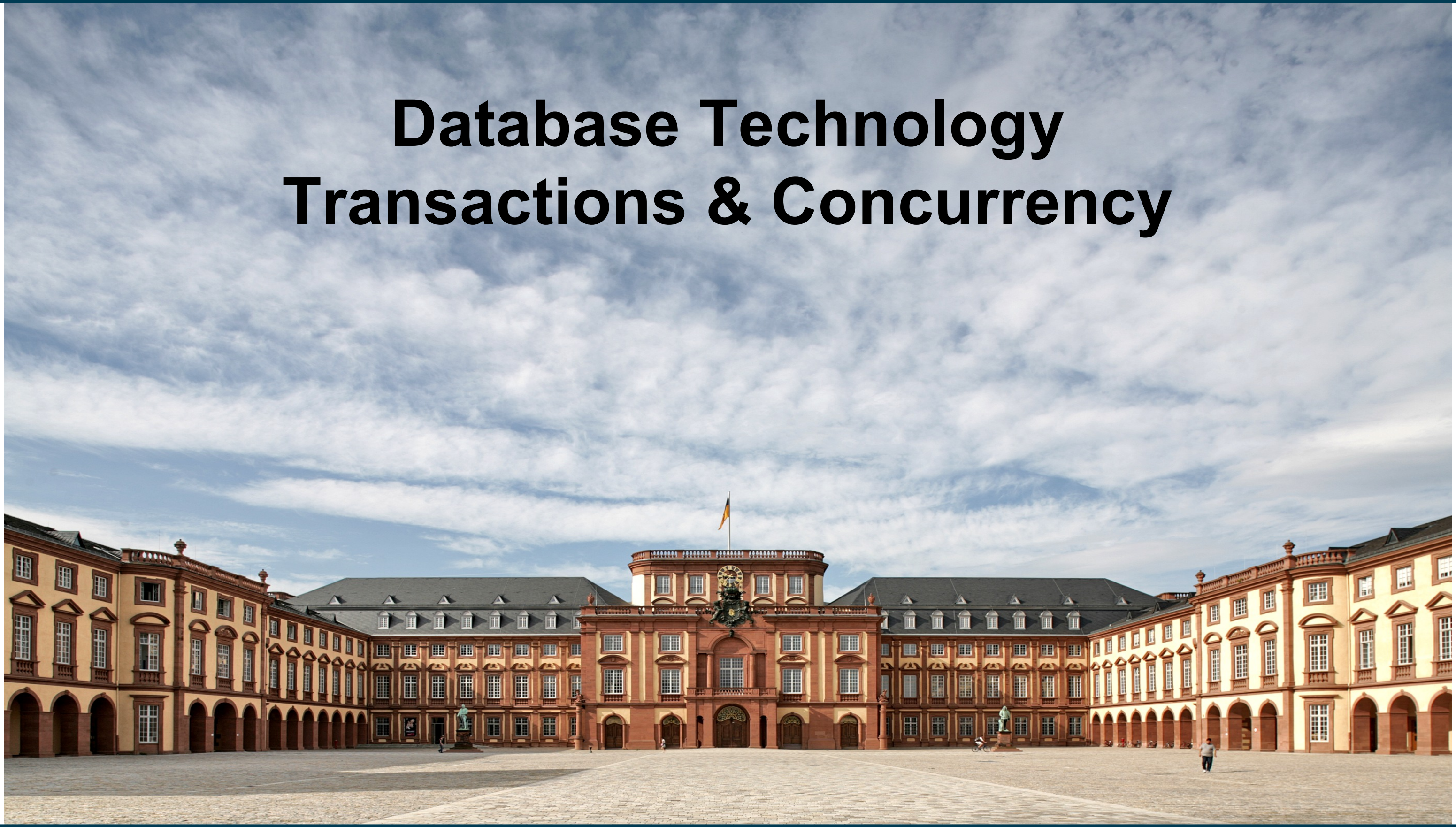
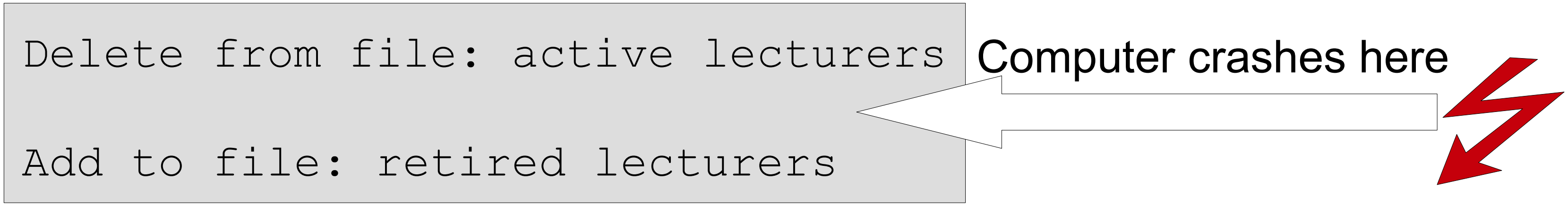


Database Technology
Transactions & Concurrency

Heiko Paulheim

Flashback to First Lecture

- We already stumbled upon transactions



```
Delete from file: active lecturers

Add to file: retired lecturers
```

Computer crashes here

File: active lecturers

Prof. Smith
Dr. Stevens
Prof. Miller

File: retired lecturers

Dr. Hawkins
Prof. Brown
Prof. Wilson

Flashback to First Lecture

- ...and we already stumbled upon concurrency

```
Read num_current_participants
      from file

If num_current_participants
      < limit
Then
      add participant to fi
```

User 1

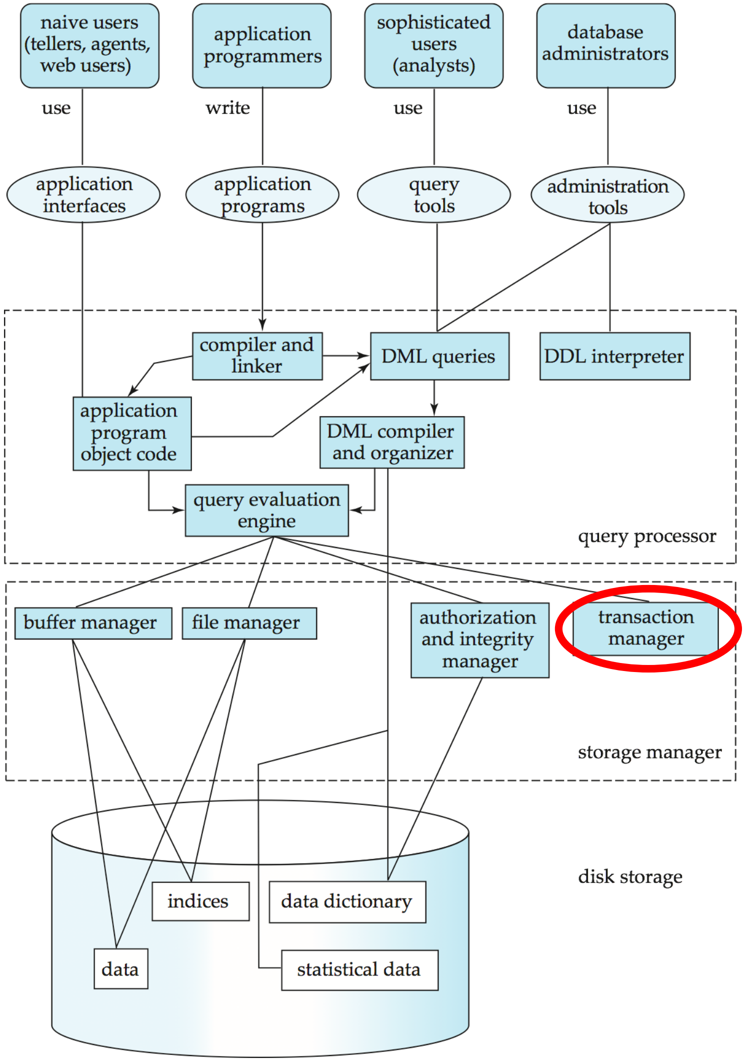
```
Read num_current_participants
      from file

If num_current_participants
      < limit
Then
      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

- Transactions
  - Concurrent Executions
  - Serializability
  - Recoverability
  - Testing for Serializability
  - Transaction Definition in SQL
- Protocols for Concurrent Execution
  - Lock-Based Protocols
  - Timestamp-Based Protocols
  - Validation-Based Protocols
  - Handling Insert and Delete Operations
  - Concurrency in Index Structures

Concept of a Transaction

- 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:
  1. **read**(*A*)
  2. $A := A - 50$
  3. **write**(*A*)
  4. **read**(*B*)
  5. $B := B + 50$
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent (=parallel) execution of multiple transactions

Requirements for Transactions

- **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
- **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

Requirements for Transactions

- **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

- **Isolation requirement**
  - if between steps 3 and 6, another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database

T1	T2
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*)	

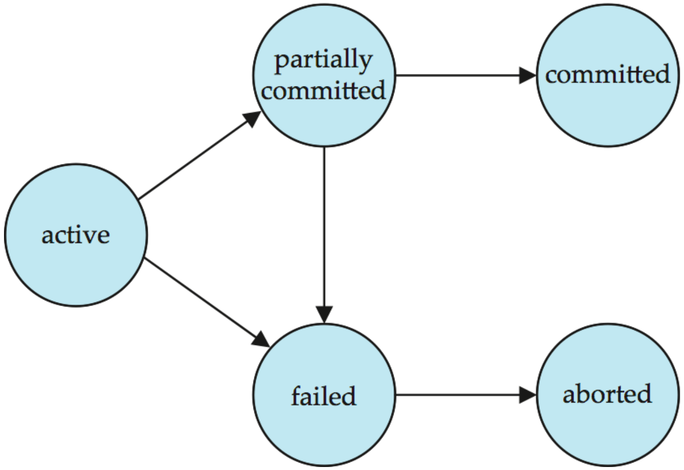
- Isolation can be ensured trivially by running transactions **serially**
  - i.e., one after the other
  - however, parallel execution is often desired due to performance benefits

ACID Properties

- **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
  - Intermediate transaction results must be hidden from other concurrently executed transactions
  - i.e., for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Transaction States

- **Active:** the initial state; transaction stays active while it is executing
- **Partially committed:** after the final statement has been executed
- **Failed:** after discovery that normal execution can no longer proceed
- **Aborted:** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Actions to be taken:
  - Restart the transaction (can be done only if no internal logical error)
  - Kill the transaction
- **Committed:** after successful completion



Concurrent Execution of Transactions

- 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

- **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** instructions as the last statement
  - By default, a transaction is assumed to execute 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*
- Serial schedule: T_1 is executed as a whole, followed by T_2 :

A = 200

A = 150

B = 400

B = 450

T_1	T_2
read(A)	
A = A – 50	
write(A)	
read(B)	
B = B + 50	
write(B)	
commit	
	read(A)
	A = A + 20
	write(A)
	read(B)
	B = B - 20
	write(B)
	commit

A = 150

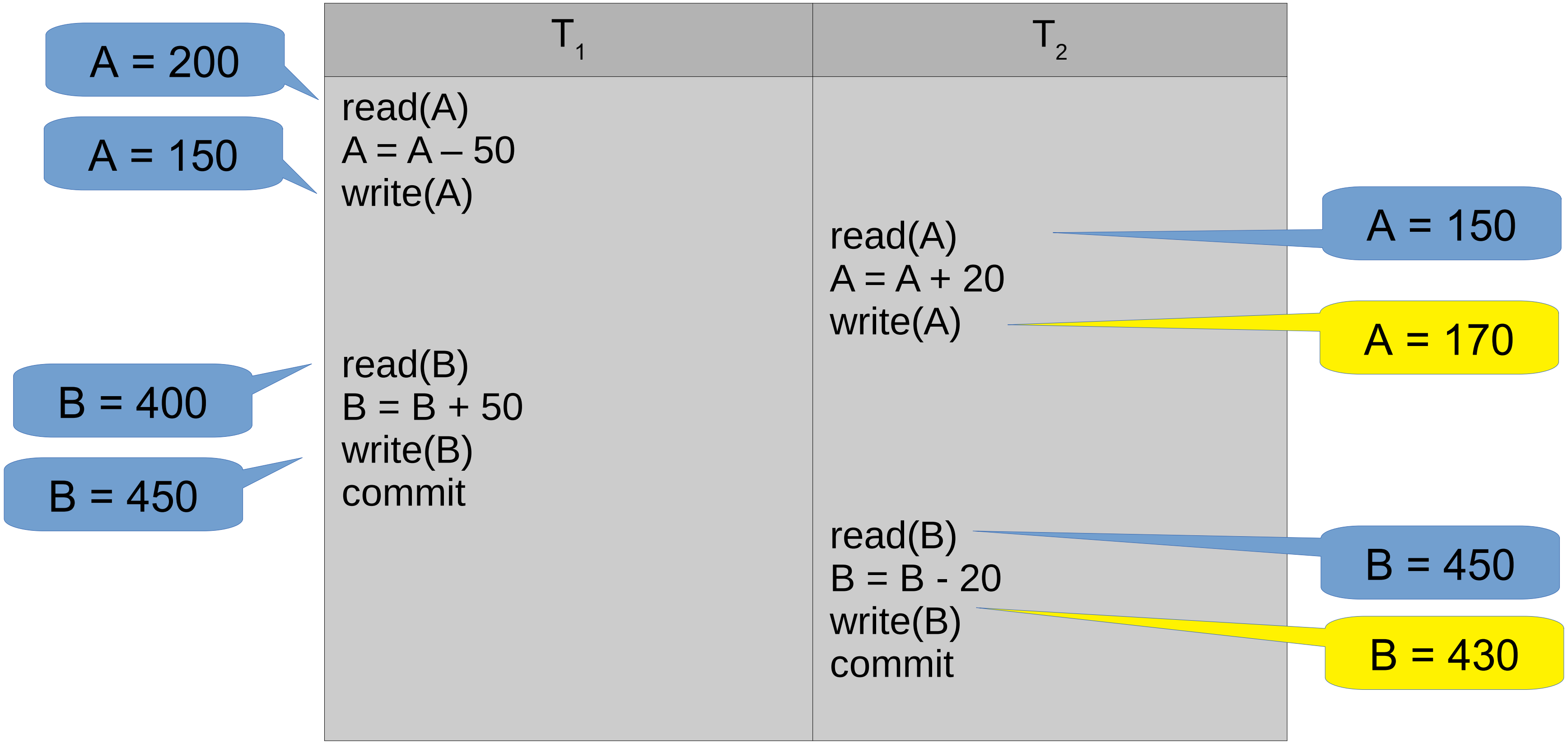
A = 170

B = 450

B = 430

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



T_1	T_2
read(A) A = A – 50 write(A)	
	read(A) A = A + 20 write(A)
read(B) B = B + 50 write(B) commit	
	read(B) B = B - 20 write(B) commit

A = 200

A = 150

A = 150

A = 170

B = 400

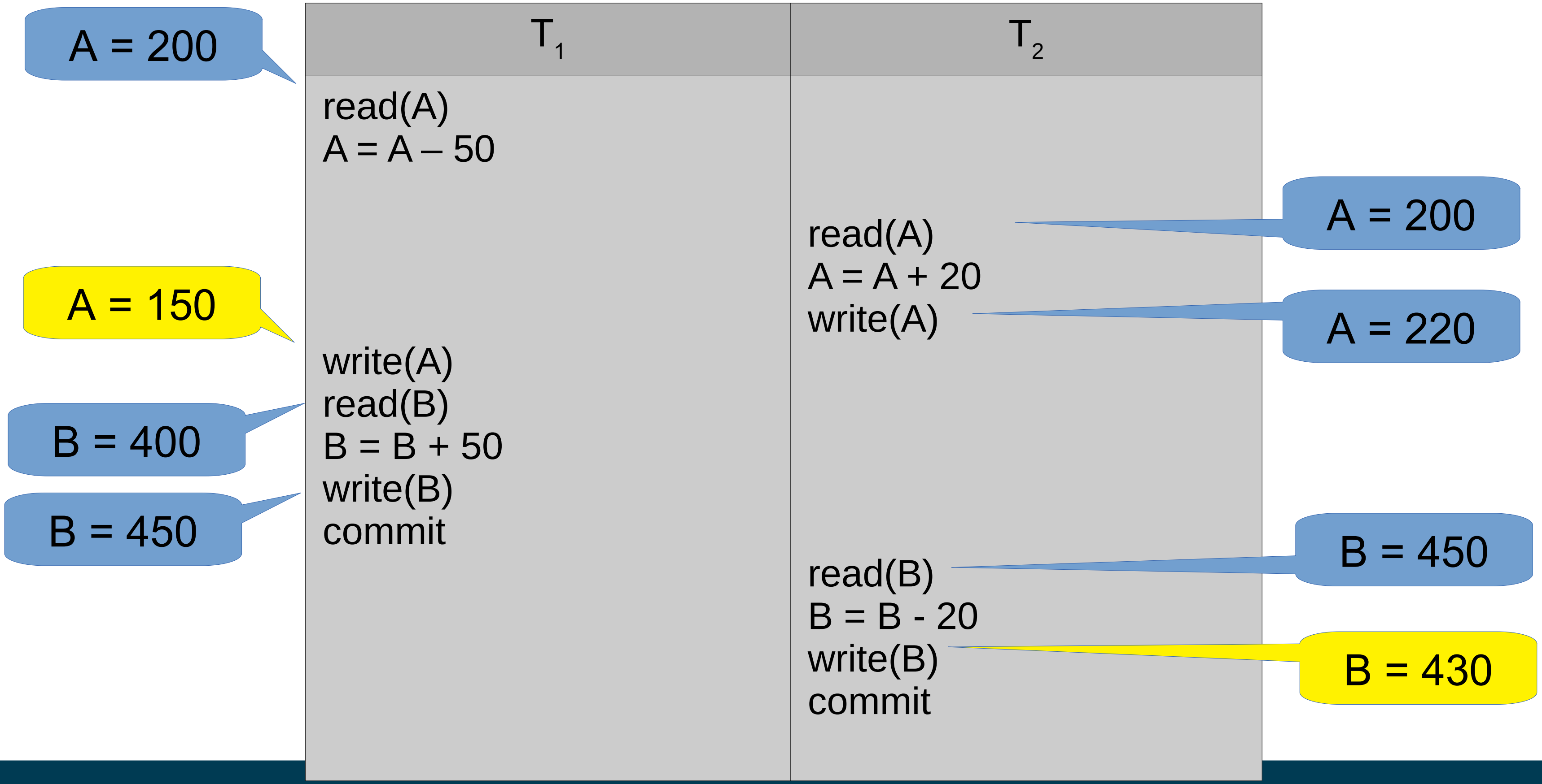
B = 450

B = 450

B = 430

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*
- The sum of A and B is not maintained!



T_1	T_2
read(A)	
A = A – 50	
	read(A)
	A = A + 20
	write(A)
write(A)	
read(B)	
B = B + 50	
write(B)	
commit	
	read(B)
	B = B - 20
	write(B)
	commit

Serializability

- 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 inbetween
  - Our simplified schedules consist of only read and write instructions

Conflicting Transactions

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively
- 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
  - 1. I_i = **read**(Q), I_j = **read**(Q). → No conflict
    2. I_i = **read**(Q), I_j = **write**(Q). → Conflict
    3. I_i = **write**(Q), I_j = **read**(Q). → Conflict
    4. I_i = **write**(Q), I_j = **write**(Q). → Conflict
    5. I_i = **write**(Q), I_j = **write**(R). → No conflict
    6. I_i = **read**(Q), I_j = ***write***(R). → No conflict
- 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 and Serializability

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

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

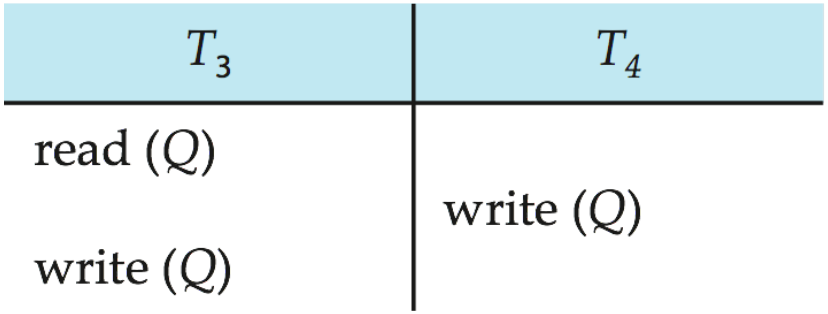
S

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

S’

Conflict Equivalence and Serializability

- Example of a schedule that is not conflict serializable:

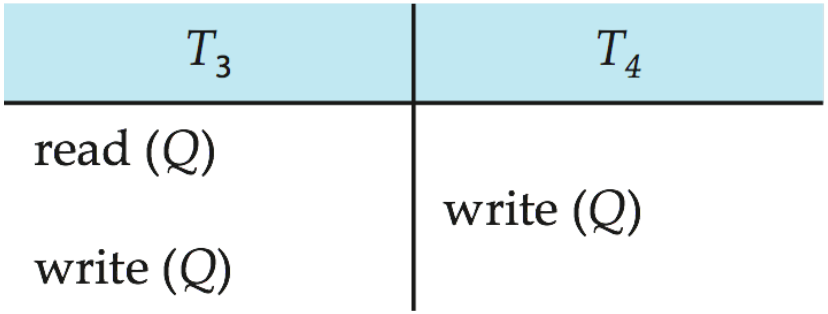


T_3	T_4
read (Q)	
	write (Q)
write (Q)	

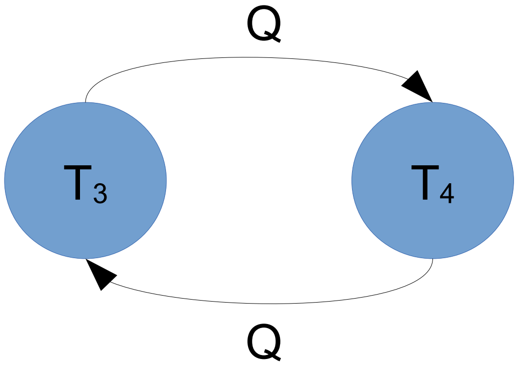
- write(Q) in T_4 conflicts both with read(Q) and with write(Q) in T_3
  - i.e., we are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >

Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph:** a directed graph where the vertices are the transactions (names)
  - We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier
  - We may label the arc by the item that was accessed
- Example:

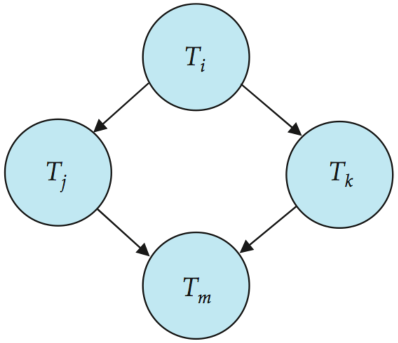


T_3	T_4
read (Q)	
	write (Q)
write (Q)	

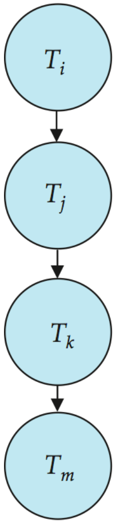


Testing for Conflict Serializability

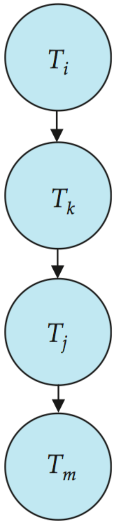
- A schedule is conflict serializable
  - if and only if its precedence graph is acyclic
  - serializability order can be obtained by a topological sorting of the graph
    - i.e., a linear order consistent with the partial order of the graph
    - Example: both (b) and (c) are possible partial orders of (a)
- Cycle-detection algorithms in O(n²) 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)



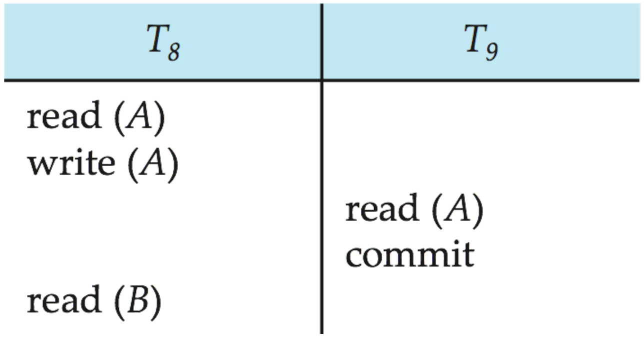
(b)



(c)

Recoverable Schedules

- Consider the following schedule:

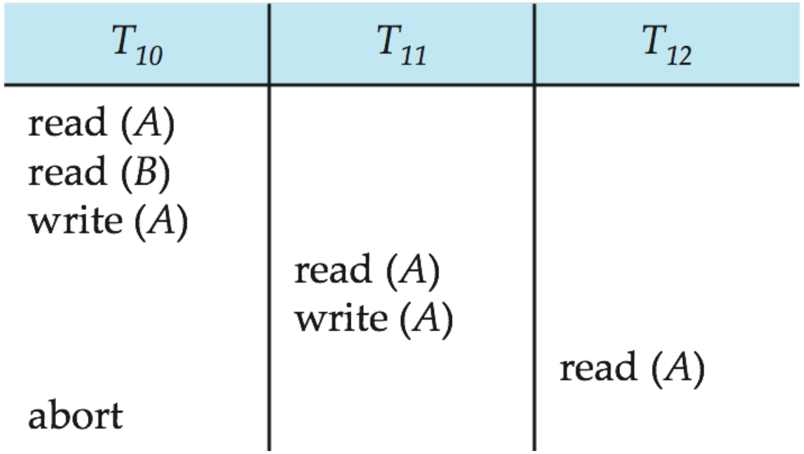


T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- What happens if T_8 should abort after T_9 commits?
  - T_9 would have read (and possibly shown to the user) an inconsistent database state
  - The DBMS should avoid those cases
- A schedule is *recoverable* if the following holds:
  - if a transaction T_j 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

Cascading Rollbacks

- Consider the following schedule:



T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

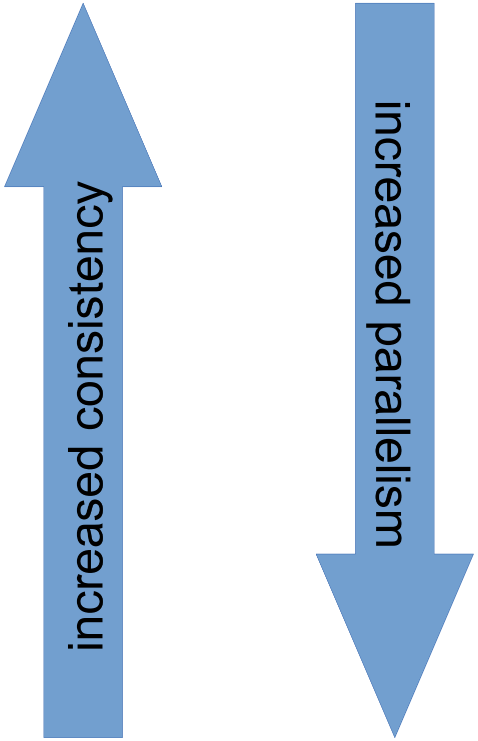
- On the abort of T_{10}
  - all three transactions need to be rolled back
  - can mean undoing a significant amount of work

Cascadeless Schedules

- 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
- Every cascadeless schedule is also recoverable
  - the reverse need not hold
- It is desirable to restrict the schedules to those that are cascadeless

Levels of Consistency

- **Serializable:** default
- **Repeatable read:**
  - only committed records to be read
  - successive reads of same record must return the same value
  - transactions may not be serializable
- **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

- 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

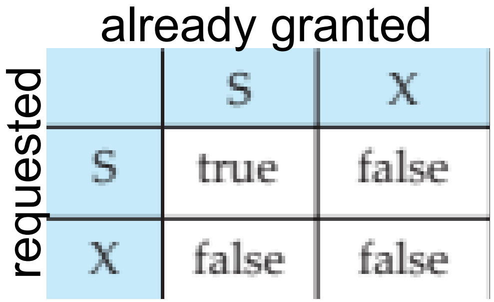
- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules
  - but provides a poor degree of parallelism
- Concurrency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

Locks

- 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

- 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



already granted

requested

	S	X
S	true	false
X	false	false

Lock-based Protocols

- Example of two transactions performing locking:

T_1:	T_2:
lock-S(*A*);	**lock-S**(*A*);
read(*A*);	**lock-S**(*B*);
unlock(*A*);	**read**(*A*);
lock-S(*B*);	**read**(*B*);
read(*B*);	**display**(*A*+*B*);
unlock(*B*);	**unlock**(*A*);
display(*A*+*B*);	**unlock**(*B*);

- 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

- 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

- 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

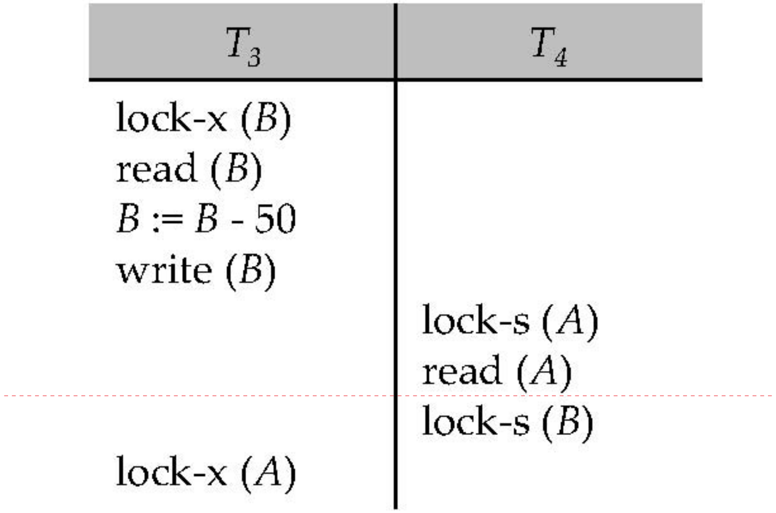
- A transaction T_i issues the standard read/write instruction, without explicit locking calls
- The operation **write*(D)*** is processed by the DBMS as:

```
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

Deadlocks

- Consider the partial schedule



T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress
  - executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on ,
  - 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 & Starvation

- Two-phase locking protocol
  - guarantees *serializability*
  - does *not* ensure freedom from deadlocks
- 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
  - The same transaction is repeatedly rolled back due to deadlocks
  - Concurrency control manager can be designed to prevent starvation

Deadlocks

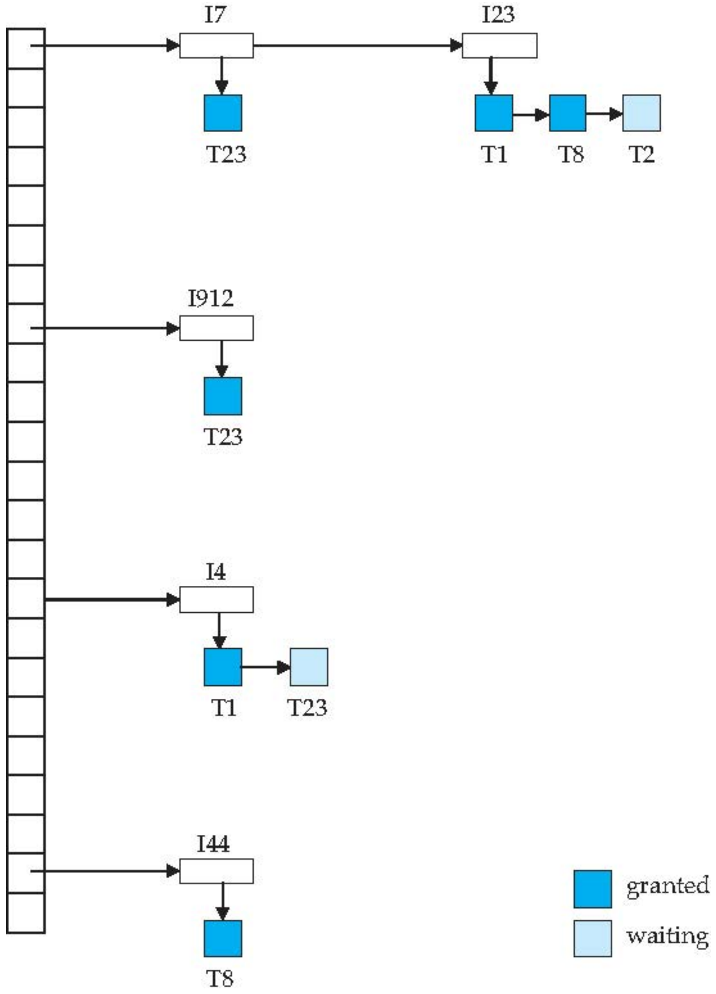
- The potential for deadlock exists in most locking protocols
  - but there are prevention mechanisms (see later)
- When a deadlock occurs
  - rollbacks are necessary
  - there is a possibility of cascading roll-backs
    - but cascading rollbacks can be expensive
- Cascading roll-back is possible under two-phase locking
- Modified protocol called **strict two-phase locking**
  - a transaction must hold all its exclusive locks until it commits/aborts
  - avoids cascading rollbacks

Implementation of Locking

- 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 requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
  - The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table

- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
  - Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item
  - granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted
  - later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep an index of locks held by each transaction, to implement this efficiently



Deadlock Prevention

- System is deadlocked:
  - there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- ***Deadlock prevention*** protocols
  - ensure that the system will *never* enter into a deadlock state
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order

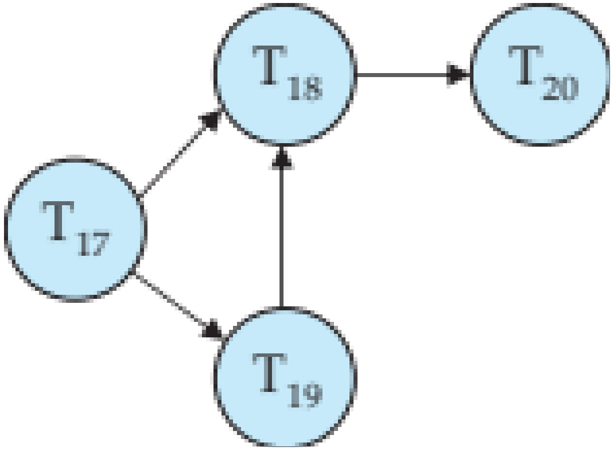
Deadlock Prevention

- **timeout-based schemes**
  - transactions wait for a lock only for a specified amount of time
    - if the lock has not been granted within that time → roll back
  - simple to implement; but starvation is possible
  - also difficult to determine good value of the timeout interval
- **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 before acquiring needed data item
- **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

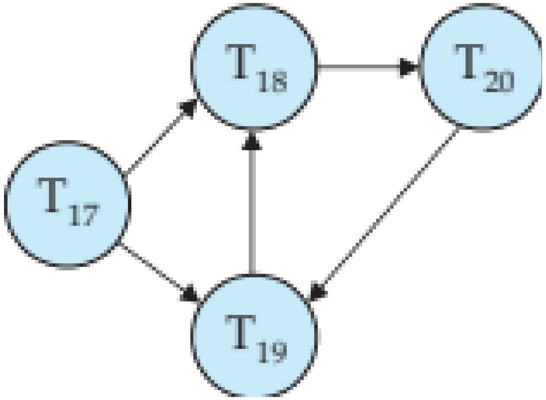
Deadlock Detection

- Deadlocks can be detected using a *wait-for graph*, which consists of a pair $G = (V,E)$
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
  - Edge from T_i to T_j implies that T_i is waiting for T_j to release a data item
- T_i requests a lock on a data item currently being locked by T_j,
  - the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph
- 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 ↔ the wait-for graph has a cycle
  - invoke a deadlock-detection algorithm periodically to look for cycles

Deadlock Detection



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - some transaction will have to rolled back (made a victim)
  - select that transaction as 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

Timestamp-based Scheduling

- Each transaction is issued a timestamp 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-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully

Timestamp-based Scheduling

- Transaction T_i issues a **read**(Q)
  - if TS(T_i) $>$ **W**-timestamp(Q)
    - execute **read** operation, set R-timestamp(Q) to **max**(R-timestamp(Q),TS(T_i))
  - if TS(T_i) $\leq$ **W**-timestamp(Q),
    then T_i needs to read a value of Q that was already overwritten

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

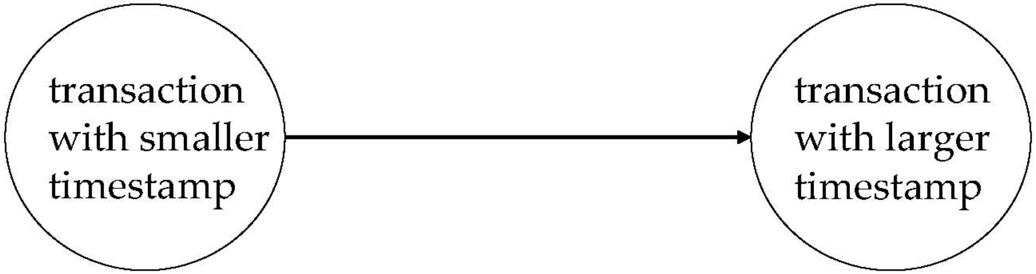
    → reject **write**, rollback T_i
  - if TS(T_i) $<$ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q

    → reject **write**, rollback T_i
  - Otherwise, execute **write** and set W-timestamp(Q) to TS(T_i)

Thomas Write Rule: we can also simply ignore this **write**

Timestamp-based Scheduling

- 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

- 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

- 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

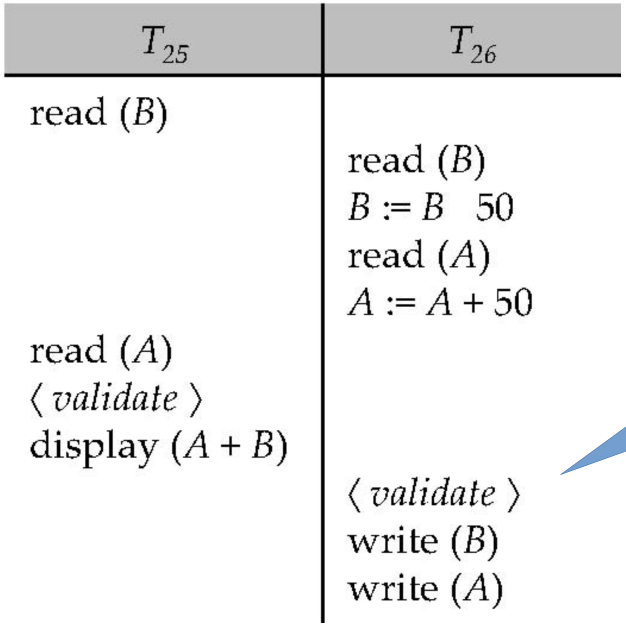
Validation Test for Transaction T_j

- 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

Validation Test for Transaction T_j

- Example schedule using validation:



T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

T_{25} has not written anything read by T_{26}

Insert and Delete Operations

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted
  - A transaction that **insert**s a new tuple into the database is given an exclusive lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**
- A transaction that scans a relation

  (e.g., read number of all accounts in Perryridge)

  and a transaction that inserts a tuple in the relation

  (e.g., insert a new account at Perryridge)

  (conceptually) conflict in spite of not accessing any tuple in common

Insert and Delete Operations

- The transaction scanning the relation is reading information that indicates what tuples the relation contains
  - while a transaction inserting a tuple updates the same information
- The conflict should be detected, e.g., by locking the information
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains
  - Transactions scanning the relation acquire a shared lock in the data item
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.
(Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions
  - Index locking protocols provide higher concurrency while preventing the phantom phenomenon
  - requiring locks on certain index buckets

Index Locking Protocol

- Index locking protocol
  - Every relation must have at least one index
  - A transaction can access tuples only after finding them through one or more indices on the relation
- A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
  - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
- A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
  - must update all indices to r
  - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
- The rules of the two-phase locking protocol must be observed
  - Guarantees that phantom phenomenon does not occur

Concurrency in Index Structures

- Indices are unlike other database items
  - their only job is to help in accessing the actual data
- Index structures are typically accessed very often
  - much more than other database items
  - Treating index-structures like other database items,
    e.g. by 2-phase locking of index nodes can lead to low concurrency
- Special protocols for index structures
  - e.g., locks on internal nodes are released early, instead of two-phase fashion
  - it is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained
  - in particular, the exact values read in an internal node of a
    B^+-tree are irrelevant so long as we end up in the correct leaf node

Concurrency in Index Structures

- Example of index concurrency protocol:Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows
- During search/insertion/deletion:
  - First lock the root node in shared mode
  - After locking all required children of a node in shared mode, release the lock on the parent node
- During insertion/deletion
  - upgrade leaf node locks to exclusive mode
- When splitting or coalescing requires changes to a parent
  - lock the parent in exclusive mode
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
  - Better protocols are available; e.g., the B-link tree protocol
    - Intuition: release lock on parent before acquiring lock on child

Summary

- 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
  - for “normal” data
  - for indices

Questions?