Database Development and Design (CPT201)

Lecture 6a: Transaction Management – Concepts of Transaction

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Transaction Concept: Atomicity, Consistency, Isolation, Durability
- Concurrent Executions and Schedule
  - Serialisability
  - Recoverability
  - Testing for Serialisability

Transaction Concept

- A transaction is a *unit* of program execution that accesses and possibly updates various data items.
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully (is committed), the database must be consistent.
  - After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Two main issues to deal with:
  - Concurrent execution of multiple transactions
  - Recovery from failures of various kinds, such as hardware failures and system crashes

ACID Properties

- Atomicity: Either all operations of the transaction are properly reflected in the database or none are.
- Consistency: Execution of a transaction in isolation 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.
  - That is, 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.

Example of Fund Transfer

- The following is a 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*)

Example of Fund Transfer cont'd

- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else inconsistency will result.
- Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.

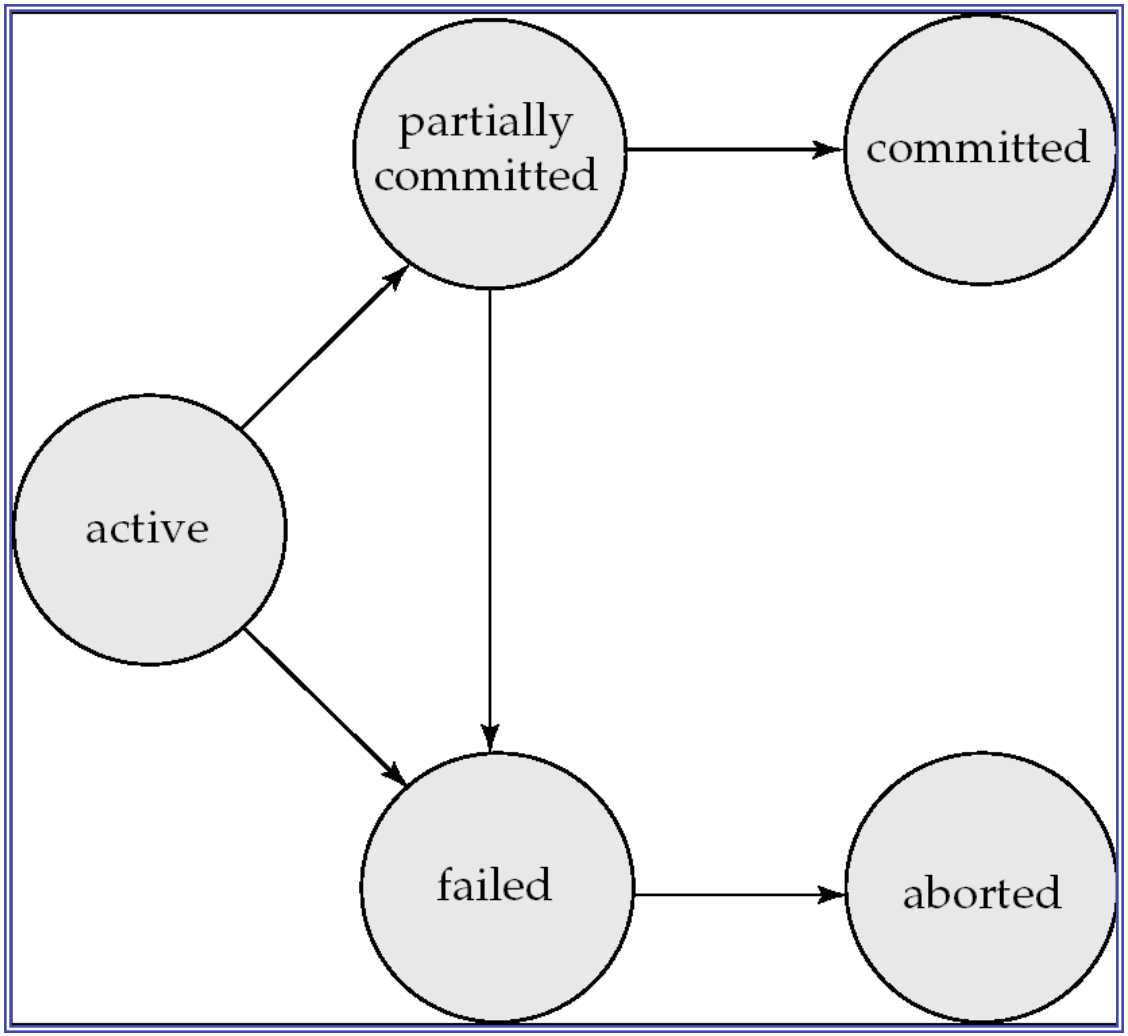
1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Example of Fund Transfer cont'd

- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
- 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 despite failures.

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Transaction States



Transaction State

- Active – the initial state; the transaction stays in this state while it is executing
- Partially committed – after the final statement has been executed.
- Failed –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.
- Committed – after successful completion.

Concurrent Executions

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

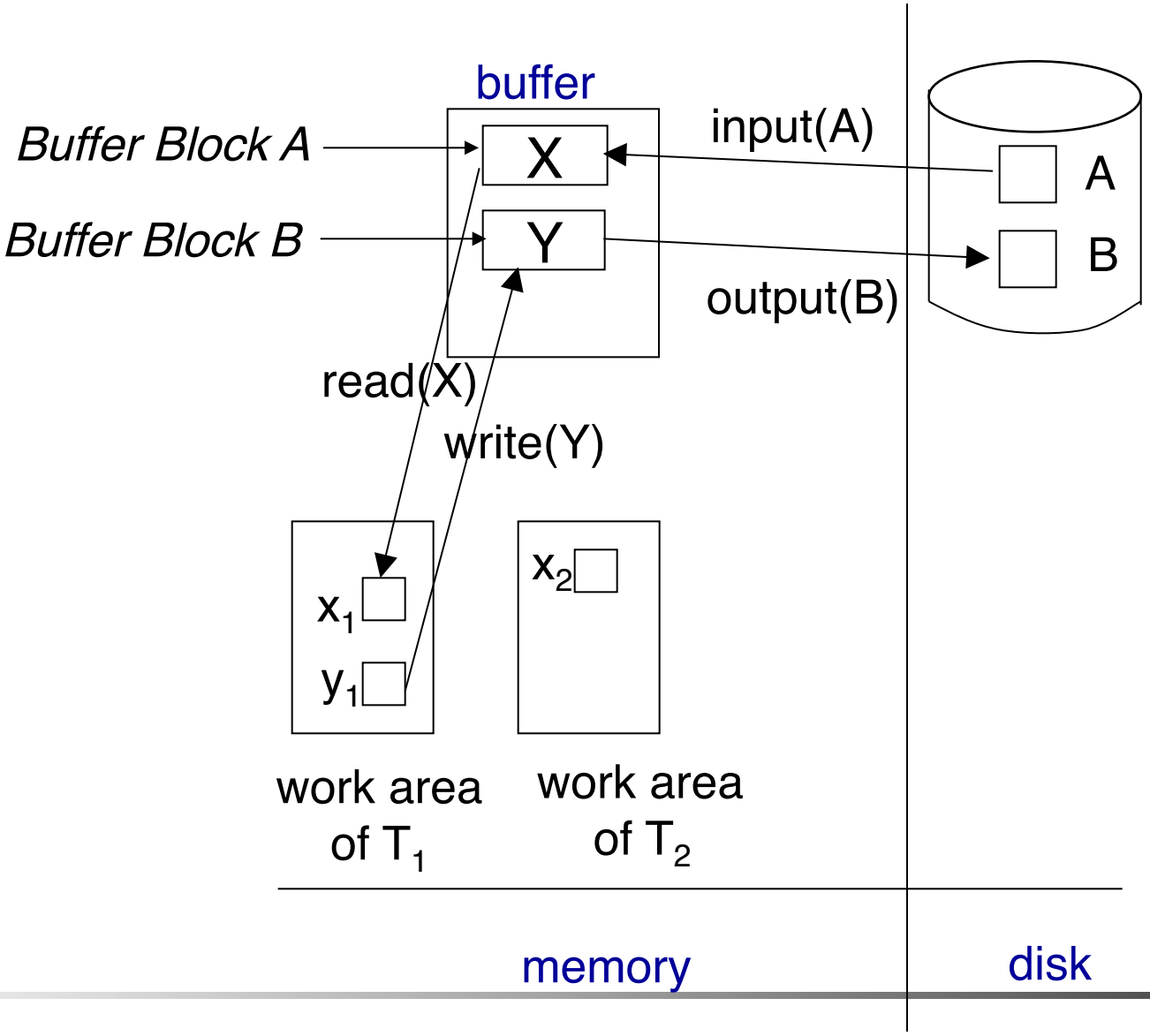
Schedules

- Schedule – a sequence of instructions that specifies the chronological order in which 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 (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement (will be omitted if it is obvious)

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input(*B*)** transfers the physical block *B* to main memory (buffer).
  - **output(*B*)** transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Example of Schedules

Let A=1,000, B=2,000. Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

2

- 1 and 2 are serial schedules;
- How much is A, B, and A+B?

Example of Schedules cont'd

Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

3

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

4

- 3 and 4 are not serial schedules;
- Does 3 preserve the value of (A+B)? How much is A and B?
- Does 4 preserve the value of (A+B)? How much is A and B?

Serialisability

- Basic Assumption – Each transaction preserves database consistency.
- A (possibly concurrent) schedule is serialisable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  - conflict serialisability
  - view serialisability
- We ignore operations other than read and write instructions for now for simplicity.
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_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 is write Q.
- Three conflicts can be identified
  - write-read (WR) conflict: reading uncommitted data
  - read-write (RW) conflict: unrepeatable reads
  - write-write (WW) conflict: overwriting uncommitted data
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule (the order of the two instructions can be swapped).

Write-Read (WR) Conflicts

- Transaction T2 reads a database object that has been modified by transaction T1 which has not committed ("dirty read")

T1	T2
read(A) A:= A – 50 write(A)	
	read(A) A:= A + A*10% write(A) read(B) B:= B + B*10% write(B) commit
read(B) B:= B + 50 write(B) Rollback	

Read-Write (RW) Conflicts

- Transaction T2 could change the value of an object that has been read by a transaction T1, while T1 is still in progress. (unrepeatable read)
- "unrepeatable read" example

T1	T2
read(A)	
	read(A)
	write(A)
	commit
read(A)	
write(A)	
commit	

T1	T2
read(A)	
	read(A)
	A:=A-1
	write(A)
	commit
read(A)	
A:=A-1	
write(A)	
commit	

Let A=1 and assume that there is an integrity constraint that prevents A from becoming negative. T1 will get an error.

Write-Write (WW) Conflicts

- Transaction T2 could overwrite the value of an object which has already been modified by T1, while T1 is still in progress. (Blind Write)
- "Blind Write" example
  - T1: set both Steven and Paul salaries at USD 1m.
  - T2: set both Steven and Paul salaries at RMB 1m.
- With this schedule, Steven has salary 1m RMB and Paul has salary 1m USD.

T1	T2
write(*Steven*)	
	write(*Paul*)
	write(*Steven*)
	commit
write(*Paul*)	
commit	

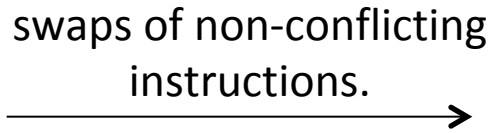
Conflict Serialisability

- 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. A schedule S is conflict serialisable 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)

3

swaps of non-conflicting instructions.



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

6

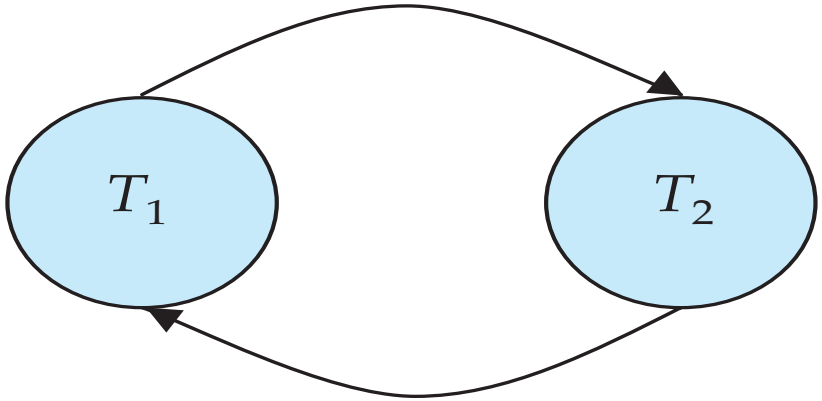
Conflict Serialisability cont'd

- Example of a schedule that is not conflict serialisable
- We are unable to swap instructions in the schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

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

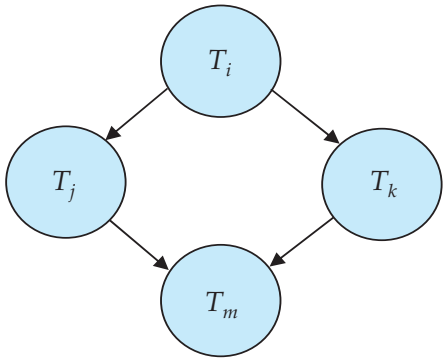
Testing for Serialisability

- Consider a schedule with a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a directed graph where
  - vertices are the transactions (names).
  - an arc (edge) is drawn from T_i to T_j if the two transactions conflict, and T_i accesses the same data item before T_j does.

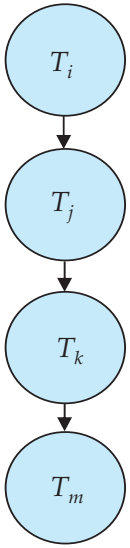


Test for Conflict Serialisability

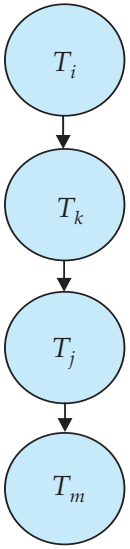
- A schedule is conflict serialisable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serialisability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - Could have more than one serialisability orders
- It is possible for two schedules to produce the same outcome, but are not conflict serialisable.



(a)



(b)



(c)

View Serialisability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  - If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  - If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  - The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serialisability cont'd

- A schedule *S* is **view serialisable** if it is view equivalent to a serial schedule.
- Every conflict serialisable schedule is also view serialisable but not vice versa.
  - Below is a schedule which is view-serialisable but *not* conflict serialisable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
  - Answer: <T27, T28, T29>
- Every view serialisable schedule that is not conflict serialisable has **blind writes**.

Test for View Serialisability

- The precedence graph test for conflict serialisability cannot be used directly to test for view serialisability.
  - Extension to test for view serialisability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serialisable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some sufficient conditions for view serialisability can still be used.
- It is not used in practice due to its high degree of computational complexity

Recoverable Schedules

- Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable as T_7 commits immediately after the read

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- If T_6 should abort, T_7 would have read (and possibly shown to the user) an inconsistent database state.
- Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed.
- If T_{10} fails (aborted later and needs to roll back), T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

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

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; 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.
- It is desirable to restrict the schedules to those that are cascadeless.

End of Lecture

- Summary
  - Transactions and the ACID properties
  - Schedules and conflict serialisable schedules
  - Potential anomalies with interleaving

- Reading
  - Textbook 6th edition, chapter 14.1, 14.2, 14.3, 14.4, 14.5, 14.6 and 14.7
  - Textbook 7th edition, chapter 17.1, 17.2, 17.3, 17.4, 17.5, 17.6 and 17.7