CS322: Database Systems

Transactions

Dr. Manas Khatua
Assistant Professor
Dept. of CSE
IIT Jodhpur
E-mail: manaskhatua@iitj.ac.in

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
  - Conflict Serializability
  - View Serializability
- Recoverability
- Implementation of Isolation
- Testing for Serializability.

Transaction Concept

- 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 execution of multiple transactions

Required Properties of a Transaction

- 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)*

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system 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.

Cont…

- **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 (Referential Integrity)
    - e.g., primary keys and foreign keys
  - **Implicit** integrity constraints (Data Integrity)
    - 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 database must be consistent

Cont…

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** 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).

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**
  - That is, one after the other.

- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

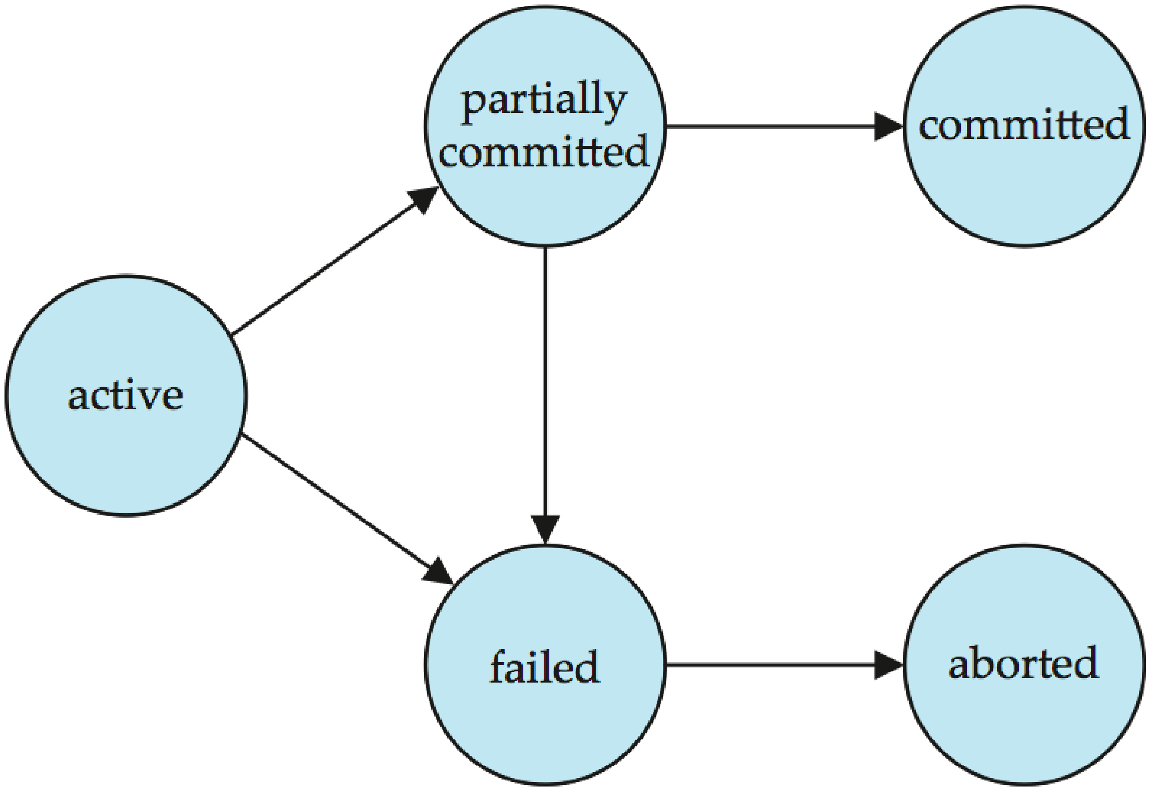
- A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **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.

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** -- after the 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. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Cont…



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **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: 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
    - Will study it after studying the notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify 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 transaction 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 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
-
- An example of a **serial** schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously.
- The following schedule is **not a serial** schedule, but it is **equivalent** to Schedule 1.

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) commit	
	read (B) $B := B + temp$ write (B) commit

Note -- In schedules 1, 2 and 3, the sum “A + B” is preserved.

Schedule 4

- The following concurrent schedule does not preserve the sum of "$A + B$"

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption**: Each transaction preserves database **consistency**.

- Thus, serial execution of a set of transactions preserves database consistency.

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflict 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

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.
- Therefore, Schedule 3 is conflict serializable.

Schedule 3

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

Schedule 6

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

Cont…

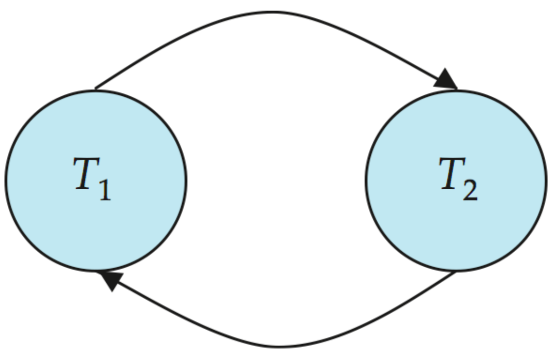
- Example of a schedule that is not conflict serializable:

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

- 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 >.

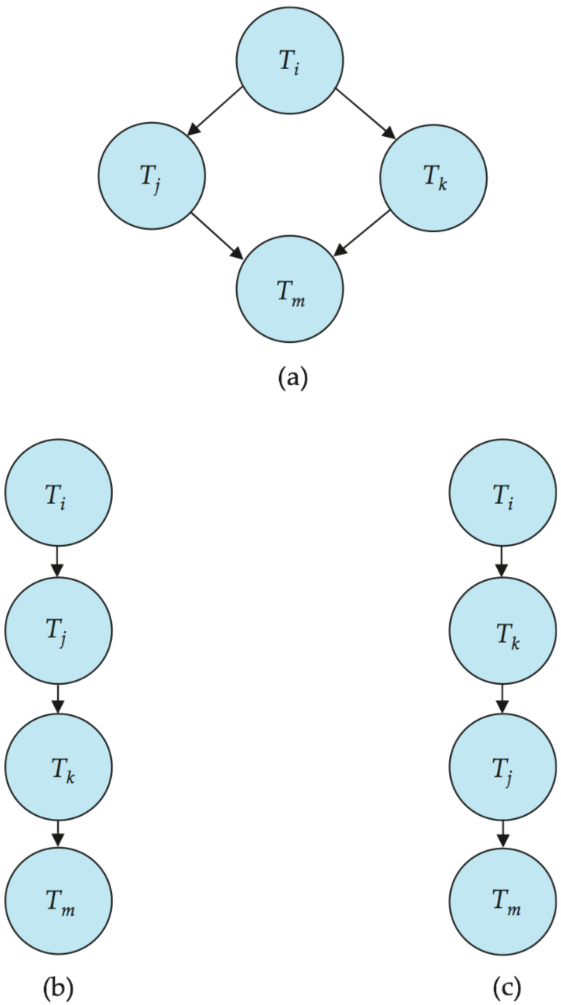
Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct 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**



Testing for Conflict Serializability

- A schedule is conflict serializable 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 serializability order can be obtained by a *topological sorting* of the graph.
  - That is, a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



Recoverable Schedules

- **Recoverable schedule** — if a transaction T_k 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_k.

- Example:

this schedule is **not recoverable** if T_9 commits immediately after the read(A) operation.

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

Recoverable Schedule

T1	T2
• Read(x)	
• Write(x)	
	• Read(y)
• Commit	
	• Read(x)

- If T_8 should abort, T_9 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 (so the schedule is recoverable)

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

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — for each pair of transactions T_i and T_k such that T_k reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_k.
- Every cascadeless schedule is also recoverable; but not the vice-versa.
- It is **desirable** to restrict the schedules to those that are cascadeless
- Example

NOT cascadeless schedule

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

cascadeless schedule

T1	T2
• Read(x)	
• Write(x)	
	• Read(y)
• Commit	
	• Read(x)

Strict Schedule

- take a scenario of a cascadeless schedule

T1	T2
	• Read(x)
• Read(x)	
• Write (x)	
	• Write(x)
• Commit	
	• Commit

- In this, the Write(x) of the transaction T2 overwrites the previous value written by T1 , and hence overwrite conflicts arise .
- This problem is taken care in Strict Schedule.

- **Strict Schedule** is a schedule in which a transaction can neither Read(x) nor Write(x) until the last transaction that wrote x has committed or aborted.

Concurrency Control

- A database **must** provide a mechanism that will **ensure** that all possible schedules are both:
  - Conflict serializable,
  - Recoverable, and
  - Preferably cascadeless & strict
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little **too late**!
- Tests for serializability help us to understand why a concurrency control protocol is correct.
- **Goal**: to develop concurrency control protocols that will assure serializability.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

View Serializability

- 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,

  1. 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.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_k (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_k .
  3. 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'.

- In loose sense, both schedules **"view"** the same data values.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone**.**

Cont…

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.

- Every conflict serializable schedule is also view serializable; But not the vice-versa
- Example: below schedule is view-serializable but *not* conflict serializable.

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

→ Serial Schedule

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

- Both the schedules **"view"** the same data values at the end.

- Every view serializable schedule that is not conflict serializable has **blind writes.**

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable 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 serializability can still be used.

Thanks!