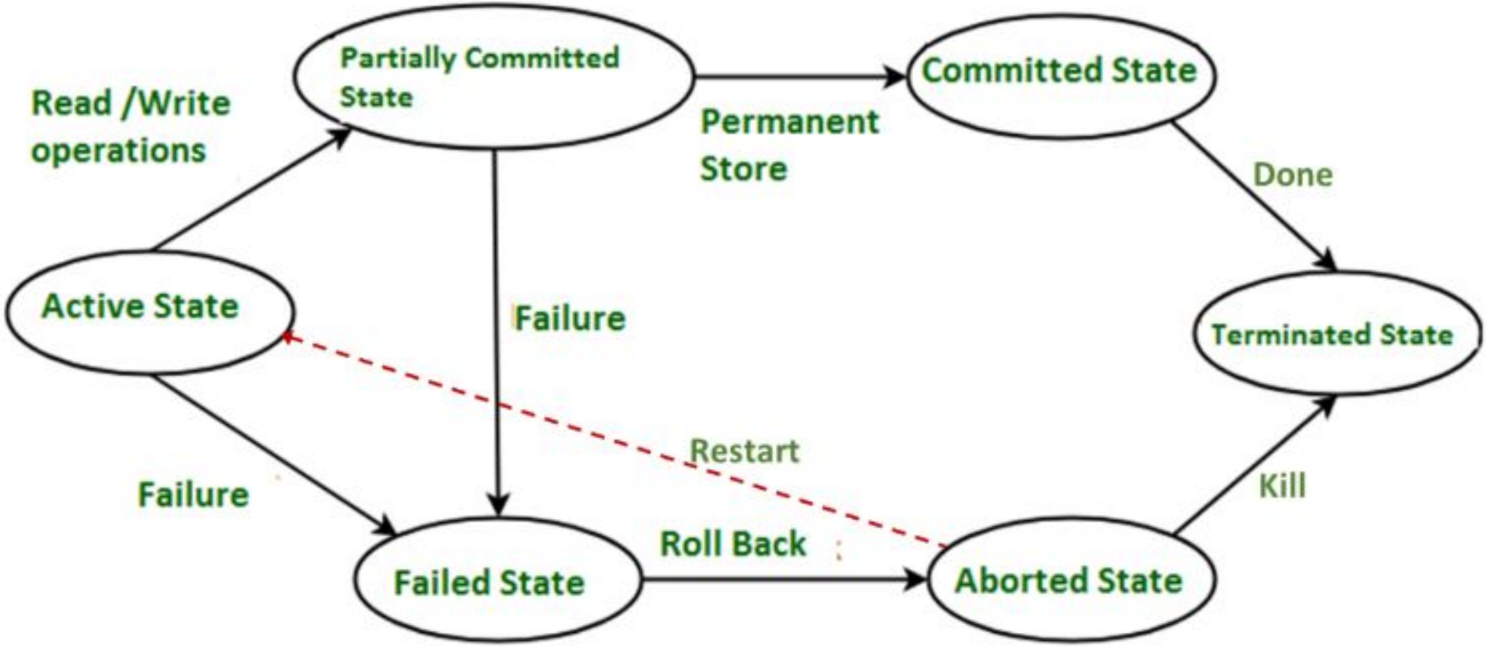
WEEK 10 SLIDES

DBMS

- A **transaction** is a *unit* of program execution that accesses and, possibly updates, various data items
- For example, 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

A **transaction** is a unit of program execution that accesses and possibly updates various data items:

- **Atomicity:** Atomicity guarantees that each transaction is treated as a single *unit*, which either succeeds completely, or fails completely
  - If any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged
  - Atomicity must be guaranteed in every situation, including power failures, errors and crashes
- **Consistency:** Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants
  - Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof
- **Isolation:** Transactions are often executed concurrently (multiple transactions reading and writing to a table at the same time)
  - Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
- **Durability:** Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (like power outage or crash)
  - This usually means that completed transactions (or their effects) are recorded in non-volatile memory



Partially Committed State
Committed State
Read /Write operations
Permanent Store
Done
Active State
Failure
Terminated State
Restart
Failure
Kill
Roll Back
Failed State
Aborted State

- **Schedule**: A sequence 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

- 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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write A	
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

- 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

Schedule 3

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

Schedule 1

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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@ Commit
45	255	300	T2, write B	@Commit

Consistent @ Commit

Inconsistent @ Transit

Inconsistent @ Commit

Note – In schedules 1, 2 and 3, the sum "$A + B$" is preserved

- **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:
  a) **Conflict Serializability**
  b) **View Serializability**

Example of Serializable Schedule

- 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

Schedule 3

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

Schedule 1

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

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@ Commit
45	255	300	T2, write B	@Commit

Consistent @ Commit
Inconsistent @ Transit
Inconsistent @ Commit

Note: In schedules 1, 2 and 3, the sum "$A + B$" is preserved

- Let I_i and I_j be two Instructions from transactions T_i and T_j respectively
- Instructions I_i and I_j conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions write to Q
  a) $I_i = \textbf{read}(Q)$, $I_j = \textbf{read}(Q)$. I_i and I_j don't conflict
  b) $I_i = \textbf{read}(Q)$, $I_j = \textbf{write}(Q)$. They conflict
  c) $I_i = \textbf{write}(Q)$, $I_j = \textbf{read}(Q)$. They conflict
  d) $I_i = \textbf{write}(Q)$, $I_j = \textbf{write}(Q)$. They conflict
- 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

- 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:
  - Swap T1.read(B) and T2.write(A)
  - Swap T1.read(B) and T2.read(A)
  - Swap T1.write(B) and T2.write(A)
  - Swap T1.write(B) and T2.read(A)

These swaps do not conflict as they work with different items (A or B) in different transactions

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

Schedule 3

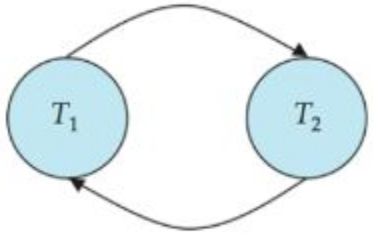
T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5

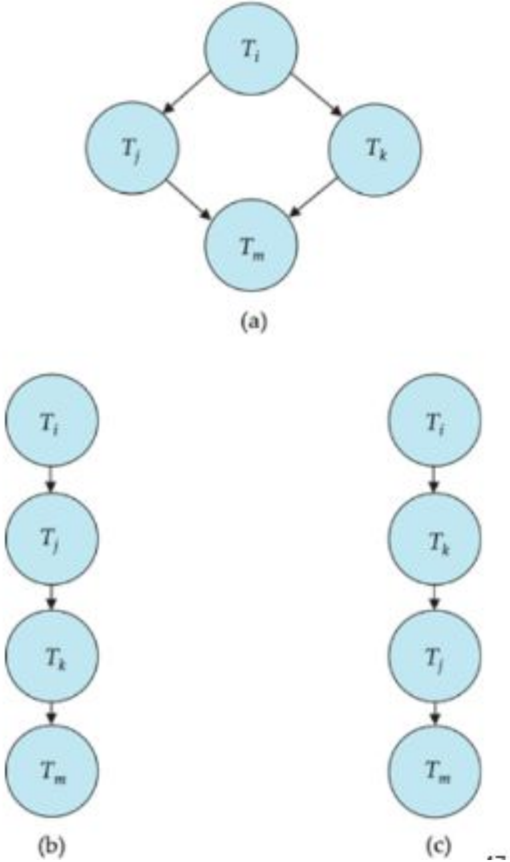
T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

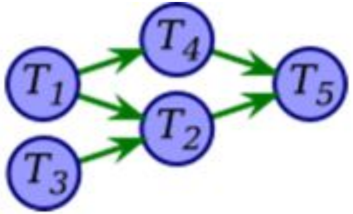
- Consider some schedule of a set of transactions $T_1, T_2, \cdots, T_n$
- **Precedence Graph**
    - A direct graph where the vertices are the transactions (names)
- We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier
-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



- *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)



- Consider the following schedule:
  - $w_1(A), r_2(A), w_1(B), w_3(C), r_2(C), r_4(B), w_2(D), w_4(E), r_5(D), w_5(E)$
- We start with an empty graph with five vertices labeled T_1, T_2, T_3, T_4, T_5.



- We go through each operation in the schedule:
  $w_1(A)$: A is subsequently read by T_2, so add edge $T_1 \to T_2$
  $r_2(A)$: no subsequent writes to A, so no new edges
  $w_1(B)$: B is subsequently read by T_4, so add edge $T_1 \to T_4$
  $w_3(C)$: C is subsequently read by T_2, so add edge $T_3 \to T_2$
  $r_2(C)$: no subsequent writes to C, so no new edges
  $r_4(B)$: no subsequent writes to B, so no new edges
  $w_2(D)$: C is subsequently read by T_2, so add edge $T_3 \to T_2$
  $w_4(E)$: E is subsequently written by T_5, so add edge $T_4 \to T_5$
  $r_5(D)$: no subsequent writes to D, so no new edges
  $w_5(E)$: no subsequent operations on E, so no new edges
- We end up with precedence graph
- This graph has no cycles, so the original schedule must be serializable. Moreover, since one way to topologically sort the graph is $T_3 - T_1 - T_4 - T_2 - T_5$, one serial schedule that is conflict-equivalent is
  - $w_3(C), w_1(A), w_1(B), r_4(B), w_4(E), r_2(A), r_2(C), w_2(D), r_5(D), w_5(E)$

- Serializability helps to ensure Isolation and Consistency of a schedule
- Yet, the Atomicity and Consistency may be compromised in the face of system failures
- Consider a schedule comprising a single transaction (obviously serial):
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)
  7. **commit** // Make the changes permanent; show the results to the user
- What if system fails after Step 3 and before Step 6?
  - Leads to inconsistent state
  - Need to rollback update of A
- This is known as **Recovery**

- 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.
- The following 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)	

- 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 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

Example of Recoverable Schedule with Cascading Rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
Failure Point				
Commit;				
		Commit;		

Rollback is possible as T2 has not committed yet. But T2 also need to be rolled back for rolling back T1.

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,
  - **Initial Read**: 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
  - **Write-Read Pair**: 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
  - **Final Write**: 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

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

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

- What serial schedule is above equivalent to?
  - $T_{27} - T_{28} - T_{29}$
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T_{29} performs the final write of Q in both schedules
- T_{28} and T_{29} perform write(Q) operations called **blind writes**, without having performed a read(Q) operation
- *Every view serializable schedule that is not conflict serializable has* **blind writes**

Lock-Based Protocols

- *A lock is a mechanism to control concurrent access to a data item*
- Data items can be locked in two modes:
  a) *exclusive (X)* mode:
     - Data item can be *both read as well as written*
     - **X-lock** is requested using **lock-X** instruction

  b) *shared (S)* mode:
     - Data item can *only be read*
     - **S-lock** is requested using **lock-S** instruction
- A transaction can unlock a data item Q by the **unlock**(Q) Instruction
- Lock requests are made to the concurrency-control manager by the programmer
- *Transaction can proceed only after request is granted*

State of the lock	Lock request type: Shared	Lock request type: Exclusive
Shared	Yes	No
Exclusive	No	No

Lock-Based Protocols: Example: Serial Schedule

- Let A and B be two accounts that are accessed by transactions T_1 and T_2.
  - Transaction T_1 transfers \$50 from account B to account A
  - Transaction T_2 displays the total amount of money in accounts A and B, that is, the sum $A + B$
- Suppose that the values of accounts A and B are \$100 and \$200, respectively
- If these transactions are executed serially, either as T_1, T_2 or the order T_2, T_1 then transaction T_2 will display the value \$300

T1:

```
lock-X(B);
read(B);
B := B – 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A);
```

T2:

```
lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display(A + B)
```

Lock-Based Protocols: Example (4): Concurrent Schedule: Deadlock

- Given, T_3 and T_4, consider Schedule 2 (partial)
- Since T_3 is holding an exclusive mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B
- Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**
- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked.
- These data items are then available to the other transaction, which can continue with its execution.

$T3$:

```
lock-X(B);
read(B);
B := B − 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A)
```

$T4$:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
display(A + B);
unlock(A);
unlock(B)
```

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)	

Schedule 2

Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release 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**
  - That is, the point where a transaction acquired its final lock

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - ▷ can acquire a lock-S on item
    - ▷ can acquire a lock-X on item
    - ▷ can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - ▷ can release a lock-S
    - ▷ can release a lock-X
    - ▷ can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions

10. Consider the following schedule **S** with three transactions T1, T2 and T3: [NAT:1 points]

$$\mathbf{S}\text{: } R2(B); R1(B); R1(A); W1(A); R3(C); W3(C);$$

The number of serial schedule for given schedule **S** is....

11. Consider the following schedule **S** with four transactions T1, T2, T3,T4: [Subendu:MCQ:2 points]

$$\textbf{S}: R3(A); W2(A); R1(A); W1(A); R3(B); W4(B);$$

Where, Ri(A) denotes a read operation by transaction Ti on a data item A, Wi(A) denotes a write operation by transaction Ti on a data item A.
What is the possible number of conflict serializable schedule of the above schedule **S**.
Ans: [redacted]

- ○ 4
- ○ 3
- ○ 1
- ○ 0

17. Consider the following two schedules **S1** and **S2** and three transactions T_1, T_2, T_3:

$$\mathbf{S1}: R_1(X);\ R_3(Y);\ W_1(X);\ R_2(X);\ W_3(Y);\ W_2(X);\ R1_(Y);\ W_1(Y);$$
$$\mathbf{S2}: R_3(Y);\ R_1(X);\ W_1(X);\ R_2(X);\ W_3(Y);\ R_1(Y);\ W_1(Y);\ W_2(X);$$

3. Consider the following schedules: [MSQ: 3 Points]

S1:W3(A), R2(A), W2(A), W3(B), W3(C), W1(C)
S2:W1(A), W3(A), W3(C), W2(A), W1(B), W3(B)

Which of the following options is/are correct?

- ○ Schedule **S1** is conflict serializable.
- ○ Schedule **S1** can be two-phase lockable.
- ○ Schedule **S2** is conflict serializable.
- ○ Schedule **S2** can be two-phase lockable.