2(

3.2 ACID Properties

AU : May-14, Marks 8

1) Atomicity :

- This property states that each transaction must be considered as a **single unit** and **must be completed fully or not completed at all**.
- No transaction in the database is left half completed.
- Database should be in a state either before the transaction execution or after the transaction execution. It **should not be in a state 'executing'**.
- For example - In above mentioned withdrawal of money transaction all the five steps must be completed fully or none of the step is completed. Suppose if transaction gets failed after step 3, then the customer will get the money but the balance will not be updated accordingly. The state of database should be either at before ATM withdrawal (i.e customer without withdrawn money) or after ATM withdrawal (i.e. customer with money and account updated). This will make the system in consistent state.

2) Consistency :

- The database must remain in consistent state after performing any transaction.
- For example : In ATM withdrawal operation, the balance must be updated appropriately after performing transaction. Thus the database can be in consistent state.

3) Isolation :

- In a database system where **more than one transaction** are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system.
- No transaction will affect the existence of any other transaction.
- **For example :** If a bank manager is checking the account balance of particular customer, then manager should see the balance either before withdrawing the money or after withdrawing the money. This will make sure that each individual transaction is completed and any other dependent transaction will get the consistent data out of it. Any failure to any transaction will not affect other transaction in this case. Hence it makes all the transactions consistent.

4) Durability :

- The database should be **strong enough** to handle any **system failure**.
- If there is any set of insert /update, then it should be able to handle and commit to the database.
- If there is any **failure**, the database should be able to **recover** it to the consistent state.
- For example : In ATM withdrawal example, if the system failure happens after Customer getting the money then the system should be strong enough to update Database with his new balance, after system recovers. For that purpose the system has to keep the **log of each transaction and its failure**. So when the system recovers, it should be able to know when a system has failed and if there is any pending transaction, then it should be updated to Database.

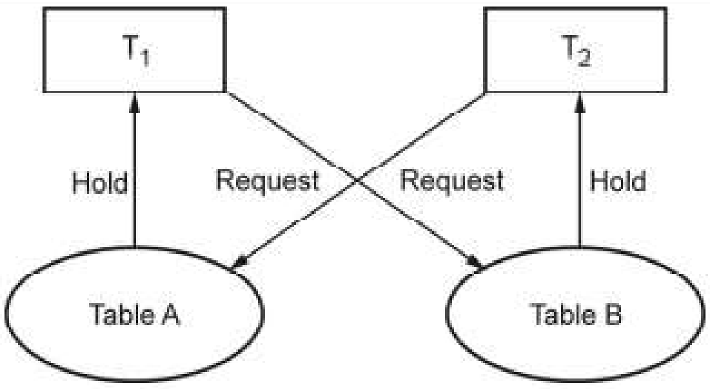
&

3.12 Dead Lock

AU : May-09, Dec. 14, 15, 16, Marks 16

Deadlock is a specific concurrency problem in which two transactions depend on each other for something.

For example – Consider that transaction T1 holds a lock on some rows of table **A** and needs to update some rows in the **B** table. Simultaneously, transaction T2 holds locks on some rows in the **B** table and needs to update the rows in the **A** table held by Transaction T1.



Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. This situation is called **deadlock** in DBMS.

Definition : Deadlock can be formally defined as - " A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. "

There are four conditions for a deadlock to occur

A deadlock may occur if all the following conditions holds true.

1. **Mutual exclusion condition :** There must be at least one resource that cannot be used by more than one process at a time.
2. **Hold and wait condition :** A process that is holding a resource can request for additional resources that are being held by other processes in the system.
3. **No preemption condition :** A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
4. **Circular wait condition :** A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ….so on and the last process is waiting for the first process. Thus making a circular chain of waiting.

Deadlock can be handled using two techniques –

1. Deadlock Prevention 2. Deadlock Detection and deadlock recovery

1. Deadlock Prevention :

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

There are two techniques used for deadlock prevention –

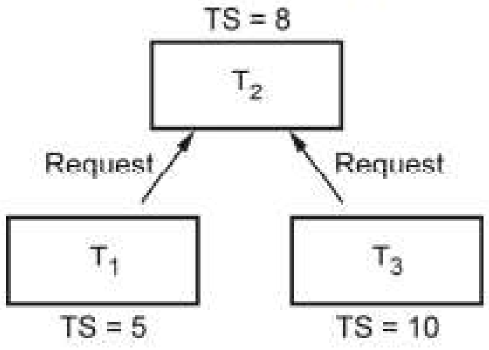
(i) Wait-Die :

- In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It **allows the older transaction to wait** until the resource is available for execution.
- Suppose there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS :
  - Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
  - Check if TS(Ti) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Timestamp is a way of assigning priorities to each transaction when it starts. If timestamp is lower then that transaction has higher priority. **That means oldest transaction has highest priority**.

For example –

Let T1 is a transaction which requests the data item acquired by Transaction T2. Similarly T3 is a transaction which requests the data item acquired by transaction T2.



Here TS(T1) i.e. Time stamp of T1 is less than TS(T3). In other words T1 is older than T3. Hence T1 is made to **wait** while T3 is **rolledback.**

(ii) Wound- wait :

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After some delay, the younger transaction is restarted but with the same timestamp.

- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Suppose T1 needs a resource held by T2 and T3 also needs the resource held by T2, with TS(T1)=5, TS(T2)=8 and TS(T3)=10, then T1 being older waits and T3 being younger dies. After the some delay, the younger transaction is restarted but with the same timestamp.

This ultimately prevents a deadlock to occur.

To summarize

	Wait-Die	**Wound-wait**
Older transaction needs a data item held by younger transaction	older transaction waits	younger transaction dies.
Younger transaction needs a data item held by older transaction	Younger transaction dies	Younger transaction dies.

2. Deadlock Detection :

- In deadlock detection mechanism, an algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred or not. If deadlock is occurrence is detected, then the system must try to recover from it.
- Deadlock detection is done using wait for graph method.

Wait For Graph

- In this method, a graph is created based on the transaction and their lock. If the created **graph has a cycle or closed loop, then there is a deadlock.**
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions in the system.
- When transaction Ti requests a data item currently being held by transaction Tj , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i .

2(ii)

3.12 Dead Lock

AU : May-09, Dec. 14, 15, 16, Marks 16

Deadlock is a specific concurrency problem in which two transactions depend on each other for something.

For example – Consider that transaction T1 holds a lock on some rows of table **A** and needs to update some rows in the **B** table. Simultaneously, transaction T2 holds locks on some rows in the **B** table and needs to update the rows in the **A** table held by Transaction T1.

Definition : Deadlock can be formally defined as - " A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. "

There are four conditions for a deadlock to occur

A deadlock may occur if all the following conditions holds true.

1. **Mutual exclusion condition :** There must be at least one resource that cannot be used by more than one process at a time.
2. **Hold and wait condition :** A process that is holding a resource can request for additional resources that are being held by other processes in the system.
3. **No preemption condition :** A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
4. **Circular wait condition :** A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ....so on and the last process is waiting for the first process. Thus making a circular chain of waiting.

Deadlock can be handled using two techniques –

1. Deadlock Prevention 2. Deadlock Detection and deadlock recovery

3(

3.10 Two Phase Locking

AU : May-14, 18, Dec.-16, Marks 7

- The two phase locking is a protocol in which there are two phases :

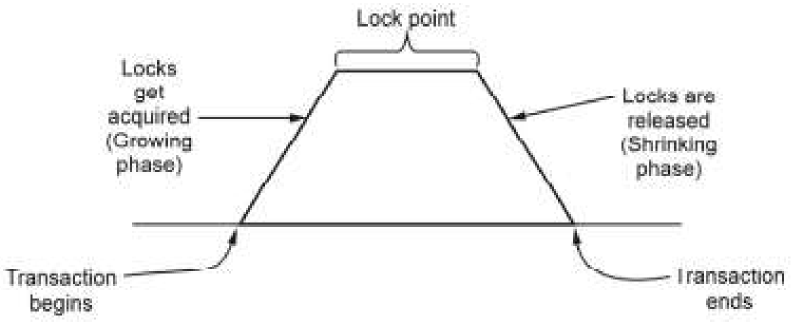
  i) **Growing phase (Locking phase) :** It is a phase in which the transaction may obtain locks but does not release any lock.

  ii) **Shrinking phase (Unlocking phase) :** It is a phase in which the transaction may release the locks but does not obtain any new lock.

- **Lock Point :** The **last lock position** or **first unlock position** is called lock point. For example

-

$$
\begin{array}{l}
\text{Lock(A)} \\
\text{Lock(B)} \\
\text{Lock(C)} \\
\left.\begin{array}{l} \text{....} \\ \text{...} \\ \text{...} \end{array}\right\} \textbf{Lock Point} \\
\text{Unlock(A)} \\
\text{Unlock(B)} \\
\text{Unlock(C)}
\end{array}
$$


Lock point
Locks get acquired (Growing phase)
Locks are released (Shrinking phase)
Transaction begins
Transaction ends

For example –

Consider following transactions

The important rule for being a two phase locking is **- All Lock operations precede all the unlock operations.**

In above transactions T1 is in two phase locking mode but transaction T2 is not in two phase locking. Because in T2, the Shared lock is acquired by data item B, then data item B is read and then the lock is released. Again the lock is acquired by data item A , then the data item A is read and the lock is then reloased. Thus we get lock-unlock-lock-unlock sequence. Clearly this is not possible in two phase locking.

Example 3.10.1 *Prove that two phase locking guarantees serializability.*

Solution:

- Serializability is mainly an issue of handling write operation. Because any inconsistency may only be created by **write** operation.
- Multiple reads on a database item can happen parallely.
- 2-Phase locking protocol restricts this unwanted read/write by applying **exclusive lock**.
- Moreover, when there is an **exclusive lock** on an item it will **only be released in shrinking phase**. Due to this restriction there is no chance of getting any inconsistent state.

The serializability using two phase locking can be understood with the help of following example

Consider two transactions

T_1	T_2
R(A)	
	R(A)
R(B)	
W(B)	

Step 1 : Now we will **apply two phase locking**. That means we will **apply locks in growing** and **shrinking** phase

T_1	T_2
Lock-S(A)	
R(A)	
	Lock-S(A)
	R(A)
Lock-X(B)	
R(B)	
W(B)	
Unlock-X(B)	
	Unlock-S(A)

Note that above schedule is serializable as it prevents interference between two transactions.

The serializability order can be obtained based on the **lock point**. The lock point is either last lock operation position or first unlock position in the transaction.

The last lock position is in T_1, then it is in T_2. Hence the serializability will be T_1->T_2 based on lock points. Hence The **serializability sequence** can be **R1(A);R2(A);R1(B);W1(B)**

Limitations of Two Phase Locking Protocol

The two phase locking protocol leads to two problems – deadlock and cascading roll back.

(1) Deadlock : The deadlock problem can not be solved by two phase locking. Deadlock is a situation in which when two or more transactions have got a lock and waiting for another locks currently held by one of the other transactions.

For example

T1	**T2**
Lock-X(A)	Lock-X(B)
Read(A)	Read(B)
A=A-50	B=B+100
Write(A)	Write(B)
Delayed, wait for T2 to release Lock on B	Delayed, wait for T1 to release Lock on A

(2) Cascading Rollback : Cascading rollback is a situation in which a single transaction failure leads to a series of transaction rollback. For example –

T1	T2	T3
Read(A)		
Read(B)		
C=A+B		
Write(C)		
	Read(C)	
	Write(C)	
		Read(C)

When T1 writes value of C then only T2 can read it. And when T2 writes the value of C then only transaction T3 can read it. But if the transaction T1 gets failed then automatically transactions T2 and T3 gets failed.

The simple two phase locking does not solve the cascading rollback problem. To solve the problem of cascading Rollback two types of two phase locking mechanisms can be used.

5 (

3.5 Serializability

AU : Dec.-15, May-15, 18, Marks 8

- When multiple transactions run concurrently, then it may lead to inconsistency of data (i.e. change in the resultant value of data from different transactions).
- Serializability is a concept that helps to identify which non serial schedule and find the transaction equivalent to serial schedule.

- There are **two types of serializabilities :** conflict serializability and view serializability

3.5.1 Conflict Serializability

Definition : Suppose T_1 and T_2 are two transactions and I_1 and I_2 are the instructions in T_1 and T_2 respectively. Then these two transactions are said to be conflict Serializable, if both the instruction access the data item d, and at least one of the instruction is write operation.

What is conflict ?: In the definition **three conditions** are specified for a conflict in conflict serializability –

1) There should be **different transactions**
2) The **operations** must be performed on **same data** items
3) **One of the operation** must be the **Write(W)** operation

- We can test a given schedule for conflict serializability by constructing a **precedence graph** for the schedule, and by searching for absence of cycles in the graph.

Testing for serializability

Following method is used for testing the serializability : To test the conflict serializability we can draw a graph G=(V,E) where V = vertices which represent the number of transactions. E = edges for conflicting pairs.

Step 1 : Create a node for each transaction.

Step 2 : Find the conflicting pairs(RW, WR, WW) on the same variable(or data item) by different transactions.

Step 3 : Draw edge for the given schedule. Consider following cases

1. Ti executes write(Q) before Tj executes read(Q), then draw edge from T_i to T_j.
2. Ti executes read(Q) before Tj executes write(Q) , then draw edge from T_i to T_j
3. Ti executes write(Q) before Tj executes write(Q), , then draw edge from T_i to T_j

Step 4 : Now, if precedence graph is cyclic then it is a non conflict serializable schedule and if the precedence graph is acyclic then it is conflict serializable schedule.

3.5.2 View Serializability

If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view serializable schedule.

View Equivalent Schedule : Consider two schedules S_1 and S_2 consisting of transactions

T_1 and T_2 respectively, then schedules S_1 and S_2 are said to be view equivalent schedule if it satisfies following three conditions :

- If transaction T_1 reads a data item A from the database initially in schedule S_1, then in schedule S_2 also, T_1 must perform the initial read of the data item X from the database. This is same for all the data items. In other words - the initial reads must be same for all data items.
- If data item A has been updated at last by transaction T_i in schedule S_1, then in schedule S_2 also, the data item A must be updated at last by transaction T_i.
- If transaction Ti reads a data item that has been updated by the transaction T_j in schedule S_1, then in schedule S_2 also, transaction T_i must read the same data item that has been updated by transaction T_j. In other words the Write-Read sequence must be same.

Steps to check whether the given schedule is view serializable or not

Step 1 : If the schedule is conflict serializable then it is surely view serializable because conflict serializability is a restricted form of view serializability.

Step 2 : If it is not conflict serializable schedule then check whether there exist any blind write operation. The blind write operation is a write operation without reading a value. If there does not exist any blind write then that means the given schedule is not view serializable. In other words if a blind write exists then that means schedule may or may not be view conflict.

Step 3 : Find the view equivalence schedule

8(i)(

3.3 Transaction States

AU : Dec.-11, May-14,18, Marks 8

Each transaction has following five states :

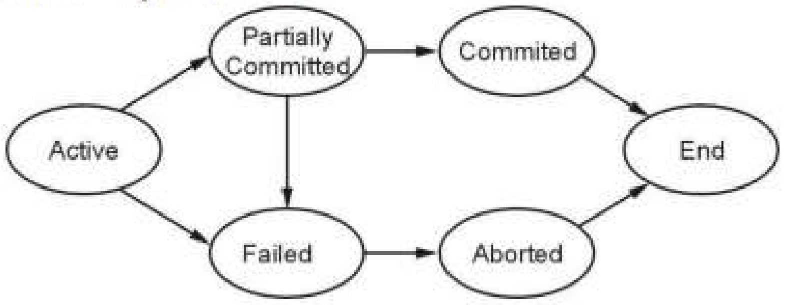


Fig. 3.3.1 Transaction States

1) **Active :** This is the first state of transaction. For example : insertion, deletion or updation of record is done here. But data is not saved to database.

2) **Partially Committed :** When a transaction executes its final operation, it is said to be in a partially committed state.

3) **Failed :** A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.

4) **Aborted :** If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. If not, it brings the database to consistent state by aborting or rolling back the transaction.

5) **Committed :** If a transaction executes all its operations successfully, it is said to be committed. This is the last step of a transaction, if it executes without fail.)

8(ii)

Possible Sequences of States for a Transaction

1. **Normal Execution (Commit Scenario):**
   - **Active → Partially Committed → Committed**
   - **Explanation:** The transaction begins in the active state, executes its operations, moves to partially committed after executing the final statement, and then commits, making its changes permanent.
2. **Failure Scenario (Abort Scenario):**
   - **Active → Failed → Aborted**
   - **Explanation:** The transaction encounters an error during execution in the active state, moves to the failed state, and then rolls back any changes, entering the aborted state.
3. **Recovery from Failure:**
   - **Active → Failed → Aborted → Active (Restarted) → Partially Committed → Committed**
   - **Explanation:** After the transaction enters the aborted state, it may be restarted by the system. After restarting, it executes again and eventually commits successfully.

Explanation of Each State Transition

- **Active to Partially Committed**: The transaction has executed its final statement and is ready to commit but hasn't made the changes permanent yet.
- **Partially Committed to Committed**: The transaction successfully commits, meaning all changes are saved to the database.
- **Active to Failed**: An error occurs during execution, possibly due to resource contention, deadlock, or a system crash, leading to failure.
- **Failed to Aborted**: The system rolls back all changes made by the transaction, returning to the state before the transaction began.
- **Aborted to Active (Restart)**: If the system decides to retry the transaction after an abort, the transaction may be restarted.
- **Aborted (Termination)**: The transaction is permanently terminated after rollback, and no further attempts are made to execute it.

These state transitions ensure consistency, durability, and atomicity in transaction management within a database.

(6) (i) Precedence graph

S_1 : T_1 : R(x) → T_2 : R(x) → T_1 : W(x) → T_2 : W(Y) → T_1 : R(Y) → T_2 : R(Y)

Conflicts in S_1 :

(i) T_1(W(Y)) → T_2(R(Y))
Write → Read conflicts

(ii) T_2(W(Y)) → T_1(R(Y))
Write → Read conflicts

• There are cycles between T_1 & T_2 due to the conflicts.

S_2 : T_3 : W(x) → T_1 : R(x) → T_1 : W(Y → T_2 : R(Z) → T_2 : W(Z) ⇒ T_3 : R(Z

Conflict in S_2 :

(i) ~~T_2~~T_3(W(x)) → T_1(R(x))
Write - Read conflict

(ii) T_2(W(Z)) → T_3(R(Z))

⇒ No cycles are there as they donot have circular dependencies.

(ii) Conflict Serializable:

$S_1 \rightarrow$ No

$S_2 \rightarrow$ Yes.

(iii) Conflict equivalent Serial schedules:

$S_1 \rightarrow$ No

$S_2 \rightarrow$ Yes

$T_3 \rightarrow T_2 \rightarrow T_1$

(iv) View Serializable:

$S_1 \rightarrow$ Yes

$T_1 \rightarrow T_2$ (or) $T_2 \rightarrow T_1$

$S_2 \rightarrow$ Yes

$T_3 \rightarrow T_1 \rightarrow T_2$

```
(7)  T1 : Read (A);
          Read (B);
          if A = 0 then B = B+1;
          write (B).

     T2 : Read (B);
          Read (A);
          if B = 0 then A = A+1;
          write (A)
```

<u>Rewrite T</u>

Rewrite T_1 & T_2

```
T1 : lock (A);
     Read (A);
     lock (B)
     Read (B);
     if A = 0 , then
          B : = B+1;
     write (B);
     unlock (B);
     unlock (A);

T2 : lock (B);
     Read (B);
     lock (A);
     Read (A);
     if B = 0, then
          A : = A + 1;
     write (A);
     unlock (A);
     unlock (B);
```

The execution of these transactions can result in a deadlock if T_1 locks A

+1; and T_2 locks B at the same time.

These deadlocks can be avoided by (i) Timeout

(ii) Lock ordering.

(i) Timeout : Abort the transaction if it takes too long.

(ii) Lock ordering : Always locks in predefined order (A first, then B)