UNIT-5

SYLLABUS:

Transaction Processing: Introduction, Transaction and System Concepts, Desirable Properties of Transactions, Characterizing Schedules Based on Recoverability & Serializability, Transaction Support in SQL.

Introduction to Concurrency Control: Two-Phase Locking Techniques: Types of Locks and System Lock Tables, Guaranteeing Serializability by Two-Phase Locking.

Introduction to Recovery Protocols: Recovery Concepts, No-UNDO/REDO Recovery Based on Deferred Update, Recovery Techniques Based on Immediate Update, Shadow Paging.

> **Part I : Transaction Processing**

Introduction to Transaction Processing

Single User Vs Multiuser Systems

- A DBMS system is said to be single user system if at the most one user at a time can use the system.
- A DBMS system is said to be multiuser system if many users can use the system.
- The access to the database system in multiuser system is concurrently.
- Most of the DBMS systems are based on multi user systems.
- Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.

Transaction, Database Items, Read and Write Operations and DBMS Buffers

- **Definition of Transaction :** A transaction can be defined as a **group of tasks** that form a single logical unit. For example - Suppose we want to withdraw ₹ 100 from an account then we will follow following operations :
  1) Check account balance
  2) If sufficient balance is present request for withdrawal.
  3) Get the money
  4) Calculate Balance = Balance – 100
  5) Update account with new balance.

The above mentioned **four steps** denote **one transaction**.

- There are two modes of concurrency -
  - **Interleaved Processing :** Concurrent execution of processes is interleaved in a single CPU.
  - **Parallel Processing :** Processes are concurrently executed in multiple CPUs.
- Basic transaction processing theory assumes interleaved concurrency.
- A database is a collection of named data items.

- Basic operations on data item A are -

  > 1. read_item(X)   2. write_item(X)

1. **read_item(X)** : This is a reading operation in which database item named X is read into a programming variable. We can name the program variable as X for simplification.

2. **write_item(X)** : This operation writes the value of program variable X into the database item which is also named as X.

- Basic unit of data transfer from the disk to the computer main memory is one block.

- **read_item(X) command includes the following steps :**

  Step 1 : Find the address of the disk block that contains item X.

  Step 2 : Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

  Step 3 : Copy item X from the buffer to the program variable named X.

- **write_item(X) command includes the following steps :**

  Step 1 : Find the address of the disk block that contains item X.

  Step 2 : Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

  Step 3 : Copy item X from the program variable named X into its correct location in the buffer.

  Step 4 : Store the updated block from the buffer back to disk.

- **Example of sample transaction performing read and write operations**

  read_item(X);
  X=X+M;
  Write_item(X);

  - **Transaction Notations :**

  The transaction notations focuses on read and write operations. For example –
Following are two transactions denoted by T1 and T2

  T1:b1;r1(X);w1(X);r1(Y);W1(Y);e1;
  T2:b2;r2(X);r2(Y);e2

  The r and w represents the read and write operations. The b1 and b2 represents the beginning and e1 and e2 represents ending of transaction.

Why Concurrency Control is Needed ?

- Concurrent execution of transactions over shared database creates several data integrity and consistency problems - these are,

1) Lost update problem :

  This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

For example - Consider following transactions,

1) Salary of employee is read during transaction T_1.
2) Salary of employee is read by another transaction T_2.
3) During transaction T_1, the salary is incremented by ₹ 200
4) During transaction T_2, the salary is incremented by ₹ 500

	T_1	T_2	
This update is lost	Read		Salary = ₹ 1000
		Read	Salary = ₹ 1000
Time	Update Increment salary by ₹ 200		Salary = ₹ 1200
Only this update is successful		Update Increment salary by ₹ 500	Salary = ₹ 1500

The result of the above sequence is that the update made by transaction T_1 is completely lost. Therefore this problem is called as **lost update problem**.

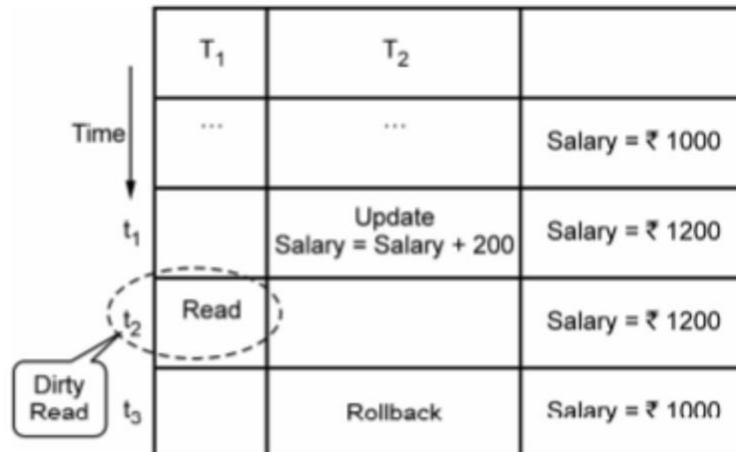**2) Dirty read or uncommited read problem or temporary update :**

The dirty read is a situation in which one transaction reads the data immediately after the write operation of previous transaction

T_1	T_2	
R(A)		
A=A+50		
W(A)		Dirty read
	R(A)	
	A=A – 20	
	W(A)	
	Commit	
Commit		

For example - Consider following transactions,

Assume initially salary is = ₹ 1000.

	T$_1$	T$_2$	
Time	...	...	Salary = ₹ 1000
t$_1$		Update Salary = Salary + 200	Salary = ₹ 1200
t$_2$	Read		Salary = ₹ 1200
t$_3$		Rollback	Salary = ₹ 1000

(Dirty Read indicated at t$_2$ Read)

1) At the time t$_1$, the transaction T$_2$ updates the salary to ₹ 1200.

2) This salary is read at time t$_2$ by transaction T$_1$. Obviously it is ₹ 1200.

3) But at the time t$_3$, the transaction T$_2$ performs rollback by undoing the changes made by T$_1$ and T$_2$ at time t$_1$ and t$_2$.

4) Thus the salary again becomes = ₹ 1000. This situation leads to **Dirty Read or Uncommited Read** because here the read made at time t$_2$ (**immediately after update of another transaction**) becomes a dirty read.

3) Non-repeatable read problem :

This problem is also known as **inconsistent analysis problem**. This problem occurs when a particular transaction sees two different values for the same row within its lifetime. For example -

		T$_1$	T$_2$	
Time	t$_1$	Read		Salary = ₹ 1000
	t$_2$		Update salary from ₹ 1000 to ₹ 1200	Salary = ₹ 1200
	t$_3$		Commit	
	t$_4$	Read		Salary = ₹ 1200
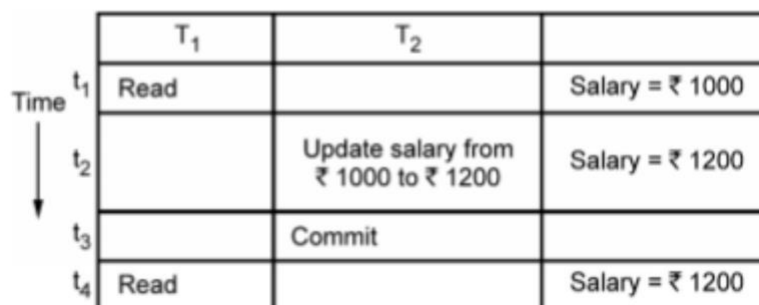
1) At time t1, the transaction T1 reads the salary as ₹ 1000.

2) At time t2 the transaction T2 reads the same salary as ₹ 1000 and updates it to ₹1200.

3) Then at time t3, the transaction T2 gets committed.

4) Now when the transaction T1 reads the same salary at time t4, it gets different value than what it had read at time t1. Now, transaction T1 cannot repeat its reading operation. Thus inconsistent values are obtained.

Hence the name of this problem is non-repeatable read or inconsistent analysis problem.

(4) Incorrect read problem :

This is a problem in which one of the transaction makes the changes in the database system and due to these changes another transaction can not read the data item which it has read just recently. For example -

Time		T_1	T_2	
	t_1	Read		Salary = ₹ 1000
	t_2		Read	Salary = ₹ 1000
	t_3	Delete salary		No salary
	t_4		Read	"Can not find salary"

(1)   At time t1, the transaction T1 reads the value of salary as ₹ 1000

(2)   At time t2, the transaction T2 reads the value of the same salary as ₹ 1000

(3)   At time t3, the transaction T1 deletes the variable salary.

(4)   Now at time t4, when T2 again reads the salary it gets error. Now transaction T2 can not identify the reason why it is not getting the salary value which is read just few time back.

This problem occurs due to changes in the database and is called **phantom read problem**.

Why Recovery is Needed ?

The important reason why recovery is needed is because some transaction gets failed. Following are the reasons that indicate why recovery is needed -

1)   **A computer failure :**
   o   The computer failure means some hardware crash or some software error may occur.
   o   During execution of transaction if such computer failure occurs then the contents of the computer's internal memory may be lost.

2) **A transaction or system error :**
   o   Some operation in the transaction may cause it to fail, such as integer overflow or division by zero.
   o   Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.

3) **Local errors or exception conditions detected by the transaction**
   o   Certain conditions occur that cancels the transactions. For example – in Banking application, if the insufficient balance is found then the transaction such as 'withdrawal of money' gets canceled.

4) **Concurrency control enforcement**

   o The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5) **Disk failure**

   o Sometimes due to read/write head crash disk blocks may lose their data. This is a severe failure.

6) **Physical problems and catastrophes**

   o It includes various problems such as power failure, fire, overwriting disks or tapes by mistake, mounting wrong tapes by operator, theft and so on.

Transaction and System Concepts

Q) Explain various states of a transaction execution.

Transaction States

Each transaction has following five 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.
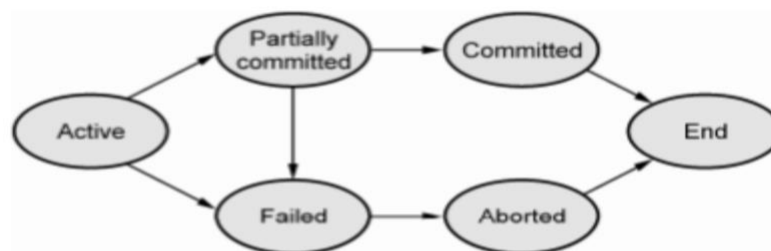
Transaction states

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.

Q) Define a transaction. Discuss the following with relevant examples:

1. **A read only transaction**
2. **A read write transaction**
3. **An aborted transaction**

Solution : 1) Read only transaction :

T1
Read(A)
Read(B)
Display(A − B)

2) A read write transaction :

T1
Read(A)
A=A+100
Write(A)

3) An aborted transaction

T1	T2	
Read(A)		Assume A=100
A=A+50		A=150
Write(A)		
	Read(A)	A=150
	A=A+100	A=250
RollBack		A=100 (restore back to original value which is before Transaction T1)
	Write(A)	

Q) Discuss ACID properties of a database transaction.

Desirable Properties of Transactions

In a database, each transaction should maintain ACID property to meet the consistency and integrity of the database.

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.

Characterizing Schedules based on Recoverability

The classification of the schedule based on recoverability is -

| 1. Recoverable Schedule | 2. Cascadeless Schedule | 3. Strict Schedule |

1. **Recoverable Schedule :** This is a kind of schedule where no transaction needs to be rolled back.

   Definition : A recoverable schedule is one where, 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 commit operation of T_j.

For example : Consider following schedule, consider A = 100

T_1	T_2
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A – 20
	W(A)
	Commit
some transaction…	
Commit	

- The above schedule is inconsistent if failure occurs after the commit of T_2.
- It is because T_2 is **dependable transaction** on T_1. A transaction is said to be dependable if it contains a **dirty read**.
- The dirty read is a situation in which one transaction reads the data immediately after the write operation of previous transaction.

T_1	T_2
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A-20
	W(A)
	Commit
Commit	

- Now if the dependable transaction i.e. T2 is committed first and then failure occurs then if the transaction T1 makes any changes then those changes will not be known to the T2. This leads to non recoverable state of the schedule.
- To make the schedule recoverable we will apply the rule that - commit the independent transaction before any dependable transaction.
- In above example independent transaction is T1, hence we must commit it before the dependable transaction i.e. T2.
- The recoverable schedule will then be -

T1	T2
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A-20
	W(A)
Commit	
	Commit

(2) Cascadeless Schedule

- **Definition :** If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written that data item is committed or aborted, then such a schedule is known as a **cascadeless schedule**.

- The cascadeless schedule allows only committed Read operation. For example :

T1	T2	T3
R(A)		
A=A+50		
W(A)		
Commit		
	R(A)	
	A=A-20	
	W(A)	
	Commit	
		R(A)
		W(A)

- In above schedule at any point if the failure occurs due to commit operation before every read operation of each transaction, the schedule becomes recoverable and atomicity can be maintained.

(3) Strict Schedule

- A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.
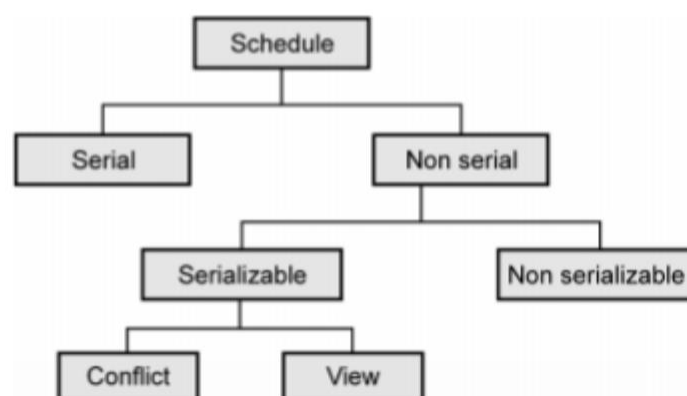
For example

T1	T2
R(A)	
A=A+50	
W(A)	
Commit	
	R(A)
	A=A-20
	W(A)
	Commit

Characterizing Schedule based on Serializibility

Concept of Schedule

   Schedule is an order of multiple transactions executing in concurrent environment. Following Fig represents the types of schedules.

Types of schedule

   Serial schedule : The schedule in which the transactions execute one after the other is called **serial schedule**. It is consistent in nature. For example : Consider following two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

All the operations of transaction T1 on data items A and then B executes and then in transaction T2 all the operations on data items A and B execute. The **R stands for read** operation and **W stands for write** operation.

Non serial schedule : The schedule in which operations present within the transaction are intermixed. This may lead to conflicts in the result or inconsistency in the resultant data.

For example -

Consider following two transactions,

T1	T2
R(A)	
W(A)	
	R(A)
	W(B)
R(A)	
W(B)	
	R(B)
	W(B)

The above transaction is said to be non serial which result in inconsistency or conflicts in the data.

Serializability of Scheduling

- 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.

For example :

T1	A	B	T2
Initial Value	100	100	
A=A – 10			
W(A)			
B=B+10			
W(B)			
	90	110	
			A=A−10
			W(A)
	80	110	

- In above transactions initially T1 will read the values from database as A=100,

  B=100 and modify the values of A and B. But transaction T2 will read the modified value i.e. 90 and will modify it to 80 and perform write operation. Thus at the end of transaction T1 value of A will be 90 but at end of transaction T2 value of A will be 80. Thus conflicts or inconsistency occurs here. This sequence can be converted to a sequence which may give us consistent result. This process is called **serializability**.

Difference between serial schedule and serializable schedule

Sr. No.	Serial schedule	Serializable schedule
1	**No concurrency** is allowed in serial schedule.	**Concurrency** is allowed in serializable schedule.
2	In serial schedule, if there are two transactions executing at the same time and no interleaving of operations is permitted, then following can be the possibilities of execution - i) Execute all the operations of transactions T_1 in a sequence and then execute all the operations of transactions T_2 in a sequence. ii) Execute all the operations of transactions T_2 in a sequence and then execute all the operations of transactions T_1 in a sequence.	In serializable schedule, if there are two transactions executing at the same time and interleaving of operations is allowed there can be different possible orders of executing an individual operation of the transactions.

Example of serial schedule		Example of serializable schedule	
T_1	T_2	T_1	T_2
Read(A)		Read(A)	
A=A−50		A=A−50	
Write(A)		Write(A)	
Read(B)			Read(B)
B=B+100			B=B+100
Write(B)			Write(B)
	Read(A)	Read(B)	
	A=A+10	Write(B)	
	Write(A)		

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

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.
- Predence graph is a directed graph, consisting of G = (V,E) where V is set of vertices and E is set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds :
  1. T_i executes write(Q) before T_j executes read(Q).
  2. T_i executes read(Q) before T_j executes write(Q).
  3. T_i executes write(Q) before T_j executes write(Q).
- A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting.**

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

Q) Check weather following schedule is conflict serializable or not conflict serializable then find the serializability order.

T1	T2	T3
R(A)		
	R(B)	
		R(B)
	W(B)	
W(A)		
		W(A)
	R(A)	
	W(A)	

Solution :

Step 1 : We will read from top to bottom, and build a precedence graph for conflicting entries :
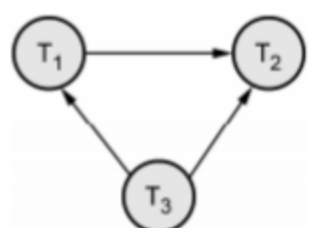
Precedence graph

Step 2 : As there is **no cycle** in the precedence graph, the given sequence is **conflict serializable**. Hence we can convert this non serial schedule to serial schedule. For that purpose we will follow these steps to find the serializable order.

Step 3 : A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

Step 4 : Find the vertex which has no incoming edge which is T1. Finally find the vertex having no outgoing edge which is T2. So in between them is T3. Hence the order will be T1 – T3 – T2.

Q) Consider the three transactions T1, T2 and T3 and schedules S1 and S2 given below. Determine whether each schedule is serializable or not? If a schedule is serializable write down the equivalent serial schedule(S).

> T1: R1(x) R1(z);W1(x);
> T2: R2(x);R2(y);W2(z);W2(y)
> T3:R3(x);R3(y);W3(y);
> S1: R1(x);R2(z);R1(z);R3(x);R3(y);W1(x);W3(y);R2(y);W2(z);W2(y);
> S2: R1(x);R2(z);R3(x);R1(z);R2(y);R3(y);W1(x);W2(z);W3(y);W2(y);

Solution : Step 1 : We will represent the schedule S1 as follows

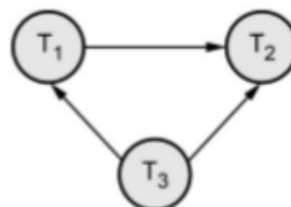| T1	T2	T3
R1(x)		
	R2(z)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
		W3(y)
	R2(y)	
	W2(z)	
	W2(y)	

Step (a) : We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them -

   i)   Both the **operations belong to different transactions**.

   ii) Both the operations are on **same data item**.

  iii) **At least one** of the two operations is a **write operation**

The conflicting entries are as follows -

T1	T2	T3
R1(x)		
	R2(z)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
		W3(y)
	R2(y)	
	W2(z)	
	W2(y)	

Step (b) : Now we will draw precedence graph as follows -

     As there is **no cycle** in the precedence graph, the given sequence is **conflict serializable**. Hence we can convert this non serial schedule to serial schedule. For that purpose we will follow these steps to find the serializable order.

Step (c) : A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

Step (d) : Find the vertex which has no incoming edge which is T3. Finally find the vertex having no outgoing edge which is T2. So in between them is T1. Hence the order will be T3- T1-T2

Step 2 : We will represent the schedule S2 as follows -

| T1	T2	T3
R1(x)		
	R2(z)	
		R3(x)
R1(z)		
	R2(y)	
		R3(y)
W1(x)		
	W2(z)	
		W3(y)
	W2(y)	

We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them -

   i)   Both the **operations belong to different transactions**.

   ii) Both the operations are on **same data item**.

   iii) **At least one** of the two operations is a **write operation**

The conflicting entries are as follows -

T1	T2	T3
R1(x)		
	R2(z)	
		R3(x)
R1(z)		
	R2(y)	
		R3(y)
W1(x)		
	W2(z)	
		W3(y)
	W2(y)	

Step (b) : Now we will draw precedence graph as follows -
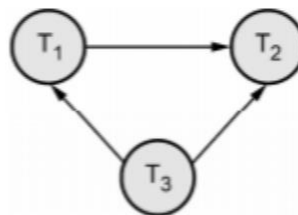
As there is **no cycle** in the precedence graph, the given sequence is **conflict serializable**. Hence we can convert this non serial schedule to serial schedule. For that purpose we will follow these steps to find the serializable order.

Step (c) :   **A serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

Step (d) : Find the vertex which has no incoming edge which is T3. Finally find the vertex having no outgoing edge which is T2. So in between them is T1. Hence the order will be T3- T1-T2

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 S1 and S2 consisting of

transactions T1 and T2 respectively, then schedules S1 and S2 are said to be view equivalent schedule if it satisfies following three conditions :

o If transaction T1 reads a data item A from the database initially in schedule S2, then in schedule S2 also, T1 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.

o If data item A has been updated at last by transaction Ti in schedule S1, then in schedule S2 also, the data item A must be updated at last by transaction Ti.

o If transaction Ti reads a data item that has been updated by the transaction Tj in schedule S1, then in schedule S2 also, transaction Ti must read the same data item that has been updated by transaction Tj. 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

Q) Consider the following schedules for checking if these are view serializable or not.

T1	T2	T3
		W(C)
	R(A)	
	W(B)	R(B)
R(C)		
		W(B)
W(B)		

Solution :

i) The initial read operation is performed by T_2 on data item A or by T_1 on data item C. Hence we will begin with T_2 or T_1. We will choose T_2 at the beginning.

ii) The final write is performed by T_1 on the same data item B. Hence T_1 will be at the last position.

iii) The data item C is written by T_3 and then it is read by T_1. Hence T_3 should appear before T_1. Thus we get the order of schedule of view serializability as $T_2 - T_1 - T_3$

Q) Consider the following schedules. The actions are listed in the order they are scheduled, and prefixed with the transaction name.

$$S_1 : T_1 : R(X), \quad T_2 : R(X), \quad T_1 : W(Y), \quad T_2 : W(Y) \; T_1 : R(Y), \quad T_2 : R(Y)$$
$$S_2 : T_3 : W(X), \quad T_1 : R(X), \quad T_1 : W(Y), \quad T_2 : R(Z), T_2 : W(Z) \quad T_3 : R(Z)$$

For each of the schedules, answer the following questions:

   i.   **What is the precedence graph for the schedule?**
  ii.   **Is the schedule conflict serializable? If so, what are all the conflict equivalent serial schedules?**
 iii.   **Is the schedule view serializable? If so, what are all the view equivalent serial schedules?**

Solution : (i) We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them -

- Both the **operations belong to different transactions**.
- Both the operations are on **same data item**.
- At least one of the two operations is a write operation

    For S_1 : From above given example in the top to bottom scanning we find the conflict as

    o $T_1 : W(Y), T_2 : W(Y)$ and
    o $T_2 : W(Y), T_1 : R(Y)$

Hence we will build the precedence graph. Draw the edge between conflicting transactions. For example in above given scenario, the conflict occurs while moving from T_1: W(Y) to T_2:W(Y). Hence edge must be from T_1 to T_2. Similarly for second conflict, there will be the edge from T_2 to T_1.

Precedence graph for S_1

For S_2 : The conflicts are

    o $T_3 : W(X), \quad T_1 : R(X)$
    o $T_2 : W(Z) \quad T_3 : R(Z)$

Hence the precedence graph is as follows -

Precedence graph for S_2

(ii)

- o S_1 is **not conflict-serializable** since the dependency **graph has a cycle**.
- o S_2 is conflict-serializable as the dependency graph is acyclic. The order T_2-T_3-T_1 is the only equivalent **serial order**.

(iii)

- o S_1 is not view serializable.
- o S_2 is trivially view-serializable as it is conflict serializable. The only serial order allowed is T_2-T_3-T_1.

Transaction Support in SQL

- Using SQL statements various transactions occur with database systems.
- A Single SQL statement is always atomic.
- With SQL there is no explicit **Begin** Transaction statement. The transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have explicit **End** statement. It can be wither COMMIT or ROLLBACK.
- Every transaction must have certain characteristics which are specified using SET TRANSACTION statement in SQL.
- Three characteristics are - 1. Access Mode 2. Diagnostic Area Size and 3. Isolation Level

(1) Access Mode :

- o In SQL, the access mode can be specified as READ ONLY or READ WRITE.
- o The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

(2) Diagnostic Area Size

- o The diagnostic area size n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

(3) Isolation Level

- o The transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system at particular instance.
- o Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- o The isolation level is denoted as <isolation>. There are four levels of transaction isolation defined by SQL -

(i) Serializable :

- This is the **highest isolation level**.
- **Serializable execution is defined** to be an execution of operations in which concurrently executing transactions **appears to be serially executing**.
- This is a default isolation level.

(ii) Repeatable Read :

- This is the **most restrictive isolation level**.
- The transaction **holds read locks on all rows** it references.
- It holds **write locks on all rows** it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

(iii) Read Committed :

- This isolation level allows only **committed data to be read**.
- Thus it does **not allows dirty read** (i.e. one transaction reading of data immediately after written by another transaction).
- The transaction hold a **read or write lock** on the current row, and thus prevent other rows from reading, updating or deleting it.

(iv) Read Uncommitted :

- It is **lowest isolation level**.
- In this level, one transaction may read not yet committed changes made by other transaction.
- This level **allows dirty reads**.
- In this level **transactions are not isolated** from each other.

o The potential problems with lower isolation level are -

1) **Dirty Read :** The dirty read is a situation in which one transaction reads the data immediately after the write operation of previous transaction.

2) **Non Repeatable Read :** Allowing another transaction to write a new value between multiple reads of one transaction is called **non repeatable read**.

3) **Phantoms :** This is a problem in which one of the transaction makes the changes in the database system and due to these changes another transaction can not read the data item which it has read just recently.

o   Possible violation of serializability

Isolation Level	Type of Problems		
	Dirty Read	Non-repeatable Read	Phantoms
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITED	No	Yes	Yes
REPEATABL READ	No	No	Yes
SERIALIZABLE	No	No	No

<div style="text-align:center; border:1px solid;">Part II : Concurrency Control in Databases</div>

Concurrency Control

- One of the fundamental properties of a transaction is **isolation**.
- When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.
- A database can have multiple transactions running at the same time. This is called **concurrency**.
- To preserve the isolation property, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called **concurrency control schemes.**
- **Definition of concurrency control :** A mechanism which ensures that simultaneous execution of more than one transactions does not lead to any database inconsistencies is called **concurrency control mechanism.**
- The concurrency control can be achieved with the help of various protocols such as - Lock based protocol, deadlock handling, multiple granularity, timestamp based protocol, and validation based protocols.
- **Definition of concurrency control :** A mechanism which ensures that simultaneous execution of more than one transactions does not lead to any database inconsistencies is called **concurrency control mechanism.**
- The concurrency control can be achieved with the help of various protocols such as - Lock based protocol, deadlock handling, multiple granularity, timestamp based protocol, and validation based protocols.

Two Phase Locking Techniques for Concurrency Control

Why Do We Need Lock ?

- One of the method to ensure the **isolation** property in transactions is to require that data items be accessed in a mutually exclusive manner. That means, while one transaction is accessing a data item, no other transaction can modify that data item.
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock on that item.**
- Thus the lock on the operation is required to ensure the isolation of transaction.

Working of Lock

- **Concept of protocol :** The lock based protocol is a mechanism in which there is exclusive use of **locks** on the data item for current transaction.
- **Types of locks :** There are two types of locks used –

```
Lock
  ├── Shared lock
  └── Exclusive lock
```

i) **Shared lock :** The shared lock is used for reading data items only. It is denoted by Lock-S. This is also called as **read lock**.

ii) **Exclusive lock :** The exclusive lock is used for both read and write operations. It is denoted as Lock-X. This is also called as **write lock**.

- The **compatibility matrix** is used while working on set of locks. The **concurrency control manager** checks the compatibility matrix before granting the lock. If the two modes of transactions are compatible to each other then only the lock will be granted.

- In a set of locks may consists of shared or exclusive locks. Following matrix represents the compatibility between modes of locks.

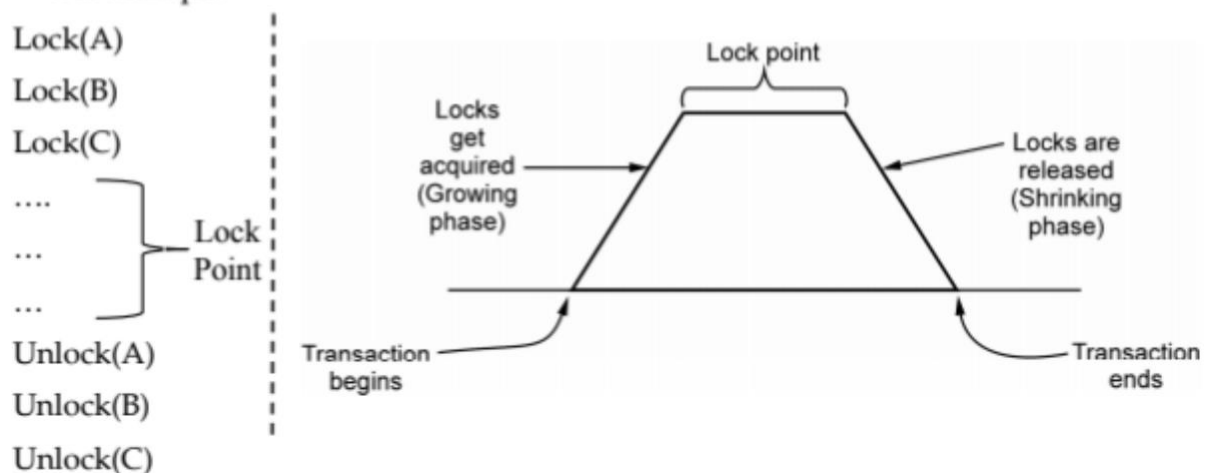|	S	X
S	T	F
X	F	F

Compatibility matrix for locks

Here T stands for True and F stands for False. If the control manager get the compatibility mode as True then it grant the lock otherwise the lock will be denied.

- **For example :** If the transaction T_1 is holding a shared lock in data item A, then the control manager can grant the shared lock to transaction T_2 as compatibility is True.

  But it cannot grant the exclusive lock as the compatibility is false. In simple words if transaction T_1 is reading a data item A then same data item A can be read by another transaction T_2 but cannot be written by another transaction.

- Similarly if an exclusive lock (i.e. lock for read and write operations) is hold on the data item in some transaction then no other transaction can acquire shared or exclusive lock as the compatibility function denotes F. That means of some transaction is writing a data item A then another transaction can not read or write that data item A.

- Hence the **rule of thumb is**

i) **Any number** of transactions can hold **shared lock** on an item.

ii) But **exclusive lock can be hold by only one transaction**.

- **Example of a schedule denoting shared and exclusive locks :** Consider following schedule in which initially A=100. We deduct 50 from A in T1 transaction and Read the data item A in transaction T2. The scenario can be represented with the help of locks and concurrency control manager as follows :

	T₁	T₂	Concurrency control manager
	T_1	T_2	Concurrency control manager
	Lock-X(A)		
Exclusive lock			Grant X(A,T_1) because in T_1 there is write operation.
	R(A)		
	A=A − 50		
	W(A)		
	Unlock(A)		
		Lock-S(A)	
Shared lock			Grant S(A,T_2) because in T_2 there is read operation
		R(A)	
		Unlock(A)	

Two Phase Locking Protocol

- 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 -

Lock(A)
Lock(B)
Lock(C)
....
... } Lock Point
...
Unlock(A)
Unlock(B)
Unlock(C)

Lock point

Locks get acquired (Growing phase)

Locks are released (Shrinking phase)

Transaction begins

Transaction ends

Consider following transactions

T₁	T₂
Lock-X(A)	Lock-S(B)
Read(A)	Read(B)
A=A−50	Unlock-S(B)
Write(A)	
Lock-X(B)	
Unlock-X(A)	
B=B+100	Lock-S(A)
Write(B)	Read(A)
Unlock-X(B)	Unlock-S(A)

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

In above transactions T_1 is in two phase locking mode but transaction T_2 is not in two phase locking. Because in T_2, 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 released. Thus we get lock-unlock-lock-unlock sequence. Clearly this is not possible in two phase locking.

Q) 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 $R_1(A);R_2(A);R_1(B);W_1(B)$

Advantages of two phase locking

(1) It ensures serializability.

Disadvantages of two phase locking protocol

(1) It leads to dealocks.

(2) It leads to cascading rollback.

Problems in two phase locking

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

T_1	T_2
Lock-X(A)	Lock-X(B)
Read(A)	Read(B)
A=A−50	B=B+100
Write(A)	Write(B)
Delayed, wait for T_2 to release Lock on B	Delayed, wait for T_1 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 -

T_1	T_2	T_3
Read(A)		
Read(B)		
C=A+B		
Write(C)		
	Read(C)	
	Write(C)	
		Read(C)

When T_1 writes value of C then only T_2 can read it. And when T_2 writes the value of C then only transaction T_3 can read it. But if the transaction T_1 gets failed then automatically transactions T_2 and T_3 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.

Types of Two Phase Locking

1) **Strict two phase locking** : The strict 2PL protocol is a basic two phase protocol **but all the exclusive mode locks** be **held until the transaction commits**. That means in other words all the **exclusive locks** are **unlocked** only after the **transaction is committed**. That also means that if T_1 has exclusive lock, then T_1 will release the exclusive lock only after commit operation, then only other transaction is allowed to read or write. For example - Consider two transactions

T_1	T_2
W(A)	
	R(A)

If we apply the locks then

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

    Thus only after commit operation in T_1, we can unlock the exclusive lock. This ensures the strict serializability.

    Thus compared to basic two phase locking protocol, the advantage of strict 2PL protocol is it ensures strict serializability.

2) **Rigorous two phase locking** : This is stricter two phase locking protocol. Here all locks are to be held until the transaction commits. The transactions can be seriealized in the order in which they commit.

Example - Consider transactions

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

If we apply the locks then

T₁
Lock-S(A)
R(A)
Lock-X(B)
R(B)
W(B)
Commit
Unlock(A)
Unlock(B)

Thus the above transaction uses rigorous two phase locking mechanism.

Q) Consider the following two transactions:

T_1: read(A)Read(B);
If A=0 then B=B+1;
Write(B)
T_2: read(B); read(A)
If B=0 then A=A+1
Write(A)

Add lock and unlock instructions to transactions T_1 and T_2, so that they observe two phase locking protocol. Can the execution of these transactions result in deadlock ?

Solution:

T₁	T₂
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Read(B)	Read(A)
if A=0 then B=B+1	if B=0 then A=A+1
Write(B)	Write(A)
Unlock(A)	Unlock(B)
Commit	Commit
Unlock(B)	Unlock(A)

This is lock-unlock instruction sequence help to satisfy the requirements for strict two phase locking for the given transactions.

The execution of these transactions result in deadlock. Consider following partial execution scenario which leads to deadlock.

T_1	T_2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Now it will wait for T_2 to release exclusive lock on A	Now it will wait for T_1 to release exclusive lock on B

Part III : Introduction to Database Recovery Protocol

Recovery Concepts

Purpose of Database Recovery

- The purpose of recovery is to bring the database into the last consistent stage prior to occurrence of failure.

- The recovery must preserve all the ACID properties of transaction. The ACID properties are - Atomicity, consistency, isolation and durability.

- Thus recovery ensures high availability of the database for transaction purpose.

- For example - If the system crashes before the amount transfer from one account to another then either one or both the accounts may have incorrect values. Here the database must be recovered before the modification takes place.

Types of Failure

There are three types of failures that occur commonly -

1) **Transaction failure :** Transactions may fail because of incorrect input, deadlock, incorrect synchronization.

2) **System failure :** System may fail because of addressing error, application error, operating system fault, RAM failure, etc.

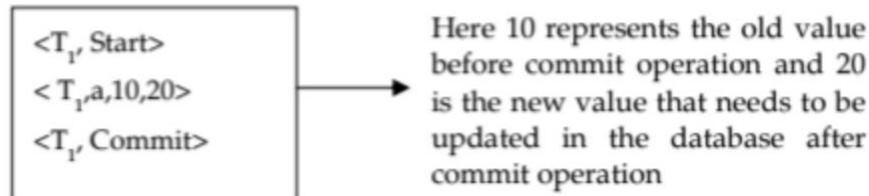3) **Media failure :** Disk head crash, power disruption, etc.

Transaction Log

- For recovery from any type of failure data values prior to modification (**BFIM - BeFore Image**) and the new value after modification (**AFIM – AFter Image**) are required. These values and other information is stored in a sequential file called **Transaction log**.

- Log is the most commonly used structure for recording the modifications that as to be made in the actual database. Hence during the recovery procedure a **log file** is maintained.

- A log record maintains four types of operations. Depending upon the type of operations there are four types of log records-
  1. **<Start>** Log record : It is represented as $<T_i, Start>$

  2. **<Update>** Log record

  3. **<Commit>** Log record : It is represented as $<T_i, Commit>$

  4. **<Abort>** Log record : It is represented as $<T_i, Abort>$

- The log contains various fields as shown in following figure. This structure is for <update> operation

Transaction ID(T_i)	Data Item Name	Old Value of Data Item	New Value of Data Item

- For example : The sample log file is

```
<T₁, Start>
< T₁,a,10,20>
<T₁, Commit>
```

Here 10 represents the old value before commit operation and 20 is the new value that needs to be updated in the database after commit operation

The log must be maintained on the stable storage and the entries in the log file are maintained before actually updating the physical database.

Concept of Data Caching

- Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk.
- The writing to the disk is controlled by **Modified** and **Pin-Unpin bits**.
    - **Pin-Unpin :** Instructs the operating system not to flush the data item.
    - **Modified :** Indicates the AFIM of the data item.

Data Update

The data can be updated using four ways. These are,

1) **Immediate Update :** In this method, as soon as the data is modified in cache, the data is updated on the disk also.

2) **Deferred Update :** All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.

3) **Shadow Update :** The modified data items of cache are not overwritten to the disk, but a separate copy of modified data items is maintained at different location on the disk.

4) **In-place Update :** The disk version of the data item is overwritten by the cache version.

UNDO/REDO(Roll-back/Roll-forward)

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force written to disk and the updates are recorded in the database.

In order to maintain the atomicity of transaction, the operations can be redone or undone.

UNDO : This is an operation in which we restore all the old values (BFIM - BeFore Modification Image) onto the disk. This is called **roll-back operation**.

REDO : This is an operation in which all the modified values(AFIM - AFter Modification Image) are restored onto the disk. This is called **roll-forward operation**.

These operations are recorded in the log as they happen.

Difference between UNDO and REDO

Sr. No.	UNDO	REDO
1.	Makes a change go away.	Reproduces a change.
2.	Used for rollback and read consistency.	Used for rolling forward the changes.
3.	Protects the database from inconsistent reads.	Protects from data loss.

Steal/No-steal and Force/No-force

There are possible ways for flushing database cache to database disk :

1. Steal : Cache can be flushed before transaction commits.

2. No-Steal : Cache cannot be flushed before transaction commit.

3. Force : Cache is immediately flushed (forced) to disk.

4. No-Force : Cache is deferred until transaction commits.

Write Ahead Logging

- Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. This rule is called the **write-ahead logging**.

- This rule is necessary because - In the event of a crash or ROLLBACK, the original content contained in the rollback journal is played back into the database file to revert the database file to its original state.

Check-pointing

- Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.

- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

- The recovery system reads the logs backwards from the end to the last checkpoint.

- Performing a checkpoint consists of the following operations :
  - o   Suspending executions of transactions temporarily;
  - o   Writing (force-writing) all modified database buffers of committed transactions out to disk;
  - o   Writing a checkpoint record to the log; and
  - o   Writing (force-writing) all log records in main memory out to disk.

- A checkpoint record usually contains additional information, including a list of transactions active at the time of the checkpoint.
- Many recovery methods (including the deferred and immediate update methods) need this information when a transaction is **rolled back**, as all transactions active at the time of the checkpoint and any subsequent ones may need to be redone.
- Since checkpoints cause some loss in performance while they are being taken, their frequency should be reduced if fast recovery is not critical.
- If we need fast recovery check-pointing frequency should be increased. If the amount of stable storage available is less, frequent check-pointing is unavoidable.

NO-UNDO/REDO Recovery based on Deferred Update

1. Deferred Database Modification :

- In this technique, the database is not updated immediately.
- Only log file is updated on each transaction.
- When the transaction reaches to its commit point, then only the database is physically updated from the log file.
- In this technique, if a transaction fails before reaching to its commit point, it will not have changed database anyway. Hence there is no need for the UNDO operation. The REDO operation is required to record the operations from log file to physical database. Hence deferred database modification technique is also called as **NO UNDO/REDO algorithm**.

  i) **UNDO (T_i)** : The transaction T_i needs to be undone if the log contains $<T_i,Start>$ but does not contain $<T_i,Commit>$. In this phase, it restores the values of all data items updated by T_i to the old values.

  ii) REDO (T_i) : The transaction Ti needs to be redone if the log contains both $<T_i,Start>$ and $<T_i,Commit>$. In this phase, the data item values are set to the new values as per the transaction. After a failure has occurred log record is consulted to determine which transaction need to be redone.

- For example :

Consider two transactions T_1 and T_2 as follows :

T_1	T_2
Read (A, a)	Read (C, c)
a = a – 10	c = c – 20
Write (A, a)	Write (C, c)
Read (B, b)	
b = b + 10	
Write (B, b)	

If T_1 and T_2 are executed serially with initial values of A = 100, B = 200 and C = 300, then the state of log and database if crash occurs

a) Just after write (B, b)
b) Just after write (C, c)
c) Just after $<T_2$, commit>

The result of above three scenarios is as follows :

Initially the log and database will be,

Log	Database
$<T_1$, Start>	
$<T_1$, A, 90>	
$<T_1$, B, 210>	
$<T_1$, Commit>	
	A = 90
	B = 210
$<T_2$, Start>	
$<T_2$, C, 280>	
$<T_2$, Commit>	
	C = 280

a) Just after write (B, b)

- Just after write operation, no commit record appears in log. Hence no write operation is performed on database. So database retains only old values. Hence A = 100 and B = 200 respectively.
- Thus the system comes back to original position and no redo operation take place.
- The incomplete transaction of T_1 can be deleted from log.

b) Just after write (C, c)

- The state of log records is as follows :
- Note that crash occurs before T_2 commits. At this point T_1 is completed successfully, so new values of A and B are written from log to database. But as T_2 is not committed, there is no redo (T_2) and the incomplete transaction T_2 can be deleted from log.
- The redo (T_1) is done as $< T_1$, commit> gets executed. Therefore A = 90, B = 210 and C = 300 are the values for database.

c) **Just after < T_2, commit>**

The log records are as follows :

< T_1, Start>

< T_1, A, 90>

< T_1, B, 210>

< T_1, Commit>

< T_2, Start>

< T_2, 6, 280>

< T_2, Commit>

←——— **Crash occurs here**

Clearly both T_1 and T_2 reached at commit point and then crash occurs. So both redo (T_1) and redo (T_2) are done and updated values will be A = 90, B = 210, C = 280.

Recovery Technique based on Immediate Update

In this technique, the database is updated during the execution of transaction even before it reaches to its commit point.

If the transaction gets failed before it reaches to its commit point, then the **ROLLBACK Operation** needs to be done to bring the database to its earlier consistent state. That means the effects of operations need to be undone on the database. For that purpose both Redo and Undo operations are both required during the recovery. This technique is known as **UNDO/ REDO** technique.

For example : Consider two transaction T_1 and T_2 as follows :

T_1	T_2
Read(A,a)	Read(C, c)
a = a – 10	c = c – 20
Write(A, a)	Write(C, c)
Read(B, b)	
b = b + 10	
Write(B, b)	

Here T_1 and T_2 are executed serially. Initially A = 100, B = 200 and C = 300.

If the crash occurs

i) **Just after Write(B, b)**   ii) **Just after Write(C, c)**

iii) **Just after < T_2, Commit>**

Then using the immediate database modification approach the result of above three scenarios can be elaborated as follows :

The contents of **log** and **database** is as follows :

Log	Database
$<T_1, Start>$	
$<T_1, A, 100, 90>$	
	A = 90
$<T_1, B, 200, 210>$	
	B = 210
$<T_1, Commit>$	
$<T_2, Start>$	
$<T_2, C, 300, 280>$	
	C = 280
$<T_2, Commit>$	

The recovery scheme uses two recovery techniques -

i) **UNDO (T_i)** : The transaction T_i needs to be undone if the log contains $<T_i, Start>$ but does not contain $<T_i, Commit>$. In this phase, it restores the values of all data items updated by T_i to the old values.

ii) **REDO (T_i)** : The transaction T_i needs to be redone if the log contains both $<T_i, Start>$ and $<T_i, Commit>$. In this phase, the data item values are set to the new values as per the transaction. After a failure has occurred log record is consulted to determine which transaction need to be redone.

   a) **Just after Write (B, b)** : When system comes back from this crash, it sees that there is

   $<T_1, Start>$ but no $<T_1, Commit>$. Hence T_1 must be undone. That means old values of A and B are restored. Thus old values of A and B are taken from log and both the transaction T_1 and T_2 are re-executed.

   b) **Just after Write (C, c)** : Here both the redo and undo operations will occur.

   c) **Undo** : When system comes back from this crash, it sees that there is $<T_2, Start>$ but no $<T_2, Commit>$. Hence T_2 must be undone. That means old values of C is restored.

   Thus old value of C is taken from log and the transaction T_2 is re-executed.

   d) **Redo** : The transaction T_1 must be done as log contains both the $<T_1, Start>$ and $<T_1, Commit>$

   So A = 90, B = 210 and C = 300

   e) **Just after $<T_2, Commit>$** : When the system comes back from this crash, it sees that there are two transaction T_1 and T_2 with both start and commit points. That means T_1 and T_2 need to be redone. So A = 90, B = 210 and C = 280.

Shadow Paging

- Shadow paging is a recovery scheme in which database is considered to be made up of number of fixed size disk pages.
- A directory or a page table is constructed with n number of pages where each ith page points to the ith database page on the disk.
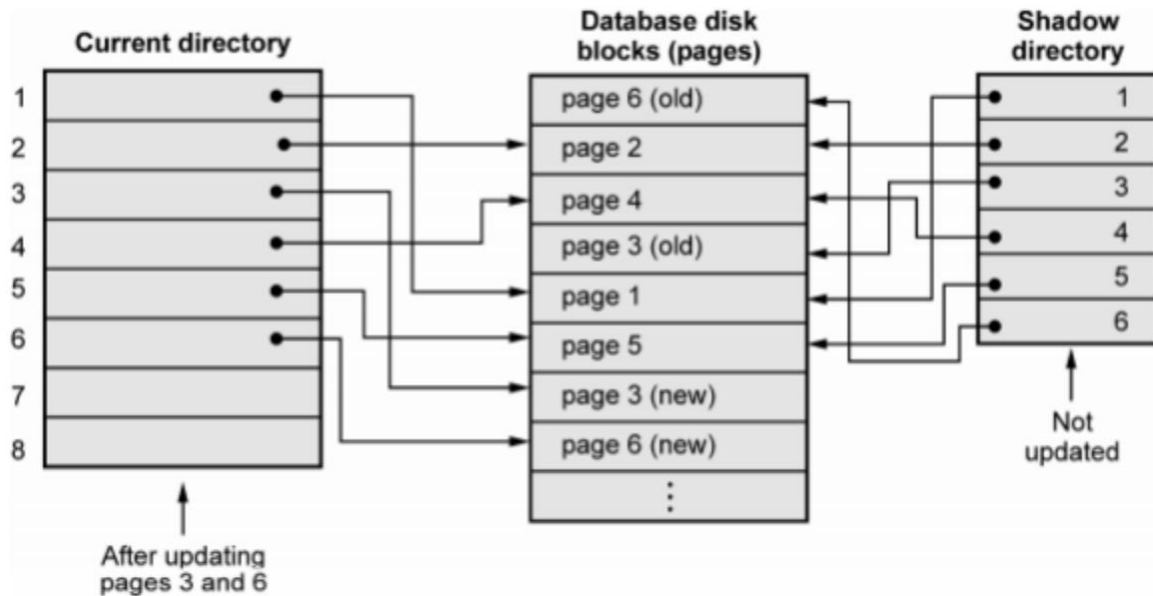
Demonstration of Shadow Paging

- The directory can be kept in the main memory.
- When a transaction begins executing, the **current directory**-whose entries point to the most recent or current database pages on disk-is copied into a another directory called **shadow directory**.
- The shadow directory is then saved on **disk** while the current directory is used by the transaction.
- During the execution of transaction, the shadow directory is never modified.
- When a write operation is to be performed then the **new copy** of modified database page is created but the old copy of database page is never overwritten. This newly created database page is written somewhere else.
- The current directory will point to newly modified web page and the shadow page directory will point to the old web page entries of database disk.
- When the failure occurs then the modified database pages and current directory is discarded.
- The state of database before the failure occurs is now available through the shadow directory and this state can be recovered using shadow directory pages.
- This technique does not require any UNDO/REDO operation.