UNIT – IV

Transaction Management: Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for serializability, Lock Based Protocols, Timestamp Based Protocols, Validation- Based Protocols, Multiple Granularity, Recovery and Atomicity, Log–Based Recovery, Recovery with Concurrent Transactions.

1. TRANSACTION

Definition: A transaction is a single logical unit consisting of one or more database access operation.

Example: Withdrawing 1000 rupees from ATM.

The following set of operations are performed to withdraw 1000 rupees from database

i.	Read current balance from Database	(Let say 5000 rupees)	one Transaction
ii.	Deduct 1000 from current balance	(5000 – 1000 = 4000)	
iii.	Update current balance in Database	(4000 rupees)	

- Every transaction is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one *consistent state* to another *consistent state*.
- DBMS transactions must satisfy ACID properties (atomic, consistent, isolated and durable).

2. ACID PROPERTIES

ACID properties are used for maintaining the integrity of database during transaction processing. ACID stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- **Atomicity**: This property ensure that either all of the tasks of a transaction are performed or none of them. In simple words it is referred as "*all or nothing rule*".

  Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails. When all parts of the transaction completed successfully, then the transaction said to be success. ("*all or nothing rule*")

  Example: Transferring $100 from account **A** to account **B**.

  (Assume initially, account **A** balance = $400 and account **B** balance = 700$.)

Transferring $100 from account **A** to account **B** has two operations

a) Debiting 100$ from **A**'s balance ($400 -$100 = $300)
b) Crediting 100$ to **B**'s balance ($700+$100 = $800)

Let's say first operation (a) passed successfully while second (b) failed, in this case **A**'s balance would be 300$ while **B** would be having 700$ instead of 800$. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.

ii. Consistency: The consistency property ensures that the database must be in consistent state before and after the transaction. There must not be any possibility that some data is incorrectly affected by the execution of a transaction.

For example, transfering funds from one account to another, the consistency property ensures that the total values of funds in both the accounts is the same before and end of the transaction. i.e., Assume initially, **A** balance = $400 and **B** balance = 700$.

The total balance of **A** + **B** = 1100$ (Before transferring 100$ from **A** to **B**)

The total balance of **A** + **B** = 1100$ (After transferring 100$ from **A** to **B**)

iii. Isolation: For every pair of transactions, one of the transactions should not start execution before the other transaction execution completed, if they use some common data variable. That is, if the transaction T1 is executing and using the data item X, then transaction T2 should not start until the transaction T1 ends, if T2 also use same data item X.

For example, Transaction **T1**: Transfer 100$ from account **A** to account **B**

Transaction **T2**: Transfer 150$ from account **B** to account **C**

Assume initially, **A** balance = **B** balance = **C** balance = $1000

	Transaction T1	**Transaction T2**
10:00 AM	Read A's balance ($1000)	Read B's balance ($1000)
10:01 AM	A balance = A Balance – 100$ (1000-100 = 900$)	B balance = B Balance – 150$ (1000-150 = 850$)
10:02 AM	Read B's balance ($1000)	Read C's balance ($1000)
10:03 AM	B balance = B Balance + 100$ (1000+100 = 1100$)	C balance = C Balance + 150$ (1000+150 = 1150$)
10:04 AM	Write A's balance (900$)	Write B's balance (850$)
10:05AM	Write B's balance (1100$)	Write C's balance (1150$)
10:06 AM	COMMIT	COMMIT

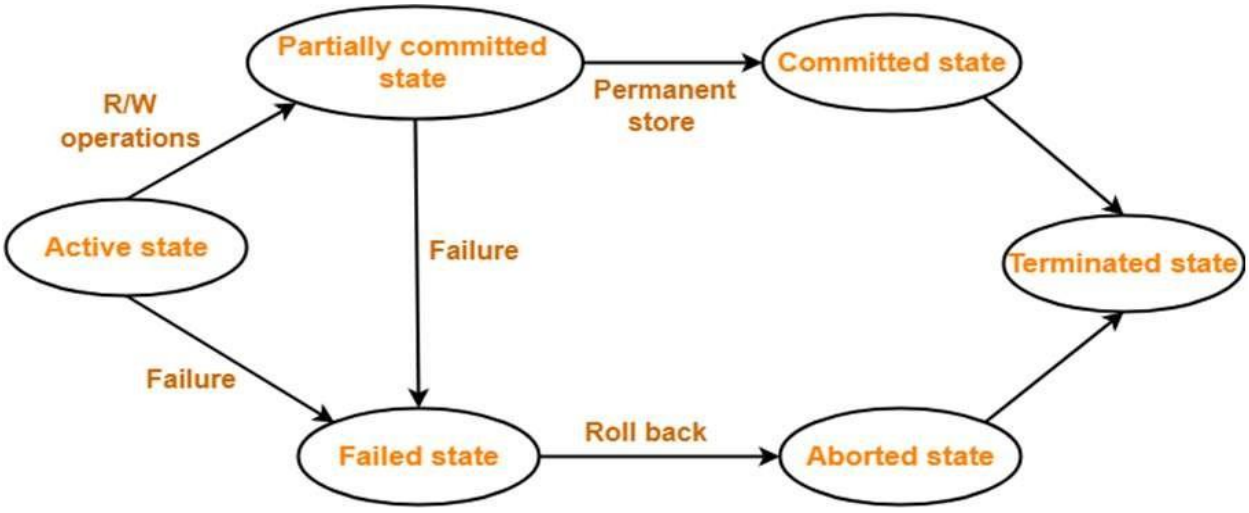
After completion of Transaction **T1** and **T2, A** balance = 900$, **B** balance = 1100$, **C** balance

=1150$. But **B** balance should be 950$. The **B** balance is wrong due to execution of T1 and T2 parallel and in both the transactions, Account **B** is common. The last write in account **B** is at 10:05 AM, so that **B** balance is 1100$ (write in account **B** at 10:04 AM is overwritten).

iv. **Durability**: Once a transaction completes successfully, the changes it has made into the database should be permanent even if there is a system failure. The recovery-management component of database systems ensures the durability of transaction. For example, assume account **A** balance = 1000$. If **A** withdraw 100$ today, then the **A** balance = 900$. After two days or a month, **A** balance should be 900$, if no other transactions done on **A**.

3. STATES OF TRANSACTION

A transaction goes through many different states throughout its life cycle. These states are called as **transaction states**. They are:



Active State:

- This is the first state in the life cycle of a transaction.
- Once the transaction starts executing, then it is said to be in active state.
- During this state it performs operations like READ and WRITE on some data items. All the changes made by the transaction are now stored in the buffer in main memory. They are not updated in database.
- From active state, a transaction can go into either a partially committed state or a failed state.

Partially Committed State:

- When the transaction executes its last statement, then the transaction is said to be in partially committed state.
- Still, all the changes made by the transaction are stored in the buffer in main memory, but they are not updated in the database.
- From partially committed state, a transaction can go into one of two states, a committed state or a failed state.

Committed State:

- After all the changes made by the transaction have been successfully updated in the database, it enters into a **committed state** and the transaction is considered to be fully committed.
- After a transaction has entered the committed state, it is not possible to roll back (undo) the transaction. This is because the system is updated into a new consistent state and the changes are made permanent.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

Failed State:

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

Aborted State:

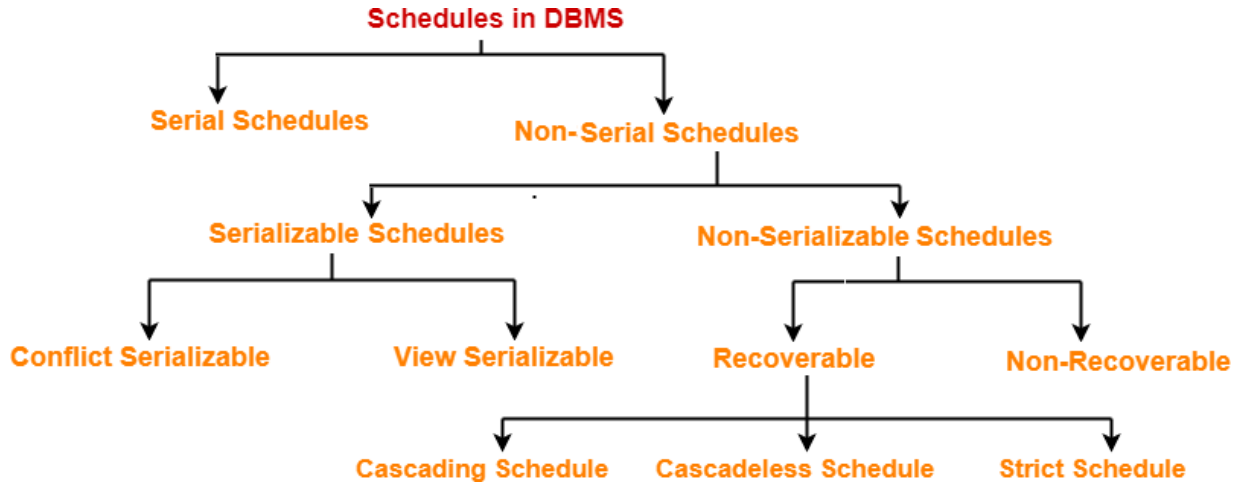
- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an **aborted state**.

Terminated State:

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

4. TYPES OF SCHEDULES – SERIALIZABILITY

In DBMS, schedules may be classified as



i. Serial Schedules:

- All the transactions execute serially one after the other.
- When one transaction executes, no other transaction is allowed to execute.

Examples:

Schedule-1

T1	T2
Read(A) A=A-100 Write(A) Read(B) B=B+100 Write(B) COMMIT	
	Read(A) A=A+500 Write(A) COMMIT

Schedule-2

T1	T2
	Read(A) A=A+500 Write(A) COMMIT
Read(A) A=A-100 Write(A) Read(B) B=B+100 Write(B) COMMIT	

In schedule 1, after T1 completes its execution, transaction T2 executes. So, schedule-1 is a *Serial Schedule*. Similarly, in schedule-2, after T2 completes its execution, transaction T1 executes. So, schedule -2 is also an example of a *Serial Schedule*.

ii. Non-Serial Schedules:

- In non-serial schedules, multiple transactions execute concurrently.
- Operations of all/some of the transactions are inter-leaved or mixed with each other.
- Some non-serial schedules may lead to inconsistency of the database and may produce wrong results.

Examples:

Schedule-1

T1	T2
Read(A)	
A=A-100	
Write(A)	
	Read(A)
	A=A+500
Read(B)	
B=B+100	
Write(B)	
COMMIT	
	Write(A)
	COMMIT

Schedule-2

T1	T2
	Read(A)
Read(A)	
A=A-100	
Write(A)	
	A=A+500
Read(B)	
B=B+100	
Write(B)	
COMMIT	
	Write(A)
	COMMIT

In schedule-1 and schedule-2, the two transactions T1 and T2 executing concurrently. The operations of T1 and T2 are interleaved. So, these schedules are **Non-Serial Schedule**.

iii. Serializable Schedules:

- A non-serial schedule of 'n' transactions is equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule**.
- In other words, the results produced by the transactions in a serial schedule are equal to the result produced by the same transactions in some non-serial schedule, then that non-serial schedule is called as serializability.
- Serializable schedules behave exactly same as serial schedules.
- Even though, Serial Schedule and Serializable Schedule produce same result, there are some differences they are

Serial Schedules	Serializable Schedules
Concurrency is not allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
It leads to less resource utilization and CPU throughput.	It improves both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.

Serializability is mainly of two types. They are:

- Conflict Serializability
- View Serializability

Conflict Serializability: If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Two operations are called as **conflicting operations** if all the following conditions hold true

(1) Both the operations belong to different transactions

(2) Both the operations are on the **same data item**

(3) At least one of the two operations is a write operation

Schedule – 1	
T1	**T2**
Read(A)	
	Read(A)

Schedule – 2	
T1	**T2**
Read(A)	
	Write(A)

Schedule - 3	
T1	**T2**
Write(B)	
	Read(A)

Schedule - 4	
T1	**T2**
Write(B)	
	Write(B)

In Schedule -1, only rule (1) & (2) are true, but rule (3) is not holding. So, the operations are not conflict.

In Schedule -2, rule (1), (2) & (3) are true. So, the operations are conflict.

In Schedule -3, only rule (1) & (3) are true, but rule (2) is not holding. So, the operations are not conflict.

In Schedule -4, rule (1), (2) & (3) are true. So, the operations are conflict.

Testing of Conflict Serializability: Precedence Graph is used to test the Conflict Serializability of a schedule. The algorithm to draw precedence graph is

(1) Draw a node for each transaction in Schedule **S.**

(2) If T_a reads X value written by T_b, then draw arrow from $T_b \rightarrow T_a$.

(3) If T_b writes X value after it has been read by T_a, then draw arrow from $T_a \rightarrow T_b$.

(4) If T_a writes X after T_b writes X, then draw arrow from $T_b \rightarrow T_a$.

If the precedence graph has no cycle, then Schedule **S** is known as conflict serializable. If a precedence graph contains a cycle, then **S** is not conflict serializable.

Problem-01: Check whether the given schedule S is conflict serializable or not.

$S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)$

Solution:

Given that $S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)$.

The schedule for the above operations is

Schedule-1

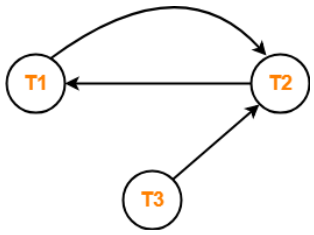
T1	T2	T3
Read(A)		
	Read(A)	
Read(B)		
	Read(B)	
		Read(B)
Write(A)		
	Write(B)	

List all the conflicting operations and determine the dependency between the transactions

(Thumb rule to find conflict operations: For each Write(X) in T_a, make a pair with each Read(X) and Write(X) in T_b. **The order is important in each pair** i.e., for example, Read after Write on X or write after read on X in the given schedule.**)**

- $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
- $R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Draw the precedence graph:



There exists a cycle in the above graph. Therefore, the schedule S is not conflict serializable.

Problem-02: Check whether the given schedule S is conflict serializable schedule.

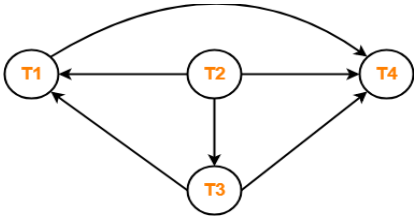
Schedule – S

T1	T2	T3	T4
	Read(X)		
		Write(X)	
		COMMIT	
Write(X)			
COMMIT			
	Write(Y)		
	Read(Z)		
	COMMIT		
			Read(X)
			Read(Y)
			COMMIT

Solution: List all the conflicting operations to determine the dependency between transactions.

$R_2(X)$, $W_3(X)$ $(T_2 \rightarrow T_3)$
$W_3(X)$, $W_1(X)$ $(T_3 \rightarrow T_1)$
$W_3(X)$, $R_4(X)$ $(T_3 \rightarrow T_4)$
$R_2(X)$, $W_1(X)$ $(T_2 \rightarrow T_1)$
$W_1(X)$, $R_4(X)$ $(T_1 \rightarrow T_4)$
$W_2(Y)$, $R_4(Y)$ $(T_2 \rightarrow T_4)$

Draw the precedence graph:



There exists no cycle in the precedence graph. Therefore, the schedule S is conflict serializable.

View Serializability: Two schedules S1 and S2 are said to be **view equivalent** if both of them satisfy the following three rules:

(1) **Initial Read:** The first read operation on each data item in both the schedule must be same.

- For each data item X, If first read on X is done by transaction T_a in schedule S1, then in schedule2 also the first read on X must be done by transaction T_a only.

(2) **Updated Read:** It should be same in both the schedules.

- If Read(X) of T_a followed by Write(X) of T_b in schedule S1, then in schedule S2 also, Read(X) of T_a must follow Write(X) of T_b .

(3) **Final write:** The final write operation on each data item in both the schedule must be same.

- For each data item X, if X has been updated at last by transaction T_i in schedule S1, then in schedule S2 also, X must be updated at last by transaction T_i.

View Serializability Definition: If a given schedule is view equivalent to some serial schedule, then it is called as a view serializable schedule.

Note: *Every conflict serializable schedule is also view serializable schedule but not vice-versa*

Problem 03: Check whether the given schedule S is view serializable or not

Schedule – 1

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Solution:

For the given schedule-1, the serial schedule can be schedule -2

Schedule-1 (S1)

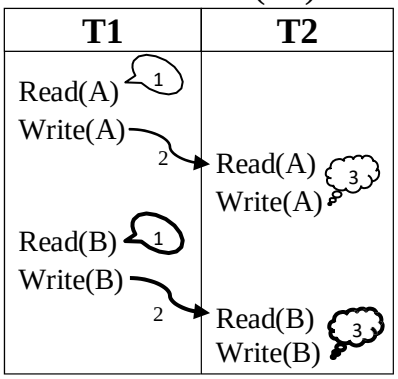
T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule-2 (S2)

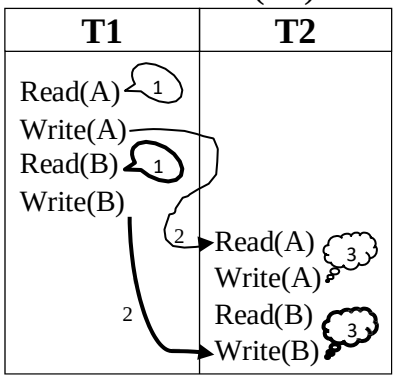
T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Now let us check whether the three rules of view-equivalent satisfy or not.

Schedule-1 (S1)



Schedule-2 (S2)



Rule 1: Initial Read

First Read(A) is by T1 in S1 and in S2 also the first Read(A) is by T1 only.

First Read(B) is by T1 in S1 and in S2 also the first Read(B) is by T1 only.

Rule 2: Updated Read

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Rule 3: Final Write

The final Write(A) is by T2 in S1 and in S2 also the final Write(A) is by T2 only

The final Write(B) is by T2 in S1 and in S2 also the final Write(B) is by T2 only

Conclusion: Hence, all the three rules are satisfied in this example, which means Schedule S1 and S2 are view equivalent. Also, it is proved that schedule S2 is the serial schedule of S1. Thus we can say that the S1 schedule is a view serializable schedule.

Note: *Other way of solving it is, if we are able to prove that S1 is conflict serializable, then S1 is also view serializable. (Refer conflict serializable problems. Every conflict serializable schedule is also view serializable but not vice-versa.)*

5. IMPLEMENTATION OF ATOMICITY AND DURABILITY

The recovery-management component of a DBMS supports atomicity and durability by a variety of schemes. The simplest scheme to implement it is Shadow copy.

Shadow copy: In shadow-copy scheme,

- A transaction that wants to update the database first creates a complete copy of the database.
- All updates are done on the new database copy, leaving the original copy, untouched.
- If at any point the transaction has to be aborted, the system simply deletes the new copy. The old copy of the database has not been affected.

- If the transaction complete successfully, then the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the original copy of the database. The old copy of the database is then deleted. Figure below depicts the scheme, showing the database state before and after the update.

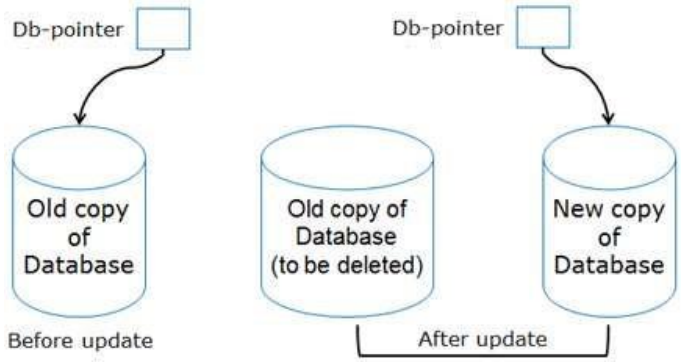


Figure: Shadow copy technique for atomicity and durability

6. RECOVERABILITY

During execution, if any of the transaction in a schedule is aborted, then this may leads the database into inconsistence state. If anything goes wrong, then the completed operations in the schedule needs to be undone. Sometimes, these undone operations may not possible. The recoverability of schedule depends on undone operations.

> If a transaction reads a data value that is updated by an uncommitted transaction, then this type of read is called as a **dirty read.**

Irrecoverable Schedule: In a schedule, if a transaction T_a performs a dirty read operation from other transaction T_b and T_a commits before T_b then such a schedule is known as an **Irrecoverable Schedule**.

Example: Consider the following schedule

T1	T2
Read(A) Write(A) | | | | | ROLLBACK	Read(A) **//Dirty Read** Write(A) COMMIT

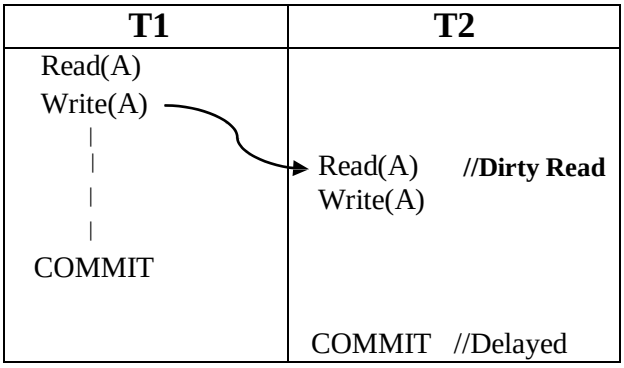
Here,

- T2 performs a dirty read operation.

- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 cannot recover since it has already committed.

Recoverable Schedules: In a schedule, if a transaction $\mathbf{T_a}$ performs a dirty read operation from other transaction $\mathbf{T_b}$ and $\mathbf{T_a}$ commit operation delayed till $\mathbf{T_b}$ commit, then such a schedule is known as an **Irrecoverable Schedule**.

Example: Consider the following schedule-



Here,

- T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.
- In case, T1 would have failed, T2 has a chance to recover by rolling back.

Checking Whether a Schedule is Recoverable or Irrecoverable:

Check if there exists any dirty read operation.

- If there does not exist any dirty read operation, then the schedule is surely recoverable.
- If there exists any dirty read operation, then
  - If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then the schedule is irrecoverable.
  - If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

7. IMPLEMENTATION OF ISOLATION

Isolation determines how transactions integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only one transaction that is accessing the resources in a database system.

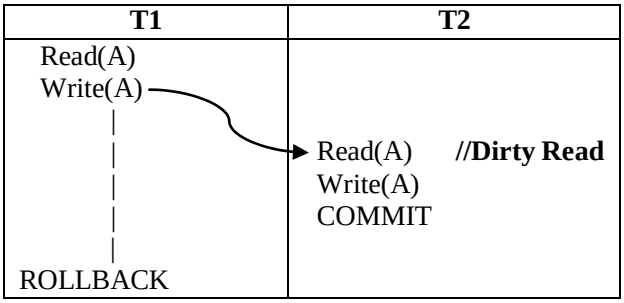
Isolation level defines the degree to which a transaction must be isolated from the data modifications made by any other transactions in the database system. The phenomena's used to define levels of isolation are:

a) Dirty Read

b) Non-repeatable Read

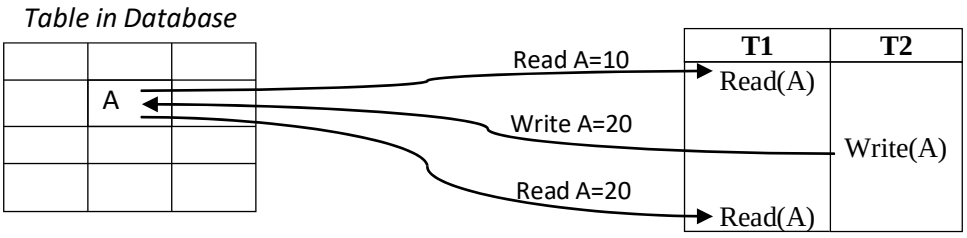
c) Phantom Read

Dirty Read: If a transaction reads a data value updated by an uncommitted transaction, then this type of read is called as dirty read.



As T1 aborted, the results produced by T2 become wrong. This is because T2 read A (Dirty Read) which is updated by T1.

Non-Repeatable Read: Non repeatable read occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **UPDATES** from another transaction.



First, T1 reads data item A and get A=10

Next, T2 writes data item A as A = 20

Last, T1 reads data item A and get A=20

Other example for Non-repeatable read:

Table: STUDENT_DATA before T2

A	B	C
100	5	**10**
101	5	20
102	6	30

Table: STUDENT_DATA after T2

A	B	C
100	5	**15**
101	5	20
102	6	30

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20) = **30**

T2: UPDATE STUDENT_DATA SET C = 15 WHERE A=100; Answer, in First row C changes to 15

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (15+20) = **35**

Phantom reads: Phantom reads occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **INSERTS** and/or **DELETES** from another transaction.

Non-repeatable read	Phantom read
When T1 perform second read, there is no change in no of rows in the given table	When T1 perform second read, the no of rows either increase or decrease.
T2 perform UPDATE operation on the given table	T2 perform INSERT and/or DELETE operation on the given table

Example for Phantom read:

Table: STUDENT_DATA before T2

A	B	C
100	5	10
101	5	20
102	6	30

Table: STUDENT_DATA after T2

A	B	C
100	5	10
101	5	20
102	6	30
103	**5**	**25**

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20) = 30

T2: INSERT INTO STUDENT_DATA VALUES(103, 5, 25); Answer, in First row C changes to 15

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20+25) = 55

Based on these three phenomena, SQL define four isolation levels. They are:

(1) **Read uncommitted:** This is the lowest level of isolation. In this level, one transaction may read the data item modified by other transaction which is not committed. It mean dirty read is allowed. In this level, transactions are not isolated from each other.

(2) **Read Committed:** This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read/write lock on the data object, and thus prevents other transactions from reading, updating or deleting it.

(3) **Repeatable Read:** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read. So other transactions cannot read, update or delete these data items.

(4) **Serializable:** This is the highest isolation level. A *serializable* execution is guaranteed to be a serial schedule. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The table given below clearly depicts the relationship between isolation levels and the read phenomena and locks.

Isolation Level	Dirty Read	Non-repeatable read	Phantom Read
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

From the above table, it is clear that serializable isolation level is better than others.

8. CONCURRENCY CONTROL

- **Concurrency** is the ability of a database to execute multiple transactions simultaneously.
- Concurrency control is a mechanism to manage the simultaneously executing multiple transactions such that no transaction interfere with other transaction.
- Executing multiple transactions concurrently improves the system performance.
- Concurrency control increases the throughput and reduces waiting time of transactions.
- If Concurrency Control is not done, then it may leads to problems like lost updates, dirty read, non-repeatable read, phantom read etc. (Refer section 7 for more details)
- **Lost Updates:** It occur when two transactions update same data item at the same time. In this the first write is lost and only the second write is visible.

Concurrency control Protocols:

The concurrency can be controlled with the help of the following Protocols

(1) Lock-Based Protocol
(2) Timestamp-Based Protocol
(3) Validation-Based Protocol

9. LOCK-BASED PROTOCOL

- Lock assures that one transaction should not retrieve or update a record which another transaction is updating.
- For example, traffic at junction, there are signals which indicate stop and go. When one side signal is green (vehicles allowed passing), then other side signals are red (locked. Vehicles not allowed passing). Similarly, in database transaction when one transaction operations are under execution, the other transactions are locked.
- If at a junction, green signal is given to more than one side, then there may be chances of accidents. Similarly, in database transactions, if the locking is not done properly, then it will display the inconsistent and corrupt data.

There are two lock modes: **(1).** Shared Lock **(2).** Exclusive Lock

Shared Locks are represented by **S**. If a transaction $\mathbf{T_i}$ apply shared lock on data item **A**, then $\mathbf{T_i}$ can only read **A** but not write into **A**. Shared lock is requested using lock-S instruction.

Exclusive Locks are represented by **X**. If a transaction $\mathbf{T_i}$ apply exclusive lock on data item **A**, then $\mathbf{T_i}$ can read as well as write data item **A**. Exclusive lock is requested using lock-X instruction.

Lock Compatibility Matrix:

- Lock Compatibility Matrix controls whether multiple transactions can acquire locks on the same resource at the same time.

		Transaction $\mathbf{T_i}$ applied	
		Shared	**Exclusive**
Transaction $\mathbf{T_j}$ request for	**Shared**	√	X
	Exclusive	X	X

- If a transaction $\mathbf{T_i}$ applied shared lock on data item **A**, then $\mathbf{T_j}$ can also be allowed to apply shared lock on **A**.
- If a transaction $\mathbf{T_i}$ applied shared lock on data item **A**, then $\mathbf{T_j}$ is not allowed to apply exclusive lock on **A**.
- If a transaction $\mathbf{T_i}$ applied exclusive lock on data item **A**, then $\mathbf{T_j}$ is not allowed to apply shared lock on **A**.
- If a transaction $\mathbf{T_i}$ applied exclusive lock on data item **A**, then $\mathbf{T_j}$ is not allowed to apply exclusive lock on it.
- Any number of transactions can hold shared locks on a data item, but if any transaction holds an exclusive lock on a data item, then other transactions are not allowed to hold any lock on that data item.

- Whenever a transaction wants to read a data item, it should apply shared lock and when a transaction wants to write it should apply exclusive lock. If the lock is not applied, then the transaction is not allowed to perform the operation.

There are four types of lock protocols available. They are:

(1) Simplistic lock protocol

- It is the simplest locking protocol.
- It considers each read/write operation of a transaction as individual.
- It allows transactions to perform write/read operation on a data item only after obtaining a lock on that data item.
- Transactions unlock the data item immediately after completing the write/read operation.
- When a transaction needs to perform many read and write operations, for each operation lock is applied before performing it and release the lock immediately after completion of the operation.

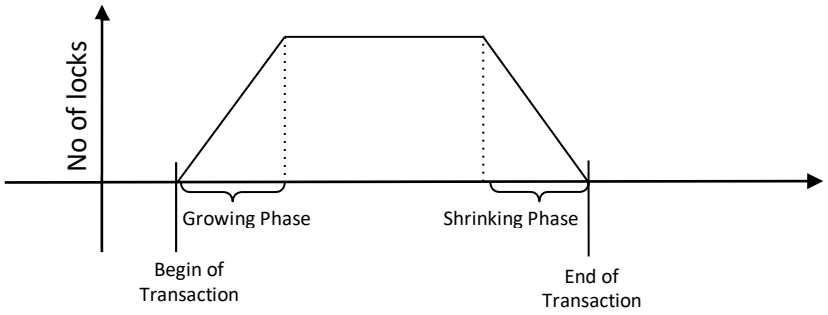
(2) Pre-claiming Lock Protocol

- In pre-claiming Lock Protocol, for each transaction a list is prepared consisting of the data items and type of lock required on each of the data item.
- Before initiating an execution of the transaction, it requests DBMS to issue all the required locks as per the list.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.

(3) Two-phase locking (2PL) protocol

- Every transaction execution starts by acquiring few locks or zero locks. During execution it acquire all other required locks one after the other.

- When a transaction releases any of the acquired locks then it cannot acquire any more new locks. But, it can only release the acquired locks one after the other during remaining execution of that transaction.



The Two Phase Locking (2PL) has two phases. They are:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released. (Only get new locks but no release of locks).

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired. (Only release locks but no more getting new locks).

Example:

Time	T1	T2
0	***LOCK-S(A)***	
1		***LOCK-S(A)***
2	Read(A)	
3		Read(A)
4	***LOCK-X(B)***	
5	--	
6	Read(B)	
7	B = B + 100	
8	Write(B)	
9	***UNLOCK(A)***	
10		***LOCK-X(C)***
11	***UNLOCK(B)***	--
12		Read(C)
13		C = C + 500
14		Write(C)
15	COMMIT	
16		***UNLOCK(A)***
17		***UNLOCK(C)***
18		COMMIT

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

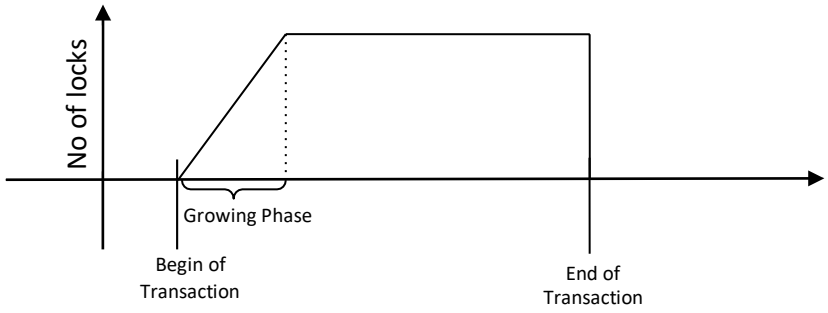
- **Growing phase:** from step 1-5 (After first lock onwards)
- **Shrinking phase:** from step 10-12 (After first unlock onwards)
- **Lock point:** at 5 (No more new locks)

Transaction T2:

- **Growing phase:** from step 2-11 (After first lock onwards)
- **Shrinking phase:** from step 17-18 (After first unlock onwards)
- **Lock point:** at 11 (No more new locks)

(4) Strict Two-phase locking (Strict-2PL) protocol

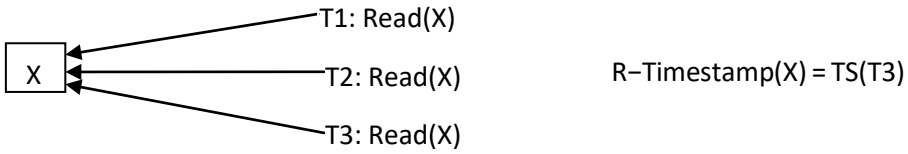
- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



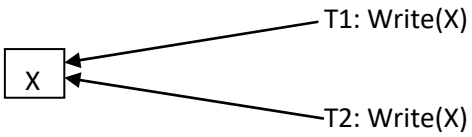
Strict-2PL does not have cascading abort as 2PL does.

10. TIMESTAMP BASED PROTOCOL

- A timestamp is issued to each transaction when it enters into the system.
- It uses either system time or logical counter as a timestamp.
- It is most commonly used concurrency protocol.
- The timestamp of transaction T is denoted as **TS(T)**.
- The system order the transactions based on their arrival time. For example, let the arrival times of transactions T1, T2 and T3 be 9:00AM, 9:01AM and 9:02AM respectively. Then TS(T1) < TS(T2) < TS(T3). (*9:00AM < 9:01AM < 9:02AM*)
- By using timestamp, the system prepares the serializability order. i.e., T1→T2→T3
- The read timestamp of data item X is denoted by **R–timestamp(X)**.
- **R–timestamp(X):** It is the time stamp of the youngest transaction that performed read operation on X.



- The write timestamp of data item X is denoted by **W–timestamp(X)**.
- **W–timestamp(X):** It is the time stamp of the youngest transaction that performed write operation on X.



R–Timestamp(X) = TS(T2)

There are mainly two Timestamp Ordering Algorithms in DBMS. They are:

- Basic Timestamp Ordering
- Thomas Write rule

(1). Basic Timestamp Ordering

- Check the following condition whenever a transaction **Ti** issues a **Read (X)** operation:
  - If W_ timestamp(X) >TS(Ti) then the operation is rejected.
  - If W_ timestamp(X) <= TS(Ti) then the operation is executed.

    (Read is not allowed by Ti, if any younger transactions than Ti write X)
- Check the following condition whenever a transaction Ti issues a **Write(X)** operation:
  - If TS(Ti) < R_ timestamp(X) then the operation is rejected. (Write is not allowed by Ti, if any younger transactions than Ti read X)
  - If TS(Ti) < W_ timestamp(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed. (Write is not allowed by Ti, if any younger transactions than Ti write X and also Ti should be rolled back and restarted later)

(2) Thomas's Write Rule

Thomas Write Rule is a timestamp-based concurrency control protocol which ignores outdated writes. It follows the following steps:

(i). If **R_TS(X) > TS(T_a)**, then abort and rollback **T_a** and reject the operation.

Transaction:**T1** *Arrival = 9:00 AM* *∴ TS(T1) = 9:00 AM*	Transaction:**T2** *Arrival = 9:02 AM* *∴ TS(T1) = 9:02 AM*	**Variable A** Initial A=100
| | | Write(A) (A=200) Reject and Rollback T1	| Read(A) (A=100) | : | (A=100)	A = 100 (R_TS(A) = 9:02AM) A = ~~200~~ 100

(ii). If **W_TS(X) > TS(T_a)**, then don't execute the Write Operation of **T_a** but continue **T_a** processing. This is a case of *Outdated or Obsolete Writes*.

Transaction:**T1** *Arrival = 9:00 AM* *∴ TS(T1) = 9:00 AM*	Transaction:**T2** *Arrival = 9:02 AM* *∴ TS(T1) = 9:02 AM*	**Variable A** Initial A=100
Write(A) (A=500) Reject but continue T1	Write(A) (A=400) : (A=400)	A = 400 (W_TS(A) = 9:02AM) A = ~~500~~ (Outdated write)

(iii). If the condition in (i) or (ii) is not satisfied, then execute Write(X) of **T_a** and set W_TS(X) to TS(**T_a**).

Outdated writes are rejected but the transaction is continued in **Thomas Write Rule** but in Basic TO protocol will reject write operation and terminate such a Transaction.

11. VALIDATION BASED PROTOCOL

In this technique, no concurrency control checking is done while the transaction is under execution. After transaction execution is completed, then only whether concurrency violated or not is checked. It is based on timestamp based protocol. Validation Based Protocol has three phases:

1. **Read phase:** In this phase, the transaction **T_a** read the value of various data items that are required by **T_a** and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
1. **Validation phase:** After Transaction **T_a** execution completed, **T_a** perform a validation test to determine whether it can copy the temporary local variable values to actual database without causing a violation of serializability.
2. **Write phase:** If the validation of the transaction is successful (valid), then the temporary results are written to the database. Otherwise the temporary local variable values of **T_a** is ignored and **T_a** is rolled back.

To perform the validation test, we need to know when the various phases of transaction **T_a** took place. We shall therefore associate three different timestamps with transaction **T_a**.

(i). **Start (T_a)**: the time when **T_a**, started its execution.

(ii). **Validation (T_a):** the time when **T_a** finished its execution and started its validation phase.

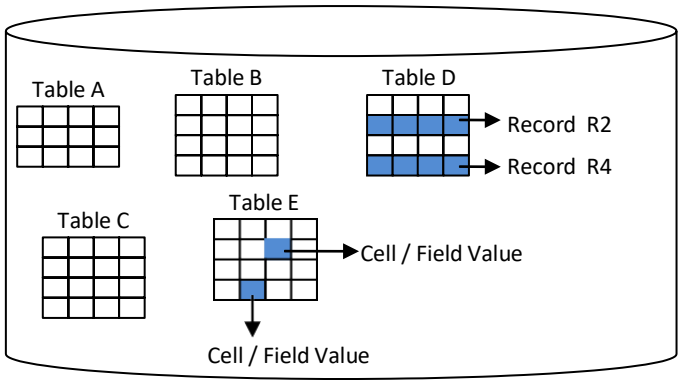
(iii). **Finish (T_a)**: the time when T_a finished its write phase.

The serializability order is determined by changing the timestamp of **T** as TS(**T**) = Validation(**T**). Hence the serializability is determined at the validation process and cannot be decided in advance. Therefore it ensures greater degree of concurrency while executing the transactions.

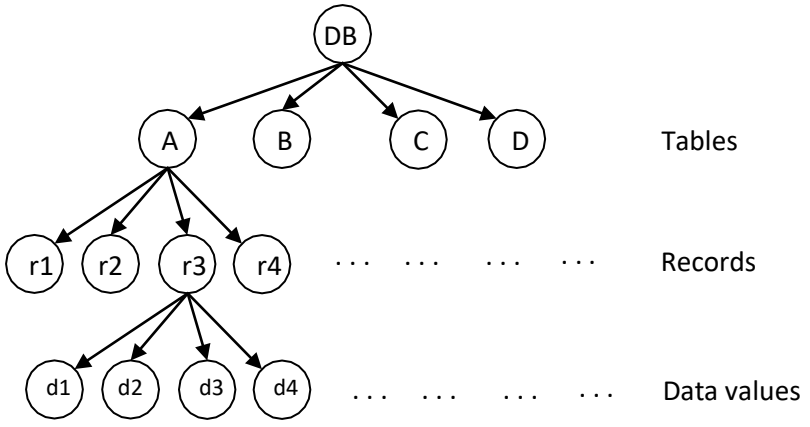
12. MULTIPLE GRANULARITY

The size of data items is often called the **data item granularity**. There exist multiple granularity levels in the DBMS. They are:

- Database
- Table
- Record / row
- Cell / field value



A database contains multiple tables. Each table contains multiple records. Each record contains multiple field values. It is shown in the above figure. For example, consider Table D and Record R2. These two are not mutually exclusive. R2 is a part of D. So granularity means different levels of data where as smaller levels are nested inside the higher levels. Inside database we have tables. Inside table we have records. Inside record we have field values. This can be represented with a tree as shown below.



A lock can be applied at a node, if and only if there does not exist any locks on the decedents (childs and grand childs) of that node. Otherwise lock cannot be applied. If lock is applied on table A, it implies that the lack is also applicable to sub-tree from node A. If lock is applied on database (at root node), it implies the lack is also applicable to all the nodes in the tree.

The larger the object size on which lock is applied, the lower the degree of concurrency permitted. On the other hand, the smaller the object size on which lock is applied, the system has to maintain larger number of locks. More locks cause a higher overhead and needs more disk space. So, what is the best object size on which lock can be applied? It depends on the types of transactions involved. If a typical transaction accesses data values from a record, it is advantageous to have the lock to that one record. On the other hand, if a transaction typically accesses many records in the same table, it may be better to have lock at that table.

Locking at higher levels needs lock details at lower levels. This information is provided by additional types of locks called **intention locks**. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

(1) **Intention-shared (IS):** It indicates that one or more shared locks will be requested on some descendant node(s).

(2) **Intention-exclusive (IX):** It indicates that one or more exclusive locks will be requested on some descendant node(s).

(3) **Shared-intention-exclusive (SIX):** It indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, the shared and exclusive locks, is shown in Figure.

Mode	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Note: In multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

13. RECOVERY AND ATOMICITY

Database needs to be recovered, when the following failures occur.

(1) Transaction failure
(2) System crash
(3) Disk failure

- **Transaction failure:** During transaction execution, if it cannot proceed further, then it needs to abort. This is known as transaction failure. A single transaction failure may influence many transactions or processes. The reasons for transaction failure are:
  - **Logical errors:** It occurs due to some code error or an internal condition error.
  - **System error:** It occurs when the DBMS itself terminates an active transaction due to deadlock or resource unavailability.
- **System crash:** The system may crash due to the external factors such as interruptions in power supply, hardware or software failure. Example: Operating System errors.
- **Disk failure:** In early days of technology evolution, hard-disk drives or storage drives used to fail frequently. Disk failure occurs due to the formation of bad sectors, disk head crash, un-reachable to the disk or any other failure which destroys all or part of disk storage.

When a system crashes, it may have many transactions being executed and many files may be opened for them. When a DBMS recovers from a crash, it must maintain the following:

- It must check the states of all the transactions that were being executed.

- Few transactions may be within the middle of some operation; the DBMS should make sure the atomicity of the transaction during this case.
- It must check for each transaction whether its execution accepted or to be rolled back.
- No transaction is allowed to be in an inconsistent state.

The following techniques facilitate a DBMS in recovering as well as maintaining the atomicity of a transaction:

- Log based recovery
- Check point
- Shadow paging

14. LOG BASED RECOVERY

The log file contains information about the start and end of each transaction and any updates done by the transaction on database items. The log file is saved onto some stable storage so that if any failure occurs, then it can be used to recover the database. The results of all the operation of transaction are first saved in the log and latter updated on the database. The log information is used to recover from system failures.

The log is a sequence of records. It contains the following entries.

- When a transaction $\mathbf{T_i}$ starts execution, the log stores: **< $\mathbf{T_i}$**, Start >
- When a transaction $\mathbf{T_i}$ modifies an item X from old value V_1 to new value V_2 , the log stores: < $\mathbf{T_i}$, X, V_1, V_2>
- When the transaction $\mathbf{T_i}$ execution completed, the log stores: < $\mathbf{T_i}$, commit>
- When the transaction $\mathbf{T_i}$ execution aborted, the log stores: < $\mathbf{T_i}$, abort>

Recovery using Log records

When the system is crashed, then the DBMS checks the log to find which transactions needs to be undo and which need to be redo. There are two major techniques for recovery from non-catastrophic transaction failures. They are deferred updates and immediate updates.

i. **Deferred database modification:** In this technique, all the changes done by the transaction are saved in the system log without modifying the actual database. Once the transaction committed, then only the changes are updated in the database. If a transaction fails before reaching its commit point, it has not changed the database in any way so

UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the system log, because their effect not yet written in the database.

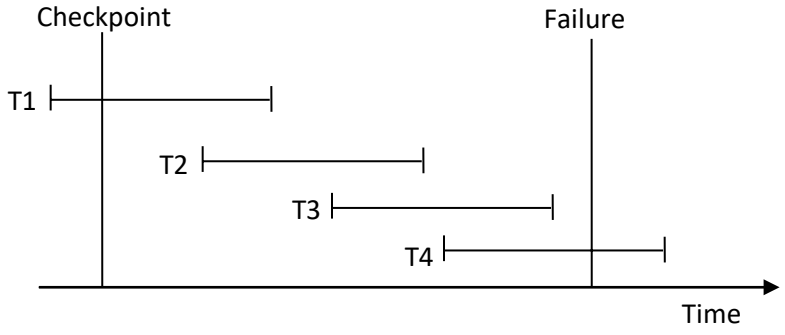
ii. **Immediate database modification:** In this technique, the database is modified immediately after every operation. However, these operations are recorded in the log file before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo.

15. CHECKPOINT – (Recovery with Concurrent Transactions)

- In order to recover database from system crashes, all the transaction operations are first saved in the log file and latter updated on the database. The log file is saved in remote location so that it can be used to recover the database. As time passes, the entries in the log file may grow too big. At the time of recovery, searching the entire log file is very time consuming and an inefficient method. To ease this situation, the concept of 'checkpoint' is introduced.
- **Checkpoint** is a mechanism where all the previous log entries are removed from the log file and their results are updated in the database. The checkpoint is like a bookmark.
- During the execution of the transactions, after executing few operations, a check point is created and saved in the log file. Now the log file contains only entries after checkpoint related to new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.

- If the recovery system sees a log with <T_i, Start> and < T_i, Commit> or just < T_i, Commit>, it puts the transaction T_i in the redo-list.

  For example: In the log file, transaction T1 have only < T_i, commit> and the transactions T2 and T3 have < T_i, Start> and < T_i, Commit>. Therefore T1, T2 and T3 transaction are added to the redo list.
- If the recovery system finds a log with < T_i, Start> but no commit or abort, then it puts the transaction T_i in undo-list.

  For example: Transaction T4 will have < T_i, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed in the middle.
- All the transactions in the undo-list are then undone and their logs are removed.
- All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

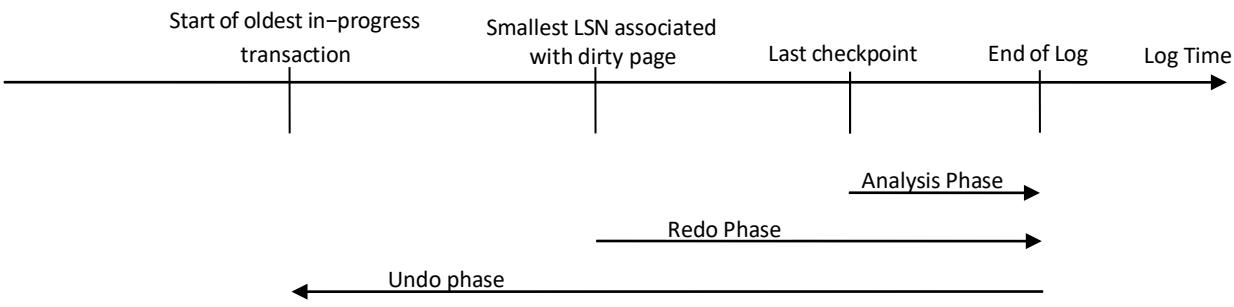
16. ARIES ALGORITHM (Algorithm for Recovery and Isolation Exploiting Semantics)

Algorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics (**ARIES**) is one of the log based recovery method. It uses the Write Ahead Log (WAL) protocol.

Write-ahead logging (WAL): In computer science, **write-ahead logging** (WAL) is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems. The change done by the transactions are first recorded in the log file and written to stable storage at remote location, before the changes are written to the database.

The recovery process of ARIES algorithm has 3 phases. They are:

(1) Analysis phase
(2) Redo Phase
(3) Undo Phase



(1) **Analysis phase:** The recovery subsystem scans the log file forward from the last checkpoint up to the end. The purpose of the scan is to obtain information about the following:

- The starting point from where the redo pass should start.
- The list of transactions to be rolled back in the undo pass.
- The list of dirty pages.

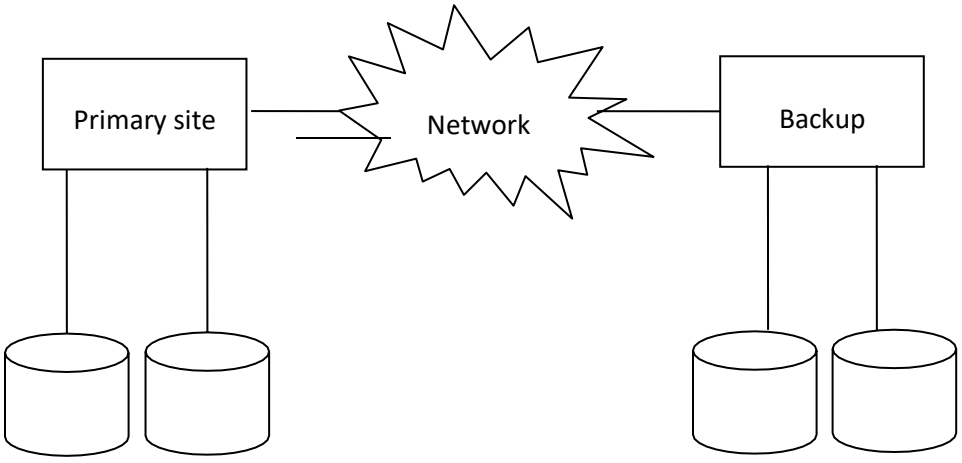
(2) **Redo:** In this phase, the log file is read forward starting from smallest LSN of a dirty page to the end and each update found in the log file is redone. The purpose of this redo pass is to repeat the history to reconstruct the database to the state present at the time of system failure.

(3) **Undo:** The log is scanned backward and updates related to loser transactions are undone. The 'loser transaction' updates are rolled back in reverse chronological order. If any of the aborted transaction operations are undone, then skip them, no need to undo once again.

17. DATABASE BACKUP

The process of creating duplicate copy of database is called database backup. Backup helps to recover against failure of media, hardware or software failures or any other kind of failures that cause a serious data crash.

Database copy is created and stored in the remote area with the help of network. This database is periodically updated with the current database so that it will be in sync with data and other details. This remote database can be updated manually called offline backup. It can be backed up online where the data is updated at current and remote database simultaneously. In this case, as soon as there is a failure of current database, system automatically switches to the remote database and starts functioning. The user will not know that there was a failure.



Some of the backup techniques are as follows:

- **Full backup or Normal backup:** Full backup is also known as Normal backup. In this, an exact duplicate copy of the original database is created and stored every time the backup made. The advantage of this type of backup is that restoring the lost data is very fast as compared to other. The disadvantage of this method is that it takes more time to backup.
- **Incremental Backup:** Instead of backup entire database every time, **backup** only the files that have been updated since the last full backup**.** For this at least weekly once normal backup has to be done. While incremental database backups do run faster, the recovery process is a bit more complicated.
- **Differential backup:** Differential is similar to incremental backup but the difference is that the recovery process is simplified by not clear the archive bit. So a file that is updated after a normal backup will be archived every time a differential backup is run until the next normal backup runs and clears the archive bit.